aws

V2 Developer Guide

Amazon Lex

Amazon Lex: V2 Developer Guide

Table of Contents

What is Amazon Lex V2?

Amazon Lex V2 is an AWS service for building conversational interfaces for applications using voice and text. Amazon Lex V2 provides the deep functionality and flexibility of natural language understanding (NLU) and automatic speech recognition (ASR) so you can build highly engaging user experiences with lifelike, conversational interactions, and create new categories of products.

Amazon Lex V2 enables any developer to build conversational chatbots quickly. With Amazon Lex V2, no deep learning expertise is necessary—to create a chatbot, you specify the basic conversation flow in the Amazon Lex V2 console. Amazon Lex V2 manages the dialog and dynamically adjusts the responses in the conversation. Using the console, you can build, test, and publish your text or voice chatbot. You can then add the conversational interfaces to chatbots on mobile devices, web applications, and chat platforms (for example, Facebook Messenger).

Amazon Lex V2 now includes advanced AI-powered features that make bot building even more powerful and accessible. **Assisted NLU** uses Large Language Models (LLMs) to improve intent classification and slot resolution while staying within your bot's configured intents and slots. This means better understanding of user requests with less training data required. Amazon Lex V2 also supports **custom vocabularies in 17 additional languages**, enabling global deployment with improved speech recognition accuracy across diverse markets.

With **Multi-Region Replication (MRR)**, you can now deploy your bots across multiple AWS regions for improved availability and disaster recovery, ensuring your conversational interfaces remain accessible to users worldwide.

Amazon Lex V2 provides integration with AWS Lambda, and you can integrate with many other services on the AWS platform, including Connect Customer, Amazon Comprehend, and Amazon Kendra. Integration with Lambda provides bots access to pre-built serverless enterprise connectors to link to data in SaaS applications such as Salesforce.

For bots created after August 17, 2022, you can use conditional branching to control the conversation flow with your bot. With conditional branching you can create complex conversations without needing to write Lambda code.

Amazon Lex V2 provides the following benefits:

- **Simplicity** – Amazon Lex V2 guides you through using the console to create your own bot in minutes. You supply a few example phrases, and Amazon Lex V2 builds a complete natural

language model through which the bot can interact using voice and text to ask questions, get answers, and complete sophisticated tasks.

- **Democratized deep learning technologies** – Amazon Lex V2 provides ASR and NLU technologies to create a Speech Language Understanding (SLU) system. Through SLU, Amazon Lex V2 takes natural language speech and text input, understands the intent behind the input, and fulfills the user intent by invoking the appropriate business function.

  Speech recognition and natural language understanding are some of the most challenging problems to solve in computer science, requiring sophisticated deep learning algorithms to be trained on massive amounts of data and infrastructure. Amazon Lex V2 puts deep learning technologies within reach of all developers. Amazon Lex V2 bots convert incoming speech to text and understand the user intent to generate an intelligent response so you can focus on building your bots with added value for your customers and define entirely new categories of products made possible through conversational interfaces.

- **Seamless deployment and scaling** – With Amazon Lex V2, you can build, test, and deploy your bots directly from the Amazon Lex V2 console. Amazon Lex V2 enables you to publish your voice or text bots for use on mobile devices, web apps, and chat services (for example, Facebook Messenger). Amazon Lex V2 scales automatically. You don't need to worry about provisioning hardware and managing infrastructure to power your bot experience.

- **Built-in integration with the AWS platform** – Amazon Lex V2 operates natively with other AWS services, such as AWS Lambda and Amazon CloudWatch. You can take advantage of the power of the AWS platform for security, monitoring, user authentication, business logic, storage, and mobile app development.

- **Cost-effectiveness** – With Amazon Lex V2, there are no upfront costs or minimum fees. You are charged only for the text or speech requests that are made. The pay-as-you-go pricing and the low cost per request make the service a cost-effective way to build conversational interfaces. With the Amazon Lex V2 free tier, you can easily try Amazon Lex V2 without any initial investment.

Common use cases for Amazon Lex V2

Amazon Lex V2 enables you to build sophisticated conversational interfaces for a wide variety of business scenarios. Here are some popular use cases:

- **Customer Support** – Create intelligent chatbots that can handle common customer inquiries, troubleshoot issues, and escalate complex problems to human agents. Integrate with your existing CRM and knowledge base systems.
- **E-commerce and Retail** – Build shopping assistant chatbots that help customers find products, check order status, process returns, and provide personalized recommendations based on purchase history.
- **Appointment Booking** – Develop scheduling chatbots for healthcare, professional services, or hospitality that can check availability, book appointments, send reminders, and handle cancellations.
- **IT Helpdesk** – Create internal support chatbots that can reset passwords, provide software installation guidance, track IT tickets, and connect employees with the right technical resources.
- **Financial Services** – Build banking chatbots that can check account balances, transfer funds, provide transaction history, and offer financial advice while maintaining strict security standards.
- **HR and Employee Services** – Develop HR assistant chatbots that can answer policy questions, help with benefits enrollment, process time-off requests, and provide onboarding support for new employees.

Amazon Lex V2 integrates seamlessly with popular communication platforms including Slack, Microsoft Teams, WhatsApp, Facebook Messenger, and custom web applications, making it easy to deploy your chatbots where your users already communicate.

Paying for Amazon Lex V2

Amazon Lex V2 charges you only for the text or speech requests that you make. This model gives you a variable-cost service that can grow with your business while giving you the cost advantages of the AWS infrastructure. For more information, see Amazon Lex V2 Pricing.

When you sign up for AWS, your AWS account is automatically signed up for all services in AWS, including Amazon Lex V2. However, you are charged only for the services that you use. If you are a new Amazon Lex V2 customer, you can get started with Amazon Lex V2 for free. For more information, see AWS free tier.

To see your bill, go to the Billing and Cost Management Dashboard in the AWS Billing and Cost Management console. To learn more about AWS account billing, see the *AWS Billing User Guide*. If you have questions concerning AWS billing and AWS accounts, contact AWS Support.

Are You a First-time User of Amazon Lex V2?

If you are a first-time user of Amazon Lex V2, we recommend that you read the following sections in order:

1. **Amazon Lex V2 core concepts** – This section introduces Amazon Lex V2 and the features that you use to create a chatbot.
2. **Getting started with Amazon Lex V2** – In this section, you set up your account and test Amazon Lex V2.
3. **API Reference** – This section contains details about API operations.

Using Amazon Lex V2 with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples

SDK documentation	Code examples
AWS Tools for PowerShell	AWS Tools for PowerShell code examples
AWS SDK for Python v4	AWS SDK for Python v4 code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Using Amazon Lex V2 with an AWS SDK

Beyond the console interface, Amazon Lex V2 provides comprehensive programmatic access through AWS SDKs. This enables you to integrate conversational AI capabilities directly into your applications, automate bot management tasks, and build scalable solutions.

When you use an AWS SDK with Amazon Lex V2, you can:

- **Automate bot creation and management** – Programmatically create, update, and deploy bots without manual console interaction
- **Integrate with existing applications** – Add conversational interfaces to web applications, mobile apps, and enterprise systems
- **Scale bot operations** – Manage multiple bots, intents, and slot types efficiently through code
- **Implement custom workflows** – Build sophisticated conversation flows that integrate with your business logic

The table below shows the AWS SDKs that support Amazon Lex V2 operations. Choose the SDK that matches your development environment and follow the provided links to get started with installation and implementation.

Programming Language	AWS SDK	Getting Started Resources
Java	AWS SDK for Java 2.x	Developer Guide API Reference
Python	AWS SDK for Python (Boto3)	Getting Started API Reference
JavaScript/ Node.js	AWS SDK for JavaScript v3	Developer Guide API Reference
.NET/C#	AWS SDK for .NET	Developer Guide API Reference
Go	AWS SDK for Go v2	Developer Guide API Reference
Ruby	AWS SDK for Ruby	Developer Guide API Reference
PHP	AWS SDK for PHP	Developer Guide API Reference

To get started with any SDK:

1. Install the SDK for your preferred programming language using the installation instructions in the Developer Guide
2. Configure your AWS credentials and region settings

3. Set up the necessary IAM permissions for Amazon Lex V2 operations
4. Review the API Reference documentation for the specific operations you need
5. Start with basic operations like creating a bot or listing existing resources

For detailed examples and step-by-step guidance on creating bots programmatically, see the SDK documentation links provided in the table above.

Latest features

This topic provides information about the latest features that Amazon Lex V2 offers:

AMAZON.BedrockAgentIntent for connecting with Bedrock Agents and Knowledge Bases

Amazon Lex V2 now provides the AMAZON.BedrockAgentIntent built-in intent to seamlessly connect your bot with Amazon Bedrock Agents and Knowledge Bases for enhanced conversational AI capabilities.

- Documentation

Assisted NLU for improved intent classification and slot resolution

Amazon Lex V2 now offers Assisted NLU to improve intent classification and slot resolution using Large Language Models (LLMs) while staying within your bot's configured intents and slots.

- Documentation

Custom vocabulary support for 17 additional languages

Amazon Lex V2 now supports custom vocabularies in 17 additional languages to improve speech recognition accuracy.

- Documentation

QinConnect built-in intent for Amazon Connect integration

Amazon Lex V2 now provides the QinConnect built-in intent to seamlessly connect your bot with Amazon Connect for enhanced contact center experiences.

- Documentation

AMAZON.Currency and AMAZON.Confirmation built-in slots now support all locales

Amazon Lex V2 now supports AMAZON.Currency and AMAZON.Confirmation built-in slot types in all locales, expanding beyond the previously supported English and Spanish locales.

- Documentation

Enhanced QnA intent with Bedrock Knowledge Base and Guardrails

The QnA built-in intent for generative AI now supports Bedrock Knowledge Base and Guardrails for more secure and accurate responses.

- Documentation

Support for Anthropic Claude 3 models

The QnA built-in intent for generative AI now supports Anthropic Claude 3 Haiku and Anthropic Claude 3 Sonnet models for enhanced conversational capabilities.

- Documentation

Global resiliency for bot replication across regions

Amazon Lex V2 now offers Global resiliency to replicate your bot in a second AWS region for improved availability and disaster recovery.

- Documentation

Regional support for AWS GovCloud (US-West)

Amazon Lex V2 is now available in AWS GovCloud (US-West).

- Amazon Lex endpoints and quotas

Generative AI features for Amazon Lex V2

Amazon Lex V2 now allows you to take advantage of Amazon Bedrock's generative AI capabilities for your bot.

- Descriptive bot builder
  - What's new post
  - Documentation
- Assisted slot resolution
  - What's new post
  - Documentation
- Utterance generation
  - What's new post
  - Documentation
- `AMAZON.QnAIntent` (Conversational FAQ)
  - What's new post
  - Documentation
- AWS Machine Learning Blog post

AMAZON.Confirmation built-in slot for Yes/No/Maybe/Don't know disambiguation.

Amazon Lex V2 now offers `AMAZON.Confimation` built-in slot to improve the accuracy of slot confirmation and Yes/No/Maybe/Don't know responses.

- Documentation

Measuring business performance with Analytics

Amazon Lex V2 now offers users the ability to view the performance of intents and slots on the Analytics dashboard.

- What's new post
- Documentation

Evaluating bot performance with Test workbench

Amazon Lex V2 now offers users the ability to create and run test sets to measure bot performance and improve bot metrics.

- What's new post
- Documentation
- AWS Machine Learning Blog post

Vertical specific bot templates

Amazon Lex V2 now offers users pre-built bot templates with ready-to-use conversation flows along with both training data and dialog prompts, for both voice and chat modalities.

- What's new post
- Documentation

Network of bots

Amazon Lex V2 now offers users the ability to combine multiple bots into a single network and the ability to route requests to the appropriate bot based on user input.

- What's new post
- Documentation

Visual conversation builder

Amazon Lex V2 now offers a drag and drop conversation builder to easily design and visualize conversation paths by using intents within a rich visual environment.

- What's new post
- Documentation
- AWS Machine Learning Blog post

Composite slot type

Amazon Lex V2 now offers users the ability to combine multiple slots into a composite slot using logical expressions.

- What's new post
- Documentation

Conditional branching

Amazon Lex V2 now offers users the ability to write conditions to better control the path that customers take through a conversation with your bot.

- What's new post
- Documentation

Automated chatbot designer

Amazon Lex V2 now offers users the option of automatically designing a chatbot from conversation transcripts. Read the for usage examples.

- What's new post
- Documentation
- AWS Machine Learning Blog post
- Amazon Lex Automated Chatbot Designer page

Runtime hints

Amazon Lex V2 now offers users the option of configuring runtime hints to improve recognition of phrases to improve elicitation of slot values.

- What's new post
- Documentation

Custom vocabulary

Amazon Lex V2 now offers users the option of creating a custom vocabulary, a list of phrases that can include proper nouns or domain-specific words, for Amazon Lex V2 to recognize in the audio input.

- What's new post
- Documentation
- AWS Machine Learning Blog post

Grammar slot type

Amazon Lex V2 now offers users the ability to author grammars in XML format following the Speech Recognition Grammar Specification (SRGS) to collect information in a conversation.

- What's new post
- Documentation
- AWS Machine Learning Blog post

Amazon Lex V2 core concepts

Amazon Lex V2 enables you to build chat applications (bots) to elicit information from users to accomplish a task. For example, you can create a chatbot to provide customer support, answer frequently asked questions, or book appointments. Following are the typical steps for working with Amazon Lex V2:

Quick Start Learning Path

New to Amazon Lex V2? Follow this progressive learning path to get started quickly:

1. **Start with a Template** (5 minutes) – Choose from pre-built chatbot templates like Customer Support FAQ, Appointment Booking, or Order Status. Templates include pre-configured intents, slots, and sample utterances.
2. **Customize Your Chatbot** (15 minutes) – Modify the template to match your specific use case. Add your own intents, update sample utterances, and configure slot types for your domain.
3. **Test and Refine** (10 minutes) – Use the built-in test console to have conversations with your chatbot. Enable Assisted NLU for improved understanding with minimal training data.
4. **Deploy and Integrate** (20 minutes) – Publish your chatbot and integrate it with your preferred platform (Slack, web app, or mobile application).

Total time to working chatbot: ~50 minutes

For a more comprehensive understanding, continue with the detailed development process below.

Detailed Development Process

For more complex bots or when building from scratch, follow this comprehensive development process:

1. Create a bot and add one or more languages. Configure the bot so that it understands the user's goal, engages in conversation with the user to elicit information, and fulfills the user's intent.
2. Test the bot. You can use the test window client provided by the Amazon Lex V2 console.
3. Publish a version and create an alias.
4. Deploy the bot. You can deploy the bot on your own applications or messaging platforms such as Facebook Messenger or Slack

Core Concepts and Terminology

Before you get started, familiarize yourself with the following Amazon Lex V2 core concepts and terminology:

- **Bot** – A bot performs automated tasks such as ordering a pizza, booking a hotel, ordering flowers, and so on. An Amazon Lex V2 bot is powered by automatic speech recognition (ASR) and natural language understanding (NLU) capabilities.

  Amazon Lex V2 bots can understand user input provided with text or speech and converse natural language.

- **Language** – An Amazon Lex V2 bot can converse in one or more languages. Each language is independent of the others, you can configure Amazon Lex V2 to converse with a user using native words and phrases. For more information, see Languages and locales supported by Amazon Lex V2.

- **Intent** – An intent represents an action that the user wants to perform. You create a bot to support one or more related intents. For example, you might create an intent that orders pizzas and drinks. For each intent, you provide the following required information:

  - **Intent name** – A descriptive name for the intent. For example, **`OrderPizza`**.
  - **Sample utterances** – How a user might convey the intent. For example, a user might say "Can I order a pizza" or "I want to order a pizza."
  - **How to fulfill the intent** – How you want to fulfill the intent after the user provides the necessary information. We recommend that you create a Lambda function to fulfill the intent.

    You can optionally configure the intent so Amazon Lex V2 returns the information back to the client application for the necessary fulfillment.

  In addition to custom intents, Amazon Lex V2 provides built-in intents to quickly set up your bot. For more information, see Built-in intents.

  Amazon Lex always includes a fallback intent for each bot. The fallback intent is used whenever Amazon Lex can't deduce the user's intent. For more information, see AMAZON.FallbackIntent.

- **Slot** – An intent can require zero or more slots, or parameters. You add slots as part of the intent configuration. At runtime, Amazon Lex V2 prompts the user for specific slot values. The user must provide values for all required slots before Amazon Lex V2 can fulfill the intent.

For example the `OrderPizza` intent requires slots such as size, crust type, and number of pizzas. For each slot, you provide the slot type and one or more prompts that Amazon Lex V2 sends to the client to elicit values from the user. A user can reply with a slot value that contains additional words, such as "large pizza please" or "let's stick with small." Amazon Lex V2 still understands the slot value.

- **Slot type** – Each slot has a type. You can create your own slot type, or you can use built-in slot types. For example, you might create and use the following slot types for the `OrderPizza` intent:
  - Size – With enumeration values `Small`, `Medium`, and `Large`.
  - Crust – With enumeration values `Thick` and `Thin`.

  Amazon Lex V2 also provides built-in slot types. For example, `AMAZON.Number` is a built-in slot type that you can use for the number of pizzas ordered. For more information, see Built-in intents.
- **Version** – A version is a numbered snapshot of your work that you can publish for use in different parts of your workflow, such as development, beta deployment, and production. Once you create a version, you can use a bot as it existed when the version was made. After you create a version, it stays the same while you continue to work on your application.
- **Alias** – An alias is a pointer to a specific version of a bot. With an alias, you can update the version the your client applications are using. For example, you can point an alias to version 1 of your bot. When you are ready to update the bot, you publish version 2 and change the alias to point to the new version. Because your applications use the alias instead of a specific version, all of your clients get the new functionality without needing to be updated.

Advanced Amazon Lex V2 Features

In addition to the core concepts above, Amazon Lex V2 includes advanced features that enhance bot capabilities:

- **Assisted NLU** – Uses Large Language Models (LLMs) to improve intent classification and slot resolution. This feature helps your bot understand user requests more accurately, even when they use different phrasing than your training examples. Assisted NLU works within your configured intents and slots, providing better understanding without requiring extensive training data.

- **Multi-turn Conversations** – Amazon Lex V2 can maintain context across multiple conversation turns, allowing for natural back-and-forth interactions. Users can provide information gradually, change their mind, or ask clarifying questions without losing the conversation context.
- **Context Switching** – Advanced bots can handle topic changes within a conversation. For example, a user might start asking about account information, then switch to placing an order, and return to the original topic. Amazon Lex V2 can manage these context switches gracefully.
- **Fallback Strategies** – When Amazon Lex V2 doesn't understand a user's request, you can configure sophisticated fallback behaviors including clarifying questions, suggestion prompts, or escalation to human agents. This ensures users always have a path forward in the conversation.
- **Conversation Flow Management** – Use conditional branching and conversation flow controls to create complex dialog patterns without writing code. You can route conversations based on user responses, slot values, or external data.

Regional Availability

For a list of the AWS Regions where Amazon Lex V2 is available, see Amazon Lex V2 endpoints and quotas in the *Amazon Web Services General Reference*.

Languages and locales supported by Amazon Lex V2

Amazon Lex V2 supports a variety of languages and locales. This topic lists the languages that are supported and the features that support these languages.

Supported languages and locales

Amazon Lex V2 supports the following languages and locales.

Code	Language and locale
af_ZA*	Afrikaans (South Africa)
ar_AE	Gulf Arabic (United Arab Emirates)
ar_SA*	Arabic (Saudi Arabia)
bg_BG*	Bulgarian (Bulgaria)

Code	Language and locale
ca_ES	Catalan (Spain)
cs_CZ*	Czech (Czech Republic)
cy_GB*	Welsh (United Kingdom)
da_DK*	Danish (Denmark)
de_AT	German (Austria)
de_CH*	German (Switzerland)
de_DE	German (Germany)
en_AB*	English (Scotland)
en_AU	English (Australia)
en_GB	English (UK)
en_IE*	English (Ireland)
en_IN	English (India)
en_NZ*	English (New Zealand)
en_US	English (US)
en_WL*	English (Wales)
en_ZA	English (South Africa)
es_419	Spanish (Latin America)
es_ES	Spanish (Spain)
es_MX*	Spanish (Mexico)
es_US	Spanish (US)

Code	Language and locale
et_ET*	Estonian (Estonia)
fa_IR*	Farsi (Iran)
fi_FI	Finnish (Finland)
fr_BE*	French (Belgium)
fr_CA	French (Canada)
fr_FR	French (France)
he_IL*	Hebrew (Israel)
hi_IN	Hindi (India)
hr_HR*	Croatian (Croatia)
hu_HU*	Hungarian (Hungary)
id_ID*	Indonesian (Indonesia)
is_IS*	Icelandic (Iceland)
it_IT	Italian (Italy)
ja_JP	Japanese (Japan)
km_KH*	Khmer (Cambodia)
ko_KR	Korean (Korea)
lt_LT*	Lithuanian (Lithuania)
lv_LV*	Latvian (Latvia)
ms_MY*	Malay (Malaysia)
nl_BE*	Dutch (Belgium)

Code	Language and locale
nl_NL	Dutch (The Netherlands)
no_NO	Norwegian (Norway)
pl_PL	Polish (Poland)
pt_BR	Portuguese (Brazil)
pt_PT	Portuguese (Portugal)
ro_RO*	Romanian (Romania)
ru_RU*	Russian (Russia)
sk_SK*	Slovak (Slovakia)
sl_SI*	Slovenian (Slovenia)
so_SO*	Somali (Somalia)
sr_RS*	Serbian (Serbia)
su_ID*	Sundanese (Indonesia)
sv_SE	Swedish (Sweden)
th_TH*	Thai (Thailand)
tl_PH*	Tagalog/Filipino (Philippines)
tr_TR*	Turkish (Turkey)
uk_UA*	Ukrainian (Ukraine)
vi_VN*	Vietnamese (Vietnam)
zh_CN	Mandarin (PRC)
zh_HK	Cantonese (Hong Kong)

Code	Language and locale
zu_ZA*	Zulu (South Africa)

Locales with limited feature support

Locales marked with an asterisk (*) have limited feature support via generative AI and third-party Automatic Speech Recognition (ASR) or Text-To-Speech (TTS). See the following table for a complete list of supported features.

Generative AI feature support

Generative AI feature support includes Assisted NLU in Primary Mode, Assisted Slot Resolution, and Intent Disambiguation.

Languages and locales supported by Amazon Lex V2 features

The following table lists Amazon Lex V2 features that are limited to certain languages and locales. All other Amazon Lex V2 features are supported in all languages and locales.

Feature	Supported languages and locales
AMAZON.AlphaNumeric	All languages and locales except Korean (ko_KR) and limited feature support locales
AMAZON.KendraSearchIntent	English (US) (en_US)
Improving speech recognition with a custom vocabulary	English (UK) (en_GB) English (US) (en_US)
Automated Chatbot Designer	English (US) (en_US)
Region availability	The following languages and locales are not available in the Asia Pacific (Singapore) (ap-southeast-1) and Africa (Cape Town) (ap-south-1) Regions: • Gulf Arabic (United Arab Emirates) (ar_AE)

Feature	Supported languages and locales
	• Catalan (Spain) (ca_ES) • Finnish (Finland) (fi_FI) • Hindi (India) (hi_IN) • Dutch (The Netherlands) (nl_NL) • Norwegian (Norway) (no_NO) • Polish (pl_PL) • Portuguese (Brazil) (pt_BR) • Portuguese (Portugal) (pt_PT) • Swedish (sv_SE) • Mandarin (PRC) (zh_CN) • Cantonese (Hong Kong) (zh_HK)
Setting intent context for your Lex V2 bot	English (US) (en_US)
Grammar slot type	English (Australia) (en_AU) English (UK) (en_GB) English (US) (en_US)
Using multiple values in a slot	English (US) (en_US)
Improving recognition of slot values with runtime hints in the conversation	English (UK) (en_GB) English (US) (en_US)
Capturing slot values with spelling styles during the conversation	English (Australia) (en_AU) English (UK) (en_GB) English (US) (en_US)
Using confidence scores to improve conversation accuracy	English (UK) (en_GB) English (US) (en_US)

Feature	Supported languages and locales
Only with Third-party ASR (Deepgram)	Dutch (Belgium) (nl_BE) French (Belgium) (fr_BE)
Only with Third-party TTS (ElevenLabs)	Afrikaans (South Africa) (af_ZA) Bulgarian (Bulgaria) (bg_BG) Croatian (Croatia) (hr_HR) English (Scotland) (en_AB) Estonian (Estonia) (et_ET) Farsi (Iran) (fa_IR) Hebrew (Israel) (he_IL) Hungarian (Hungary) (hu_HU) Indonesian (Indonesia) (id_ID) Latvian (Latvia) (lv_LV) Lithuanian (Lithuania) (lt_LT) Malay (Malaysia) (ms_MY) Serbian (Serbia) (sr_RS) Slovak (Slovakia) (sk_SK) Slovenian (Slovenia) (sl_SI)

Regions

For a list of AWS Regions where Amazon Lex V2 is available, see AWS regions and endpoints in the AWS General Reference.

Getting started with Amazon Lex V2

Overview

If you are new to Amazon Lex V2, we recommend that you read Amazon Lex V2 core concepts first.

Prerequisites Checklist

Before you begin building your first Amazon Lex V2 bot, ensure you have the following:

- **AWS Account** – You'll need an active AWS account. If you don't have one, you can create it for free and take advantage of the AWS Free Tier, which includes 10,000 text requests and 5,000 speech requests per month for Amazon Lex V2. For detailed setup instructions, see Sign Up for AWS.
- **IAM User with Permissions** – Create an IAM user with appropriate Amazon Lex V2 permissions. We'll guide you through creating an administrator user for getting started. For comprehensive information about IAM permissions and policies, see Identity and access management for Amazon Lex V2.
- **Supported Browser** – Use a modern web browser (Chrome, Firefox, Safari, or Edge) to access the Amazon Lex V2 console.
- **Time Commitment** – Allow 30-60 minutes to complete the getting started exercises, depending on your experience level.

Cost Estimate: The exercises in this guide use AWS Free Tier resources. If you're within the free tier limits, there should be no charges. Otherwise, costs are typically under $1 for completing all exercises.

Introduction to Amazon Lex V2

Before you can use Amazon Lex V2, you will need to create an AWS account and obtain an AWS account ID. You create a user account and an IAM role which gives you the permissions needed to use Amazon Lex V2

After creating your accounts, you can start using Amazon Lex V2 with the AWS Console or with the API.

Amazon Lex V2 provides API operations that you can integrate with your existing applications. For a list of supported operations, see the API Reference. You can use any of the following options:

- AWS SDK — When using the SDKs your requests to Amazon Lex V2 are automatically signed and authenticated using the credentials that you provide. We recommend that you use an SDK to build your application.
- AWS CLI — You can use the AWS CLI to access any Amazon Lex V2 feature without having to write any code.

Step 1: Set up an AWS Account and create an administrator User

Before you use Amazon Lex V2 for the first time, complete the following tasks:

1. Sign Up for AWS
2. Create an IAM user

Sign Up for AWS

If you already have an AWS account, skip this task.

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all services in AWS, including Amazon Lex V2. You are charged only for the services that you use.

With Amazon Lex V2, you pay only for the resources that you use. If you are a new AWS customer, you can get started with Amazon Lex V2 for free. For more information, see AWS Free Usage Tier.

If you already have an AWS account, skip to the next task. If you don't have an AWS account, use the following procedure to create one.

To create an AWS account

- To create an AWS account, see How do I create and activate a new AWS account? in the AWS Knowledge Center.

Write down your AWS account ID because you'll need it for the next task.

Create an IAM user

Services in AWS, such as Amazon Lex V2, require that you provide credentials when you access them so that the service can determine whether you have permissions to access the resources owned by that service.

Create an IAM user account to access your account for Amazon Lex V2. For comprehensive guidance on IAM user creation, see Getting started with IAM in the *IAM User Guide*.

- Use AWS Identity and Access Management (IAM) to create an IAM user
- Add the user to an IAM group with administrative permissions
- Grant administrative permissions to the IAM user that you created.

You can then access AWS using a special URL and the IAM user's credentials.

The Getting Started exercises in this guide assume that you have a user (`adminuser`) with administrator privileges. Follow the procedure to create `adminuser` in your account.

To create an administrator user and sign in to the console

1. Create an administrator user called `adminuser` in your AWS account. For instructions, see Creating Your First IAM user and Administrators Group in the *IAM User Guide*.
2. As a user, you can sign in to the AWS Management Console using a special URL. For more information, How Users Sign In to Your Account in the *IAM User Guide*.

For more information about IAM, see the IAM User Guide.

Common Setup Issues

If you encounter issues during account setup, here are common solutions:

- **Permission Denied Errors** – Ensure your IAM user has the necessary Amazon Lex V2 permissions. For getting started, administrator permissions are recommended. For detailed information about Amazon Lex V2 permissions, see the Security section in this guide.
- **Console Access Issues** – Make sure you're signing in with the correct IAM user URL for your account, not the root AWS console. For more information about IAM best practices, see the Security section in this guide.

- **Region Availability** – Amazon Lex V2 is not available in all AWS regions. Check the Amazon Lex endpoints and quotas to confirm availability in your preferred region. For information about supported locales, see Languages and locales supported by Amazon Lex V2.
- **Free Tier Limits** – Monitor your usage to stay within free tier limits: 10,000 text requests and 5,000 speech requests per month. For detailed pricing information, see Amazon Lex Pricing.
- **Quota Limits** – If you encounter service limits, review the current quotas and request increases if needed. For more information, see Guidelines and best practices.

For additional troubleshooting help, see the Monitoring section in this guide for information about error logs and debugging.

Grant programmatic access

For information about how to get your access key ID and secret access key, see Understanding and getting your AWS credentials in the AWS General Reference.

Developer Environment Setup

For Developers: If you plan to use the AWS CLI or SDKs, you'll also want to configure your development environment. Install the AWS CLI and configure it with your access keys. For detailed instructions, see Installing the AWS CLI. For SDK examples, see Bot examples.

Next step

Step 2: Getting started (console)

Step 2: Getting started (console)

Console Overview

The easiest way to learn how to use Amazon Lex V2 is by using the console. Choose the learning path that best matches your role and experience level:

Choose Your Learning Path

Choose the learning path that best matches your role and experience level:

- **No-Code Path (Business Users)** – Perfect for business analysts, product managers, and non-technical users who want to build bots using only the console interface.
  - Start with: Quick Start: Create a chatbot in 5 minutes
  - Then try: Exercise 1: Create a chatbot from a template
  - Learn: Best Practices for Getting Started
- **Developer Path** – Ideal for software developers who want to integrate Amazon Lex V2 with applications, use APIs, and implement custom business logic.
  - Start with: Exercise 1: Create a chatbot from a template
  - Advanced: Exercise 3: Build an advanced customer service chatbot
  - Then explore: Integrating an AWS Lambda function into your Amazon Lex V2 bot
  - Review: Bot examples
  - Advanced: API integration and SDK usage
- **Enterprise Path** – Designed for architects and engineers planning large-scale deployments with security, compliance, and scalability requirements.
  - Start with: Exercise 1: Create a chatbot from a template
  - Enterprise example: Exercise 3: Build an advanced customer service chatbot
  - Review: Security and IAM best practices
  - Explore: Use Global Resiliency to deploy bots to other Regions
  - Plan: Monitoring, logging, and cost optimization

Foundational Exercises

Regardless of your chosen path, we recommend starting with these foundational exercises:

- Exercise 1 — Create an Amazon Lex V2 bot using a template, a predefined bot that provides all of the necessary bot configuration. You do only a minimum of work to test the end-to-end setup.
- Exercise 2 — Review the JSON structures sent between your client application and an Amazon Lex V2 bot.
- Exercise 3 — Build an advanced customer service chat bot that demonstrates enterprise-level capabilities including order management, intelligent up-selling, lead generation, and AI-powered features for personalized customer experiences.

Quick Start: Create a chatbot in 5 minutes

This quick start guide helps you create a working Amazon Lex V2 chatbot in just 5 minutes using pre-built templates. You'll have a functional chatbot that you can immediately test and customize for your needs.

Step 1: Choose a Template

Amazon Lex V2 provides several pre-built templates for common use cases:

- **Customer Support FAQ** – Handle common customer service inquiries
- **Appointment Booking** – Schedule and manage appointments
- **Order Status** – Check order information and delivery status
- **IT Helpdesk** – Provide technical support and troubleshooting

To select a template

1. Open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/.
2. Choose **Create bot**.
3. Select **Start with a template**.
4. Choose the **Customer Support FAQ** template for this quick start.
5. Enter a bot name, such as **`MyFirstChatbot`**.
6. Choose **Create**.

Step 2: Quick Customization (Optional)

The template comes pre-configured, but you can quickly customize it for your specific needs:

To customize your chatbot

1. In the bot overview, review the pre-configured intents.
2. Choose an intent to modify, such as **GetAccountInfo**.
3. Add your own sample utterances that match how your customers might phrase requests.
4. Update the response messages to match your brand voice.
5. Choose **Save intent**.

Step 3: Test Your Bot

Test your chatbot immediately using the built-in test console:

To test your chatbot

1. Choose **Build** to compile your bot.
2. Wait for the build to complete (usually 1-2 minutes).
3. In the test console on the right, type: **`I need help with my account`**
4. Press Enter and observe the bot's response.
5. Try other phrases to test the bot's understanding.

Step 4: Deploy Your Bot

Once you're satisfied with your chatbot's responses, deploy it for use:

To deploy your chatbot

1. Choose **Publish** from the bot actions menu.
2. Create a new version by choosing **Create version**.
3. Create an alias (such as "Production") that points to your version.
4. Choose your integration method.

Next Steps

Congratulations! You now have a working Amazon Lex V2 chatbot. Here's what you can do next:

- **Enable Assisted NLU** – Improve understanding with AI-powered natural language processing.
- **Add More Intents** – Expand your chatbot's capabilities with additional use cases.
- **Integrate with Lambda** – Add business logic and external system integration. See Integrating an AWS Lambda function into your Amazon Lex V2 bot.
- **Set Up Monitoring** – Track usage and performance. See Measuring operational metrics with Amazon CloudWatch.
- **Learn Advanced Features** – Explore conversation flow management, multi-turn dialogs, and context switching.

For a more detailed walkthrough, continue with Exercise 1: Create a chatbot from a template.

Exercise 1: Create a chatbot from a template

In this exercise, you create your first Amazon Lex V2 chatbot and test it in the Amazon Lex V2 console. For this exercise, you use the **OrderFlowers** template, which demonstrates a practical, real-world use case for e-commerce.

OrderFlowers Bot Example

You use the **OrderFlowers** template to create an Amazon Lex V2 chatbot that can handle flower ordering requests. This example demonstrates how businesses can automate order taking with intelligent chatbots. For more information about the structure of a bot, see Amazon Lex V2 core concepts.

- **Intents** – The bot includes one main intent:
  - `OrderFlowers` - Handles flower ordering requests by collecting the flower type, pickup date, and pickup time
- **Slot types** – The bot uses built-in slot types that automatically recognize and handle common data formats:
  - AMAZON.Date - Recognizes dates like "tomorrow", "next Friday", or "March 15th"
  - AMAZON.Time - Recognizes times like "2 PM", "noon", or "quarter past three"
  - `FlowerTypes` (custom) - Specific flower varieties like "roses", "tulips", "lilies"
- **Slots** – The OrderFlowers intent requires the following information before the bot can fulfill the flower order:
  - `FlowerType` (FlowerTypes custom type) - The type of flowers to order
  - `PickupDate` (AMAZON.Date type) - When to pick up the flowers
  - `PickupTime` (AMAZON.Time type) - What time to pick up the flowers
- **Sample Utterances** – The following sample utterances show natural ways users might request flower orders:
  - "I would like to pick up flowers"
  - "I want to order some flowers"
  - "Can I get flowers for pickup?"
  - "I need to buy flowers"
- **Prompts** – After the bot identifies the intent, it uses the following prompts to fill the slots:

- Prompt for the `FlowerType` slot – "What type of flowers would you like to order?"
- Prompt for the `PickupDate` slot – "What day do you want the {FlowerType} to be picked up?"
- Prompt for the `PickupTime` slot – "At what time do you want the {FlowerType} to be picked up?"
- Confirmation statement – "Okay, your {FlowerType} will be ready for pickup by {PickupTime} on {PickupDate}. Does this sound okay?"

Create Your Bot

To create an Amazon Lex V2 bot (Console)

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Create bot**.
3. For the **Creation method**, choose **Start with an example**.
4. In the **Example bots** section, choose **OrderFlowers** from the list.
5. In the **Bot configuration** section give the bot a name and an optional description. The name must be unique in your account.
6. In the **Permissions** section, choose **Create a new role with basic Amazon Lex permissions**. This will create an AWS Identity and Access Management (IAM) role with the permissions that Amazon Lex V2 needs to run your bot.
7. In the **Children's Online Privacy Protection Act (COPPA)** section, make the appropriate choice.
8. In the **Session timeout** and **Advanced settings** sections, leave the defaults.
9. Choose **Next**. Amazon Lex V2 creates your bot.

Add a Language to Your Bot

After you create your bot, you must add one or more languages that the bot supports. A language contains the intents, slot types, and slots that the bot uses to converse with users.

To add a language to a bot

1. In the **Language** section, choose a supported language, and add a description.

2. Leave the **Voice interaction** and **Intent classification confidence score threshold** fields with their defaults.
3. Choose **Done** to add the language to the bot.

Test Your Bot

After you choose **Done**, the console opens the intent editor. You can use the intent editor to examine the intents used by the bot. When you are done examining the bot, you can test it.

To test the OrderFlowers bot

1. Choose **Build** at the top of the page. Wait for the bot to build.
2. When the build is complete, choose **Test** to open the test window.
3. Test the bot. Start the conversation with one of the sample utterances, such as "I would like to pick up flowers."

Enable NLU to Improve Understanding

Now that you have a working chatbot, let's enhance it with Assisted NLU to improve intent recognition and slot resolution. Assisted NLU uses Large Language Models (LLMs) to better understand user requests, even when they use different phrasing than your training examples.

To enable Assisted NLU

1. In the Amazon Lex V2 console, navigate to your bot's settings.
2. In the left navigation pane, choose **Bot settings**.
3. Under **Assisted NLU**, choose **Enable**.
4. Choose **Save** to apply the changes.
5. Build your bot again to apply the Assisted NLU enhancement.

Test the Improvement: Try these variations in your test console to see how Assisted NLU handles different phrasings:

- "I want to buy some roses" (should trigger OrderFlowers intent and capture FlowerType)
- "Can I get flowers delivered tomorrow?" (should trigger OrderFlowers intent and capture PickupDate)

- "I need tulips for pickup at 3 PM" (should trigger OrderFlowers intent and capture FlowerType and PickupTime)

Notice how the chatbot can understand these natural variations without requiring you to add them as explicit sample utterances. This is powered by Assisted NLU, which uses AI to improve natural language understanding.

Next steps

Now that you've created you first bot using a template, you can use the console to create your own bot. For instruction on creating a custom bot, and for more information about creating bots, see Working with Amazon Lex V2 bots.

Exercise 2: Review the conversation flow

In this exercise you review the JSON structures that are sent between your client application and the Amazon Lex V2 bot that you created in Exercise 1: Create a chatbot from a template. The conversation uses the RecognizeText operation to generate the JSON structures. The RecognizeUtterance returns the same information as HTTP headers in the response.

The JSON structures are divided by each turn of the conversation. A *turn* is a request from the client application and a response from the bot.

Turn 1

During the first turn of the conversation, the client application initiates the conversation with your bot. Both the URI and the body of the request provide information about the request.

```
POST /bots/botId/botAliases/botAliasId/botLocales/localeId/sessions/sessionId/text
 HTTP/1.1
Content-type: application/json

{
    "text": "I would like to order flowers"
}
```

- The URI identifies the bot that the client application is communicating with. It also includes a session identifier generated by the client application that identifies a specific conversation between a user and the bot.

- The body of the request contains the text that the user typed to the client application. In this case, only the text is sent, however your application can send additional information, such as request attributes or session state. For more information, see the RecognizeText operation.

From `text`, Amazon Lex V2 detects the user's intent, to order flowers. Amazon Lex V2 chooses one of the intent's slots (`FlowerType`) and one of the prompts for the slot, and then sends the following response to the client application. The client displays the response to the user.


```
{
    "interpretations": [
        {
            "intent": {
                "confirmationState": "None",
                "name": "OrderFlowers",
                "slots": {
                    "FlowerType": null,
                    "PickupDate": null,
                    "PickupTime": null
                },
                "state": "InProgress"
            },
            "nluConfidence": {
                "score": 0.95
            }
        },
        {
            "intent": {
                "name": "FallbackIntent",
                "slots": {}
            }
        }
    ],
    "messages": [
        {
            "content": "What type of flowers would you like to order?",
            "contentType": "PlainText"
        }
    ],
    "sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",
    "sessionState": {
        "dialogAction": {
```

```
            "slotToElicit": "FlowerType",
            "type": "ElicitSlot"
        },
        "intent": {
            "confirmationState": "None",
            "name": "OrderFlowers",
            "slots": {
                "FlowerType": null,
                "PickupDate": null,
                "PickupTime": null
            },
            "state": "InProgress"
        },
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
    }
}
```

Turn 2

In turn 2, the user responds to the prompt from the Amazon Lex V2 bot in turn 1 with a value that fills the `FlowerType` slot.

```
{
    "text": "1 dozen roses"
}
```

The response for turn 2 shows the `FlowerType` slot filled and provides a prompt to elicit the next slot value.

```
{
    "interpretations": [
        {
            "intent": {
                "confirmationState": "None",
                "name": "OrderFlowers",
                "slots": {
                    "FlowerType": {
                        "value": {
                            "interpretedValue": "dozen roses",
                            "originalValue": "dozen roses",
```

```
                        "resolvedValues": []
                    }
                },
                "PickupDate": null,
                "PickupTime": null
            },
            "state": "InProgress"
        },
        "nluConfidence": {
            "score": 0.98
        }
    },
    {
        "intent": {
            "name": "FallbackIntent",
            "slots": {}
        }
    }
],
"messages": [
    {
        "content": "What day do you want the dozen roses to be picked up?",
        "contentType": "PlainText"
    }
],
"sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",
"sessionState": {
    "dialogAction": {
        "slotToElicit": "PickupDate",
        "type": "ElicitSlot"
    },
    "intent": {
        "confirmationState": "None",
        "name": "OrderFlowers",
        "slots": {
            "FlowerType": {
                "value": {
                    "interpretedValue": "dozen roses",
                    "originalValue": "dozen roses",
                    "resolvedValues": []
                }
            },
            "PickupDate": null,
            "PickupTime": null
```

```
            },
            "state": "InProgress"
        },
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
    }
}
```

Turn 3

In turn 3, the user responds to the prompt from the Amazon Lex V2 bot in turn 2 with a value that fills the `PickupDate` slot.

```
{
    "text": "next monday"
}
```

The response for turn 3 both the `FlowerType` and `PickupDate` slots filled and provides a prompt to elicit the last slot value.

```
{
    "interpretations": [
        {
            "intent": {
                "confirmationState": "None",
                "name": "OrderFlowers",
                "slots": {
                    "FlowerType": {
                        "value": {
                            "interpretedValue": "dozen roses",
                            "originalValue": "dozen roses",
                            "resolvedValues": []
                        }
                    },
                    "PickupDate": {
                        "value": {
                            "interpretedValue": "2022-12-28",
                            "originalValue": "next monday",
                            "resolvedValues": [
                                "2021-01-04"
                            ]
```

```
                    }
                },
                "PickupTime": null
            },
            "state": "InProgress"
        },
        "nluConfidence": {
            "score": 1.0
        }
    },
    {
        "intent": {
            "name": "FallbackIntent",
            "slots": {}
        }
    }
],
"messages": [
    {
        "content": "At what time do you want the 1 dozen roses to be picked up?",
        "contentType": "PlainText"
    }
],
"sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",
"sessionState": {
    "dialogAction": {
        "slotToElicit": "PickupTime",
        "type": "ElicitSlot"
    },
    "intent": {
        "confirmationState": "None",
        "name": "OrderFlowers",
        "slots": {
            "FlowerType": {
                "value": {
                    "interpretedValue": "dozen roses",
                    "originalValue": "dozen roses",
                    "resolvedValues": []
                }
            },
            "PickupDate": {
                "value": {
                    "interpretedValue": "2021-01-04",
                    "originalValue": "next monday",
```

```
                        "resolvedValues": [
                            "2021-01-04"
                        ]
                    }
                },
                "PickupTime": null
            },
            "state": "InProgress"
        },
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f",
        "sessionAttributes": {}
    }
}
```

Turn 4

In turn 4, the user provides the final slot value for the intent, the time that the flowers are picked up.

```
{
    "text": "5 in the evening"
}
```

In the response, Amazon Lex V2 sends a confirmation prompt to the user to confirm that the order is correct. The `dialogAction` is set to `ConfirmIntent` and the `confirmationState` is None.

```
{
    "interpretations": [
        {
            "intent": {
                "confirmationState": "None",
                "name": "OrderFlowers",
                "slots": {
                    "FlowerType": {
                        "value": {
                            "interpretedValue": "dozen roses",
                            "originalValue": "dozen roses",
                            "resolvedValues": []
                        }
                    },
```

```
                    "PickupDate": {
                        "value": {
                            "interpretedValue": "2021-01-04",
                            "originalValue": "next monday",
                            "resolvedValues": [
                                "2021-01-04"
                            ]
                        }
                    },
                    "PickupTime": {
                        "value": {
                            "interpretedValue": "17:00",
                            "originalValue": "5 evening",
                            "resolvedValues": [
                                "17:00"
                            ]
                        }
                    }
                },
                "state": "InProgress"
            },
            "nluConfidence": {
                "score": 1.0
            }
        },
        {
            "intent": {
                "name": "FallbackIntent",
                "slots": {}
            }
        }
    ],
    "messages": [
        {
            "content": "Okay, your dozen roses will be ready for pickup by 17:00 on
2021-01-04.  Does this sound okay?",
            "contentType": "PlainText"
        }
    ],
    "sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",
    "sessionState": {
        "dialogAction": {
            "type": "ConfirmIntent"
        },
```

```
        "intent": {
            "confirmationState": "None",
            "name": "OrderFlowers",
            "slots": {
                "FlowerType": {
                    "value": {
                        "interpretedValue": "dozen roses",
                        "originalValue": "dozen roses",
                        "resolvedValues": []
                    }
                },
                "PickupDate": {
                    "value": {
                        "interpretedValue": "2021-01-04",
                        "originalValue": "next monday",
                        "resolvedValues": [
                            "2021-01-04"
                        ]
                    }
                },
                "PickupTime": {
                    "value": {
                        "interpretedValue": "17:00",
                        "originalValue": "5 evening",
                        "resolvedValues": [
                            "17:00"
                        ]
                    }
                }
            },
            "state": "InProgress"
        },
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
    }
}
```

Turn 5

In the final turn, the user responds with to the confirmation prompt.

```
{
    "text": "yes"
```

```
}
```

In the response, Amazon Lex V2 sends indicates that the intent has been fulfilled by setting the `confirmationState` to `Confirmed` and the `dialogAction` to close. All of the slot values are available to the client application.

```
{
    "interpretations": [
        {
            "intent": {
                "confirmationState": "Confirmed",
                "name": "OrderFlowers",
                "slots": {
                    "FlowerType": {
                        "value": {
                            "interpretedValue": "dozen roses",
                            "originalValue": "dozen roses",
                            "resolvedValues": []
                        }
                    },
                    "PickupDate": {
                        "value": {
                            "interpretedValue": "2021-01-04",
                            "originalValue": "next monday",
                            "resolvedValues": [
                                "2021-01-04"
                            ]
                        }
                    },
                    "PickupTime": {
                        "value": {
                            "interpretedValue": "17:00",
                            "originalValue": "5 evening",
                            "resolvedValues": [
                                "17:00"
                            ]
                        }
                    }
                },
                "state": "Fulfilled"
            },
            "nluConfidence": {
```

```
                "score": 1.0
            }
        },
        {
            "intent": {
                "name": "FallbackIntent",
                "slots": {}
            }
        }
    ],
    "messages": [
        {
            "content": "Thanks. ",
            "contentType": "PlainText"
        }
    ],
    "sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",
    "sessionState": {
        "dialogAction": {
            "type": "Close"
        },
        "intent": {
            "confirmationState": "Confirmed",
            "name": "OrderFlowers",
            "slots": {
                "FlowerType": {
                    "value": {
                        "interpretedValue": "dozen roses",
                        "originalValue": "dozen roses",
                        "resolvedValues": []
                    }
                },
                "PickupDate": {
                    "value": {
                        "interpretedValue": "2021-01-04",
                        "originalValue": "next monday",
                        "resolvedValues": [
                            "2021-01-04"
                        ]
                    }
                },
                "PickupTime": {
                    "value": {
                        "interpretedValue": "17:00",
```

```
                        "originalValue": "5 evening",
                        "resolvedValues": [
                            "17:00"
                        ]
                    }
                }
            },
            "state": "Fulfilled"
        },
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
    }
}
```

Exercise 3: Build an advanced customer service chatbot

In this advanced exercise, you'll build a sophisticated customer service chatbot for an e-commerce company. This bot demonstrates enterprise-level capabilities including order management, intelligent upselling, lead generation, and revenue optimization. The chatbot uses AI-powered features to provide personalized customer experiences and drive business growth.

SmartCommerce Customer Service Bot Overview

The **SmartCommerce Customer Service Bot** is designed to handle complex customer interactions while maximizing revenue opportunities. This example demonstrates how businesses can automate customer service while driving sales growth through intelligent conversation management.

- **Custom Intents** – The bot includes multiple custom intents for comprehensive customer service:
  - `CheckOrderStatus` - Verifies and provides order status information
  - `ProcessReturn` - Handles return requests and exchanges
  - `UpsellProducts` - Recommends additional products and services
  - `CaptureLeadInfo` - Collects customer information for lead generation
  - `ScheduleCallback` - Books customer service callbacks
- **Built-in Intents** – Leverages Amazon Lex V2 built-in intents for common interactions:
  - AMAZON.HelpIntent - Provides help and guidance
  - AMAZON.CancelIntent - Handles cancellation requests
  - AMAZON.StopIntent - Ends conversations gracefully
- **Custom Slot Types** – Specialized slot types for business-specific data:

- `ProductCategories` - Electronics, Clothing, Home, Books, Sports
- `ReturnReasons` - Defective, Wrong Size, Changed Mind, Not as Described
- `ContactPreferences` - Email, SMS, Phone Call, In-App Notification
- `CustomerTiers` - Bronze, Silver, Gold, Platinum

- **Built-in Slot Types** – Uses Amazon Lex V2 built-in slots for common data formats:
  - AMAZON.Date - For delivery dates and callback scheduling
  - AMAZON.Time - For callback times and delivery windows
  - AMAZON.EmailAddress - For customer email collection
  - AMAZON.PhoneNumber - For customer phone numbers
  - AMAZON.Number - For order numbers and quantities

Detailed Intent Configuration

CheckOrderStatus Intent

This intent handles order verification and status inquiries, providing customers with real-time order information while identifying upselling opportunities.

- **Required Slots:**
  - `OrderNumber` (AMAZON.Number) - Customer's order number
  - `CustomerEmail` (AMAZON.EmailAddress) - Email for verification
- **Sample Utterances:**
  - "What's the status of my order {OrderNumber}"
  - "I need to check on order number {OrderNumber}"
  - "Where is my package"
  - "Track my order {OrderNumber}"
  - "Has my order shipped yet"
- **AI-Powered Features:**
  - Uses Assisted NLU to understand variations like "my package", "my stuff", "my delivery"
  - Automatically extracts order numbers from natural speech patterns

UpsellProducts Intent

This intent proactively identifies revenue opportunities by recommending complementary products and premium services based on customer history and current interaction context.

- **Required Slots:**
  - `ProductCategory` (ProductCategories) - Category of interest
  - `CustomerTier` (CustomerTiers) - Customer loyalty level
  - `Budget` (AMAZON.Number) - Customer's budget range
- **Sample Utterances:**
  - "Show me related products"
  - "What else would go with this"
  - "Do you have any deals"
  - "I'm interested in {ProductCategory} items"
  - "What's on sale today"
- **Revenue Optimization Features:**
  - Dynamic pricing based on customer tier
  - Personalized recommendations using purchase history
  - Limited-time offer creation to drive immediate purchases

CaptureLeadInfo Intent

This intent systematically collects customer information for lead generation, creating valuable data that can be sold to aggregators or used for targeted marketing campaigns.

- **Required Slots:**
  - `CustomerFirstName` (AMAZON.FirstName) - Customer's first name for personalization
  - `CustomerLastName` (AMAZON.LastName) - Customer's last name for sales history lookup
  - `CustomerEmail` (AMAZON.EmailAddress) - Email address
  - `CustomerPhone` (AMAZON.PhoneNumber) - Phone number
  - `InterestCategory` (ProductCategories) - Product interests
  - `ContactPreference` (ContactPreferences) - Preferred contact method
- **Sample Utterances:**

- "I want to hear about new products"
- "Sign me up for updates"
- "I'm interested in {InterestCategory}"
- "Keep me informed about sales"
- "I want to join your mailing list"
- "My name is {CustomerFirstName} {CustomerLastName}"
- "I'm {CustomerFirstName} {CustomerLastName} and I want personalized recommendations"

- **Lead Generation Features:**
  - Structured data collection for CRM integration
  - Lead scoring based on interaction quality
  - Automated lead qualification and routing
  - Data formatting for aggregator sales
- **Sales History Integration:**
  - Customer name lookup in S3-stored purchase history tables
  - Purchase pattern analysis for personalized recommendations
  - Cross-category purchase behavior identification
  - Seasonal buying trend analysis for targeted offers
  - Customer lifetime value calculation for tier-based pricing

Multi-Turn Conversation Flow

The SmartCommerce bot is designed to handle complex, multi-turn conversations that span multiple intents and maintain context throughout the interaction. This creates a natural, human-like experience while maximizing revenue opportunities.

Example Conversation Flow

Turn 1-3: Order Status Check

```
Customer: "Hi, I need to check on my order"
Bot: "I'd be happy to help you check your order status. Can you provide your order
 number?"
Customer: "It's order 12345"
```

```
Bot: "Thank you. To verify your identity, can you provide the email address associated
 with this order?"
Customer: "john.doe@email.com"
Bot: "Perfect! Order #12345 is currently being prepared for shipment. You should
 receive it by tomorrow.
     Since you're a Gold tier customer, would you like to hear about some exclusive
 deals on related items?"
```

Turn 4-6: Upselling Opportunity

```
Customer: "Sure, what do you have?"
Bot: "Based on your Electronics purchase, I can offer you a 20% discount on premium
 accessories.
     What's your budget range for additional items?"
Customer: "Maybe around $50"
Bot: "Excellent! I have a wireless charging pad normally $60, now $48 with your Gold
 discount.
     Plus, if you add it to your current order, shipping is free. Should I add this to
 your cart?"
```

Turn 7-11: Lead Generation with Sales History Lookup

```
Customer: "Not right now, but I'd like to hear about future deals"
Bot: "Absolutely! I can set you up to receive personalized offers. What's your first
 name?"
Customer: "John"
Bot: "Thanks John! And your last name so I can look up your purchase history for better
 recommendations?"
Customer: "Smith"
Bot: "Perfect! Let me check your purchase history... I see you've bought electronics
 and home items before.
     Based on your past purchases, I can offer you exclusive deals on smart home
 accessories.
     How would you prefer to be contacted?"
Customer: "Email is fine"
Bot: "Great! I'll make sure john.doe@email.com gets personalized deals based on your
 purchase history.
     You're all set for our premium recommendations program!"
```

Context Management Features

- **Session Persistence:** Maintains customer information across multiple intents
- **Intent Chaining:** Seamlessly transitions between order checking, upselling, and lead capture
- **Dynamic Personalization:** Adjusts offers based on customer tier and purchase history
- **Revenue Tracking:** Monitors upsell success rates and lead quality scores

AI-Powered Features

This chatbot leverages multiple AI capabilities to provide intelligent, personalized customer service while driving business growth.

Assisted NLU Implementation

Assisted NLU uses Large Language Models to understand customer intent even when they use non-standard phrasing or combine multiple requests in a single utterance.

- **Natural Language Understanding:**
  - "My stuff hasn't arrived yet" → CheckOrderStatus intent
  - "I want to return this junk" → ProcessReturn intent
  - "Show me what else you got" → UpsellProducts intent
- **Multi-Intent Recognition:**
  - "Check my order 12345 and sign me up for deals" → CheckOrderStatus + CaptureLeadInfo

Generative Slot Resolution

Uses AI to extract slot values from complex, natural language inputs without requiring exact matches to training data.

- **Smart Extraction:**
  - "I bought some electronics last week, order number was something like 12345" → OrderNumber: 12345, ProductCategory: Electronics
  - "Call me on my mobile at 555-123-4567 or email me at john@company.com" → Phone: 555-123-4567, Email: john@company.com

Sentiment Analysis Integration

Monitors customer sentiment throughout the conversation to adjust approach and escalate when necessary.

- **Sentiment-Based Routing:**
  - Positive sentiment → Aggressive upselling approach
  - Neutral sentiment → Standard service with gentle upselling
  - Negative sentiment → Focus on problem resolution, minimal upselling

Order Verification and Confirmation System

The bot includes a comprehensive confirmation system that verifies customer identity, confirms order details, and sends confirmations via multiple channels.

Multi-Factor Verification Process

- **Primary Verification:**
  - Order number validation
  - Email address confirmation
  - Last 4 digits of payment method (for sensitive operations)
- **Secondary Verification:**
  - Shipping address confirmation
  - Purchase date validation
  - Product details verification

Multi-Channel Confirmation System

The bot automatically sends confirmations through multiple channels based on customer preferences and action type.

- **Email Confirmations:**
  - Order status updates with tracking information
  - Return authorization with prepaid shipping labels
  - Upsell purchase confirmations with delivery details
  - Lead capture confirmation with welcome offers

- **SMS Confirmations:**
  - Immediate order status updates
  - Delivery notifications with time windows
  - Flash sale alerts for interested customers
- **In-App Notifications:**
  - Real-time order updates
  - Personalized product recommendations
  - Loyalty program updates

Point of Sale (POS) Integration

The chatbot integrates directly with POS systems to process transactions, apply discounts, and update inventory in real-time.

Real-Time Transaction Processing

- **Upsell Transaction Flow:**

```
1. Customer accepts upsell offer
2. Bot validates inventory availability
3. Bot applies customer-tier discount
4. Bot processes payment using stored payment method
5. Bot updates order with additional items
6. Bot sends confirmation via preferred channel
7. Bot updates customer profile with purchase data
```

- **Revenue Tracking:**
  - Upsell conversion rates by customer tier
  - Average order value increase per interaction
  - Revenue attribution to chatbot interactions
  - Customer lifetime value impact

Dynamic Pricing Engine

- **Tier-Based Pricing:**

- Bronze: Standard pricing
- Silver: 5% discount on upsells
- Gold: 10-20% discount on upsells
- Platinum: 25% discount + free shipping

- **Contextual Pricing:**
  - Problem resolution scenarios: Additional discount to maintain satisfaction
  - High-value customers: Exclusive pricing tiers
  - Inventory clearance: Aggressive discounting for slow-moving items

Advanced Lead Generation System

The chatbot includes a sophisticated lead generation system that captures, qualifies, and monetizes customer data through multiple revenue streams.

Multi-Touch Lead Capture Strategy

- **Opportunistic Capture:**
  - During order status checks: "Would you like updates on similar products?"
  - After problem resolution: "Can we notify you about product improvements?"
  - During upsell interactions: "Should we alert you to future deals in this category?"
- **Value-Added Capture:**
  - Exclusive member pricing access
  - Early access to new product launches
  - Personalized product recommendations
  - Birthday and anniversary special offers

Automated Lead Qualification

- **Scoring Criteria:**
  - Purchase history value: 0-25 points
  - Engagement level: 0-20 points
  - Contact information completeness: 0-15 points
  - Product category interest breadth: 0-15 points

- Response to upsell attempts: 0-25 points
- **Lead Categories:**
  - Hot Leads (80-100 points): Immediate sales team contact
  - Warm Leads (60-79 points): Automated nurture sequence
  - Cold Leads (40-59 points): Monthly newsletter
  - Prospects (0-39 points): Quarterly promotional emails

Lead Data Monetization for Aggregators

The system formats and packages lead data for sale to third-party aggregators, creating an additional revenue stream.

- **Data Packages:**
  - *Premium Package:* Full contact info, purchase history, preferences, engagement scores
  - *Standard Package:* Contact info, basic preferences, category interests
  - *Basic Package:* Email address, primary interest category
- **Compliance Features:**
  - GDPR compliance with explicit consent tracking
  - CCPA compliance with opt-out mechanisms
  - CAN-SPAM compliance for email marketing
  - Data retention and deletion policies
- **Revenue Optimization:**
  - Dynamic pricing based on lead quality scores
  - Exclusive data partnerships with premium aggregators
  - Real-time bidding for high-value leads

Revenue Optimization Strategies

Every interaction is designed to maximize revenue through intelligent upselling, cross-selling, and customer lifetime value optimization.

Intelligent Upselling Strategies

- **Contextual Upselling:**

- Order status check → "Your order is ready! Add expedited shipping for just $5?"
- Return request → "Instead of returning, would you like to exchange for our upgraded model?"
- Product inquiry → "This item pairs perfectly with [complementary product] - bundle and save 15%"

- **Urgency-Based Selling:**
  - "This offer expires in 24 hours"
  - "Only 3 left in stock at this price"
  - "Flash sale ends at midnight"

Cross-Selling Opportunities

- **Product Ecosystem Selling:**
  - Electronics → Accessories, warranties, installation services
  - Clothing → Matching items, care products, styling services
  - Home goods → Complementary items, maintenance products, design consultations
- **Service Upselling:**
  - Extended warranties and protection plans
  - Premium customer support tiers
  - Subscription services and auto-delivery
  - Professional installation and setup

Customer Lifetime Value Optimization

- **Loyalty Program Integration:**
  - Automatic tier upgrades based on chatbot interactions
  - Bonus points for engaging with upsell offers
  - Exclusive chatbot-only rewards and discounts
- **Retention Strategies:**
  - Proactive problem resolution to prevent churn
  - Personalized win-back offers for inactive customers
  - Anniversary and milestone celebration offers

Implementation Procedures

Follow these step-by-step procedures to build your advanced customer service chatbot with all the revenue optimization features.

Prerequisites and Setup Requirements

Before building the SmartCommerce bot, ensure you have the necessary AWS account setup, permissions, and understand the service considerations.

AWS Account and Access Requirements

- **AWS Account:** You need an active AWS account with billing enabled. If you don't have one, sign up at aws.amazon.com.
- **IAM Permissions:** Your AWS user or role must have the following permissions:
  - `lex:*` - Full Amazon Lex V2 permissions for bot creation and management
  - `iam:CreateRole` - To create service roles for the bot
  - `iam:AttachRolePolicy` - To attach policies to service roles
  - `lambda:CreateFunction` - For Lambda integration (optional)
  - `logs:CreateLogGroup` - For Amazon CloudWatch Logs logging
- **Region Selection:** Choose an AWS region that supports Amazon Lex V2 and your target audience. Recommended regions include:
  - US East (N. Virginia) - us-east-1
  - US West (Oregon) - us-west-2
  - Europe (Ireland) - eu-west-1
  - Asia Pacific (Sydney) - ap-southeast-2

Service Limits and Quotas

Amazon Lex V2 has several service limits that may affect your bot development and deployment:

- **Bot Limits:**
  - Maximum 100 bots per account per region
  - Maximum 100 intents per bot
  - Maximum 100 slot types per bot
  - Maximum 200 slots per intent

- **Runtime Limits:**
  - Maximum 15-minute session timeout
  - Maximum 1,000 requests per second (can be increased)
  - Maximum 1,500 characters per text input
- **Training Data Limits:**
  - Maximum 1,500 sample utterances per intent
  - Maximum 10,000 slot type values
  - Maximum 140 characters per sample utterance

> **Note**
>
> If you need higher limits for production use, you can request quota increases through the AWS Support Center.

Cost Considerations

Understanding Amazon Lex V2 pricing helps you plan and budget for your chatbot deployment:

- **Request-Based Pricing:**
  - Text requests: $0.00075 per request after the first 10,000 requests per month
  - Speech requests: $0.004 per request after the first 1,000 requests per month
- **Free Tier:**
  - 10,000 text requests per month for the first year
  - 1,000 speech requests per month for the first year
- **Additional Costs:**
  - Lambda functions (if used): $0.20 per 1M requests + compute time
  - Amazon CloudWatch Logs logs: $0.50 per GB ingested
  - Data transfer costs for external integrations
- **Cost Optimization Tips:**
  - Use session attributes to reduce redundant API calls
  - Implement efficient conversation flows to minimize turns
  - Monitor usage through CloudWatch Logs to identify optimization opportunities

> **Important**
>
> For current pricing information, visit the Amazon Lex Pricing page as rates may change.

Lambda Integration Overview

While this exercise focuses on the Lex bot configuration, the advanced features described (POS integration, lead generation, revenue optimization) would typically require Lambda functions for backend processing.

- **When Lambda is Needed:**
  - Order status verification against external databases
  - Real-time inventory checks for upselling
  - Customer data storage and retrieval
  - Sales history lookup in S3 tables using customer first and last name
  - Purchase pattern analysis for personalized recommendations
  - Payment processing integration
  - Email and SMS confirmation sending
- **S3 Sales History Integration:**
  - Query S3 tables using CustomerFirstName and CustomerLastName slots
  - Analyze purchase history to identify buying patterns and preferences
  - Generate personalized product recommendations based on past purchases
  - Calculate customer lifetime value for dynamic pricing strategies
  - Identify cross-sell opportunities from purchase correlation analysis
- **Basic Lambda Setup Requirements:**
  - Lambda execution role with appropriate permissions
  - VPC configuration if accessing private resources
  - Environment variables for configuration
  - Error handling and logging implementation
- **Integration Points:**
  - Intent fulfillment - Process completed intents
  - Slot validation - Validate user inputs in real-time

- Dialog management - Control conversation flow

> **Note**
>
> For this exercise, we'll focus on the Lex bot configuration. Lambda integration can be added later as your requirements evolve. The bot will work with static responses for testing and demonstration purposes.

Pre-Implementation Checklist

Before proceeding with the bot creation, verify you have:

- ✓ Active AWS account with billing enabled
- ✓ Appropriate IAM permissions for Lex and related services
- ✓ Selected target AWS region
- ✓ Reviewed service limits and quotas
- ✓ Understood cost implications
- ✓ Planned Lambda integration approach (if needed)
- ✓ Access to AWS Management AWS Management Console

Create the SmartCommerce Bot

To create the SmartCommerce customer service bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Create bot**.
3. For the **Creation method**, choose **Create a blank bot**.
4. In the **Bot configuration** section:
   - Bot name: **`SmartCommerceBot`**
   - Description: **`Advanced customer service chatbot with upselling, lead generation, and revenue optimization`**
5. In the **Permissions** section, choose **Create a new role with basic Amazon Lex permissions**.

6. In the **Children's Online Privacy Protection Act (COPPA)** section, choose **No**.
7. In the **Session timeout** section, set to **`15 minutes`** to allow for complex multi-turn conversations.
8. Choose **Next**.
9. Add English (US) as the language and choose **Done**.

Create Custom Slot Types

To create the custom slot types

1. In the left navigation pane, choose **Slot types**.
2. Choose **Add slot type** and create the following slot types:

   a. **ProductCategories:**

   - Electronics
   - Clothing
   - Home
   - Books
   - Sports
   - Beauty
   - Automotive

   b. **ReturnReasons:**

   - Defective
   - Wrong Size
   - Changed Mind
   - Not as Described
   - Damaged in Shipping
   - Better Price Found

   c. **ContactPreferences:**

   - Email
   - SMS

- Phone Call
- In-App Notification
- Mail

d. **CustomerTiers:**

- Bronze
- Silver
- Gold
- Platinum

Create Custom Intents

To create the CheckOrderStatus intent

1. In the left navigation pane, choose **Intents**.
2. Choose **Add intent** and name it **`CheckOrderStatus`**.
3. Add the following sample utterances:

   - "What's the status of my order {OrderNumber}"
   - "I need to check on order number {OrderNumber}"
   - "Where is my package"
   - "Track my order {OrderNumber}"
   - "Has my order shipped yet"
   - "When will my order arrive"
   - "I want to know about my delivery"
4. Add the following slots:

   - `OrderNumber` (AMAZON.Number) - Required

     Prompt: "Can you provide your order number?"
   - `CustomerEmail` (AMAZON.EmailAddress) - Required

     Prompt: "To verify your identity, please provide the email address associated with this order."
5. Set the confirmation prompt: "Let me check order #{OrderNumber} for {CustomerEmail}. Is this correct?"

6. Set the fulfillment message: "I found your order! Order #{OrderNumber} is {OrderStatus}. {UpsellMessage}"

To create the UpsellProducts intent

1. Create a new intent named **`UpsellProducts`**.
2. Add sample utterances:
   - "Show me related products"
   - "What else would go with this"
   - "Do you have any deals"
   - "I'm interested in {ProductCategory} items"
   - "What's on sale today"
   - "Show me more options"
3. Add slots:
   - `ProductCategory` (ProductCategories) - Required

     Prompt: "What type of products are you interested in?"
   - `Budget` (AMAZON.Number) - Optional

     Prompt: "What's your budget range for additional items?"

To create the CaptureLeadInfo intent

1. Create a new intent named **`CaptureLeadInfo`**.
2. Add sample utterances:
   - "I want to hear about new products"
   - "Sign me up for updates"
   - "I'm interested in {InterestCategory}"
   - "Keep me informed about sales"
   - "I want to join your mailing list"
   - "Send me deals and offers"
   - "My name is {CustomerFirstName} {CustomerLastName}"

- "I'm {CustomerFirstName} {CustomerLastName} and I want personalized recommendations"

3. Add slots:

   - `CustomerFirstName` (AMAZON.FirstName) - Required

     Prompt: "What's your first name for personalized service?"
   - `CustomerLastName` (AMAZON.LastName) - Required

     Prompt: "And your last name so I can look up your purchase history?"
   - `CustomerEmail` (AMAZON.EmailAddress) - Required
   - `CustomerPhone` (AMAZON.PhoneNumber) - Optional
   - `InterestCategory` (ProductCategories) - Required
   - `ContactPreference` (ContactPreferences) - Required

Enable AI-Powered Features

To enable Assisted NLU and other AI features

1. In the left navigation pane, choose **Bot settings**.
2. Under **Assisted NLU**, choose **Enable**.
3. Under **Generative AI**, enable:

   - Assisted slot resolution
   - Descriptive bot building
4. Under **Sentiment Analysis**, choose **Enable**.
5. Choose **Save**.

Test and Deploy the Bot

To test the SmartCommerce bot

1. Choose **Build** to compile your bot.
2. Once the build is complete, choose **Test**.
3. Test the following conversation flows:

   a. **Order Status Flow:**

- User: "Check my order 12345"
- Bot: Requests email verification
- User: Provides email
- Bot: Provides status and offers upsell

b. **Upselling Flow:**

- User: "Show me deals"
- Bot: Asks for product category
- User: "Electronics"
- Bot: Presents personalized offers

c. **Lead Generation Flow:**

- User: "Sign me up for updates"
- Bot: Collects contact information
- User: Provides details
- Bot: Confirms subscription and offers immediate discount

4. Verify that AI features are working:

- Test natural language variations like "my stuff hasn't arrived"
- Verify sentiment analysis adjusts bot responses
- Confirm slot resolution works with complex inputs

5. Once testing is complete, choose **Publish** to deploy your bot.

Performance Analytics and Optimization

Monitor and optimize your SmartCommerce bot's performance using comprehensive analytics and revenue tracking.

Key Performance Metrics

- **Revenue Metrics:**
  - Upsell conversion rate by customer tier
  - Average order value increase per interaction
  - Revenue per conversation

    - Customer lifetime value impact
- **Lead Generation Metrics:**
    - Lead capture rate by interaction type
    - Lead quality scores and conversion rates
    - Data monetization revenue from aggregator sales
    - Email subscription growth rate
- **Operational Metrics:**
    - Intent recognition accuracy
    - Slot filling success rate
    - Conversation completion rate
    - Customer satisfaction scores

Continuous Optimization Strategies

- **A/B Testing:**
    - Test different upsell messaging approaches
    - Compare aggressive vs. gentle lead capture techniques
    - Optimize discount percentages for maximum conversion
- **Machine Learning Optimization:**
    - Analyze successful conversation patterns
    - Identify optimal timing for upsell offers
    - Refine customer tier classification algorithms

Exercise Conclusion

Congratulations! You've successfully built an advanced customer service chatbot that demonstrates enterprise-level capabilities for revenue optimization and customer engagement. This SmartCommerce bot showcases how businesses can leverage Amazon Lex V2's AI-powered features to:

- **Maximize Revenue:** Through intelligent upselling, cross-selling, and dynamic pricing strategies
- **Generate Leads:** By systematically capturing and qualifying customer information for multiple revenue streams

- **Enhance Customer Experience:** Using AI to understand natural language and provide personalized interactions
- **Optimize Operations:** Through automated customer service that scales efficiently while maintaining quality

The techniques demonstrated in this exercise can be adapted and expanded for various industries and use cases, providing a foundation for building sophisticated conversational AI solutions that drive business growth.

Next Steps

To further enhance your SmartCommerce bot, consider implementing:

- **Advanced Integrations:**
  - CRM system integration for complete customer profiles
  - Inventory management system connections for real-time availability
  - Payment processing integration for seamless transactions
- **Multi-Channel Deployment:**
  - Website chat widget integration
  - Social media platform connections
  - Voice interface implementation
- **Advanced Analytics:**
  - Custom dashboard development for business metrics
  - Predictive analytics for customer behavior
  - ROI tracking and attribution modeling

Best Practices for Getting Started

Conversation Design Principles

Following these conversation design principles from the start will help you build more effective, maintainable, and user-friendly Amazon Lex V2 chatbots that provide natural interactions.

Core Design Principles

- **Start with User Goals** - Design your bot around what users want to accomplish, not what your system can do. Focus on the user's journey and desired outcomes.
- **Use Natural Language** - Write prompts and responses conversationally. Avoid technical jargon and speak as a helpful human would.
- **Provide Clear Options** - When users get stuck, offer specific examples of what they can say rather than generic help text.
- **Keep It Simple** - Start with basic functionality and gradually add complexity. Users should be able to complete common tasks quickly.
- **Handle Errors Gracefully** - When the bot doesn't understand, provide helpful guidance rather than just saying "I don't understand."
- **Confirm Important Actions** - Always confirm before taking actions that can't be easily undone, like placing orders or deleting information.
- **Provide Escape Routes** - Always give users a way to start over, get help, or connect with a human when needed.

Conversation Flow Best Practices

- **Set Clear Expectations** - Let users know what the bot can and cannot do early in the conversation.
- **Use Progressive Disclosure** - Ask for information one piece at a time rather than overwhelming users with multiple questions.
- **Provide Context** - Remind users what information you've already collected and what you still need.
- **Make Corrections Easy** - Allow users to correct information without starting over completely.

Real-World Use Cases and Examples

These practical examples show how to apply conversation design principles to common scenarios that new Amazon Lex V2 users encounter.

Use Case 1: Appointment Booking

Scenario: A medical office wants to automate appointment scheduling.

Challenge: Users need to provide multiple pieces of information (service type, date, time, contact info) and may want to change details.

Solution Approach:

- **Start Broad:** "What type of appointment would you like to schedule?" (dental, eye exam, consultation)
- **Narrow Down:** "When would you prefer your dental appointment?" (Accept flexible input like "next week" or "Friday afternoon")
- **Confirm and Offer Changes:** "I have you scheduled for a dental cleaning on Friday, March 15th at 2 PM. Does this work for you?"
- **Handle Changes:** If user says "Can we do 3 PM instead?", update the time without restarting the entire process.

Key Techniques:

- Use AMAZON.Date and AMAZON.Time for flexible date/time input
- Create custom slot types for appointment types
- Use confirmation prompts before finalizing bookings

Use Case 2: Order Status Inquiry

Scenario: An e-commerce company wants customers to check order status without calling support.

Challenge: Users might not have their order number handy, or might ask in different ways.

Solution Approach:

- **Multiple Entry Points:** Accept "Where's my order?", "Track my package", or "Order status"
- **Flexible Identification:** "I can help you track your order. Do you have your order number, or would you prefer to use your email address?"
- **Helpful Guidance:** "Your order number is usually in your confirmation email and starts with 'ORD-'"
- **Clear Results:** "Your order #ORD-12345 shipped yesterday and will arrive tomorrow by 8 PM. Would you like tracking details?"

Key Techniques:

- Use built-in slot types like AMAZON.AlphaNumeric for order numbers
- Provide multiple ways to identify orders (email, phone, order number)
- Give clear, actionable information in responses

Use Case 3: FAQ and Support

Scenario: A software company wants to handle common support questions automatically.

Challenge: Users ask the same questions in many different ways, and some issues require human help.

Solution Approach:

- **Broad Intent Recognition:** Train intents to recognize variations like "I can't log in", "Login problems", "Password not working"
- **Guided Troubleshooting:** "Let's try a few quick steps. First, have you tried resetting your password?"
- **Escalation Path:** "If these steps don't work, I can connect you with our support team. Would you like me to do that?"
- **Follow-up:** "Did this solve your problem? You can always say 'help' if you need assistance with something else."

Key Techniques:

- Use many sample utterances to capture different ways users express problems
- Provide step-by-step guidance rather than long explanations
- Always offer a path to human support when automated help isn't sufficient

Handling Complex Scenarios

Even simple bots encounter complex situations. Here's how to handle them gracefully:

- **When Users Change Their Mind:**
  - User: "Actually, I want roses instead of tulips"
  - Bot: "No problem! I've updated your order to roses. Your roses will be ready for pickup on Friday at 2 PM. Is that correct?"

- **When Information is Unclear:**
  - User: "I need flowers for next week"
  - Bot: "I'd be happy to help! Which day next week works best for you? You can say something like 'Monday' or 'next Tuesday'."
- **When the Bot Doesn't Understand:**
  - User: "I want the purple ones with the long stems"
  - Bot: "I want to make sure I get this right. Are you looking for purple flowers? We have purple roses, purple tulips, and purple irises available."
- **When Users Need to Start Over:**
  - User: "Cancel this, I want to start over"
  - Bot: "Of course! I've cleared your current order. What type of flowers would you like to order today?"

Bot examples

This section provides examples of Amazon Lex bots that you can use as a starting point for your own applications.

Topics

- Call center agent assistant

Call center agent assistant

In this tutorial, you use Amazon Lex V2 with Amazon Kendra to build an agent assist bot that assists customer support agents and publish it as a web application. Amazon Kendra is an enterprise search service that uses machine learning to search through documents to find answers. For more information about Amazon Kendra, see the *Amazon Kendra Developer Guide*.

Amazon Lex V2 bots are widely used in call centers as the first point of contact for customers. A bot is often capable of resolving customer questions. When a bot can't answer a question, it transfers the conversation to a customer support employee.

In this tutorial, we create an Amazon Lex V2 bot that agents use to answer customer queries in real time. By reading the answers that the bot provides, the agent is spared from looking up answers manually.

The bot and web application that you create in this tutorial helps agents respond to customers efficiently and accurately by quickly providing the right resources. The following diagram shows how the web application works.

As the diagram shows, the Amazon Kendra index of documents is stored in an Amazon Simple Storage Service (Amazon S3) bucket. If you don't already have an S3 bucket, you can set one up

when you create the Amazon Kendra index. In addition to Amazon S3, you will use Amazon Cognito for this tutorial. Amazon Cognito manages permissions for deploying the bot as a web application.

In this tutorial, you create an Amazon Kendra index that provides answers to customer questions, create the bot and add intents that allow it to suggest answers based on the conversation with the customer, set up Amazon Cognito to manage access permissions, and deploy the bot as a web application.

Estimated time: 75 minutes

Estimated cost: $2.50 per hour for an Amazon Kendra index and $0.75 for 1000 Amazon Lex V2 requests. Your Amazon Kendra index continues to run after you are finished with this exercise. Be sure to delete it to avoid unnecessary costs.

Note: Make sure that you choose the same AWS Region for all the services used in this tutorial.

Topics

Step 1: Create an Amazon Kendra Index

Begin by creating an Amazon Kendra index of documents that answer customer questions. An index provides a search API for client queries. You create the index from source documents. Amazon Kendra returns answers it finds in indexed documents to the bot, which displays them to the agent.

The quality and accuracy of the responses suggested by Amazon Kendra depend on the documents that you index. Documents should include files that are frequently accessed by the agent and must be stored in an S3 bucket. You can index unstructured and semi-structured data in .html, Microsoft Office (.doc, .ppt), PDF, and text formats.

To create an Amazon Kendra index, see Getting started with an S3 bucket (console) in the *Amazon Kendra Developer Guide*.

To add questions and answers (FAQs) that help answer customer queries, see Adding questions and answers in the *Amazon Kendra Developer Guide*. For this tutorial, use the ML_FAQ.csv file on GitHub.

Next step

Step 2: Create an Amazon Lex V2 Bot

Step 2: Create an Amazon Lex V2 Bot

Amazon Lex V2 provides an interface between the call center agent and the Amazon Kendra index. It keeps track of the conversation between the agent and the customer and calls the `AMAZON.KendraSearchIntent` intent based on the questions the customer asks. An *intent* is an action that the user wants to perform.

Amazon Kendra searches the indexed documents and returns an answer to Amazon Lex V2 that it displays in the bot. This answer is visible only to the agent.

To create an agent assistant bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. In the navigation pane, choose **Bots**.
3. Choose **Create**.
4. Choose **Custom bot** and configure the bot.
    a. **Bot name** – Enter a name that indicates the bot's purpose, such as **`AgentAssistBot`**.
    b. **Output voice** – Choose **None**.
    c. **Session timeout** – Enter **5**.
    d. **COPPA** – Choose **No**.
5. Choose **Create**. After creating the bot, Amazon Lex V2 displays the bot editor tab.

Next step

Step 3: Add a Custom and Built-in Intent

Step 3: Add a Custom and Built-in Intent

An *intent* represents an action that the call center agent wants the bot to perform. In this case, the agent wants the bot to suggest responses and helpful resources based on the agent's conversation with the customer.

Amazon Lex V2 has two types of intents: custom intents and built-in intents. `AMAZON.KendraSearchIntent` is a built-in intent. The bot uses the `AMAZON.KendraSearchIntent` intent to query the index and display the responses suggested by Amazon Kendra.

The bot in this example doesn't need a custom intent. However, to build the bot, you must create at least one custom intent with at least one sample utterance. This intent is required only to build your agent assistant bot. It doesn't perform any other function. The utterance for the intent must not answer any of the questions that the customer might ask. This ensures that the `AMAZON.KendraSearchIntent` is called to answer customer queries. For more information, see AMAZON.KendraSearchIntent.

To create the required custom intent

1. On the **Getting started with your bot** page, choose **Create intent**.
2. For **Add intent**, choose **Create intent**.
3. In the **Create intent** dialog box, enter a descriptive name for the intent, such as **`RequiredIntent`**.
4. For **Sample utterances**, enter a descriptive utterance, such as **`Required utterance`**.
5. Choose **Save intent**.

To add the `AMAZON.KendraSearchIntent` intent and response message

1. In the navigation pane, choose the plus sign (+) next to **Intents**.
2. Choose **Search existing intents**.
3. In the **Search intents** box, enter **`AMAZON.KendraSearchIntent`**, then choose it from the list.
4. Give the intent a descriptive name, such as **`AgentAssistSearchIntent`**, then choose **Add**.
5. In the intent editor, choose **Amazon Kendra query** to open the query options.
6. Choose the index that you want the intent to search,
7. In the **Response** section, add the following three messages to a message group.

```
I found an answer for the customer query: ((x-amz-lex:kendra-search-response-
question_answer-question-1)) and the answer is ((x-amz-lex:kendra-search-response-
question_answer-answer-1)).
I found an excerpt from a helpful document: ((x-amz-lex:kendra-search-response-
document-1)).
I think this answer will help the customer: ((x-amz-lex:kendra-search-response-
answer-1)).
```

8. Choose **Save intent**.
9. Choose **Build** to build the bot.

Next step

Step 4: Set up Amazon Cognito

Step 4: Set up Amazon Cognito

To manage permissions and users for the web application, you need to set up Amazon Cognito. Amazon Cognito ensures that the web application is secure and has access control. Amazon Cognito uses identity pools to provide AWS credentials that grant your users access to other AWS services. For this tutorial, it provides access to Amazon Lex V2.

When creating an identity pool, Amazon Cognito provides you with AWS Identity and Access Management (IAM) roles for authenticated and unauthenticated users. You modify the IAM roles by adding policies that grant access to Amazon Lex V2.

To set up Amazon Cognito

1. Sign into the AWS Management Console and open the Amazon Cognito console at https://console.aws.amazon.com/cognito/.
2. Choose **Manage Identity Pools**.
3. Choose **Create new identity pool**.
4. Configure the identity pool.
    a. **Identity pool name** – Enter a name that indicates the pool's purpose, such as **`BotPool`**.
    b. In the **Unauthenticated identities** section, choose **Enable access to unauthenticated identities**.

5. Choose **Create Pool**.
6. On the **Identify the IAM roles to use with your new identity pool** page, choose **View Details**.
7. Record the IAM role names. You will modify them later.
8. Choose **Allow**.
9. On the **Getting Started with Amazon Cognito** page, for **Platform**, choose **JavaScript**.
10. In the **Get AWS Credentials** section, find and record the **Identity pool ID**.
11. To allow access to Amazon Lex V2, modify the authenticated and unauthenticated IAM roles.

    a. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
    b. In the navigation pane, under **Access Management**, choose **Roles**.
    c. In the search box, enter the name of the authenticated IAM role and choose the checkbox next to it.

        i. Choose **Attach policies**.
        ii. In the search box, enter **`AmazonLexRunBotsOnly`** and choose the checkbox next to it.
        iii. Choose **Attach policy**.

    d. Enter the name of the unauthenticated IAM role in the search box and choose the checkbox next to it.

        i. Choose **Attach policies**.
        ii. In the search box, enter **`AmazonLexRunBotsOnly`** and choose the checkbox next to it.
        iii. Choose **Attach policy**.

Next step

Step 5: Deploy Your Bot as a Web Application

Step 5: Deploy Your Bot as a Web Application

To deploy your bot as a web application

1. Download the repository at https://github.com/awsdocs/amazon-lex-developer-guide/blob/master/example_apps/agent_assistance_bot/ to your computer.

2. Navigate to the downloaded repository and open the index.html file in an editor.
3. Make the following changes.

    a. In the `AWS.config.credentials` section, enter your Region name and your identity pool ID.

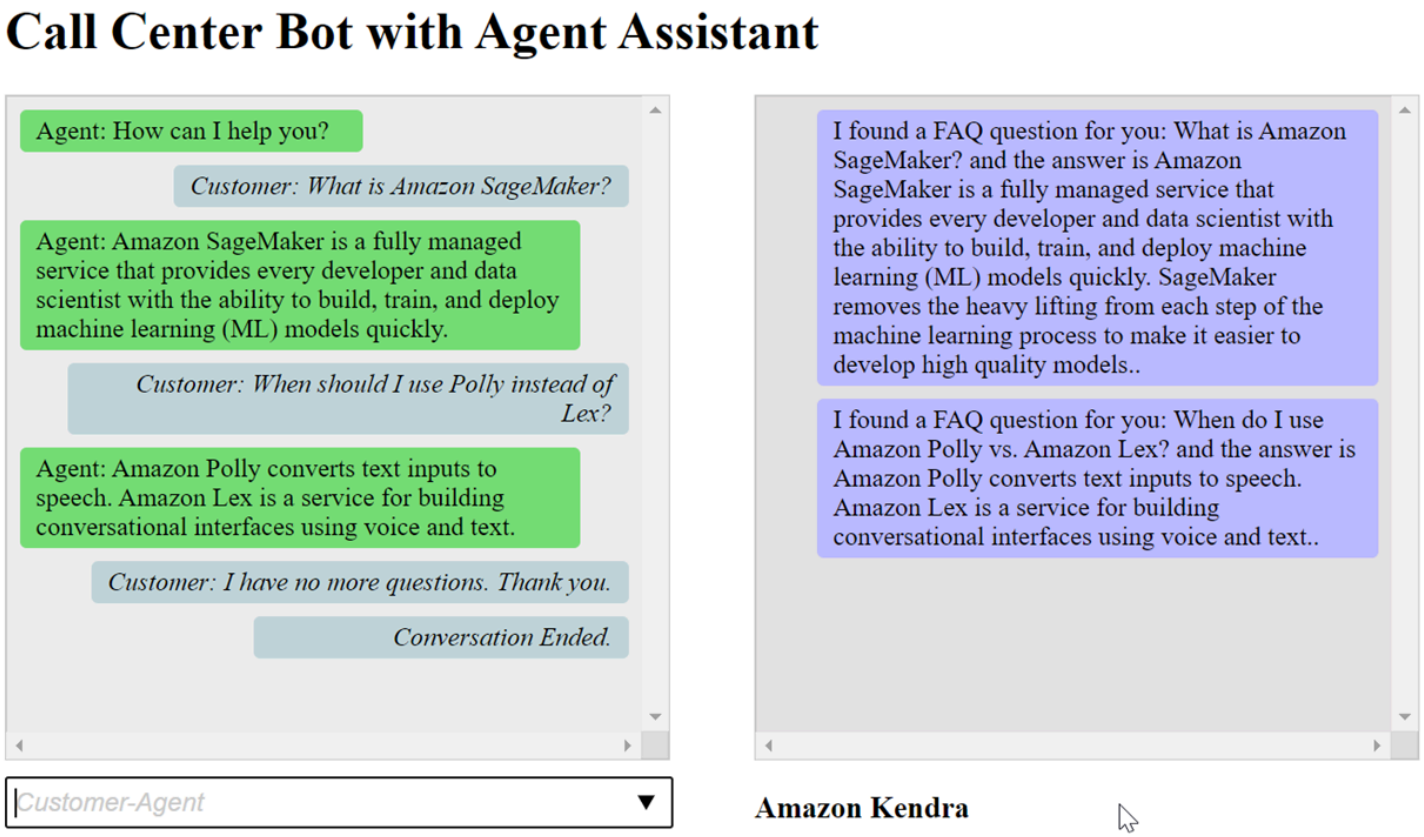
    b. In the `Amazon Lex V2 runtime parameters` section, enter the bot name.

    c. Save the file.

Step 6: Use the Bot

For demo purposes, you provide input to the bot as the customer and as the agent. To differentiate between the two, questions asked by the customer begin with “Customer:” and answers provided by the agent begin with “Agent:”. You can choose from a menu of suggested inputs.

Run your web application by opening `index.html` to engage in a conversation with your bot like this:.



The `pushChat()` function in the index.html file is explained below.

```
            var endConversationStatement = "Customer: I have no more questions. Thank
you."
            // If the agent has to send a message, start the message with 'Agent'
            var inputText = document.getElementById('input');
            if (inputText && inputText.value && inputText.value.trim().length > 0 &&
inputText.value[0]=='Agent') {
                showMessage(inputText.value, 'agentRequest','conversation');
                inputText.value = "";
            }
            // If the customer has to send a message, start the message with 'Customer'
            if(inputText && inputText.value && inputText.value.trim().length > 0 &&
inputText.value[0]=='Customer') {
                // disable input to show we're sending it
                var input = inputText.value.trim();
                inputText.value = '...';
                inputText.locked = true;
                customerInput = input.substring(2);

                // Send it to the Lex runtime
                var params = {
                    botAlias: '$LATEST',
                    botName: 'KendraTestBot',
                    inputText: customerInput,
                    userId: lexUserId,
                    sessionAttributes: sessionAttributes
                };

                showMessage(input, 'customerRequest', 'conversation');
                if(input== endConversationStatement){
                    showMessage('Conversation
Ended.','conversationEndRequest','conversation');
                }
                lexruntime.postText(params, function(err, data) {
                    if (err) {
                        console.log(err, err.stack);
                        showMessage('Error:  ' + err.message + ' (see console for
details)', 'lexError', 'conversation1')
                    }

                    if (data &&input!=endConversationStatement) {
                        // capture the sessionAttributes for the next cycle
                        sessionAttributes = data.sessionAttributes;
```

```
                    showMessage(data, 'lexResponse', 'conversation1');
                }
                // re-enable input
                inputText.value = '';
                inputText.locked = false;
            });
        }
        // we always cancel form submission
        return false;
```

When you provide input as a customer, the Amazon Lex V2 runtime API sends it to Amazon Lex V2.

The `showMessage(daText, senderRequest, displayWindow)` fuction displays the conversation between the agent and the customer in the chat window. Responses suggested by Amazon Kendra are shown in an adjacent window. The conversation ends when customer says **“I have no more questions. Thank you.”**

Note: Please delete your Amazon Kendra index when not in use.

Working with Amazon Lex V2 bots

You create an Amazon Lex V2 bot to interact with your users to elicit information to accomplish a task. For example, you can create a bot that gathers the information needed to order a bouquet of flowers or to book a hotel room.

To build a bot, you need the following information:

1. The language that the bot uses to interact with the customer. You can choose one or more languages, each language contains independent intents, slots, and slot types.
2. The intents, or goals, that the bot helps the user fulfill. A bot can contain one or more intents, such as ordering flowers, or booking a hotel and rental car. You need to decide which statements, or utterances, that the user makes to initiate the intent.
3. The information, or slots, that you need to gather from the user to fulfill an intent. For example, you might need to get the type of flowers from the user or the start date of a hotel reservation. You need to define one or more prompts that Amazon Lex V2 uses to elicit the slot value from the user.
4. The type of the slots that you need from the user. You may need to create a custom slot type, such as a list of flowers that a user can order, or you can use a built-in slot type, such as using the `AMAZON.Date` slot type for the start date of a reservation.
5. The user interaction flow within and between intents. You can configure the conversation flow to define the interaction between the user and the bot once the intent is invoked. You can create a Lambda function to validate and fulfill the intent.

Topics

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Changes to conversation flows in Amazon Lex V2

On August 17, 2022 Amazon Lex V2 released a change to the way that conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation.

Before the change, Amazon Lex V2 managed the conversation by eliciting slots based on their priorities in intent. You could modify this behavior dynamically and change the conversation path based on user inputs by using DialogAction in Lambda function. This could be done by keeping track of the conversation's current state and programmatically deciding what to do next based on the session state.

With this change, you can create conversational paths and conditional branches using the Amazon Lex V2 console or APIs without using a Lambda function. Amazon Lex V2 tracks the state of the conversation and controls what to do next based on conditions defined when the bot is created. This enables you to easily create complex conversations while designing your bot.

These changes give you complete control over the conversation with your customer. However, you are not required to define a path. If you do not specify a conversation path, Amazon Lex V2 creates a default path based on the priority of slots in your intent. You can continue to use Lambda functions to define conversation paths dynamically. In such a scenario, the conversation resumes based on the session state configured in the Lambda function.

This update provides the following:

- A new console experience for creating bots with complex conversation flows.
- Updates to the existing APIs for creating bots to support the new conversation flows.
- An initial response to send a message on intent invocation.
- New responses for slot elicitation, Lambda invocation as dialog code hook and confirmation.

- Ability to specify next steps at each turn of the conversation.
- Evaluation of conditions to design multiple conversation paths.
- Setting of slot values and session attributes at any point during the conversation.

Note the following for older bots:

- Bots created before August 17, 2022 continue to use the old mechanism to manage conversation flows. Bots created after that date use the new way of conversation flow management.
- New bots created via imports after August 17, 2022 use the new conversation flow management. Imports on existing bots continues to use the old way of conversation management.
- To enable the new conversation flow management for a bot created before August 17, 2022, export the bot, and then import the bot using a new bot name. The newly created bot from the import uses the new conversation flow management.

Note the following for new bots created after August 17, 2022:

- Amazon Lex V2 follows the defined conversation flow exactly as designed to deliver the desired experience. You should configure all flow branches in order to avoid default conversation paths during runtime.
- Conversation steps following a code hook should be fully configured, because incomplete steps can lead to bot failure. We recommend that you validate bots created before August 17, 2022, because for these bots, there is no automatic validation of conversation steps following a code hook.

Different ways to create a bot with Amazon Lex V2

You can create a bot with Amazon Lex V2 in the following ways:

1. Use the Amazon Lex V2 console to create a bot using a website interface. For more information, see Creating a bot using the Amazon Lex V2 console.
2. Use the Descriptive Bot Builder to create a bot using Amazon Bedrock's generative AI capabilities. For more information, see Use a description to build a bot in Lex V2 with the descriptive bot builder.
3. Use bot templates to create a preconfigured bot that matches common business use-cases. For more information, see Creating Amazon Lex V2 bots using templates.

4. Use an AWS SDK to create a bot using API operations.
5. Use the Automated Chatbot designer to create a bot using existing chat transcripts between agents and customers. For more information, see Creating Amazon Lex V2 bots using the Automated Chatbot Designer.
6. Import an existing bot definition. For more information, see Importing bots in Lex V2.
7. Use CloudFormation to create a bot. For more information, see Creating Amazon Lex V2 resources with AWS CloudFormation.

Topics

Creating a bot using the Amazon Lex V2 console

You create an Amazon Lex V2 bot to interact with your users to elicit information to accomplish a task. For example, you can create a bot that gathers the information needed to order flowers or to book a hotel room. To create a bot using the AWS Console, start by defining the name, description, and some basic information.

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Create bot**.
3. In the **Creation method** section, choose **Traditional** and then select **Create a blank bot**.
4. In the **Bot configuration** section, give the bot a name and an optional description.
5. In the **IAM permissions** section, choose an AWS Identity and Access Management (IAM) role that provides Amazon Lex V2 permission to access other AWS services, such as Amazon CloudWatch. You can have Amazon Lex V2 create the role, or you can choose an existing role with CloudWatch permissions.
6. In the **Children's Online Privacy Protection Act (COPPA)** section, choose the appropriate response.
7. In the **Idle session timeout** section, choose the duration that Amazon Lex V2 keeps a session with a user open. Amazon Lex V2 maintains session variables for the duration of the session so that your bot can resume a conversation with the same variables.

8. In the **Advanced settings** (optional) section, add tags that help identify the bot, control access and monitor resources.
9. Choose **Next** to create the bot and move to adding a language.

Creating Amazon Lex V2 bots using templates

Amazon Lex V2 offers pre-built solutions to create experiences at scale and drive digital engagement. The pre-built bot templates automates and standardizes client experiences. The bot templates provide ready-to-use conversation flows along with both training data and dialog prompts, for both voice and chat modalities. You can expedite the delivery of bot solutions while optimizing resources, so that you can focus on customer relationships.

You can create pre-built bots based on your business use case. You can use the CloudFormation console to select the pre-built options for the related services, such as Amazon S3, Connect Customer and DynamoDB.

Currently, Amazon Lex V2 supports the following business verticals:

- Financial services
- Retail orders
- Auto insurance
- Telecommunications
- Airline services
- Additional templates will be available in the future.

You can build a bot with the business solution template provided, and customize it for your business requirements.

> **Note**
>
> The templates create resources outside of Amazon Lex V2 through CloudFormation stacks. The stack may need to be modified in other consoles such as Lambda and DynamoDB.

Prerequisites to build and deploy the bot template:

- An AWS account
- Access to the following AWS services:
  - Amazon Lex V2 to create bots
  - Lambda for the business login functions
  - DynamoDB to create the tables
  - IAM access to create policies and roles
  - AWS CloudFormation to run the stack
- IAM access and secret key credentials
- Connect Customer instance (optional)

> **Note**
>
> The use of different AWS services incurs respective usage costs for each service.

To build a bot from Amazon Lex V2 templates:

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Select **Bot templates** from the left navigation pane.
3. Select which business vertical you want to use for your bot template. NOTE: There are 5 bot templates currently available. More to come soon.
4. Select **Create** for the template you want to use. A tab opens in CloudFormation where you can edit the parameters for the CloudFormation stack. All the options are already completed for the template you have chosen. You can also learn more about how the bot template works by selecting **Learn more**.
5. In the CloudFormation console, CloudFormation creates a default configuration for each of the values for the template you have chosen. You can also select your own stack name, CloudFormation parameters, Amazon DynamoDB table, and (optional) Connect Customer parameters.
6. At the bottom of the window, select **Create stack**.
7. CloudFormation processes the request in the background for several minutes to configure your new bot. NOTE: The process automatically creates resources for a DynamoDB table, an Connect Customer contact flow, and an Connect Customer instance. You can track the progress

in the CloudFormation console, and then navigate back to the Amazon Lex V2 console once the CloudFormation stack creation is completed.

8. If successfully built, a message appears and you can select **Go to bots list** to go to the **Bots** page, where you find your new bot that is ready for your testing and use.

Configuring your bot template

Lambda functions – The bot template automatically creates the needed Lambda functions for your deployment. If multiple bots are part of the template solution, then multiple Lambda functions are listed in the CloudFormation parameters. If you have existing Lambda functions to deploy with your bot, you can enter the name of your custom Lambda function.

Amazon DynamoDB – The bot template automatically creates the DynamoDB table needed to load your sample policy data. You can also enter the name of your custom DynamoDB table. Your custom DynamoDB table should be formatted in the same way as the default table created by the bot template deployment.

Connect Customer – You can configure your Connect Customer instance to work with your new bot template by entering the ConnectInstanceARN and a unique ContactFlowName. With the use of Connect Customer, you can test your bot using an IVR system from end-to-end.

Troubleshooting your bot template

- Check that you have the proper permissions to create the template that you are choosing. Users need CloudFormation:CreateStack permission along with permissions for the AWS resources that are listed within the template. A list of resources that need user permissions are at the bottom of the **Create template** page.
- If your bot template fails to be created, the red banner within the Amazon Lex V2 console provides a link to the CloudFormation stack that is responsible for creating the template. Within the CloudFormation console, you can view the events tab to see the specific error that caused the template to fail. Once you have reviewed the CloudFormation error, see Troubleshooting CloudFormation for more information.
- Bot templates work with the sample data only. You must populate the DynamoDB table with your data to make the templates work with your custom data.

Creating Amazon Lex V2 bots using the Automated Chatbot Designer

The Automated Chatbot Designer helps you design bots from existing conversation transcripts. It analyzes the transcripts and suggests an initial design with intents and slot types. You can iterate on the bot design, add prompts, build, test, and deploy the bot.

After you create a new bot or add a language to your bot using the Amazon Lex V2 console or API, you can upload transcripts of conversations between two parties. The automated chatbot designer analyzes the transcripts and determines the intents and slot types for the bot. It also labels the conversations that influenced the creation of a particular intent or slot type for your review.

You use the Amazon Lex V2 console or the API to analyze conversation transcripts and suggest intents and slot types for a bot.

> **Note**
>
> You can only use transcripts in the English (US) language.

You can review the suggested intents and slot types after the chatbot designer finishes the analysis. After you've added a suggested intent or slot type, you can modify it or delete it from the bot design using the console or the API.

The automated chatbot designer supports conversation transcript files using the Contact Lens for Connect Customer schema. If you are using a different contact center application, you must transform the conversation transcripts to the format used by the chatbot designer. For information, see Input transcript format.

To use the automated chatbot designer, you must allow the IAM role that is running the designer access. For the specific IAM policy, see Allow users to use the Automated Chatbot Designer. To enable Amazon Lex V2 to encrypt output data with an optional AWS KMS key, you need to update the key with the policy shown in Allow users to use a AWS KMS key to encrypt and decrypt files.

> **Note**
>
> If you use a KMS key, you must provide a **KMS key policy**, regardless of the IAM role used.

Topics

- Importing conversation transcripts
- Creating intents and slot types
- Input transcript format
- Output transcript format

Importing conversation transcripts

Importing conversation transcripts is a three-step process:

1. Prepare the transcripts for importing by converting them to the correct format. If you are using Contact Lens for Connect Customer the transcripts are already in the correct format.
2. Upload the transcripts to an Amazon S3 bucket. If you are using Contact Lens, your transcripts are already in an S3 bucket.
3. Analyze the transcripts using the Amazon Lex V2 console or API operations. The time that it takes to complete training depends on the volume of transcripts and the complexity of the conversation. Typically, 500 lines of transcripts are analyzed every minute.

Each of these steps is described in the following sections.

Importing transcripts from Contact Lens for Connect Customer

The Amazon Lex V2 automated chatbot designer is compatible with Contact Lens transcript files. To use Contact Lens transcript files, you must turn on Contact Lens and note the location of its output files.

To export transcripts from Contact Lens

1. Turn on Contact Lens in your Connect Customer instance. For instructions, see Enable Contact Lens for Connect Customer in the *Connect Customer administrator guide*.
2. Note the location of the S3 bucket that Connect Customer is using for your instance. To see the location, open the **Data storage** page in the Connect Customer console. For instructions, see Update instance settings in the *Connect Customer administrator guide*.

After you have turned on Contact Lens and noted the location of your transcript files, go to Analyze your transcripts using Amazon Lex V2 console for instructions to import and analyze your transcripts.

Prepare transcripts

Prepare your transcripts by creating transcript files.

- Create one transcript file per conversation listing the interaction between the parties. Each interaction in the conversation can span multiple lines. You can provide both redacted and non-redacted versions of the conversation.
- The file must be in the JSON format specified in Input transcript format.
- You must provide at least 1,000 conversational turns. To improve the discovery of your intents and slot types, you should provide around 10,000 or more conversational turns. The automated chatbot designer will only process the first 700,000 turns.
- There is no limit to the number of transcript files that you can upload, nor is there a file size restriction.

If you plan to filter the transcripts that you import by date, the files must be in the following directory structure:

```
<path or bucket root>
   --> yyyy
      --> mm
         --> dd
            --> transcript files
```

The transcript file must contain the date in the format "yyyy-mm-dd" somewhere in the file name.

To export transcripts from other contact center applications

1. Use your contact center application's tools to export conversations. The conversation must contain at least the information specified in Input transcript format.
2. Transform the transcripts produced by your contact center application to the format described in Input transcript format. You are responsible for performing the transformation.

We provide three scripts for preparing transcripts. They are:

- A script to combine Contact Lens transcripts with Amazon Lex V2 conversation logs. Contact Lens transcripts don't include parts of Connect Customer conversations that interact with Amazon Lex V2 bots. The script requires conversation logs to be turned on for Amazon Lex V2,

and appropriate permissions to query conversation log CloudWatch Logs and Contact Lens S3 buckets.

- A script to transform Amazon Transcribe call analytics to the Amazon Lex V2 input format.
- A script to transform Connect Customer chat transcripts to the Amazon Lex V2 input format.

You can download the scripts from this GitHub repository: https://github.com/aws-samples/amazon-lex-bot-recommendation-integration .

Upload your transcripts to an S3 bucket

If you are using Contact Lens, your transcript files are already contained in an S3 bucket. For the location and file names of your transcript files, see Example Contact Lens output files in the *Connect Customer administrator guide*.

If you are using another contact center application and you have not set up an S3 bucket for your transcript files, follow this procedure. Otherwise, if you have an existing S3 bucket, after logging in to the Amazon S3 console, follow this procedure starting with step 5.

To upload files to an S3 bucket

1. Sign in to the AWS Management Console and open the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. Choose **Create bucket**.
3. Give the bucket a name and choose a Region. The Region must be the same one that you use for Amazon Lex V2. Set the other options as required for your use case.
4. Choose **Create bucket**.
5. From the list of buckets, choose an existing bucket or the bucket that you just created
6. Choose **Upload**.
7. Add the transcript files that you want to upload.
8. Choose **Upload**.

Analyze your transcripts using Amazon Lex V2 console

You can only use automated bot design in an empty language. You can add a new language to an existing bot, or create a new bot.

To create a new language in a new bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Create bot**
3. Choose **Start with Automated Chatbot Designer**. Fill out the information to create your new bot.
4. Choose **Next**
5. In **Add language to bot** fill out the information for the language.
6. In the **Transcript file location on S3** section, choose the S3 bucket that contains your transcript files and the local path to the files if necessary.
7. You can optionally choose the following:
   - A AWS KMS key to encrypt the transcript data during processing. If you don't select a key, a service AWS KMS key is used.
   - To filter the transcripts to a specific date range. If you choose to filter the transcripts, they must be in the correct folder structure. For more information, see Prepare transcripts.
8. Choose **Done**.

Wait for Amazon Lex V2 to process the transcript. You see a completion message when the analysis is complete.

How to stop analyzing your transcript

In case you need to stop the analysis of the transcripts you have uploaded, you can stop a running `BotRecommendation` job, which has a `BotRecommendationStatus` status as processing. You can click on the **Stop processing** button present on the banner after submitting a job from the console or by using CLI SDK for the `StopBotRecommendation` API. For more information, see StopBotRecommendation

After calling the `StopBotRecommendation`, the internal `BotRecommendationStatus` is set to `Stopping` and you are not charged. To make sure the job has stopped, you can call the `DescribeBotRecommendation` API and verify that the `BotRecommendationStatus` is `Stopped`. This usually takes 3-4 minutes.

You are not charged for the processing after the `StopBotRecommendation` API is called.

Creating intents and slot types

After the chatbot designer creates intents and slot types, you select the intents and slot types to add to your bot. You can review the details of each intent and slot type to help you decide which recommendations are the most relevant to your use case.

You can click on a recommended intent's name to view the sample utterances and slots that the chatbot designer has suggested. If you select **Show associated transcripts**, you can also scroll through the conversations that you provided. These transcripts influence the chatbot designer's recommendation of this intent. If you click on a sample utterance, you can review the primary conversation and the relevant turn of dialog, which influenced that specific utterance.

You can click on a specific slot type's name to view the slot values that have been recommended. If you select **Show associated transcripts**, you can review the conversations that influenced this slot type, with the agent prompt that elicits for the slot type highlighted. If you click on a specific slot type value, you can review the primary conversation and the relevant turn of dialog that influenced this value.

To review and add intents and slot type

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of bots, choose the bot you want to work with.
3. Choose **View languages**.
4. From the list of languages, choose the language to work with.
5. In **Conversation structure**, choose **Review**.
6. In the list of intents and slot types, choose the ones to add to the bot. You can choose an intent or slot type to see details and the associated transcripts.

Intents are sorted by the confidence that Amazon Lex V2 has that the intent is associated with the processed transcripts.

Input transcript format

The following is the input file format for generating intents and slot types for your bot. The input file must contain these fields. Other fields are ignored.

The input format is compatible with the output format from Contact Lens for Connect Customer. If you are using Contact Lens, you don't need to modify your transcript files. For more information, see Example Contact Lens output files. If you are using another contact center application, you must transform your transcript file to this format.

```
{
    "Participants": [
        {
            "ParticipantId": "string",
            "ParticipantRole": "AGENT | CUSTOMER"
        }
    ],
    "Version": "1.1.0",
    "ContentMetadata": {
        "RedactionTypes": [
            "PII"
        ],
        "Output": "Raw | Redacted"
    },
    "CustomerMetadata": {
        "ContactId": "string"
    },
    "Transcript": [
        {
            "ParticipantId": "string",
            "Id": "string",
            "Content": "string"
        }
    ]
}
```

The following fields must be present in the input file:

- **Participants** Identifies the participants in the conversation and the role that they play.
- **Version** The version of the input file format. Always "1.1.0".
- **ContentMetadata** Indicates whether you removed sensitive information from the transcript. Set the `Output` field to "Raw" if the transcript contains sensitive information.
- **CustomerMetadata** A unique identifier for the conversation.
- **Transcript** The text of the conversation between parties in the conversation. Each turn of the conversation is identified with a unique identifier.

Output transcript format

The output transcript format is nearly the same as the input transcript format. However it also includes some customer metadata and a field listing segments that influenced the suggestion of intents and slot types. You can download the output transcript from the **Review** page in the console or using the Amazon Lex V2 API. For more information, see Input transcript format.

```
{
    "Participants": [
        {
            "ParticipantId": "string",
            "ParticipantRole": "AGENT | CUSTOMER"
        }
    ],
    "Version": "1.1.0",
    "ContentMetadata": {

        "RedactionTypes": [
            "PII"
        ],
        "Output": "Raw | Redacted"
    },
    "CustomerMetadata": {
        "ContactId": "string",
        "FileName": "string",
        "InputFormat": "Lex"
    },
    "InfluencingSegments": [
        {
            "Id": "string",
            "StartTurnIndex": number,
            "EndTurnIndex": number,
            "Intents": [
                {
                    "Id": "string",
                    "Name": "string",
                    "SampleUtteranceIndex": [
                        {
                            "Index": number,
                            "Content": "String"
                        }
                    ]
                }
```

```
            ],
            "SlotTypes": [
                {
                    "Id": "string",
                    "Name": "string",
                    "SlotValueIndex": [
                        {
                            "Index": number,
                            "Content": "String"
                        }
                    ]
                }
            ]
        }
    ],
    "Transcript": [
        {

            "ParticipantId": "string",
            "Id": "string",
            "Content": "string"
        }
    ]
}
```

- **CustomerMetadata** – There are two fields added to the `CustomerMetadata` field, the name of the input file that contains the conversation and the input format, which is always "Lex".
- **InfluencingSegments** – Identifies the segments of the conversation that influenced the suggestion of an intent or slot type. The ID of the intent or slot type identifies the specific one influenced by the conversation.

Adding a new language to an Amazon Lex V2 bot

You add one or more languages and locales to your bot to enable it to communicate with users in their languages. You define the intents, slots, and slot types separately for each language so that the utterances, prompts, and slot values are specific to the language.

Your bot must contain at least one language.

To add a language to your bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. In the **Bots** section, choose the bot you want to add a language to.
3. In the **Add languages** section, click **View languages**.
4. In the **All languages** section, click **Add language**.
5. In the **Add a new language** section, choose **Add a language from scratch**.
6. In the **Language details** section, choose the language that you want to add.
7. If your bot supports voice interaction, in the **Voice** section, choose the Amazon Polly voice that Amazon Lex V2 uses to communicate with the user. If your bot doesn't support voice, choose **None**.
8. In the **Classification confidence score threshold**section, set the value that Amazon Lex V2 uses to determine whether an intent is correct. You can adjust this value after testing your bot.
9. Choose **Add**.

Adding intents

Intents are the goals that your users want to accomplish, such as ordering flowers or booking a hotel. Your bot must have at least one intent.

By default, all bots contain a single built-in intent, the fallback intent. This intent is used when Amazon Lex V2 does not recognize any other intent. For example, if a user says "I want to order flowers" to a hotel booking intent, the fallback intent is triggered.

To add an intent

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of bots, choose the bot that you want to add the intent to, then from **Add languages** choose **View languages**.
3. Choose the language to add the intent to, then choose **Intents**.
4. Choose **Add intent**, give your intent a name, and then choose **Add**.
5. In the intent editor, add the details of your intent.

- **Conversation flow** – Use the conversation flow diagram to see how a dialog with your bot might look. You can choose different sections of the conversation to jump to that section of the intent editor.
- **Intent details** – Give the intent a name and description to help identify the purpose of the intent. You can also see the unique identifier that Amazon Lex V2 assigned to the intent.
- **Contexts** – Set the input and output contexts for the intent. A context is a state variable associated with an intent. An output context is set when an intent is fulfilled. An intent with an input context can only be recognized if the context is active. An intent with no input contexts can always be recognized.
- **Sample utterances** – You should provide 10 or more phrases that you expect your users to use to initiate an intent. Amazon Lex V2 generalizes from these phrases to recognize that the user wants to initiate the intent.
- **Initial response** – The initial message sent to the user after the intent is invoked. You can provide responses, initialize values, and define the next step that Amazon Lex V2 takes to respond to the user at the beginning of the intent.
- **Slots** – Define the slots, or parameters, required to fulfill the intent. Each slot has a type that defines the values that can be entered in the slot. You can choose from your custom slot types, or you can choose a built-in slot type.
- **Confirmation** – These prompts and responses are used to confirm or decline fulfillment of the intent. The confirmation prompt asks the user to review slot values. For example, "I've booked a hotel room for Friday. Is this correct?" The declination response is sent to the user when they decline the confirmation. You can provide responses, set values, and define the next step that Amazon Lex V2 takes corresponding to a confirmation or declination response from the user.
- **Fulfillment** – Response sent to the user during the course of fulfillment. You can set fulfillment progress updates at the start of fulfillment and periodically while the fulfillment is in progress. For example, "I'm changing your password, this may take a few minutes" and "I'm still working on your request." Fulfillment updates are used only for streaming conversations. You can also set a post-fulfillment success message, a failure message, and a timeout message. You can send post-fulfillment messages for both streaming and regular conversations. For example, if the fulfillment succeeds, you can send "I've changed your password." If the fulfillment doesn't succeed, you can send a response with more information, such as "I couldn't change your password, contact the help desk for assistance." If the fulfillment takes longer than the configured timeout period, you can send a message

informing the user, such as "Our servers are very busy right now. Try your request again later." You can provide responses, set values, and define the next step that Amazon Lex V2 takes to respond to the user.

- **Closing responses** – Response sent to the user after the intent is fulfilled and all other messages are played. For example, a thank you for booking a hotel room. Or it can prompt the user to start a different intent, such as, "Thank you for booking a room, would you like to book a rental car?" You can provide responses and configure follow-up next actions after fulfilling the intent and responding with the closing response.
- **Code hooks** – Indicate whether you are using an AWS Lambda function to initialize the intent and validate user input. You specify the Lambda function in the alias that you use to run the bot.

6. Choose **Save intent** to save the intent.

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Configuring prompts in a specific order

You can configure the bot to play messages in a predefined order by checking the box for **Play messages in order**. Otherwise, the bot plays the message and the variations in random order.

Ordered prompts allow the message and variations of a message group to play in order among retries. You can use alternate rephrasing of a message when an invalid response for the prompt is given by the user, or for intent confirmation. Up to two variations of the original message may be set in each slot. You can choose whether to play the messages in order or randomly.

Ordered prompt supports all four types of messages: text, custom payload response, SSML, and card group. Responses are ordered within the same message group. Different message groups are independent.

Topics

Sample utterances

You create sample utterances that are variations of phrases that you expect users to use to initiate an intent. For example, for a **`BookFlight`** intent, you might include utterances such as the following:

1. I want to book a flight
2. help me get a flight.
3. plane tickets, please!
4. flight from {*`DepartureCity`*} to {*`DestinationCity`*}

You should provide 10 or more sample utterances. Give samples that represent a wide range of sentence structures and words that users may utter. Consider incomplete sentences as well, such as in examples 3 and 4 above. You can also use slots that you have defined for the intent in a sample utterance by wrapping curly braces around the slot name, as in {*`DepartureCity`*} in example 4. If you include slot names in a sample utterance, Amazon Lex V2 fills the slots of the intent with the values that the user provides in the utterance.

A variety of sample utterances helps Amazon Lex V2 generalize to effectively recognize that the user wants to initiate the intent.

You can add sample utterances in the intent editor, visual conversation builder, or with the CreateIntent or UpdateIntent API operations. You can also generate sample utterances automatically by taking advantage of Amazon Bedrock's generative AI capabilities. For more information, see Use utterance generation to generate sample utterances for intent recognition.

Use the Intent editor or Visual conversation builder

1. In the Intent editor, navigate to the **Sample utterances** section. In the Visual conversation builder, find the **Sample utterances** section in the **Start** block.

2. In the box with the transparent text **`I want to book a flight`**, type a sample utterance. Select **Add utterance** to add the utterance.
3. View the sample utterances you have added in either **Preview** or **Plain text** mode. In **Plain text**, each line is a separate utterance. In **Preview mode**, hover over an utterance to reveal the following options:
   - Select the text box to edit the utterance.
   - Select the x button on the right of the text box to delete the utterance.
   - Drag the button on the left of the text box to change the order of sample utterances.
4. Use the search bar at the top to search through your sample utterances and the dropdown menu next to it to sort by the order you added the utterances or in alphabetical order.

Use an API operation

1. Create a new intent with the CreateIntent operation or update an existing one with the UpdateIntent operation.
2. The API request includes a `sampleUtterances` field, which maps to an array of SampleUtterance objects.
3. For each sample utterance that you want to add, append a `SampleUtterance` object to the array. Add the sample utterance as the value of the `utterance` field.
4. To edit and delete sample utterances, send an `UpdateIntent` request. The list of utterances you provide in the `sampleUtterances` field replaces the existing utterances.

> **Important**
>
> Any field that you leave blank in the `UpdateIntent` request will cause existing configurations in the intent to be deleted. Use the DescribeIntent operation to return the bot configuration and copy any configurations that you do not want to be deleted into the `UpdateIntent` request.

Intent structure

Intents are the goals that your users want to accomplish, such as ordering flowers or booking a hotel. Your bot must have at least one intent. An intent is made up of the following components

- **Initial response** – The initial message sent to the user after the intent is invoked. You can set responses, initialize values, and define the next step that your bot takes to respond to the user at the beginning of the intent.
- **Slots** – The parameters required to fulfill an intent. Each slot has a type that defines the values that can be entered in the slot. You can choose from your custom slot types or you can choose a built-in slot type.
- **Confirmation** – After the conversation with the user is complete and the slot values for the intent are filled, you can set a confirmation prompt to ask the user if the slot values are correct.
- **Fulfillment** – The response sent to a user during the course of fulfillment. You can set fulfillment progress updates at the start of fulfillment and continue sending periodic updates while the fulfillment is in progress. You can also set a post-fulfillment success message, a failure message, and a timeout message.
- **Closing response** – The closing response sent to the user after their intent is fulfilled. You can set the closing response to end the conversation, or you can set it to let the user know that they can continue with another intent.

Topics

Initial response

The initial response is sent to the user after Amazon Lex V2 determines the intent and before it starts to elicit slot values. You can use this response to inform the user of the intent that was recognized and to prepare them for the information that you collect to fulfill the intent.

For example, if the intent is to schedule a service appointment for a car, the initial response might be:

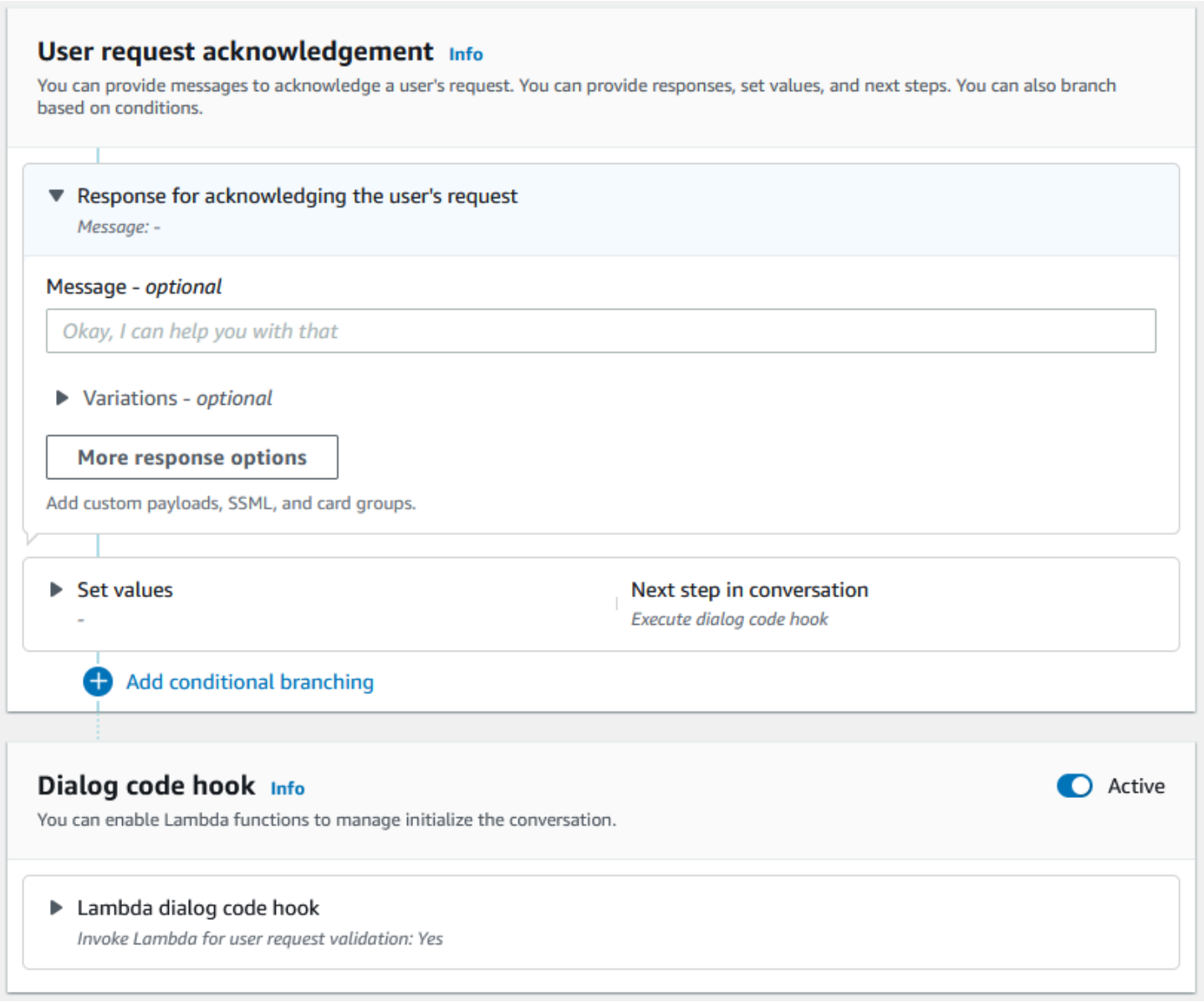
> I can help you schedule an appointment. You'll need to provide the make, model, and year of your car.

An initial response message isn't required. If you don't provide one, Amazon Lex V2 continues to follow the next step of the initial response.

You can configure the following options within the initial response:

- **Configure next step** – You can provide the next step in the conversation such as jumping to a specific dialog action, eliciting a particular slot, or jumping to a different intent. For more information, see Configure next steps in the conversation.
- **Set values** – You can set values for slots and session attributes. For more information, see Set values during the conversation
- **Add conditional branching** – You can apply conditions after playing the initial response. When a condition evaluates to true, the actions that you define are taken. For more information, see Add conditions to branch conversations.
- **Execute dialog code hook** – You can define a Lambda code hook to initialize data and execute business logic. For more information, see Invoke dialog code hook. If the option to execute Lambda function is enabled for the intent, the dialog code hook is executed by default. You can disable dialog code hook by toggling the **Active**button.

In the absence of a condition or an explicit next step, Amazon Lex V2 moves to the next slot in priority order.



Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Slots

Slots are values provided by the user to fulfill the intent. There are two types of slots:

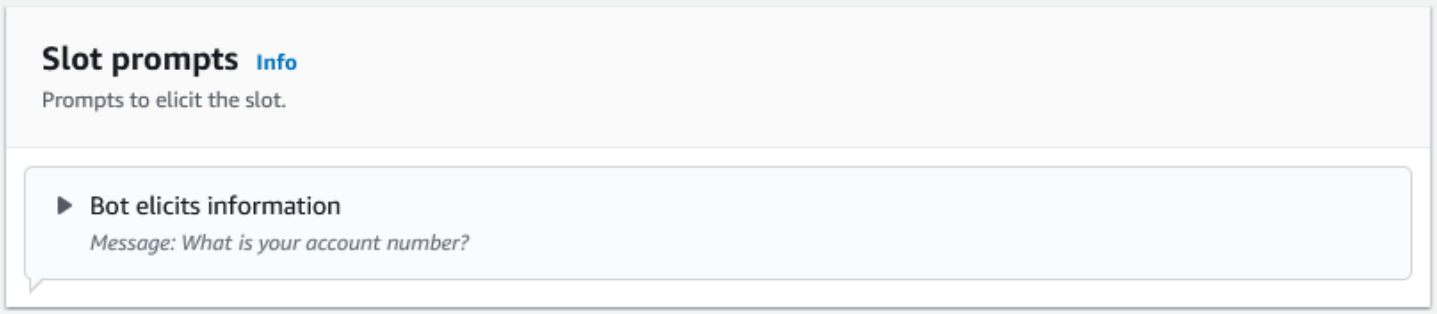
- **Built-in slot type** – You can use built-in slot types to capture standard values such as number, name, and city. For a list of supported built-in slot types, see Built-in slot types.
- **Custom slot type** – You can use custom slot types to capture custom values specific to the intent. For example, you can use a custom slot type to capture account type as “Checking” or “Savings”. For more information, see Custom slot type.

To define a slot in an intent, you have to configure the following:

- **Slot info** – This field contains a name and an optional description for the slot. For example, you can provide slot name as “AccountNumber” to capture account numbers. If the slot is required as part of the conversation flow for fulfilling the intent, it must be marked as required.
- **Slot type** – A slot type defines the list of values that a slot can accept. You can create a custom slot type or use a pre-defined slot type.
- **Slot prompt** – A slot prompt is a question posed to the user to gather information. You can configure the number of retries used to gather information and the variation of the prompt used for each retry. You can also enable a Lambda function invocation after each retry to process the input captured and attempt to resolve to a valid input.
- **Wait and Continue (optional)** – By enabling this behavior, users can say phrases such as "hold on a second" to make the bot wait for them to find the information and provide it. This is enabled only for streaming conversations. For more information, see Enabling the Amazon Lex V2 bot to wait for the user to provide more information during a pause.
- **Slot capture responses** – You can configure a success response and a failure response based on the outcome of capturing the slot value from user input.
- **Conditional branching** – You can apply conditions after playing the initial response. When a condition evaluates to true, the actions that you define are taken. For more information, see Add conditions to branch conversations.
- **Dialog code hook** – You can also use a Lambda code hook to validate the slot values and execute business logic. For more information, see Invoke dialog code hook.
- **User input type** – You can configure input type so the bot can accept a specific modality. By default, both audio and DTMF modalities are accepted. You can selectively set it to audio only or DTMF only.

- **Audio input timeouts and lengths** – You can configure audio timeouts including voice timeout and silence timeout. Also, you can set the max audio length.
- **DTMF input timeout, characters, and lengths** – You can set the DTMF timeout along with the deletion character and the end character. Also, you can set the max DTMF length.
- **Text length** – You can set the max length for text modality.

After the slot prompt is played, the user provides the slot value as an input. If Amazon Lex V2 does not understand a slot value provided by the user, it retries eliciting the slot until it understands a value or until it exceeds the maximum number of retries that you configured for the slot. Using the advanced retry settings you can configure the timeouts, restrict the type of input, and enable or disable interrupt for the initial prompt and retries. After each attempt at capturing the input, Amazon Lex V2 can call the Lambda function configured for the bot with an invocation label provided for retries. You can use the Lambda function, for example, to apply your business logic to attempt resolving it to a valid value. This Lambda function can be enabled within **Advanced options** for slot prompts.



You can define responses that the bot should send to the user once the slot value is entered or if the maximum number of retries is exceeded. For example, for a bot for scheduling service for a car, you can send a message to the user when the vehicle identification number (VIN) is entered:

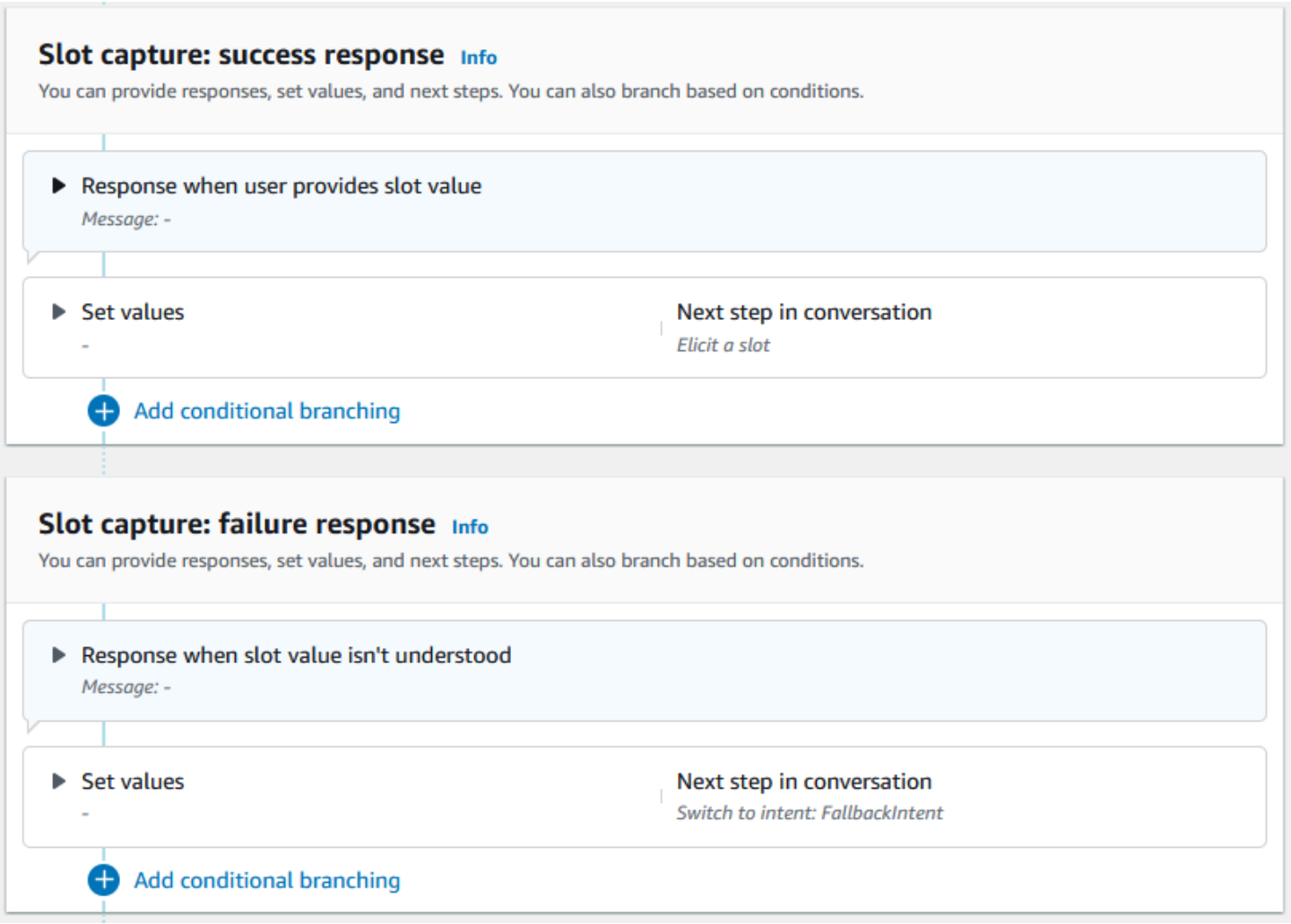
> Thank you for providing the VIN number of your car. I will now proceed to schedule an appointment.

You can create two responses:

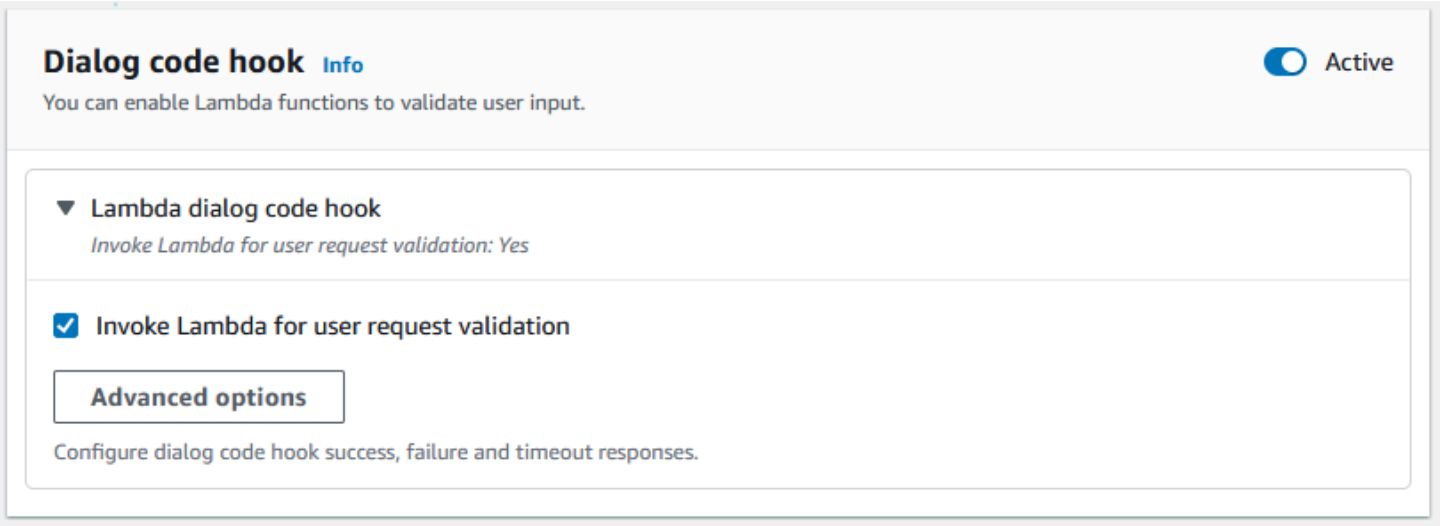
- **Success response** – sent when Amazon Lex V2 understands a slot value.
- **Failure response** – sent when Amazon Lex V2 can't understand a slot value from the user after the maximum number of retries.

You can set values, configure the next steps, and apply conditions that correspond to each response to design the conversation flow.

In the absence of a condition or an explicit next step, Amazon Lex V2 moves to the next slot in priority order.



You can use a Lambda function to validate a slot value that a user has entered and determine what the next action should be. For example, you can use the validation function to make sure that the entered value falls in the correct range, or that is correctly formatted. To activate the Lambda function, choose the **Invoke Lambda function** checkbox and the **Active** button in the **Dialog code hook** section. You can specify an invocation label for the dialog code hook. This invocation label can be used in Lambda function to write the business logic corresponding to the slot elicitation.



Slots that are not required for the intent are not part of the main conversation flow. However, if a user utterance contains a value that your bot identifies as corresponding to an optional slot, it can popluate the slot with that value. For example, if you configure a business intelligence bot to have an optional `City` slot and the user utterance **`What is the sales for April in San Diego?`**, the bot fills the optional slot with **`San Diego`**. You can configure the business logic to use the optional slot value, if present.

Slots not required for the intent cannot be elicited using next steps. These steps can be populated only during intent elicitation (as in the preceding example) or can be elicited by setting the dialog state within the Lambda function. If the slot is elicited using the Lambda function, you must use the Lambda function to decide the next step in the conversation after the slot elicitation is completed. To enable support for next step while building the bot, you must mark the slot as required for the intent.

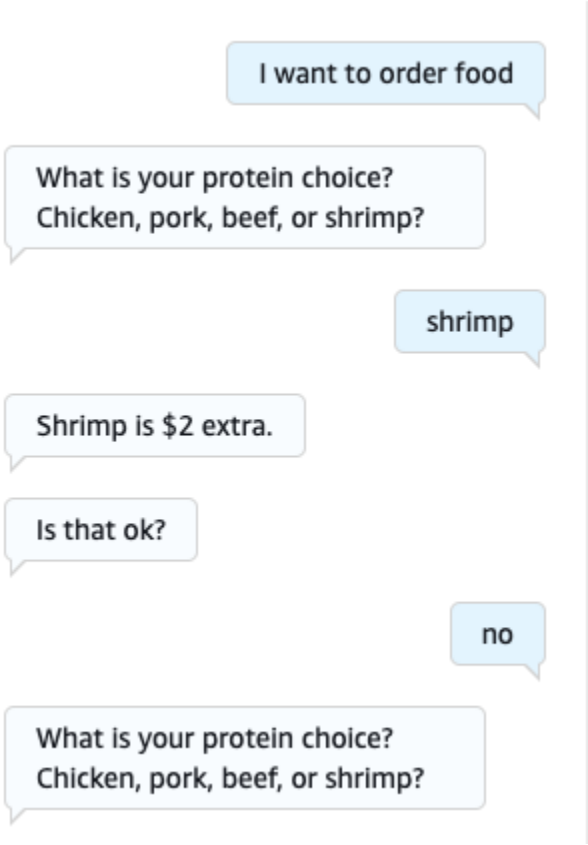
> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

The following topics describe how to configure a bot to re-elicit a slot value that has already been filled and how to create a slot that consists of multiple values:

Topics

Re-eliciting slots

You can configure your bot to re-elicit a slot that has already been filled by setting that slot value to **`null`** and setting the next step in the conversation to loop back to eliciting that slot. For example, you may want to re-elicit a slot after your customer declines a confirmation of the slot elicitation based on extra information, as in the following conversation:



You can configure a loop from the confirmation response back to re-elicit the slot with either the intent editor or the Using Visual conversation builder.

> **Note**
>
> You can loop back to re-elicit a slot at any point in the conversation provided that you set that slot value to **`null`** beforehand.

Reproducing the above example with the intent editor

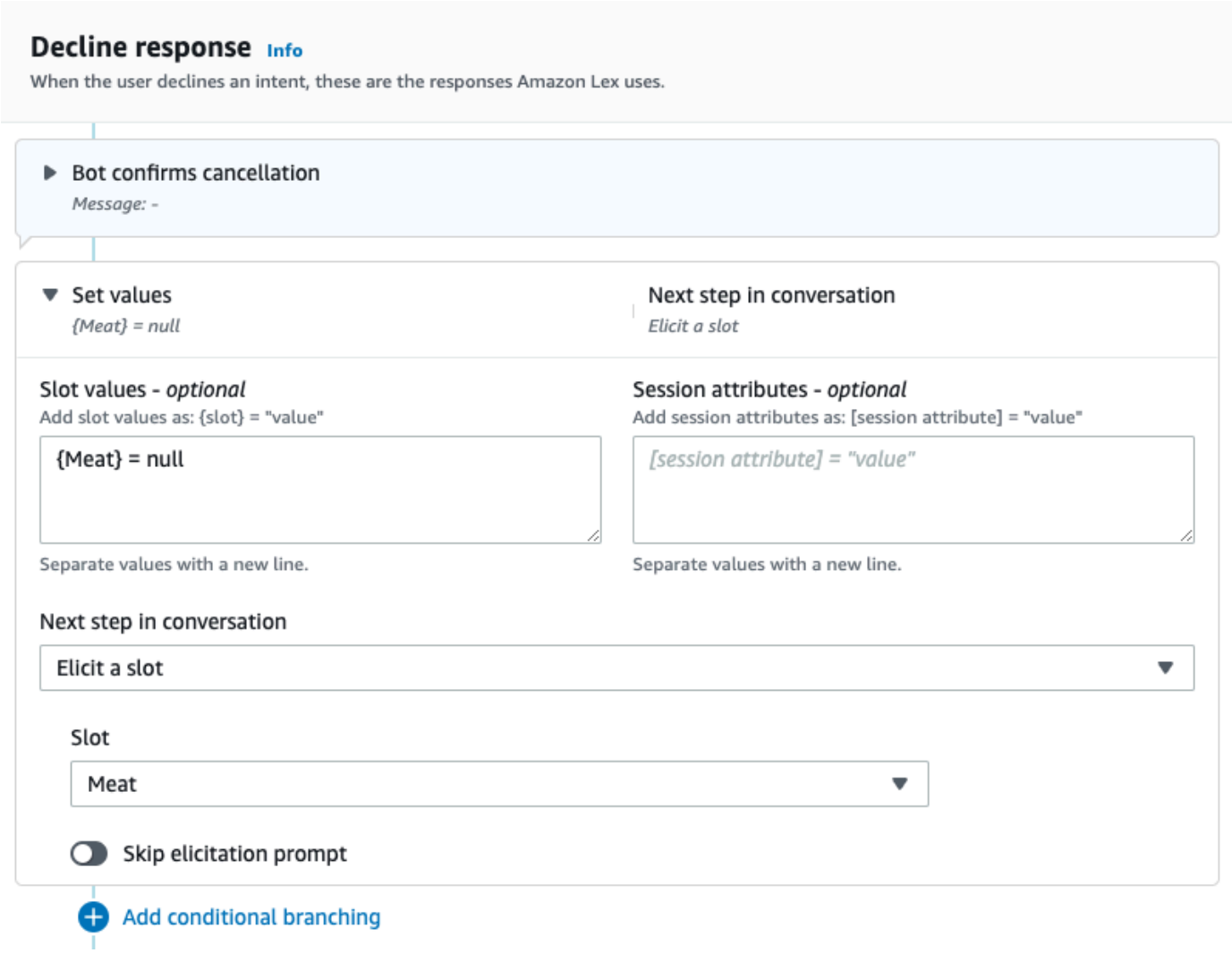
1. In the **Confirmation** section of the intent editor, select the right arrow next to **Prompts to confirm the intent** to expand the section.
2. Select **Advanced options** at the bottom.
3. In the **Decline response** section, select the right arrow next to **Set values** to expand the section. Fill in this section with the following steps, as in the image below:

    a. Set the slot value you want to re-elicit to **`null`**. In this example, we want to re-elicit the `Meat` slot, so we input **`{Meat} = null`** in the **Slot values** section.

    b. In the dropdown menu under **Next step in conversation**, choose **Elicit a slot**.

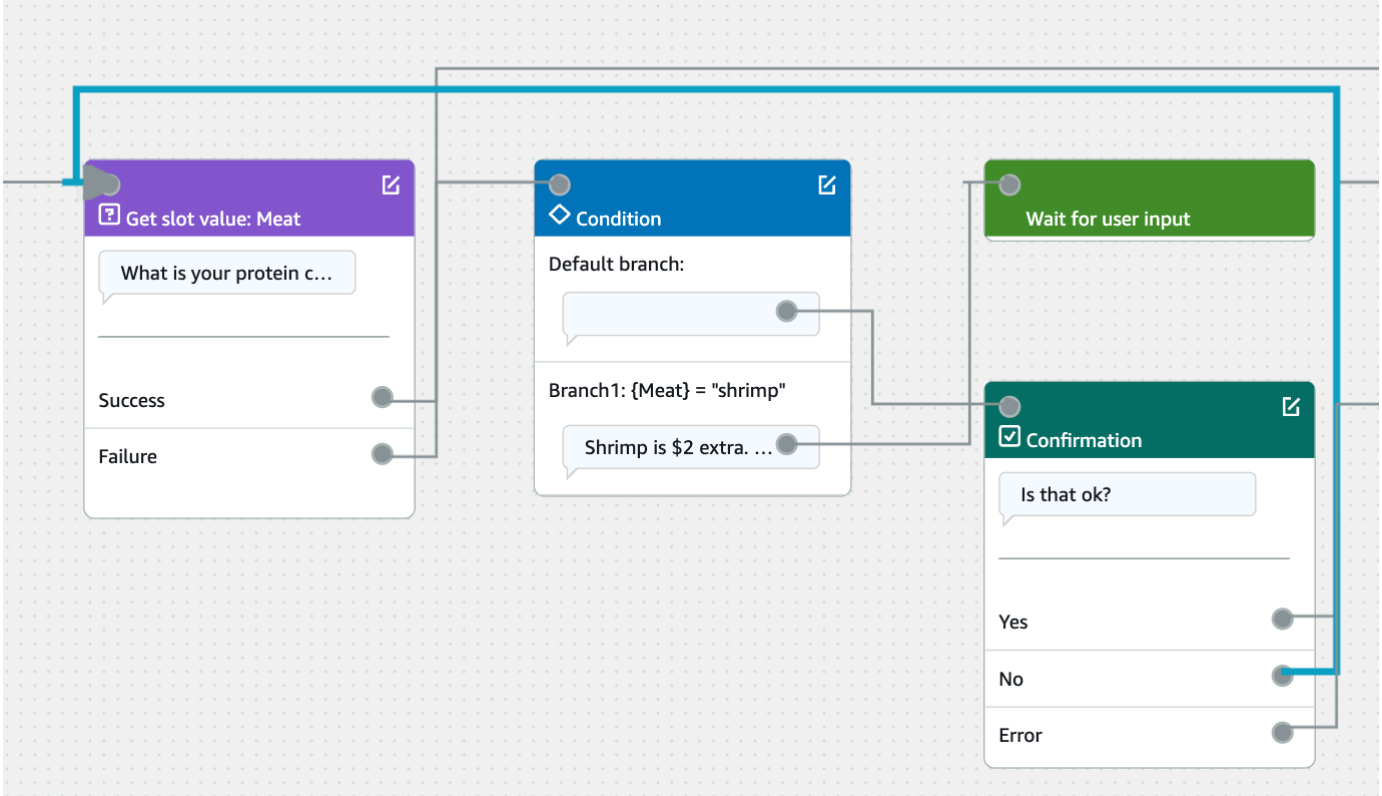
    c. A **Slot** section will appear. In the dropdown menu under it, choose the slot you want to re-elicit.

    d. Select **Update options** to confirm your changes.

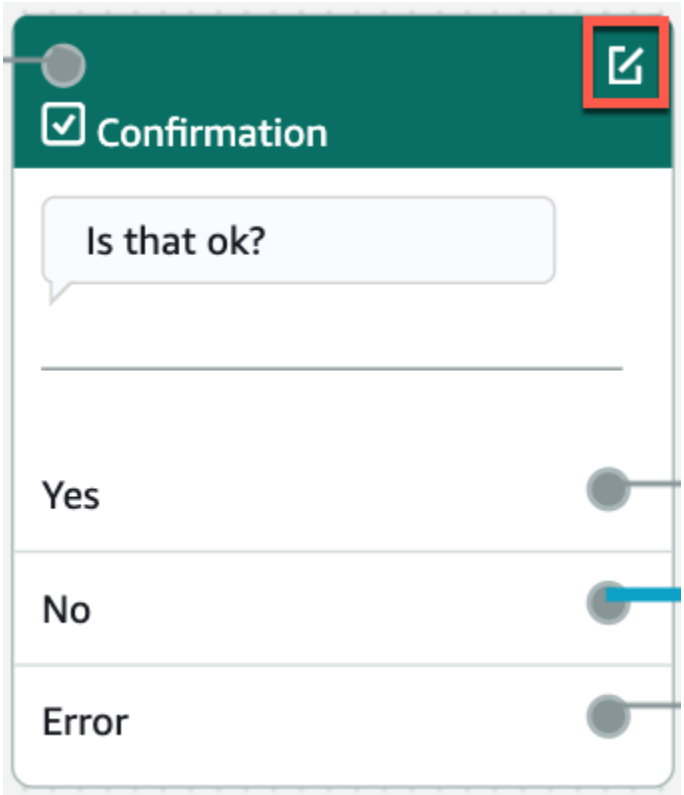


Reproducing the above example with the Visual conversation builder

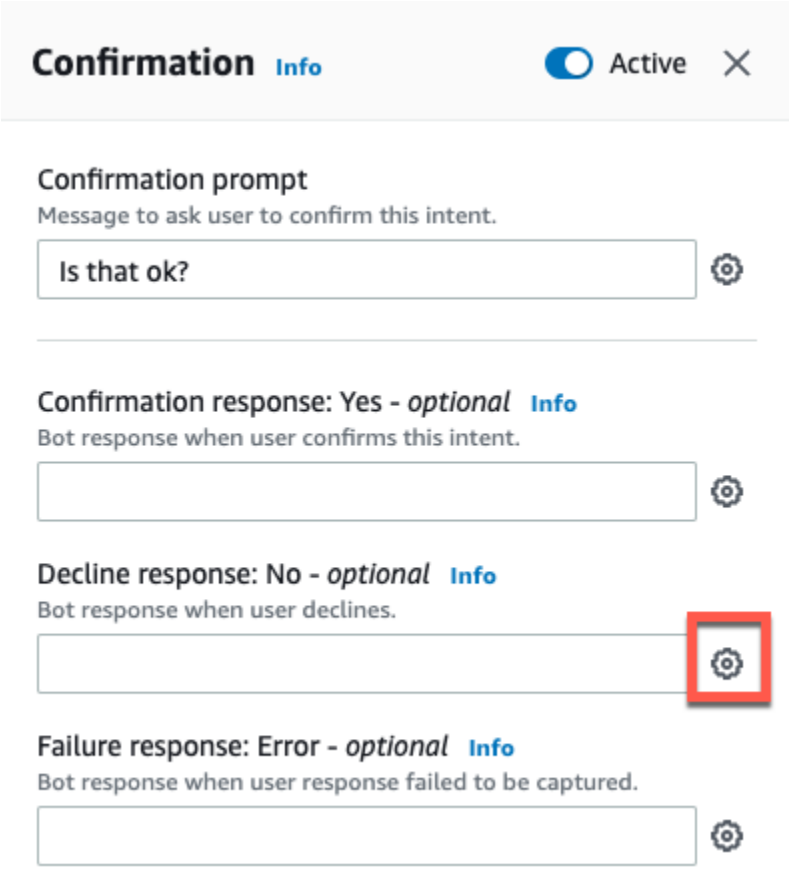
1. Create a connection from the **No** port of the **Confirmation** block to the incoming port of the **Get slot value: Meat** block.



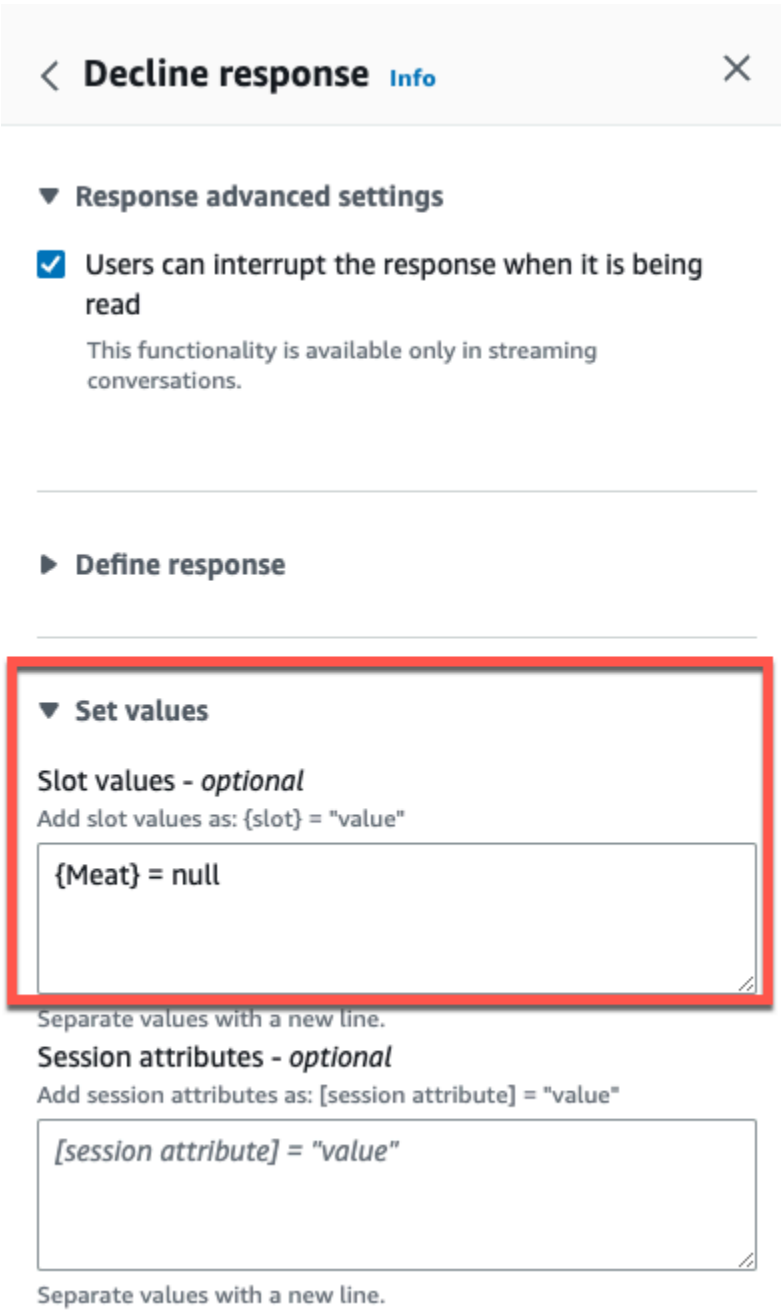
2. Select the **Edit** icon in the top right corner of the **Confirmation** block.



3. Select the gear icon next to the bot response in the **Decilne response** section.



4. In the **Set values** section, add "{Meat} = null" in the **Slot values** box.



5. Select **Save Intent.**

Using multiple values in a slot

> **Note**
>
> Multiple value slots are only supported in the English (US) language.

For some intents, you might want to capture multiple values for a single slot. For example, a pizza ordering bot might have an intent with the following utterance:

```
I want a pizza with {toppings}
```

The intent expects that the `{toppings}` slot contains a list of the toppings that the customer wants on their pizza, for example "pepperoni and pineapple".

To configure a slot to capture multiple values, you set the `allowMultipleValues` field on the slot to true. You can set the field using the console or with the CreateSlot or UpdateSlot operation.

You can only mark slots with custom slot types as multi-value slots.

For a multi-value slot, Amazon Lex V2 returns a list of slot values in the response to the RecognizeText or RecognizeUtterance operation. The following is the slot information returned for the utterance "I want a pizza with pepperoni and pineapple" from the OrderPizza bot.

```
    "slots": {
        "toppings": {
            "shape": "List",
            "value": {
                "interpretedValue": "pepperoni and pineapple",
                "originalValue": "pepperoni and pineapple",
                "resolvedValues": [
                    "pepperoni and pineapple"
                ]
            },
            "values": [
                {
                    "shape": "Scalar",
                    "value": {
                        "interpretedValue": "pepperoni",
                        "originalValue": "pepperoni",
                        "resolvedValues": [
                            "pepperoni"
                        ]
                    }
                },
                {
                    "shape": "Scalar",
                    "value:": {
                        "interpretedValue": "pineapple",
```

```
                    "originalValue": "pineapple",
                    "resolvedValues": [
                        "pineapple"
                    ]
                }
            }
        ]
    }
}
```

Multi-valued slots always return a list of values. When the utterance only contains one value, the list of values returned only contains one response.

Amazon Lex V2 recognizes multiple values separated by spaces, commas (,), and the conjunction "and". Multi-value slots work with both text and voice input.

You can use multi-valued slots in prompts. For example, you can set the confirmation prompt for an intent to

```
Would you like me to order your {toppings} pizza?
```

When Amazon Lex V2 sends the prompt to the user, it sends "Would you like me to order your pepperoni and pineapple pizza?"

Multi-valued slots support single default values. If multiple default values are provided, Amazon Lex V2 populates the slot with only the first available value. For more information, see Using default slot values in intents for your Lex V2 bot.

You can use slot obfuscation to mask the values of a multi-value slot in conversation logs. When you obfuscate slot values, the value of each of the slot values is replaced with the name of the slot. For more information, see Obscuring slot values in conversation logs from Lex V2.

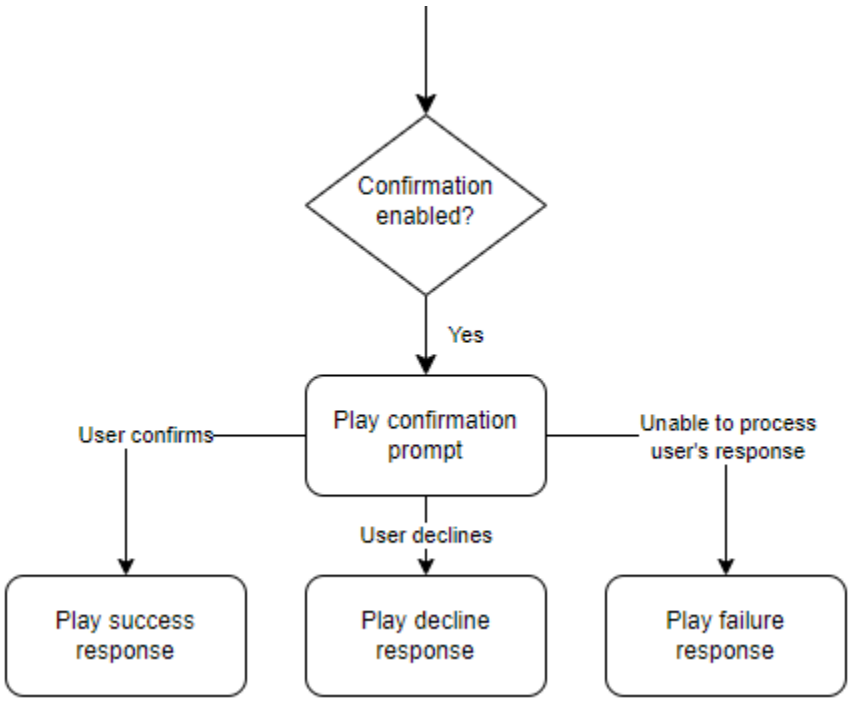
Confirmation

After the conversation with the user is complete and the slot values for the intent are filled, you can configure a confirmation prompt to ask the user if the slot values are correct. For example, a bot that schedules service appointments for cars might prompt the user with the following:

I've got service for your 2017 Honda Civic scheduled for March 25th at 3:00 PM. Is that all right?

You can define 3 types of responses to the confirmation prompt:

- **Confirmation response** – This response is sent to the user when the user confirms the intent. For example, after the user replies "yes" to the prompt "do you want to place the order?"
- **Decline response** – This response is sent to the user when the user declines the intent. For example, after the user replies "no" to the prompt "do you want to place the order?"
- **Failure response** – This response is sent to the user when the confirmation prompt can't be processed. For example, if the user's response couldn't be understood or couldn't be resolved to a yes or a no.

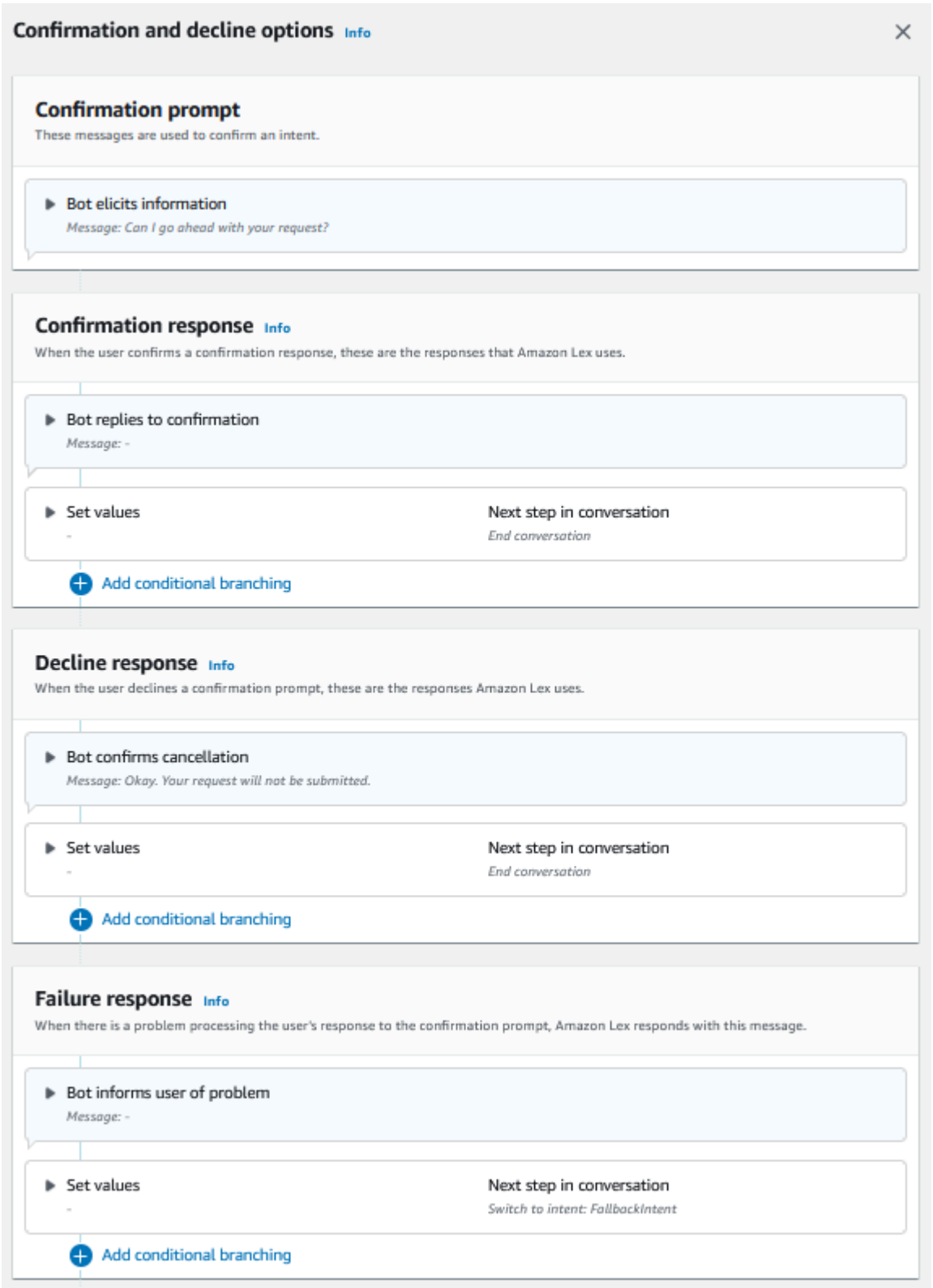


If you don't specify a confirmation prompt, Amazon Lex V2 moves to the fulfillment step or the closing response.

You can set values, configure the next steps, and apply conditions corresponding to each response to design the conversation flow. In the absence of a condition or an explicit next step, Amazon Lex V2 moves to the fulfillment step.

You can also enable the dialog code hook to validate the information captured in the intent prior to sending it for fulfillment. To use a code hook, enable the dialog code hook in the confirmation prompt advanced options. In addition, configure the next step of the previous state to execute the dialog code hook. For more information, see Invoke dialog code hook.

> **Note**
>
> If you use a code hook to trigger the confirmation step at runtime, you must mark the confirmation step as **Active** at build time.


Confirmation and decline options Info
Confirmation prompt
These messages are used to confirm an intent.
Bot elicits information
Message: Can I go ahead with your request?
Confirmation response Info
When the user confirms a confirmation response, these are the responses that Amazon Lex uses.
Bot replies to confirmation
Message: -
Set values
-
Next step in conversation
End conversation
Add conditional branching
Decline response Info
When the user declines a confirmation prompt, these are the responses Amazon Lex uses.
Bot confirms cancellation
Message: Okay. Your request will not be submitted.
Set values
-
Next step in conversation
End conversation
Add conditional branching
Failure response Info
When there is a problem processing the user's response to the confirmation prompt, Amazon Lex responds with this message.
Bot informs user of problem
Message: -
Set values
-
Next step in conversation
Switch to intent: FallbackIntent
Add conditional branching

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Using a Lambda function to validate an intent.

You can define a Lambda code hook to validate the intent before you send it for fulfillment. To use a code hook, enable the dialog code hook in the confirmation prompt advanced options.

When you use a code hook, you can define the actions that Amazon Lex V2 takes after the code hook runs. You can create three types of responses:

- **Success response** – Sent to the user when the code hook completes successfully.
- **Failure response** – Sent to the user when the code hook doesn't run successfully or when the code hook returns `Failure` in the response.
- **Timeout response** – Sent to the user when the code hook does not complete in its configured timeout period.

Fulfillment

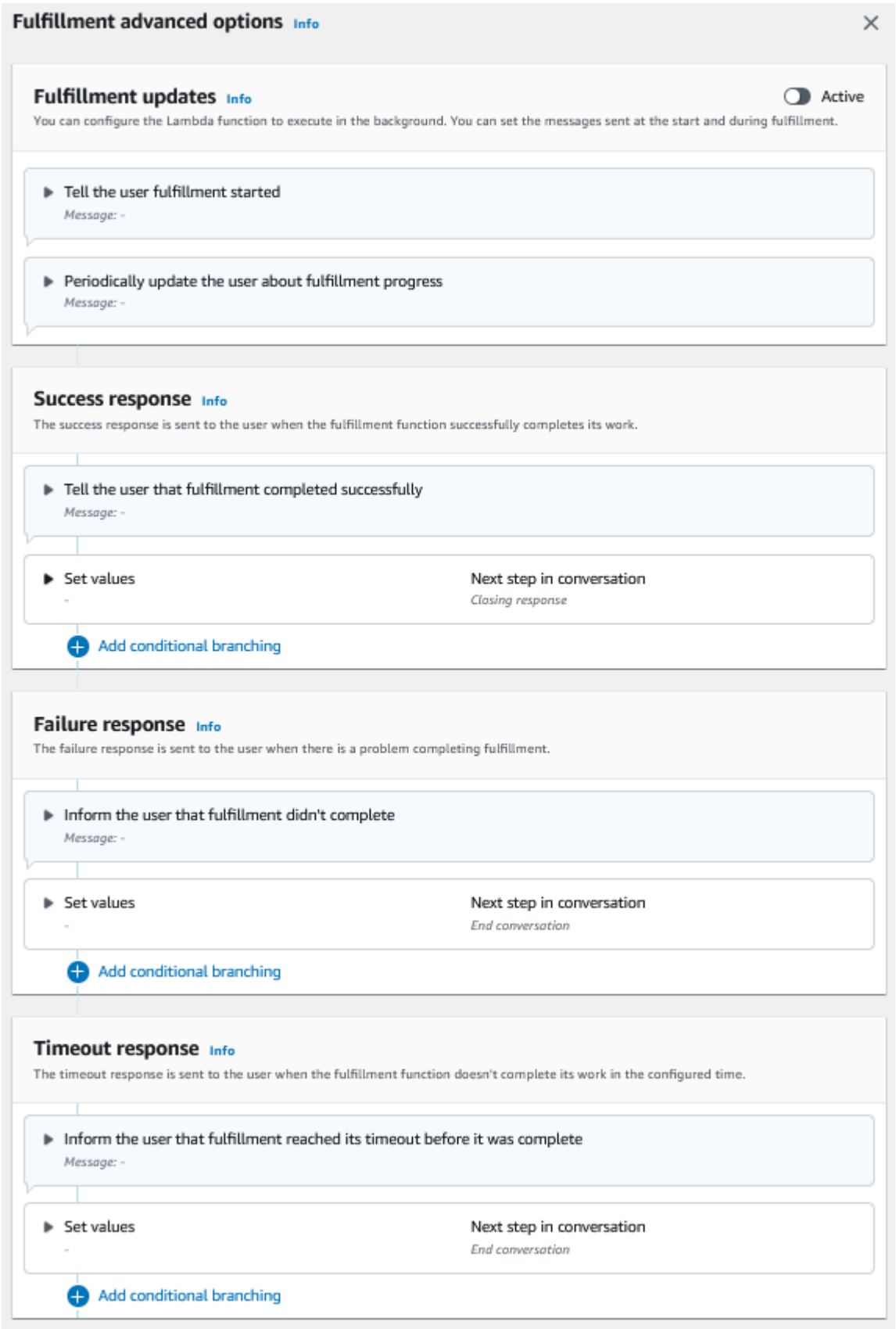
After all the slot values are provided by the user for the intent, Amazon Lex V2 fulfills the user's request. You can configure the following options for fulfillment.

- **Fulfillment code hook** – You can use this option to control the fulfillment Lambda invocation. If the option is disabled, the fulfillment succeeds without invoking the Lambda function.
- **Fulfillment updates** – You can enable fulfillment updates for Lambda functions that take more than a few seconds to complete, so that the user knows that the process is in progress. For more information, see Configuring fulfillment progress updates for your Lex V2 bot. This functionality is only available for streaming conversations.
- **Fulfillment responses** – You can configure a success response, a failure response, and a timeout response. The appropriate response is returned to the user based on the status of the fulfillment Lambda invocation.

There are three possible fulfillment responses:

- **Success response** – A message sent when the fulfillment Lambda completes successfully.
- **Failure response** – A message sent if the fulfillment failed or Lambda can't be completed for some reason.
- **Timeout response** – A message sent if the fulfillment Lambda function doesn't finish within the configured timeout.

You can set values, configure the next steps, and apply conditions corresponding to each response to design the conversation flow. In the absence of a condition or an explicit next step, Amazon Lex V2 moves to closing response.


Fulfillment advanced options Info
Fulfillment updates Info
Active
You can configure the Lambda function to execute in the background. You can set the messages sent at the start and during fulfillment.
Tell the user fulfillment started
Message: -
Periodically update the user about fulfillment progress
Message: -
Success response Info
The success response is sent to the user when the fulfillment function successfully completes its work.
Tell the user that fulfillment completed successfully
Message: -
Set values
-
Next step in conversation
Closing response
Add conditional branching
Failure response Info
The failure response is sent to the user when there is a problem completing fulfillment.
Inform the user that fulfillment didn't complete
Message: -
Set values
-
Next step in conversation
End conversation
Add conditional branching
Timeout response Info
The timeout response is sent to the user when the fulfillment function doesn't complete its work in the configured time.
Inform the user that fulfillment reached its timeout before it was complete
Message: -
Set values
-
Next step in conversation
End conversation
Add conditional branching

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

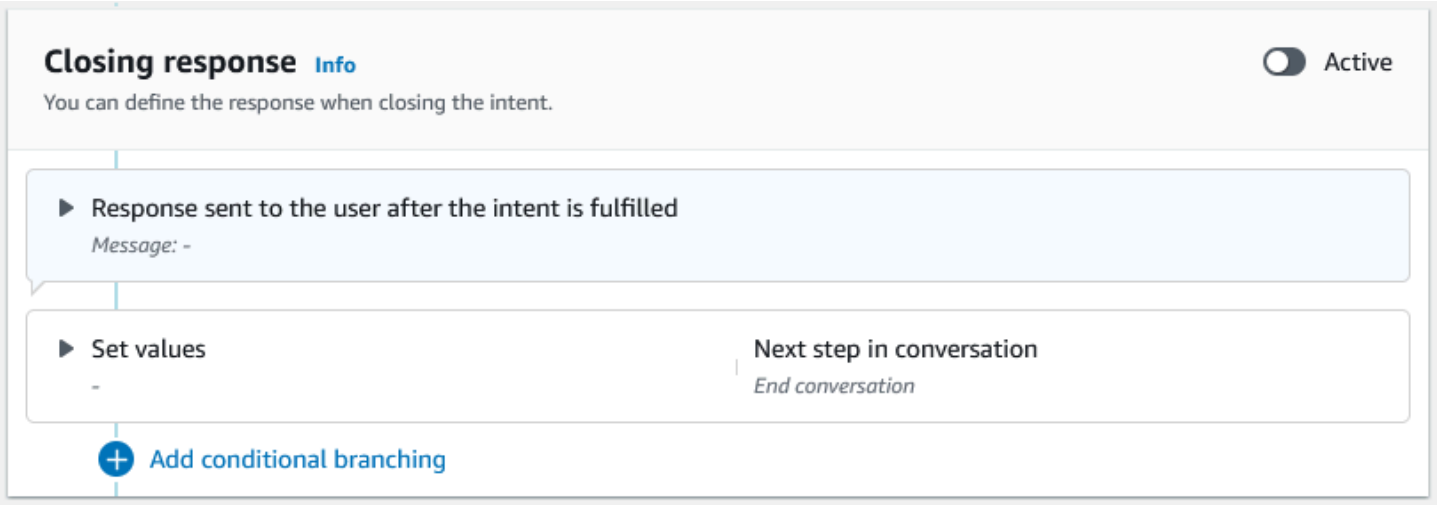
Closing response

The closing response is sent to your user after their intent is fulfilled. You can use the closing response to end the conversation, or you can use it to let the user know that they can continue with another intent. For example, in a travel booking bot, you can set the closing response for the book hotel room intent to this:

```
All right, I've booked your hotel room. Is there anything else I can help you with?
```

You can set values, configure the next steps, and apply conditions after the closing response to the design the conversation path. In the absence of a condition or an explicit next step, Amazon Lex V2 ends the conversation.

If you don't supply a closing response, or if none of the conditions evaluates to true, Amazon Lex V2 ends the conversation with your bot.



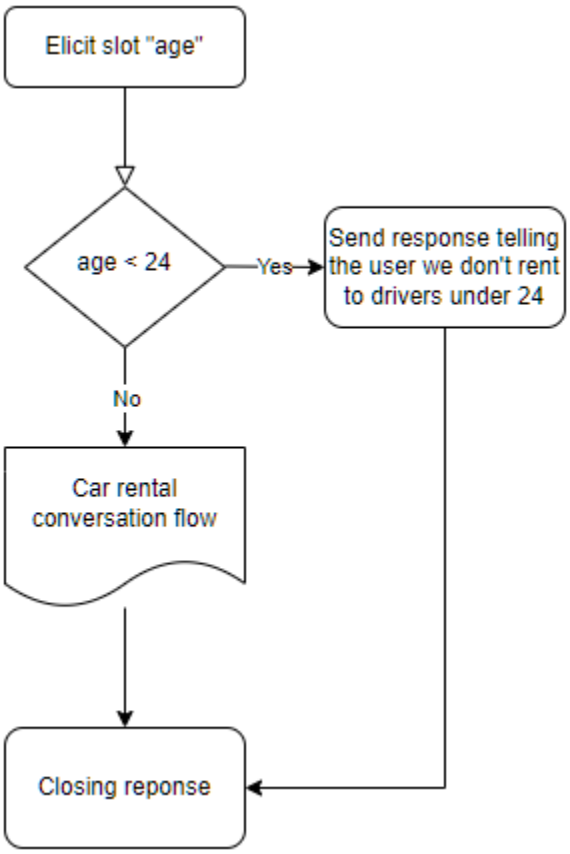
Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Creating conversation paths

Typically, Amazon Lex V2 manages the flow of conversations with your users. For simple bots, the default flow can be enough to create a good experience for your users. However, for more complex bots, you might want to take control of the conversation and direct the flow into more complex paths.

For example, in a bot that books car rentals, you might not rent to younger drivers. In this case, you can create a condition that checks to see if a driver is below a certain age, and if so, jump to the closing response.



To design such interactions, you can configure the next step at each point in the conversation, evaluate conditions, set values and invoke code hooks.

Conditional branching helps you create paths for your users through complex interactions. You can use a conditional branch at any point that you pass control of the conversation to your bot. For example, you can create a condition before the bot elicits the first slot value, you can create a condition between eliciting each slot value, or you can create a condition before the bot closes the conversation. For a list of the places that you can add conditions, see Adding intents.

When you create a bot, Amazon Lex V2 creates a default path through the conversation based on the priority order of the slots. To customize the conversation path, you can modify the next step at any point in the conversation. For more information, see Configure next steps in the conversation.

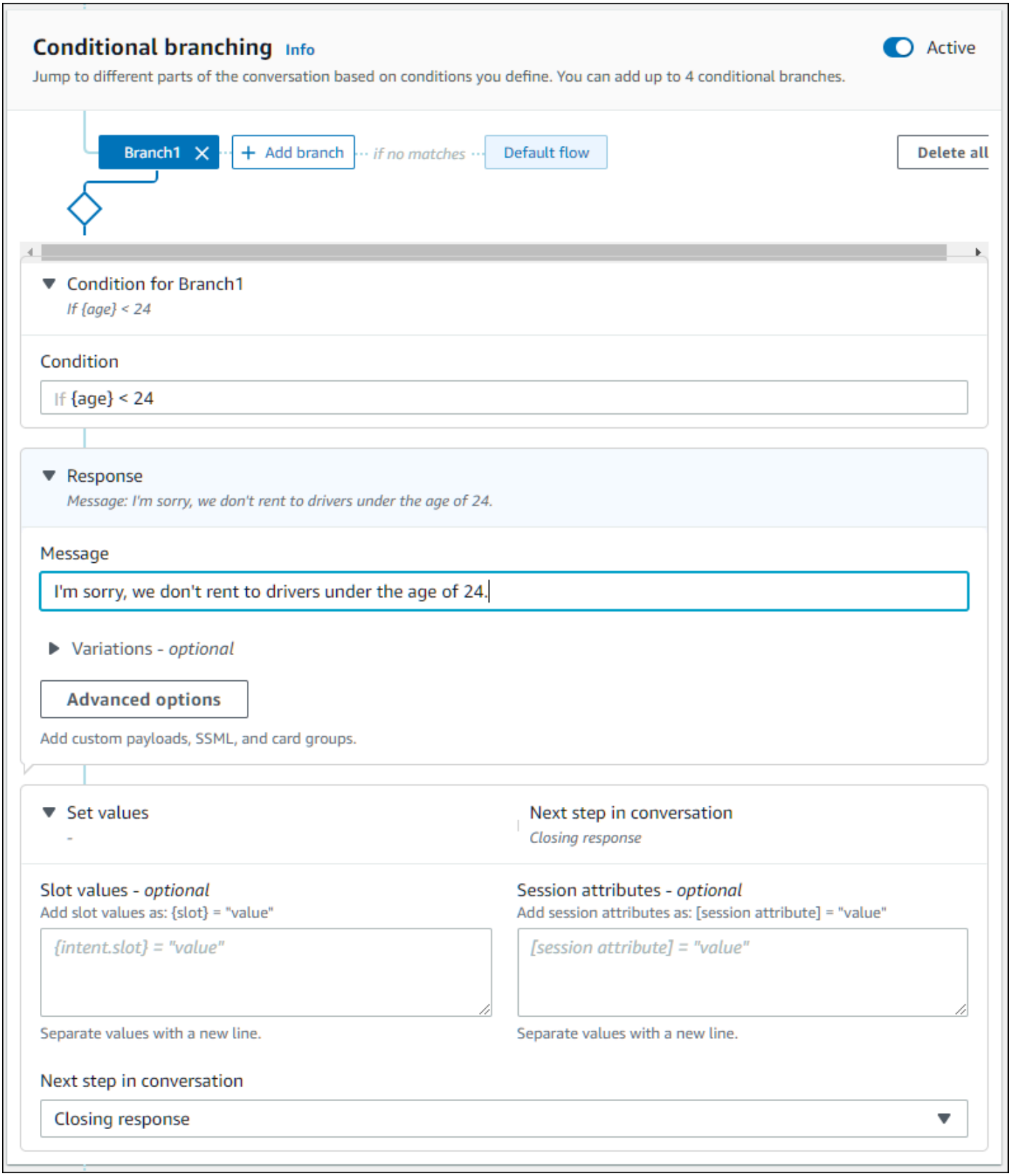
To create alternative paths based on conditions, you can use a conditional branch at any point in the conversation. For example, you can create a condition before the bot elicits the first slot value. You can create a condition between eliciting each slot value, or you can create a condition before the bot closes the conversation. For a list of the places allowing you to add conditions, see Add conditions to branch conversations.

You can set conditions based on slot values, session attributes, the input mode and input transcript, or a response from Amazon Kendra.

You can set slot and session attribute values at each point in the conversation. For more information, see Set values during the conversation.

You can also set the next action to dialog code hook to run a Lambda function. For more information, see Invoke dialog code hook.

The following image shows the creation of a path for a slot in the console. In this example, Amazon Lex V2 will elicit the slot "age". If the value of the slot is less than 24, Amazon Lex V2 jumps to the closing response, otherwise Amazon Lex V2 will follow the default path.


Conditional branching Info
Active
Jump to different parts of the conversation based on conditions you define. You can add up to 4 conditional branches.
Branch1
Add branch
if no matches
Default flow
Delete all
Condition for Branch1
If {age} < 24
Condition
If {age} < 24
Response
Message: I'm sorry, we don't rent to drivers under the age of 24.
Message
I'm sorry, we don't rent to drivers under the age of 24.
Variations - optional
Advanced options
Add custom payloads, SSML, and card groups.
Set values
-
Next step in conversation
Closing response
Slot values - optional
Add slot values as: {slot} = "value"
{intent.slot} = "value"
Separate values with a new line.
Session attributes - optional
Add session attributes as: [session attribute] = "value"
[session attribute] = "value"
Separate values with a new line.
Next step in conversation
Closing response

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Configure next steps in the conversation

You can configure a next step at each stage of the conversation to design conversations. Typically, Amazon Lex V2 automatically configures the default next steps for each stage of the conversation as per the following order.

Initial Response → Slot Elicitation → Confirmation (if active) → Fulfillment (if active) → Closing Response (if active) → End conversation

You can modify the default next steps and design the conversation based on the expected user experience. The following next steps can be configured at each stage of the conversation:

Jump to

- **Initial response** – The conversation is restarted from the beginning of the intent. You can choose to skip the initial response while configuring this next step
- **Elicit a slot** – You can elicit any slot in the intent.
- **Evaluate conditions** – You can evaluate conditions and branch conversation at any step of the conversation.
- **Invoke dialog code hook** – You can invoke business logic at any step.
- **Confirm intent** – The user will be prompted to confirm the intent.
- **Fulfill intent** – The fulfillment of the intent will begin as a next step.
- **Closing response** – The closing response will be returned to the user.

Switch to

- **Intent** – You can transition to a different intent and continue the conversation for this intent. You can optionally skip the initial response of the intent while making the transition.

- **Intent: specific slot** – You can directly elicit a specific slot in a different intent if you have already captured some slot values in the current intent.

Wait for user input – The bot waits for the user to provide inputs for recognizing any new intent. You can configure prompts such as "Is there anything else I can help you with?" before setting this next step. The bot will be in `ElicitIntent` dialog state.

End conversation – The conversation with the bot is closed.

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Set values during the conversation

Amazon Lex V2 provides the ability to set slot values and session attribute values at every step of the conversation. You can then use these values during the conversation to evaluate conditions or use them during intent fulfillment.

You can set slot values for the current intent. If the next step in the conversation is to invoke another intent, you can set slot values of the new intent.

If the assigned slot is not filled, or if the JSON path cannot be parsed, then the attribute will be set to `null`.

Use the following syntax when using slot values and session attributes:

- **Slot values** – surround the slot name with braces ("{ }"). For slot values in the current intent, you only need to use the slot name. For example, `{slot}`. If you are setting a value in the next intent, you must use both the intent name and the slot name to identify the slot. For example, `{intent.slot}`.

  Examples:

  - `{PhoneNumber} = "1234567890"`

- `{CheckBalance.AccountNumber} = "99999999"`
- `{BookingID} = "ABC123"`
- `{FirstName} = "John"`

The value of a slot can be any of the following:

- a constant string
- a JSON path that refers to the transcriptions block in the Amazon Lex response (for en-US and en-GB)
- a session attribute

Examples:

- `{username} = "john.doe"`
- `{username_confidence} = $.transcriptions[0].transcriptionConfidence`
- `{username_slot_value} = [username]`

> **Note**
>
> Slot values can also be set to `null`. If you need to re-elicit a slot value that has been filled, you must set the value to `null` before prompting the customer for the slot value again. If the assigned slot is not filled, or if the JSON path cannot be parsed, then the attribute will be set to `null`.

- **Session attributes** – surround the attribute name with square brackets ("[ ]"). For example, `[sessionAttribute]`.

  Examples:

  - `[username] = "john.doe"`
  - `[username_confidence] = $.transcriptions[0].transcriptionConfidence`
  - `[username_slot_value] = {username}`

  The value of the session attribute can be any of the following:

  - a constant string
  - a JSON path that refers to the transcriptions block in the Amazon Lex response (for en-US and en-GB)
  - a slot value reference

> **Note**
>
> If the assigned slot is not filled, or if the JSON path cannot be parsed, then the attribute will be set to `null`.

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

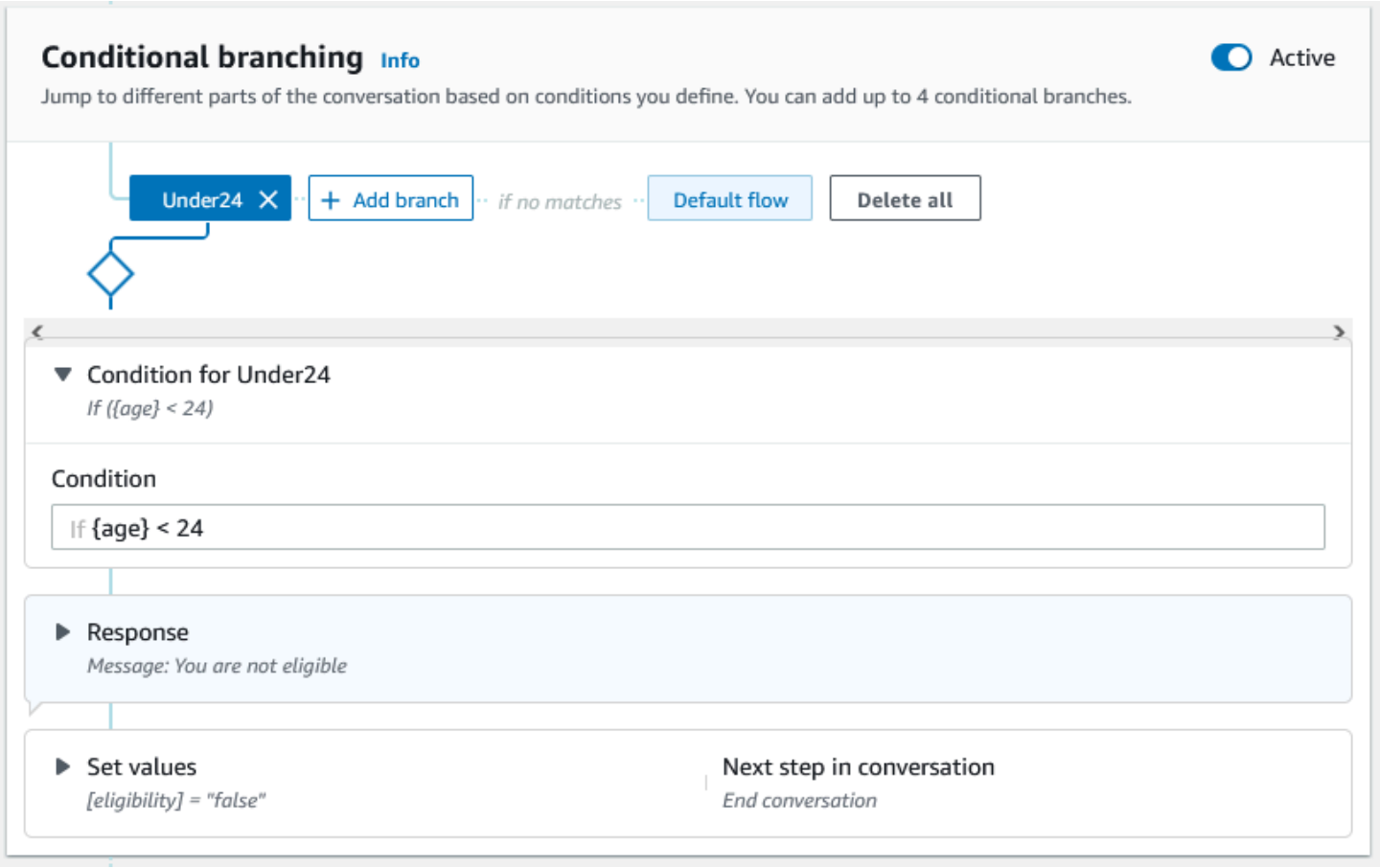
Add conditions to branch conversations

You can use *conditional branching* to control the path that your customer takes through the conversation with your bot. You can branch the conversation based on slot values, session attributes, the contents of the input mode and input transcript fields, or a response from Amazon Kendra.

You can define up to four branches. Each branch has a condition that must be satisfied in order for Amazon Lex V2 to follow that branch. If none of the branches has its condition satisfied, a default branch is followed.

When you define a branch, you define the action that Amazon Lex V2 should take if the conditions corresponding to that branch evaluate to true. You can define any of the following actions:

- A response sent to the user.
- Slot values to apply to slots.
- Session attribute values for the current session.
- The next step in the conversation. For more information, see Creating conversation paths.



Each conditional branch has a Boolean expression that must be satisfied for Amazon Lex V2 to follow the branch. There are comparison and Boolean operators, functions, and quantifier operators that you can use for your conditions. For example, the following condition returns true if the {age} slot is less than 24.

```
{age} < 24
```

The following condition returns true if the {toppings} multi-value slot contains the word "pineapple".

```
{toppings} CONTAINS "pineapple"
```

You can combine multiple comparison operators with a Boolean operator for more complex conditions. For example, the following condition returns true if the {make} slot value is "Honda" and the {model} slot value is "Civic". Use parentheses to set the evaluation order.

```
({make} = "Honda") AND ({model} = "Civic")
```

The following topics provide details on the conditional branch operators and functions.

> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Topics

Comparison operators

Amazon Lex V2 supports the following comparison operators for conditions:

- Equals (=)
- Not equals (!=)
- Less than (<)
- Less than or equals (<=)
- Greater than (>)
- Greater than or equals (>=)

When using a comparison operator, it uses the following rules.

- The left-hand side must be a reference. For example, to reference a slot value, you use `{slotName}`. To reference a session attribute value, you use `[attribute]`. For input mode and input transcript, you use `$.inputMode` and `$.inputTranscript`.

- The right-hand side must be a constant and the same type as the left hand side.
- Any expression referencing an attribute which has not been set is treated as invalid, and is not evaluated.
- When you compare a multi-valued slot, the value used is a comma-separated list of all interpreted values.

Comparisons are based on the slot type of the reference. They are resolved as follows:

- **Strings** – strings are compared based on their ASCII representation. The comparison is case-insensitive.
- **Numbers** – number-based slots are converted from the string representation to a number and then compared.
- **Date/Time** – time-based slots are compared based on the time series. The earlier date or time is considered smaller. For durations, shorter periods are considered smaller.

Boolean operators

Amazon Lex V2 supports Boolean operators to combine comparison operators. They let you create statements similar to the following:

```
({number} >= 5) AND ({number} <= 10)
```

You can use the following Boolean operators:

- AND (&&)
- OR (||)
- NOT (!)

Quantifier operators

Quantifier operators evaluate the elements of a sequence and determine if one or more elements satisfy the condition.

- **CONTAINS** – determines if the specified value is contained in a multi-valued slot and returns true if it is. For example, `{toppings} CONTAINS "pineapple"` returns true if the user ordered pineapple on their pizza.

Functions

Functions must be prefixed with the string `fn.`. The argument to the function is a reference to a slot, session attribute, or request attribute. Amazon Lex V2 provides two functions for getting information from the values of slots, sessionAttribute, or requestAttribute.

- **fn.COUNT()** – counts the number of values in a multi-valued slot.

  For example, if the slot `{toppings}` contains the value "pepperoni, pineapple":

  ```
  fn.COUNT({toppings}) = 2
  ```

- **fn.IS_SET()** – value is true if a slot, session attribute, or request attribute is set in the current session.

  Based on the previous example:

  ```
  fn.IS_SET({toppings})
  ```

- **fn.LENGTH()** – value is the length of the value of the session attribute, slot value, or slot attribute which is set in the current session. This function does not support multi-value slots or composite slots.

  Example:

  If the slot `{credit-card-number}` contains the value "123456781234":

  ```
  fn.LENGTH({credit-card-number}) = 12
  ```

Sample conditional expressions

Here are some sample conditional expressions. NOTE: `$.` represents the entry point to the Amazon Lex V2 JSON response. The value following `$.` will be parsed within the Amazon Lex V2 response to retrieve the value. Conditional expressions using the JSON path reference to transcriptions block in the Amazon Lex V2 response will only be supported in the same locales which support ASR transcription scores.

Value type	Use case	Conditional expression
Custom slot	`pizzaSize` slot value is equal to large	`{pizzaSize} = "large"`

Value type	Use case	Conditional expression
Custom slot	pizzaSize is equal to large or medium	{pizzaSize} = "large" OR {pizzaSize} = "medium"
Custom slot	Expressions with ( ) and AND/OR	{pizzaType} = "pepperoni" OR {pizzaSize} = "medium" OR {pizzaSize} = "small"
Custom slot (Multi-Valued Slot)	Check if one of the topping is Onion	{toppings} CONTAINS "Onion"
Custom slot (Multi-Valued Slot)	Number of toppings are more than 3	fn.COUNT({topping}) > 2
AMAZON.AlphaNumeric	bookingID is ABC123	{bookingID} = "ABC123"
AMAZON.Number	age slot value is greater than 30	{age} > 30
AMAZON.Number	age slot value is equal to 10	{age} = 10
AMAZON.Date	dateOfBirth slot value before 1990	{dateOfBirth} < "1990-10-01"
AMAZON.State	destinationState slot value is equal to Washington	{destinationState} = "washington"
AMAZON.Country	destinationCountry slot value is not United States	{destinationCountry} != "united states"
AMAZON.FirstName	firstName slot value is John	{firstName} = "John"

Value type	Use case	Conditional expression
AMAZON.PhoneNumber	phoneNumber slot value is 716767891932	{phoneNumer} = 716767891932
AMAZON.Percentage	Check if percentage slot value is greater than or equals 78	{percentage} >= 78
AMAZON.EmailAddress	emailAddress slot value is userA@hmail.com	{emailAddress} = "userA@hmail.com"
AMAZON.LastName	lastName slot value is Doe	{lastName} = "Doe"
AMAZON.City	City slot value is equal to Seattle	{city} = "Seattle"
AMAZON.Time	Time is after 8 PM	{time} > "20:00"
AMAZON.StreetName	streetName slot value is Boren Avenue	{streetName} = "boren avenue"
AMAZON.Duration	travelDuration slot value is less than 2 hours	{travelDuration} < P2H
Input mode	Input mode is speech	$.inputMode = "Speech"
Input transcript	Input transcript is equal to "I want a large pizza"	$.inputTranscript = "I want a large pizza"
Session attribute	check customer_subscript ion_type attribute	[customer_subcript ion_type] = "yearly"
Request attribute	check retry_enabled flag	((retry_enabled)) = "TRUE"

Value type	Use case	Conditional expression
Kendra response	Kendra response contains FAQ	`fn.IS_SET(((x-amz-lex:kendra-search-response-question_answer-question-1) ))`
Conditional expression with transcriptions	Conditional expressions using transcriptions JSON path	`$.transcriptions[0].transcriptionConfidence < 0.8 AND $.transcriptions[1].transcriptionConfidence > 0.5`
Set session attributes	Set session attributes using transcriptions JSON path and slot values	`[sessionAttribute] = "$.transcriptions...." AND [sessionAttribute] = "{<slotName>}"`
Set slot values	Set slot values using session attributes and transcriptions JSON path	`{slotName} = [<sessionAttribute>] AND {slotName} = "$.transcriptions...."`

Note

`slotName` refers to the name of a slot in the Amazon Lex V2 bot. If the slot is not resolved (null), or if the slot does not exist, then the assignments are ignored at runtime. `sessionAttribute` refers to the name of the session attribute that is set by the customer at build time.

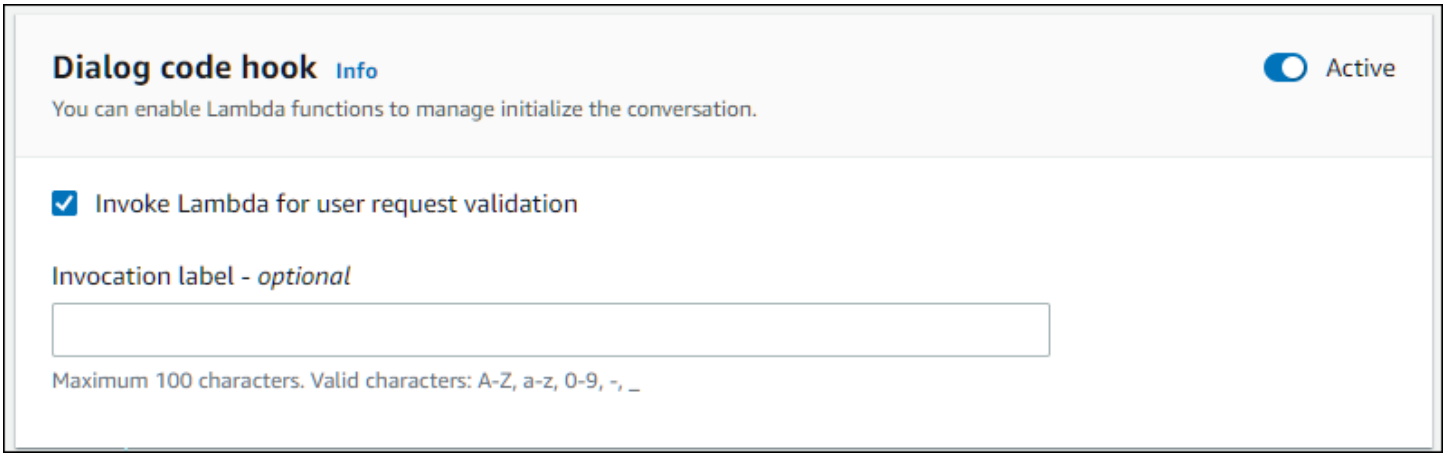
Invoke dialog code hook

At each step in the conversation when Amazon Lex V2 sends a message to the user, you can use a Lambda function as the next step in the conversation. You can use the function to implement business logic based on current state of the conversation.

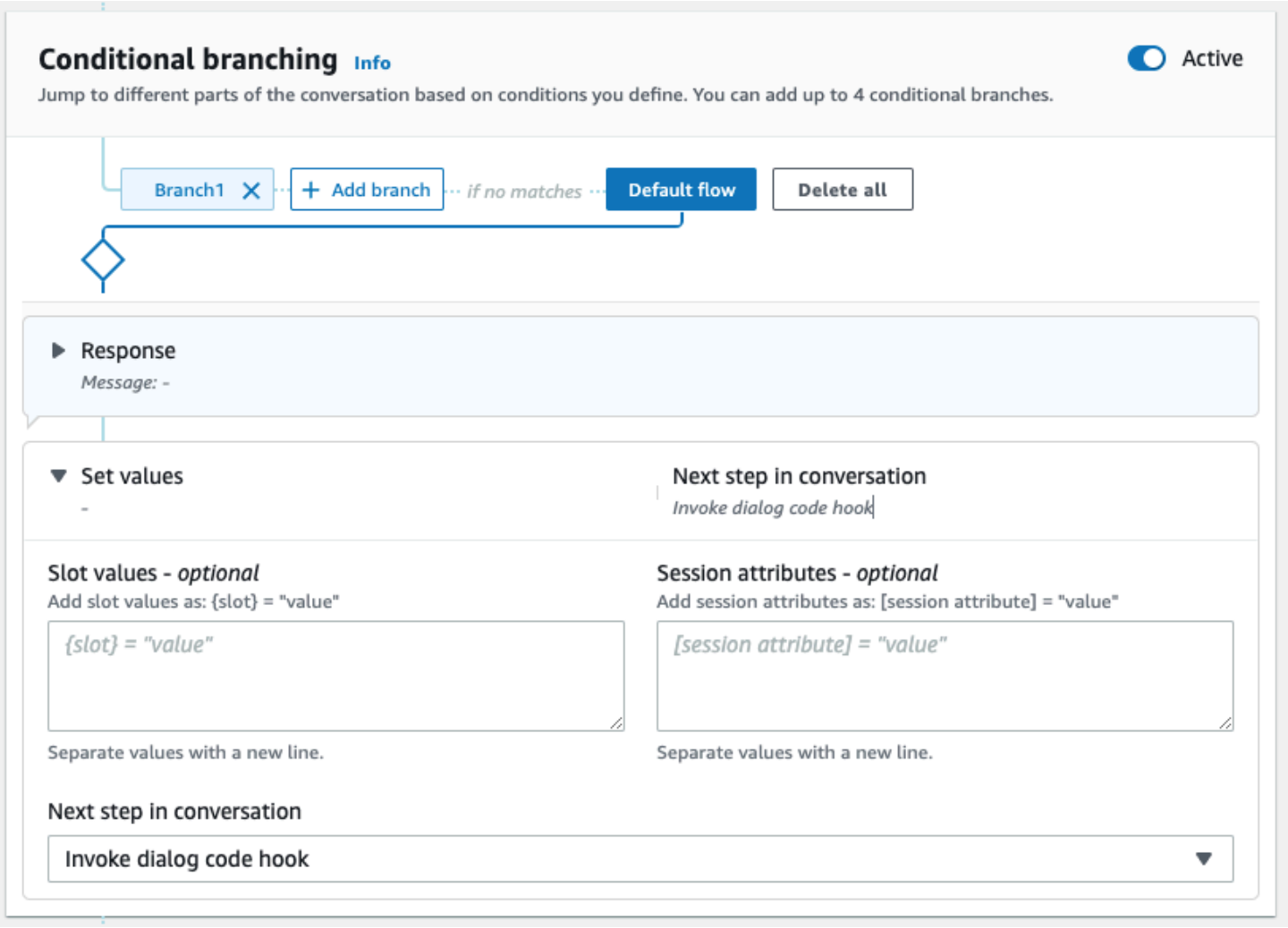
The Lambda function that runs is associated with the bot alias that you are using. To invoke Lambda function across all dialog code hooks in your intent, you must select **Use a Lambda function for initializing and validation** for the intent. For more information on choosing a Lambda function, see Creating an AWS Lambda function for your Amazon Lex V2 bot.

There are two steps to using a Lambda function. First, you must activate the dialog code hook at any point in the conversation. Second, you must set the next step in the conversation to use the dialog code hook.

The following image shows the dialog code hook activated.

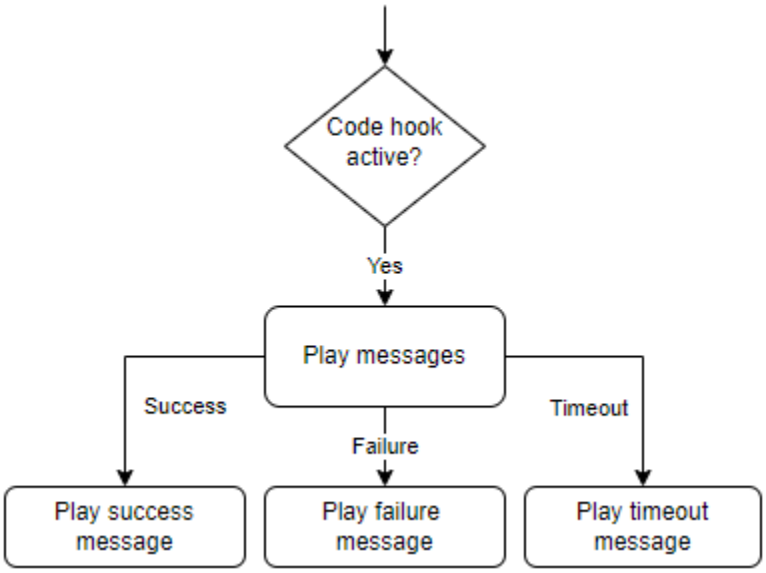


Next, set the code hook as the next action for the conversation step. You can do this by configuring the next step in conversation to Invoke dialog code hook. The following image shows a conditional branch where invoking the dialog code hook is the next step for the default path of the conversation.



When code hooks are active, you can set three responses to return to the user:

- **Success** – Sent when the Lambda function completed successfully.
- **Failure** – Sent if there was a problem with running the Lambda function, or the Lambda function returned an `intent.state` value of `Failed`.
- **Timeout** – Sent if the Lambda function did not complete in its configured timeout period.



Choose **Lambda dialog code hook** and then choose **Advanced options** to see the three options for responses that correspond to the Lambda function invocation. You can set values, configure the next steps, and apply conditions corresponding to each response to design the conversation flow. In the absence of a condition or an explicit next step, Amazon Lex V2 decides the next step based on the current state of the conversation.

On the **Advanced options** page you can also choose to enable or disable your Lambda function invocation. When the function is enabled, the dialog code hook is invoked with Lambda invocation, followed by the success, failure or timeout message based on Lambda invocation results. When the function is disabled, Amazon Lex V2 doesn't run the Lambda function and proceeds as if the dialog code hook is successful.

You can also set an invocation label that is sent to the Lambda function when it is invoked by this message. You can use this to help identify the section of your Lambda function to run.

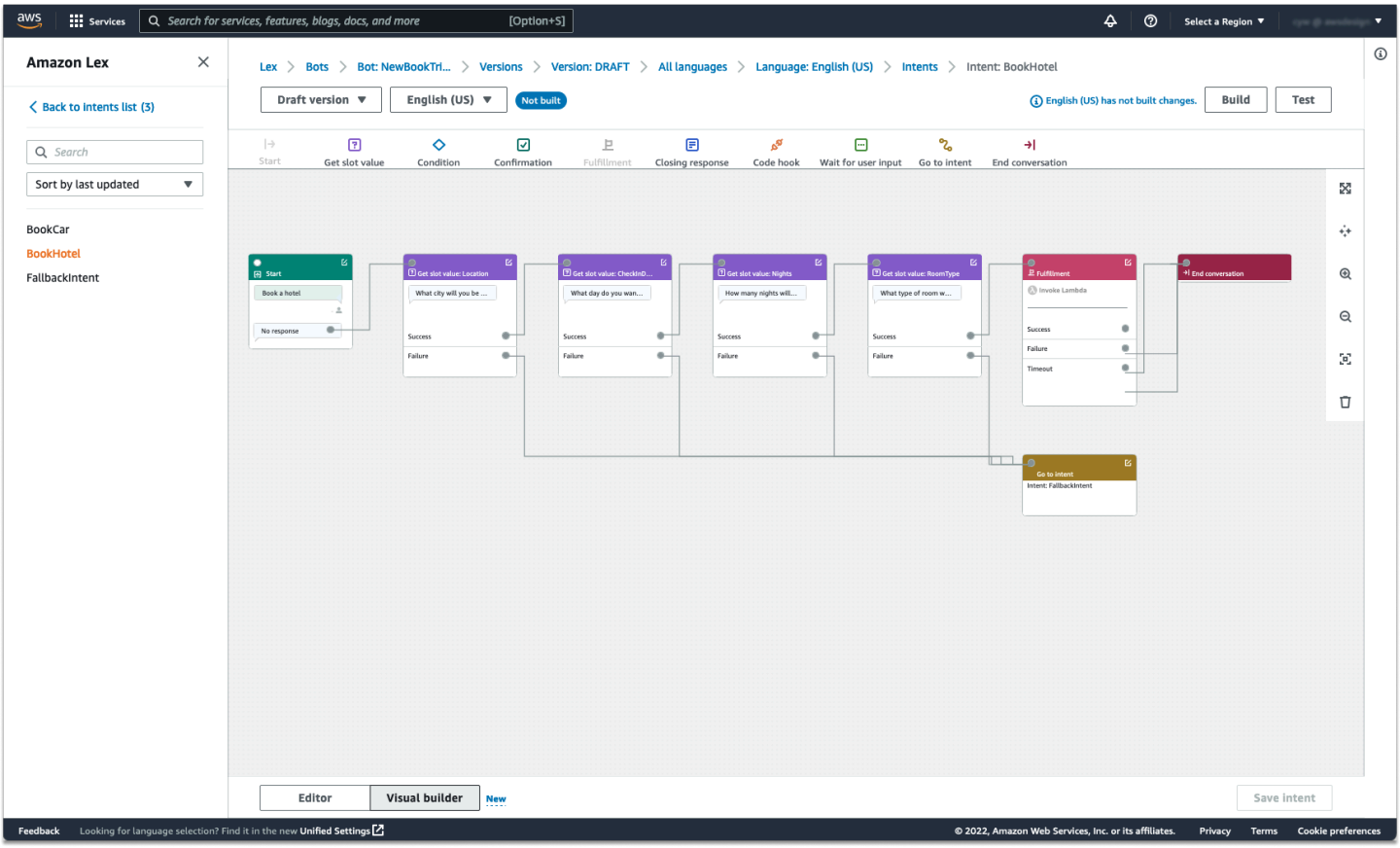
> **Note**
>
> On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Using Visual conversation builder

Visual conversation builder is a drag and drop conversation builder to easily design and visualize conversation paths by using intents within a rich visual environment.

To access the visual conversation builder

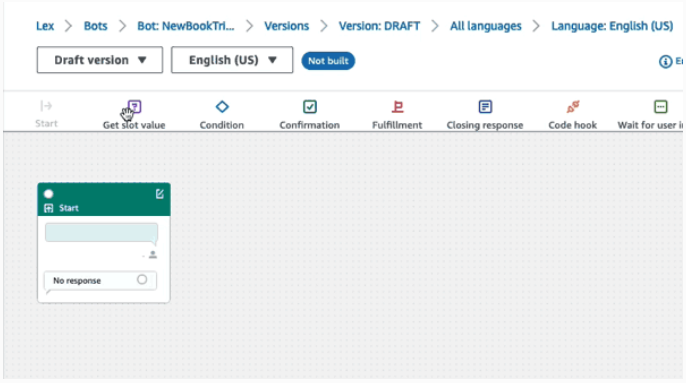
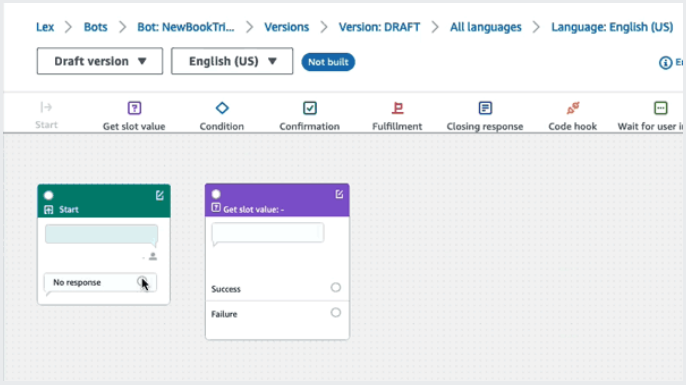
1. In the Amazon Lex V2 console, choose a bot and select **Intents** from the left navigation pane.
2. Go to the intent editor in one of the following ways:
   - Select **Add intent** at the top-right corner of the **Intents** section, and then choose to add either an empty intent or a built-in intent.
   - Choose the name of an intent from the **Intents** section.
3. In the intent editor, select **Visual builder** in the pane at the bottom of the screen to access the Visual conversation builder.
4. To return to the menu intent editor interface, select **Editor**.

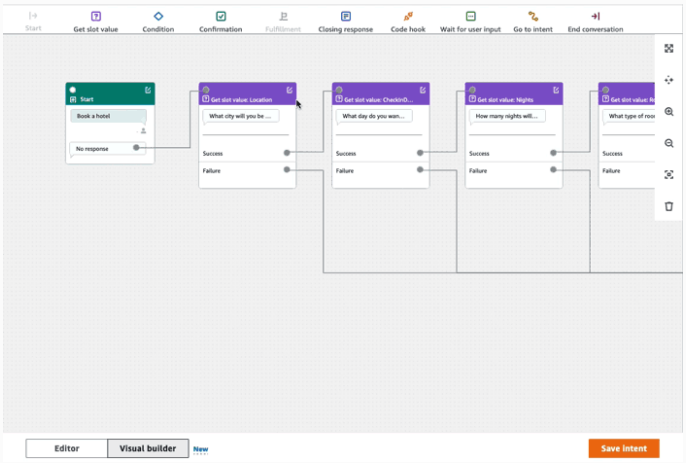
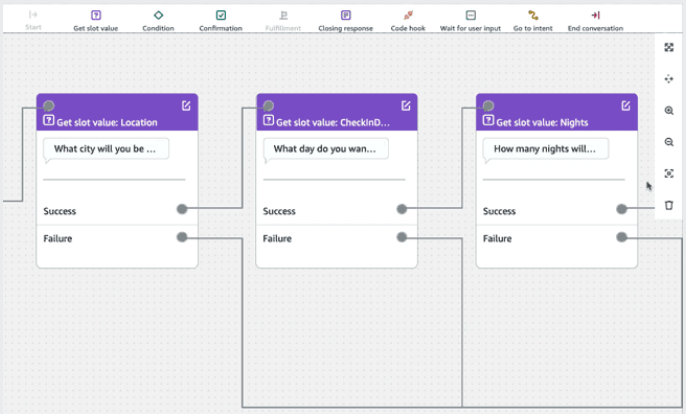
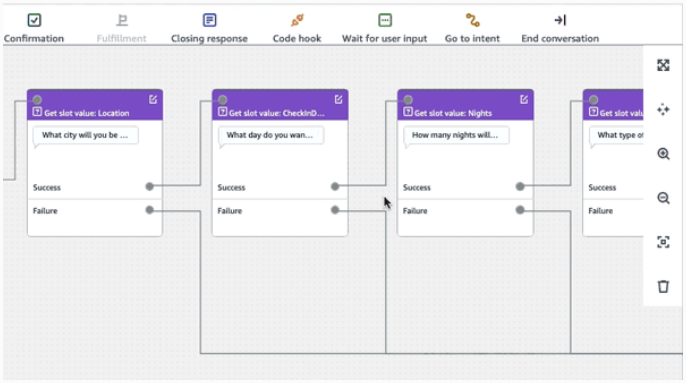


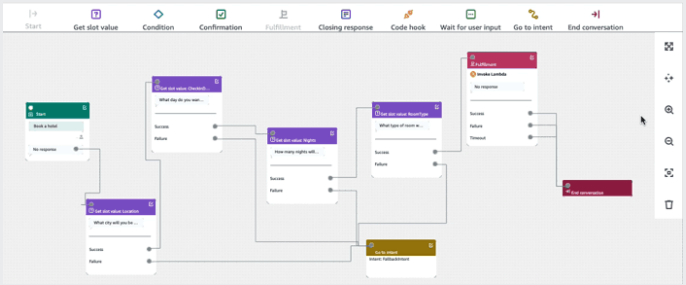
Visual conversation builder offers a more intuitive user interface with the ability to visualize and modify the conversation flow. By dragging and dropping the blocks, you can extend an existing flow or reorder the conversation steps. You can develop conversation flow with complex branching without writing any Lambda code.

This change helps to decouple the conversation flow design from other business logic in Lambda. Visual conversation builder can be used in conjunction with the existing intent editor and can be used to build conversation flows. However, it is recommended to use the visual editor view for more complex conversation flows.

When you save an intent, Amazon Lex V2 can auto-connect intents when it determines that there are missed connections, Amazon Lex V2 suggests a connection, or you can select your own connection for the block.

Action	Example
Adding a block to the workspace	
Making a connection between blocks	

Action	Example
Opening the configuration panel on a block	
Zoom to fit	
Delete a block from the conversation flow	

Action	Example
Auto clean the workspace	

Terminology:

Block – The basic building unit of a conversation flow. Each block has a specific functionality to handle different use cases of a conversation.

Port – Each block contains ports, which can be used to connect one block to another. Blocks can contain input ports and output ports. Each output port represents a particular functional variation of a block (such as errors, timeouts, or success).

Edge – An edge is a connection between the output port of one block to the input port of another block. It is a part of a branch in a conversation flow.

Conversation flow – A set of blocks connected by edges that describes intent level interactions with a customer.

Blocks

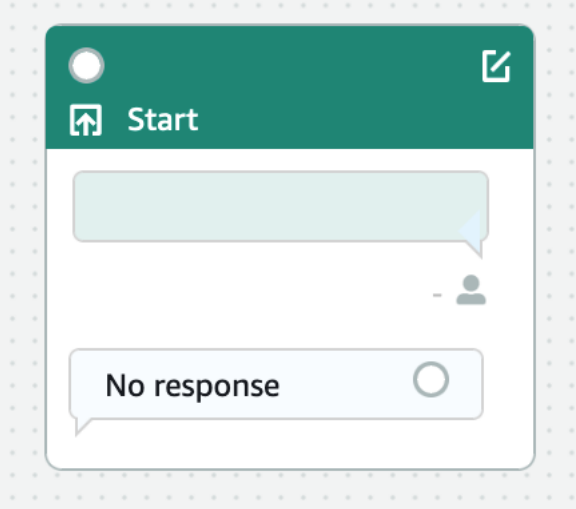
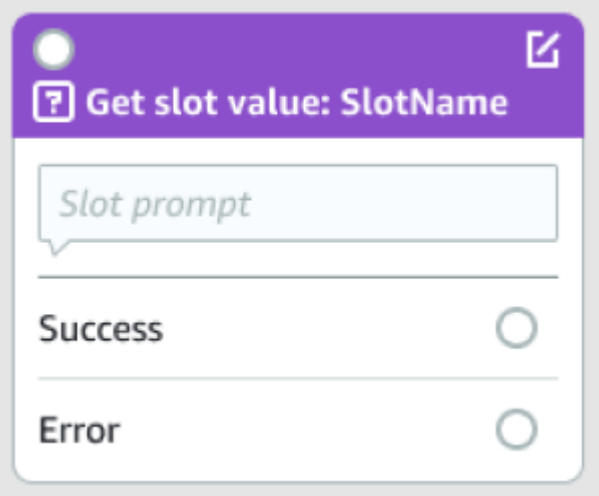
Blocks are the building blocks of a conversation flow design. They represent different states within the intent, that spans from the start of the intent, to user input, to the closing.

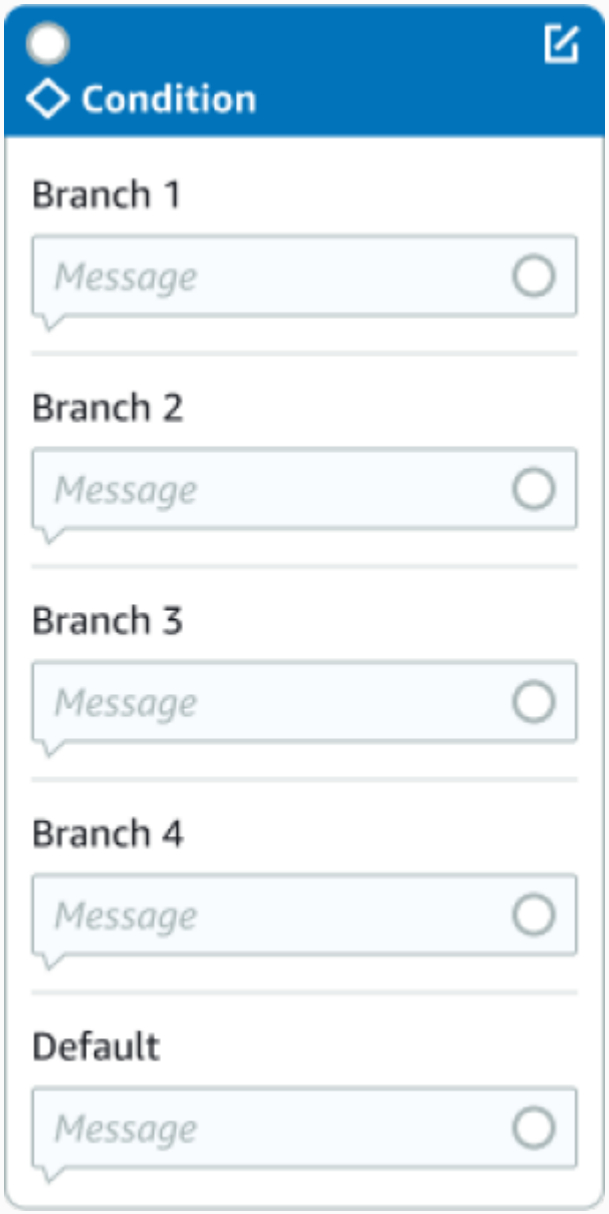
Each block has an entry point and one or many exit points based on the block type. Each exit point can be configured with a corresponding message as the conversation proceeds through the exit points. For blocks with multiple exit points, exit points relate to the status corresponding to the node. For a condition node, the exit points represent the different conditions.

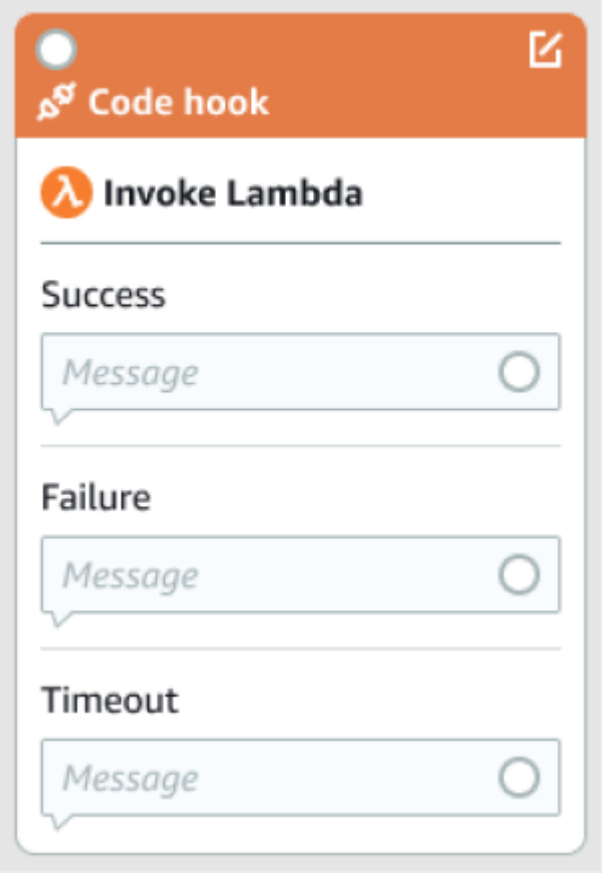
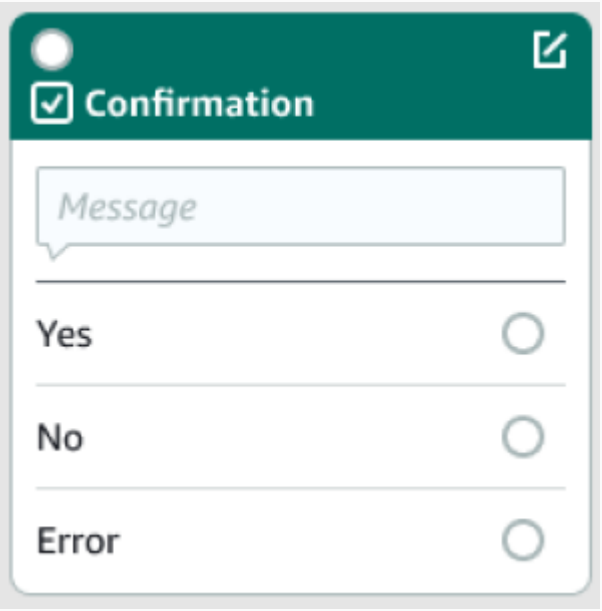
Each block has a configuration panel, which opens by clicking on the **Edit** icon on the top right corner of the block. The configuration panel contains detailed fields that can be configured to correspond with each block.

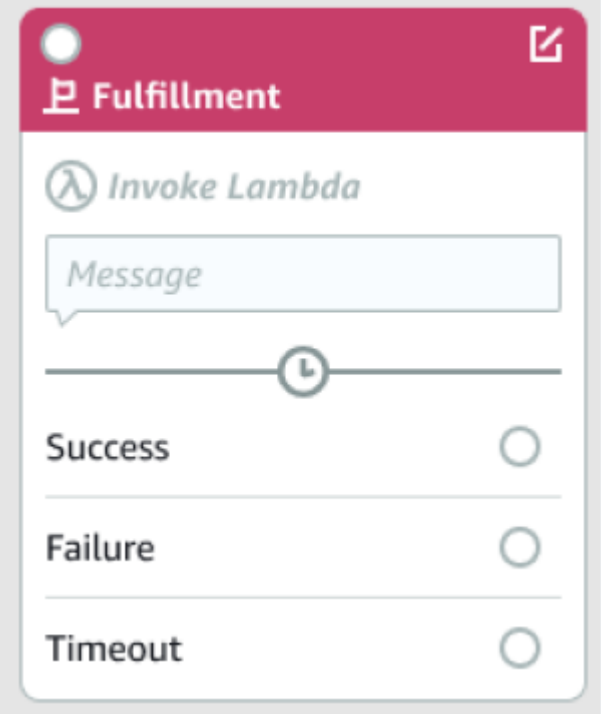
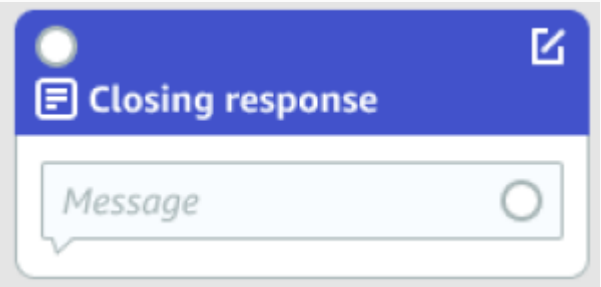
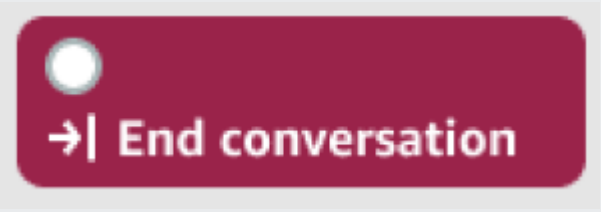
The bot prompts and messages can be configured directly on the node by dragging a new block, or they can be modified within the right panel, along with other attributes of the block.

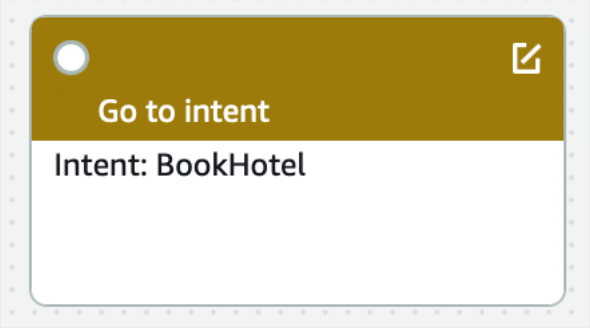
Block types – Here are the block types that you can use with visual conversation builder.

Block Type	Block
Start – The root or first block of the conversation flow. This block can also be configured such that the bot can send an initial response (message the intent has been recognized). For more information, see Initial response.	
Get slot value – This block tries to elicit value for a single slot. This block has a setting to wait for customer response to the slot elicitation prompt. For more information, see Slots.	

Block Type	Block
Condition – This block contains condition als. It contains up to 4 custom branches (with conditions) and one default branch. For more information, see Add conditions to branch conversations.	

Block Type	Block
Dialog code hook – This block handles invocation of the dialog Lambda function. This block contains bot responses based on dialog Lambda function succeeding, failing, or timing out. For more information, see Invoke dialog code hook.	
Confirmation – This block queries the customer before fulfillment of the intent. It contains bot responses based on customer saying yes or no to the confirmation prompt. For more information, see Confirmation.	

Block Type	Block
Fulfillment – This block handles fulfillment of intent, usually after slots elicitation. It can be configured to invoke Lambda functions, as well as respond with messages, if fulfillment succeeds or fails. For more information, see Fulfillment.	
Closing response – This block allows the bot to respond with a message before ending the conversation. For more information, see Closing response.	
End conversation – This block indicates the end of the conversation flow.	
Wait for user input – This block can be used to capture input from the customer and switch to another intent based on the utterance.	

Block Type	Block
Go to intent – This block can be used to go to a new intent, or to directly elicit a specific slot of that intent.	

Port types

All blocks contain one input port, which is used to connect its parent blocks. The conversation can only flow to a particular block's input port from its parent block's output port. However, blocks can contain zero, one, or many output ports. The blocks without any output ports signify the end of the conversation flow in the current intent (`GoToIntent`, `EndConversation`,`WaitForUserInput`).

Rules of intent design:

- All flows in an intent begin with the start block.
- Messages corresponding to each exit point are optional.
- You can configure the blocks to set values corresponding to each exit point in the configuration panel.
- Only a single start, confirmation, fulfillment and closing blocks can exist in a single flow within an intent. Multiple conditions, dialog code hook, get slot values, end conversation, transfer, and wait for user input blocks may exist.
- A condition block cannot have a direct connection to a condition block. The same applies for dialog code hook.
- Circular flows are allowed three blocks, but an incoming connector to Start Intent is not allowed.
- An optional slot doesn't have an incoming connector or an outgoing connection and is primarily used to capture any data present during intent elicitation. Every other slot that is part of the conversation path must be a mandatory slot.

Blocks:

- The start block must have an outgoing edge.
- Every get slot value block must have an outgoing edge from the success port, if the slot is required.
- Every condition block must have an outgoing edge from each branch if the block is active.
- A condition block cannot have more than one parent.
- An active condition block must have an incoming edge.
- Every active code hook block must have an outgoing edge from each port: success, failure, and timeout.
- An active code hook block must have an incoming edge.
- An active confirmation block must have an incoming edge.
- An active fulfillment block must have an incoming edge.
- An active closing block must have an incoming edge.
- A condition block must have at least one non-default branch.
- A go to intent block must have an intent specified.

Edges:

- A condition block cannot be connected to another condition block.
- A code hook block cannot be connected to another code hook block.
- A condition block can only be connected to zero or one code hook block.
- The connection (code hook -> condition -> code hook) is not valid.
- A fulfillment block cannot have a code hook block as a child.
- A condition block, which is a child of the fulfillment block, cannot have a code hook block child.
- A closing block cannot have a code hook block as a child.
- A condition block that is a child of the closing block cannot have a code hook block child.
- A start, confirmation, or get slot value block can have no more than one code hook block in its dependency chain.

Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user

takes through the conversation. For more information, see Changes to conversation flows in Amazon Lex V2. Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

Built-in intents

For common actions, you can use the standard built-in intents library. To create an intent from a built-in intent, choose a built-intent in the console, and give it a new name. The new intent has the configuration of the base intent, such as the sample utterances.

In the current implementation, you can't do the following:

- Add or remove sample utterances from the base intent
- Configure slots for built-in intents

To add a built-in intent to a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot to add the built-in intent to.
3. In the left menu, choose the language and then choose **Intents**.
4. Choose **Add intent**, and then choose **Use built-in intent**.
5. In **Built-in intent**, choose the intent to use.
6. Give the intent a name, and then choose **Add**.
7. Use the intent editor to configure the intent as required for your bot.

Topics

- AMAZON.BedrockAgentIntent
- AMAZON.CancelIntent
- AMAZON.FallbackIntent
- AMAZON.HelpIntent
- AMAZON.KendraSearchIntent
- AMAZON.PauseIntent

- AMAZON.QnAIntent
- AMAZON.QnAIntent (multiple use support)
- AMAZON.QinConnectIntent
- AMAZON.RepeatIntent
- AMAZON.ResumeIntent
- AMAZON.StartOverIntent
- AMAZON.StopIntent

AMAZON.BedrockAgentIntent

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
> 2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

Activates Amazon Bedrock Agents that are defined in the intent to respond to customer requests and activate agentic workflows to achieve the task defined. This feature is available in all Amazon Lex V2 supported locales and all commercial regions where both Amazon Lex V2 and Amazon Bedrock Agents are present.

If this intent is overriding `FallbackIntent`, the intent is activated when an utterance is not classified into any of the other intents present in the bot, otherwise it is only activated when an utterance is classified into this intent. It's important to note that this intent will not be activated for missed utterances when eliciting a slot value.

Once recognized by your Amazon Lex V2 bot, the `AMAZON.BedrockAgentIntent`, activates the defined `BedrockAgent` or `BedrockKnowledgeBase` to respond to the customer. If you are using Amazon Bedrock Agents, the conversation stays in the `BedrockAgentIntent` and the user requests are relayed to Agents, until the Amazon Bedrock Agent determines that the conversation is marked `FINISH`. Only after that, Amazon Lex V2 assumes control of the conversation and adheres to next steps defined in the `AMAZON.BedrockAgentIntent`.

Responds to customer questions by using an Amazon Bedrock Agents and Knowledge Bases to answer customer questions and provide detailed responses.

> **Warning**
>
> You can't use the `AMAZON.BedrockAgentIntent` without sample utterances, `AMAZON.QnAIntent` without sample utterances and the `AMAZON.KendraSearchIntent` in the same bot locale.

If you select this intent, you configure the following fields and then select Add to add the intent.

- Amazon Bedrock Agent Id - The identifier of the Amazon Bedrock Agent. Choose the Bedrock Agent that you want to use.
- Amazon Bedrock Agent Alias Id - The Alias identifier of the Amazon Bedrock Agent.

> **Important**
>
> When creating the Amazon Bedrock Agent to be used with Amazon Lex V2, verify that **User Input** under **Additional Settings** is `ENABLED`. This setting is critical to allow Agents to ask clarifying or follow-up questions, and it allows Amazon Lex V2 to delegate back to Agents to complete the corresponding task.

(Optional) You can also add a BedrockAgentIntent with these options:

- Amazon Bedrock model - Choose the provider and foundation model to use for this intent. Currently, some Anthropic Claude models are supported.
- Amazon Bedrock Knowledge Base - If you choose this option, specify the ID of the Amazon Bedrock Knowledge Base. You can find the ID by checking the details page of the Amazon Bedrock Knowledge Base in the console, or by sending a `GetKnowledgeBase` request.

The responses from the BedrockAgentIntent will be stored into the session and request attributes as shown below:

- `x-amz-lex:bedrock-agent-search-response` – The response from the Amazon Bedrock Agent to the question or utterance.

- `x-amz-lex:bedrock-knowledge-base-search-response-source` – Points to the document, or list of documents, used to generate the response if using the Amazon Bedrock Knowledge Base configuration.
- `x-amz-lex:bedrock-agent-action-group-invocation-input` - Object containing input values collected by the agent action group. For more information on agent action groups, see ActionGroupInvocationInput.
- `x-amz-lex:bedrock-agent-knowledge-base-lookup-input` - Object containing the Amazon Bedrock Knowledge Base lookup related details.
- `x-amz-lex:bedrock-agent-agent-collaborator-details` – Object containing details of the input and output from the sub agents that were invoked as part of Multi-agent collaboration invocations.

For more information, see Using BedrockAgentIntent to use a Bedrock Agent in Amazon Lex.

AMAZON.CancelIntent

Responds to words and phrases that indicate the user wants to cancel the current interaction. Your application can use this intent to remove slot type values and other attributes before ending the interaction with the user.

Common utterances:

- cancel
- never mind
- forget it

AMAZON.FallbackIntent

When a user's input to an intent isn't what a bot expects, you can configure Amazon Lex V2 to invoke a *fallback intent*. For example, if the user input "I'd like to order candy" doesn't match an intent in your `OrderFlowers` bot, Amazon Lex V2 invokes the fallback intent to handle the response.

The built-in `AMAZON.FallbackIntent` intent type is added to your bot automatically when you create a bot using the console or when you add a locale to a bot using the CreateBotLocale operation.

Invoking a fallback intent uses two steps. In the first step the fallback intent is matched based on the input from the user. When the fallback intent is matched, the way the bot behaves depends on the number of retries configured for a prompt.

Amazon Lex V2 matches the fallback intent in these situations:

- The user's input to an intent doesn't match the input that the bot expects
- Audio input is noise, or text input isn't recognized as words.
- The user's input is ambiguous and Amazon Lex V2 can't determine which intent to invoke.

The fallback intent is invoked when:

- An intent doesn't recognize the user input as a slot value after the configured number of tries.
- An intent doesn't recognize the user input as a response to a confirmation prompt after the configured number of tries.

You can't add the following to a fallback intent:

- Utterances
- Slots
- A confirmation prompt

Using a Lambda Function with a Fallback Intent

When a fallback intent is invoked, the response depends on the setting of the `fulfillmentCodeHook` parameter to the CreateIntent operation. The bot does one of the following:

- Returns the intent information to the client application.
- Calls the alias's validation and fulfillment Lambda function. It calls the function with the session variables that are set for the session.

For more information about setting the response when a fallback intent is invoked, see the `fulfillmentCodeHook` parameter of the CreateIntent operation.

If you use the Lambda function with your fallback intent, you can use this function to call another intent or to perform some form of communication with the user, such as collecting a callback number or opening a session with a customer service representative.

A fallback intent can be invoked multiple times in the same session. For example, suppose that your Lambda function uses the `ElicitIntent` dialog action to prompt the user for a different intent. If Amazon Lex V2 can't infer the user's intent after the configured number of tries, it invokes the fallback intent again. It also invokes the fallback intent when the user doesn't respond with a valid slot value after the configured number of tries.

You can configure your Lambda function to keep track of the number of times that the fallback intent is called using a session variable. Your Lambda function can take a different action if it is called more times than the threshold that you set in your Lambda function. For more information about session variables, see Setting session attributes for your Lex V2 bot.

AMAZON.HelpIntent

Responds to words or phrases that indicate the user needs help while interacting with your bot. When this intent is invoked, you can configure your Lambda function or application to provide information about the your bot's capabilities, ask follow up questions about areas of help, or hand the interaction over to a human agent.

Common utterances:

- help
- help me
- can you help me

AMAZON.KendraSearchIntent

To search documents that you have indexed with Amazon Kendra, use the `AMAZON.KendraSearchIntent` intent. When Amazon Lex V2 can't determine the next action in a conversation with the user, it triggers the search intent.

The `AMAZON.KendraSearchIntent` is available only in the English (US) (en-US) locale and in the US East (N. Virginia), US West (Oregon) and Europe (Ireland) Regions.

Amazon Kendra is a machine-learning-based search service that indexes natural language documents such as PDF documents or Microsoft Word files. It can search indexed documents and return the following types of responses to a question:

- An answer
- An entry from a FAQ that might answer the question
- A document that is related to the question

For an example of using the `AMAZON.KendraSearchIntent`, see Example: Creating a FAQ Bot for an Amazon Kendra Index.

If you configure an `AMAZON.KendraSearchIntent` intent for your bot, Amazon Lex V2 calls the intent whenever it can't determine the user utterance for an intent. If there is no response from Amazon Kendra, the conversation continues as configured in the bot.

> **Note**
>
> Amazon Lex V2 currently does not support the `AMAZON.KendraSearchIntent` during slot elicitation. If Amazon Lex V2 can't determine the user utterance for a slot, it calls the `AMAZON.FallbackIntent`.

When you use the `AMAZON.KendraSearchIntent` with the `AMAZON.FallbackIntent` in the same bot, Amazon Lex V2 uses the intents as follows:

1. Amazon Lex V2 calls the `AMAZON.KendraSearchIntent`. The intent calls the Amazon Kendra `Query` operation.
2. If Amazon Kendra returns a response, Amazon Lex V2 displays the result to the user.
3. If there is no response from Amazon Kendra, Amazon Lex V2 re-prompts the user. The next action depends on response from the user.
   - If the response from the user contains an utterance that Amazon Lex V2 recognizes, such as filling a slot value or confirming an intent, the conversation with the user proceeds as configured for the bot.
   - If the response from the user does not contain an utterance that Amazon Lex V2 recognizes, Amazon Lex V2 makes another call to the `Query` operation.

4. If there is no response after the configured number of retries, Amazon Lex V2 calls the `AMAZON.FallbackIntent` and ends the conversation with the user.

There are three ways to use the `AMAZON.KendraSearchIntent` to make a request to Amazon Kendra:

- Let the search intent make the request for you. Amazon Lex V2 calls Amazon Kendra with the user's utterance as the search string. When you create the intent, you can define a query filter string that limits the number of responses that Amazon Kendra returns. Amazon Lex V2 uses the filter in the query request.
- Add additional query parameters to the request to narrow the search results using your Lambda function. You add a `kendraQueryFilterString` field that contains Amazon Kendra query parameters to the `delegate` dialog action. When you add query parameters to the request with the Lambda function, they take precedence over the query filter that you defined when you created the intent.
- Create a new query using the Lambda function. You can create a complete Amazon Kendra query request that Amazon Lex V2 sends. You specify the query in the `kendraQueryRequestPayload` field in the `delegate` dialog action. The `kendraQueryRequestPayload` field takes precedence over the `kendraQueryFilterString` field.

To specify the `queryFilterString` parameter when you create a bot, or to specify the `kendraQueryFilterString` field when you call the `delegate` action in a dialog Lambda function, you specify a string that is used as the attribute filter for the Amazon Kendra query. If the string isn't a valid attribute filter, you'll get an `InvalidBotConfigException` exception at runtime. For more information about attribute filters, see Using document attributes to filter queries in the *Amazon Kendra Developer Guide*.

To have control over the query that Amazon Lex V2 sends to Amazon Kendra, you can specify a query in the `kendraQueryRequestPayload`field in your Lambda function. If the query isn't valid, Amazon Lex V2 returns an `InvalidLambdaResponseException` exception. For more information, see the Query operation in the *Amazon Kendra Developer Guide*.

For an example of how to use the `AMAZON.KendraSearchIntent`, see Example: Creating a FAQ Bot for an Amazon Kendra Index.

IAM Policy for Amazon Kendra Search

To use the `AMAZON.KendraSearchIntent` intent, you must use a role that provides AWS Identity and Access Management (IAM) policies that enable Amazon Lex V2 to assume a runtime role that has permission to call the Amazon Kendra `Query` intent. The IAM settings that you use depend on whether you create the `AMAZON.KendraSearchIntent` using the Amazon Lex V2 console, or using an AWS SDK or the AWS Command Line Interface (AWS CLI). When you use the console, you can choose between adding permission to call Amazon Kendra to the Amazon Lex V2 service-linked role or using a role specifically for calling the Amazon Kendra `Query` operation. When you use the AWS CLI or an SDK to create the intent, you must use a role specifically for calling the `Query` operation.

Attaching Permissions

You can use the console to attach permissions to access the Amazon Kendra `Query` operation to the default Amazon Lex V2 service-linked role. When you attach permissions to the service-linked role, you don't have to create and manage a runtime role specifically to connect to the Amazon Kendra index.

The user, role, or group that you use to access the Amazon Lex V2 console must have permissions to manage role policies. Attach the following IAM policy to the console access role. When you grant these permissions, the role has permissions to change the existing service-linked role policy.

JSON

```
{
"Version":"2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "iam:AttachRolePolicy",
            "iam:PutRolePolicy",
            "iam:GetRolePolicy"
        ],
        "Resource": "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexBots*"
    },
    {
        "Effect": "Allow",
```

```
        "Action": "iam:ListRoles",
        "Resource": "*"
    }
]
}
```

Specifying a Role

You can use the console, the AWS CLI, or the API to specify a runtime role to use when calling the Amazon Kendra `Query` operation.

The user, role, or group that you use to specify the runtime role must have the `iam:PassRole` permission. The following policy defines the permission. You can use the `iam:AssociatedResourceArn` and `iam:PassedToService` condition context keys to further limit the scope of the permissions. For more information, see IAM and AWS STS Condition Context Keys in the *AWS Identity and Access Management User Guide*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::111122223333:role/role"
        }
    ]
}
```

The runtime role that Amazon Lex V2 needs to use to call Amazon Kendra must have the `kendra:Query` permissions. When you use an existing IAM role for permission to call the Amazon Kendra `Query` operation, the role must have the following policy attached.

You can use the IAM console, the IAM API, or the AWS CLI to create a policy and attach it to a role. These instructions use the AWS CLI to create the role and policies.

> **Note**
>
> The following code is formatted for Linux and MacOS. For Windows, replace the Linux line continuation character (\) with a caret (^).

To add Query operation permission to a role

1. Create a document called **`KendraQueryPolicy.json`** in the current directory, add the following code to it, and save it
2. In the AWS CLI, run the following command to create the IAM policy for running the Amazon Kendra `Query` operation.

```
aws iam create-policy \
--policy-name query-policy-name \
--policy-document file://KendraQueryPolicy.json
```

3. Attach the policy to the IAM role that you are using to call the `Query` operation.

```
aws iam attach-role-policy \
--policy-arn arn:aws:iam::account-id:policy/query-policy-name
--role-name role-name
```

You can choose to update the Amazon Lex V2 service-linked role or to use a role that you created when you create the `AMAZON.KendraSearchIntent` for your bot. The following procedure shows how to choose the IAM role to use.

To specify the runtime role for AMAZON.KendraSearchIntent

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot that you want to add the `AMAZON.KendraSearchIntent` to.
3. Choose the plus (+) next to **Intents**.
4. In **Add intent**, choose **Search existing intents**.
5. In **Search intents**, enter **`AMAZON.KendraSearchIntent`** and then choose **Add**.
6. In **Copy built-in intent**, enter a name for the intent, such as **`KendraSearchIntent`**, and then choose **Add**.

7. Open the **Amazon Kendra query** section.
8. For **IAM role** choose one of the following options:

   - To update the Amazon Lex V2 service-linked role to enable your bot to query Amazon Kendra indexes, choose **Add Amazon Kendra permissions**.
   - To use a role that has permission to call the Amazon Kendra `Query` operation, choose **Use an existing role**.

Using Request and Session Attributes as Filters

To filter the response from Amazon Kendra to items related to current conversation, use session and request attributes as filters by adding the `queryFilterString` parameter when you create your bot. You specify a placeholder for the attribute when you create the intent, and then Amazon Lex V2 substitutes a value before it calls Amazon Kendra. For more information about request attributes, see Setting request attributes for your Lex V2 bot. For more information about session attributes, see Setting session attributes for your Lex V2 bot.

The following is an example of a `queryFilterString` parameter that uses a string to filter the Amazon Kendra query.

```
"{"equalsTo": {"key": "City", "value": {"stringValue": "Seattle"}}}"
```

The following is an example of a `queryFilterString` parameter that uses a session attribute called `"SourceURI"` to filter the Amazon Kendra query.

```
"{"equalsTo": {"key": "SourceURI","value": {"stringValue": "[FileURL]"}}}"
```

The following is an example of a `queryFilterString` parameter that uses a request attribute called `"DepartmentName"` to filter the Amazon Kendra query.

```
"{"equalsTo": {"key": "Department","value": {"stringValue": "((DepartmentName))"}}}"
```

The `AMAZON.KendraSearchInteng` filters use the same format as the Amazon Kendra search filters. For more information, see Using document attributes to filter search results in the *Amazon Kendra developer guide*.

The query filter string used with the `AMAZON.KendraSearchIntent` must use lower-case letters for the first letter of each filter. For example, the following is a valid query filter for the `AMAZON.KendraSearchIntent`.

```
{
"andAllFilters": [
    {
        "equalsTo": {
            "key": "City",
            "value": {
                "stringValue": "Seattle"
            }
        }
    },
    {
        "equalsTo": {
            "key": "State",
            "value": {
                "stringValue": "Washington"
            }
        }
    }
]
}
```

Using the Search Response

Amazon Kendra returns the response to a search in a response from the intent's `IntentClosingSetting` statement. The intent must have a `closingResponse` statement unless a Lambda function produces a closing response message.

Amazon Kendra has five types of responses.

- The following two responses require an FAQ to be set up for your Amazon Kendra index. For more details, see Adding questions and answers directly to a index.
  - `x-amz-lex:kendra-search-response-question_answer-question-<N>` – The question from a FAQ that matches the search.
  - `x-amz-lex:kendra-search-response-question_answer-answer-<N>` – The answer from a FAQ that matches the search.
- The following three responses require a data source to be set up for your Amazon Kendra index. For more details, see Creating a data source.

- `x-amz-lex:kendra-search-response-document-<N>` – An excerpt from a document in the index that is related to the text of the utterance.
- `x-amz-lex:kendra-search-response-document-link-<N>` – The URL of a document in the index that is related to the text of the utterance.
- `x-amz-lex:kendra-search-response-answer-<N>` – An excerpt from a document in the index that answers the question.

The responses are returned in `request` attributes. There can be up to five responses for each attribute, numbered 1 through 5. For more information about responses, see Types of response in the *Amazon Kendra Developer Guide*.

The `closingResponse` statement must have one or more message groups. Each message group contains one or more messages. Each message can contain one or more placeholder variables that are replaced by request attributes in the response from Amazon Kendra. There must be at least one message in the message group where all of the variables in the message are replaced by request attribute values in the runtime response, or there must be a message in the group with no placeholder variables. The request attributes are set off with double parentheses ("((" "))"). The following message group messages match any response from Amazon Kendra:

- “I found a FAQ question for you: ((x-amz-lex:kendra-search-response-question_answer-question-1)), and the answer is ((x-amz-lex:kendra-search-response-question_answer-answer-1))”
- “I found an excerpt from a helpful document: ((x-amz-lex:kendra-search-response-document-1))”
- “I think the answer to your questions is ((x-amz-lex:kendra-search-response-answer-1))”

Using a Lambda Function to Manage the Request and Response

The `AMAZON.KendraSearchIntent` intent can use your dialog code hook and fulfillment code hook to manage the request to Amazon Kendra and the response. Use the dialog code hook Lambda function when you want to modify the query that you send to Amazon Kendra, and the fulfillment code hook Lambda function when you want to modify the response.

Creating a Query with the Dialog Code Hook

You can use the dialog code hook to create a query to send to Amazon Kendra. Using the dialog code hook is optional. If you don't specify a dialog code hook, Amazon Lex V2 constructs a query from the user utterance and uses the `queryFilterString` that you provided when you configured the intent, if you provided one.

You can use two fields in the dialog code hook response to modify the request to Amazon Kendra:

- `kendraQueryFilterString` – Use this string to specify attribute filters for the Amazon Kendra request. You can filter the query using any of the index fields defined in your index. For the structure of the filter string, see Using document attributes to filter queries in the *Amazon Kendra Developer Guide*. If the specified filter string isn't valid, you will get an `InvalidLambdaResponseException` exception. The `kendraQueryFilterString` string overrides any query string specified in the `queryFilterString` configured for the intent.
- `kendraQueryRequestPayload` – Use this string to specify an Amazon Kendra query. Your query can use any of the features of Amazon Kendra. If you don't specify a valid query, you get a `InvalidLambdaResponseException` exception. For more information, see Query in the *Amazon Kendra Developer Guide*.

After you have created the filter or query string, you send the response to Amazon Lex V2 with the `dialogAction` field of the response set to `delegate`. Amazon Lex V2 sends the query to Amazon Kendra and then returns the query response to the fulfillment code hook.

Using the Fulfillment Code Hook for the Response

After Amazon Lex V2 sends a query to Amazon Kendra, the query response is returned to the `AMAZON.KendraSearchIntent` fulfillment Lambda function. The input event to the code hook contains the complete response from Amazon Kendra. The query data is in the same structure as the one returned by the Amazon Kendra `Query` operation. For more information, see Query response syntax in the *Amazon Kendra Developer Guide*.

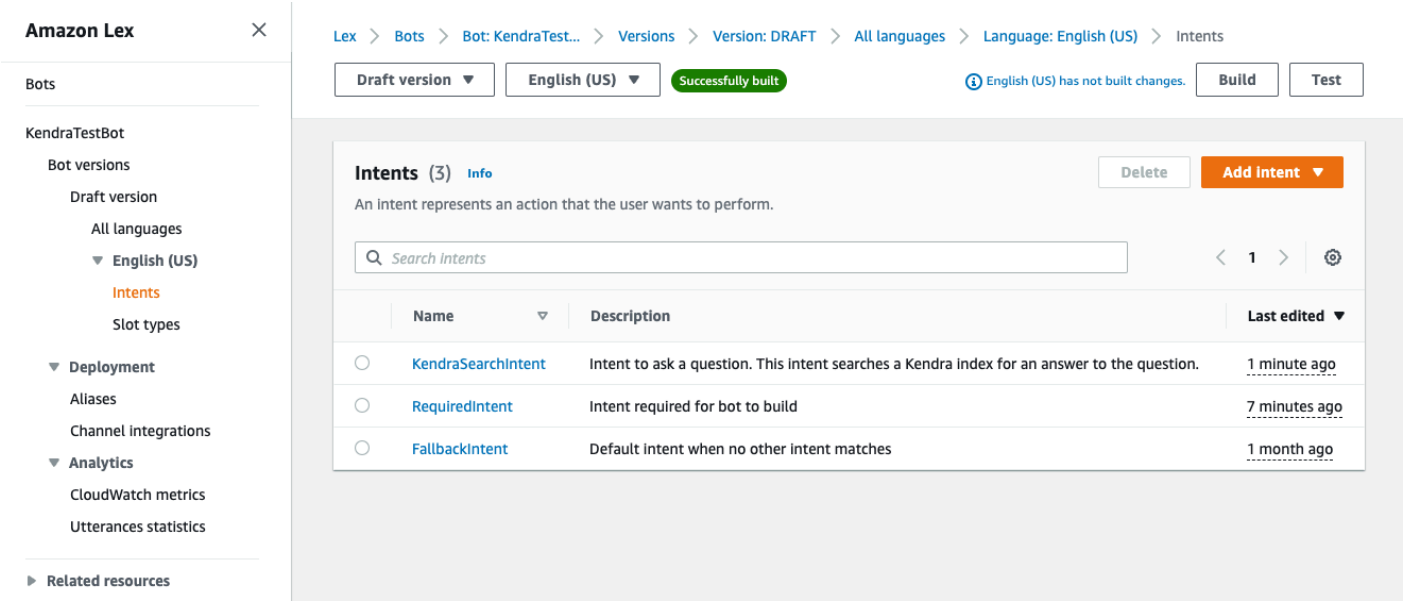
The fulfillment code hook is optional. If one does not exist, or if the code hook doesn't return a message in the response, Amazon Lex V2 uses the `closingResponse` statement for responses.

Example: Creating a FAQ Bot for an Amazon Kendra Index

This example creates an Amazon Lex V2 bot that uses an Amazon Kendra index to provide answers to users' questions. The FAQ bot manages the dialog for the user. It uses the `AMAZON.KendraSearchIntent` intent to query the index and to present the response to the user. Here is a summary of how you will create your FAQ bot using an Amazon Kendra index:

1. Create a bot that your customers will interact with to get answers from your bot.
2. Create a custom intent. Because the `AMAZON.KendraSearchIntent` and `AMAZON.FallbackIntent` are backup intents, your bot requires at least one other intent

that must contain least one utterance. This intent enables your bot to build, but is not used otherwise. Your FAQ bot will therefore contain at least three intents, as in the image below:



3. Add the `AMAZON.KendraSearchIntent` intent to your bot and configure it to work with your Amazon Kendra index.
4. Test the bot by making a query and verifying that the results from your Amazon Kendra index are documents that answer the query.

Prerequisites

Before you can use this example, you need to create an Amazon Kendra index. For more information, see Getting started with the Amazon Kendra console in the *Amazon Kendra Developer Guide*. For this example, choose the sample dataset (**Sample AWS documentation**) as your data source.

To create a FAQ bot:

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. In the navigation pane, choose **Bots**.
3. Choose **Create bot**.
    a. For the **Creation method**, choose **Create a blank bot**.
    b. In the **Bot configuration** section, give the bot a name that indicates its purpose, such as **`KendraTestBot`**, and an optional description. The name must be unique in your account.

c. In the **IAM Permissions** section, choose **Create a role with basic Amazon Lex permissions**. This will create an AWS Identity and Access Management (IAM) role with the permissions that Amazon Lex V2 needs to run your bot.

d. In the **Children's Online Privacy Protection Act (COPPA)** section, choose **No**.

e. In the **Idle session timeout** and **Advanced settings** sections, leave the default settings and choose **Next**.

f. Now you are in the **Add language to bot** section. In the menu under **Voice interaction**, select **None. This is only a text based application**. Leave the default settings for the remaining fields.

g. Choose **Done**. Amazon Lex V2 creates your bot and a default intent called **NewIntent**, and takes you to the page to configure this intent

To successfully build a bot, you must create at least one intent that is separate from the `AMAZON.FallbackIntent` and the `AMAZON.KendraSearchIntent`. This intent is required to build your Amazon Lex V2 bot, but isn't used for the FAQ response. This intent must contain at least one sample utterance and the utterance must not apply to any of the questions that your customer asks.

To create the required intent:

1. In the **Intent details** section, give the intent a name, such as **`RequiredIntent`**.
2. In the **Sample utterances** section, type an utterance in the box next to **Add utterance**, such as **`Required utterance`**. Then choose **Add utterance**.
3. Choose **Save intent**.

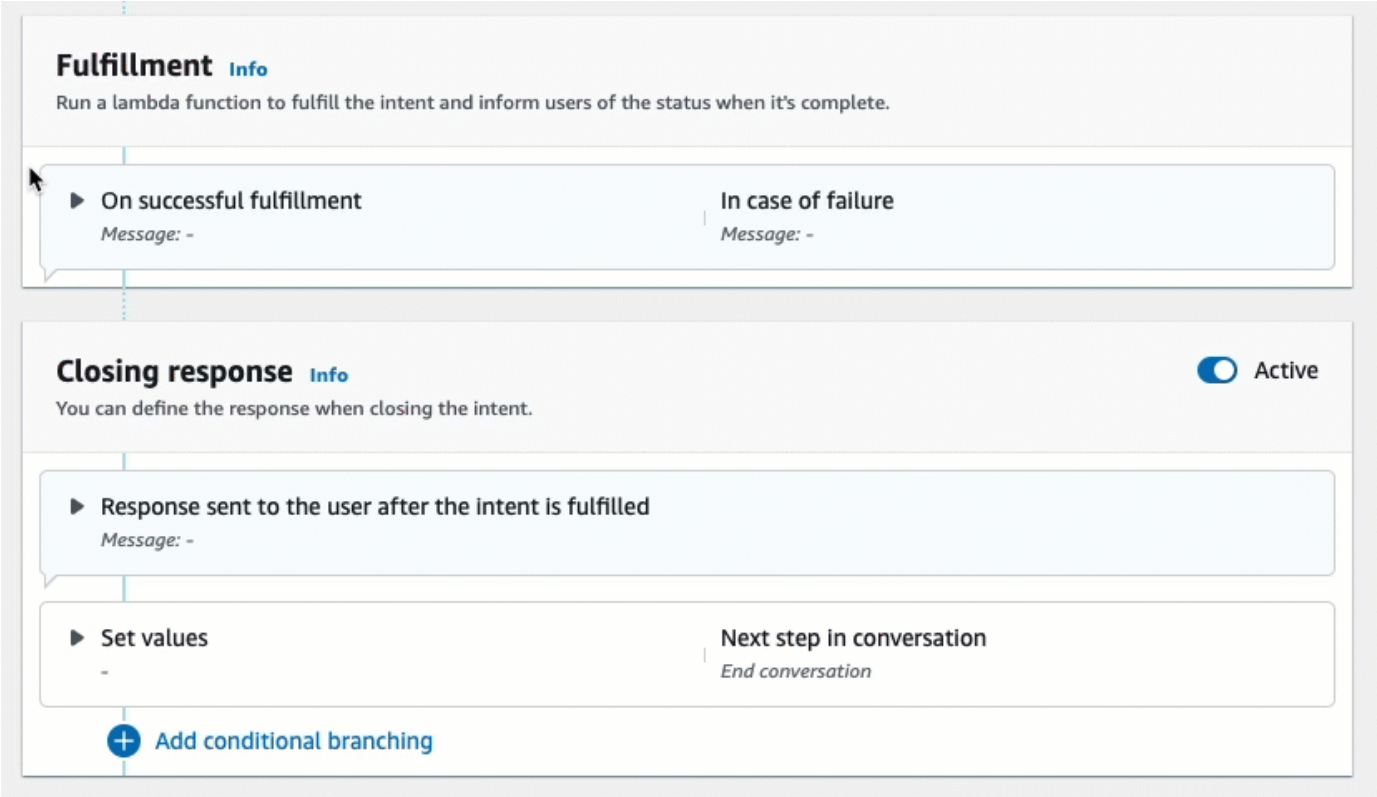
Create the intent to search an Amazon Kendra index and the response message that it should return.

To create an AMAZON.KendraSearchIntent intent and response message:

1. Select **Back to intents list** in the navigation pane to return to the **Intents** page for your bot. Choose **Add intent** and select **Use built-in intent** from the dropdown menu.
2. In the box that pops up, select the menu under **Built-in intent.** Enter **`AMAZON.KendraSearchIntent`** in the search bar and then choose it from the list.
3. Give the intent a name, such as **`KendraSearchIntent`**.

4. From the **Amazon Kendra index** dropdown menu, choose the index that you want the intent to search. The index that you created in the **Prerequisites** section should be available.
5. Select **Add**.
6. In the intent editor, scroll down to the **Fulfillment** section, select the right arrow to expand the section, and add the following message in the box under **On successful fulfillment**:

```
I found a link to a document that could help you: ((x-amz-lex:kendra-search-
response-document-link-1)).
```



For more information about the Amazon Kendra Search Response, see Using the Search Response.

7. Choose **Save intent**, and then choose **Build** to build the bot. When the bot is ready, the banner at the top of the screen turns green and displays a success message.

Finally, use the console test window to test responses from your bot.

To test your FAQ bot:

1. After the bot is successfully built, choose **Test**.

2. Enter **`What is Amazon Kendra?`** in the console test window. Verify that the bot responds with a link.
3. For more information about configuring `AMAZON.KendraSearchIntent`, see AMAZON.KendraSearchIntent and KendraConfiguration.

AMAZON.PauseIntent

Responds to words and phrases that enable the user to pause an interaction with a bot so that they can return to it later. Your Lambda function or application needs to save intent data in session variables, or you need to use the GetSession operation to retrieve intent data when you resume the current intent.

Common utterances:

- pause
- pause that

AMAZON.QnAIntent

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
> 2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

Responds to customer questions by using an Amazon Bedrock FM to search and summarize FAQ responses. This intent is activated when an utterance is not classified into any of the other intents present in the bot. Note that this intent will not be activated for missed utterances when eliciting a slot value. Once recognized, the `AMAZON.QnAIntent`, uses the specified Amazon Bedrock model to search the configured Amazon Bedrock Knowledge Base and respond to the customer question.

> **Warning**
>
> You can't use the `AMAZON.QnAIntent` and the `AMAZON.KendraSearchIntent` in the same bot locale.

The following knowledge store options are available. You must have already created the knowledge store and indexed the documents within it.

- OpenSearch Service domain – Contains indexed documents. To create a domain, follow the steps at Creating and managing Amazon OpenSearch Service domains.
- Amazon Kendra index – Contains indexed FAQ documents. To create a Amazon Kendra index, follow the steps at Creating an index.
- Amazon Bedrock Knowledge Base – Contains indexed data sources. To set up a Amazon Bedrock Knowledge Base, follow the steps at Building a Knowledge Base.

If you select this intent, you configure the following fields and then select **Add** to add the intent.

- **Bedrock model** – Choose the provider and foundation model to use for this intent. Make sure to check the latest available models and the deprecation schedule and plan migrations accordingly. For more information, see Model lifecycle.
- **Knowledge store** – Choose the source from which you want the model pull information from to answer customer questions. The following sources are available.
  - **OpenSearch** – Configure the following fields.
    - **Domain endpoint** – Provide the domain endpoint that you made for the domain or that was provided to you after domain creation.
    - **Index name** – Provide the index to search. For more information, see Indexing data in Amazon OpenSearch Service.
    - Choose how you want to return the response to the customer.
      - **Exact response** – When this option is enabled, the value in the Answer field is used as is for the bot response. The configured Amazon Bedrock foundation model is used to select the exact answer content as-is, without any content synthesis or summarization. Specify the name of the question and answer fields that were configured in the OpenSearch database.

    - **Include fields** – Returns an answer generated by the model using the fields you specify. Specify the name of up to five fields that were configured in the OpenSearch database. Use a semicolon (;) to separate fields.

- **Amazon Kendra** – Configure the following fields.
    - **Amazon Kendra index** – Select the Amazon Kendra index that you want your bot to search.
    - **Amazon Kendra filter** – To create a filter, select this checkbox. For more information on the Amazon Kendra search filter JSON format, see Using document attributes to filter search results.
    - **Exact response** – To let your bot return the exact response returned by Amazon Kendra, select this checkbox. Otherwise, the Amazon Bedrock model you select generates a response based on the results.

      > **Note**
      >
      > To use this feature, you must first add FAQ questions to your index by following the steps at Adding frequently asked questions (FAQs) to an index.

- **Amazon Bedrock Knowledge Base** – If you choose this option, specify the ID of the Amazon Bedrock Knowledge Base. You can find the ID by checking the details page of the Amazon Bedrock Knowledge Base in the console, or by sending a GetKnowledgeBase request.
    - **Exact response** – When this option is enabled, the value in the Answer field is used as is for the bot response. The configured Amazon Bedrock foundation model is used to select the exact answer content as-is, without any content synthesis or summarization. To use exact response for Amazon Bedrock Knowledge Base you need to do the following:
        - Create individual JSON files with each file containing an answer field that contains the exact response that needs to be returned to end-user.
        - When indexing these documents in Bedrock Knowledge Base, select **Chunking strategy** as **No Chunking**..
        - Define the answer field in Amazon Lex V2, as the Answer field in the Bedrock Knowledge Base.

The responses from the QnAIntent will be stored into the request attributes as shown below:

- `x-amz-lex:qnA-search-response` – The response from the QnAIntent to the question or utterance.

- `x-amz-lex:qnA-search-response-source` – Points to the document or list of documents used to generate the response.
- `x-amz-lex:qna-additional-context` – The additional context used by the QnAIntent to generate the response.

Additional model configurations

When AMAZON.QnAIntent is invoked it uses a default prompt template that combines instructions and context with the user query to construct the prompt that's sent to the model for response generation. You can also provide a custom prompt or update the default prompt to match your requirements.

You can engineer the prompt template with the following tools:

Prompt placeholders – Pre-defined variables in AMAZON.QnAIntent for Amazon Bedrock that are dynamically filled in at runtime during the bedrock call. In the system prompt, you can see these placeholders surrounded by the $ symbol. The following list describes the placeholders you can use:

Variable	Replaced by	Model	Required?
$query_results$	The retrieved results for the user query from the Knowledge Store	Selected Bedrock Model	Yes
$output_instruction$	Underlying instructions for formatting the response generation and citations. Differs by model. If you define your own formatting instructions, we suggest that you remove this placeholder.	Selected Bedrock Model	No

Variable	Replaced by	Model	Required?
$additional_context$	The additional context used by the QnAIntent to generate the response	Selected Bedrock Model	No
$locale$	The language in which the bot will answer customer queries	Selected Bedrock Model	No

Default prompt being used is:

```
$query_results$

$additional_context$

Please only follow the instructions in <instruction> tags below.
<instruction>
Given the conversation history, <additional_context> and <Context>:
(1) first, identify the user query intent and classify it as one of the categories:
 FAQ_QUERY, OTHER_QUERY, GIBBERISH, GREETINGS, AFFIRMATION, CHITCHAT, or MISC;
(2) second, if the intent is FAQ_QUERY, predict the most relevant grounding passage(s)
 by providing the passage id(s) or output CANNOTANSWER;
(3) then, generate a concise, to-the-point FAQ-style response in $locale$ locale
 ONLY USING the grounding content in <Context> and <additional_context>; or output
 CANNOTANSWER if the user query/request cannot be directly answered with the grounding
 content. DO NOT mention about the grounding passages such as ids or other meta data;
 do not create new content not presented in <Context>. Do NOT respond to query that is
 ill-intented or off-topic;
(4) lastly, provide the confidence level of the above prediction as LOW, MID or HIGH.
</instruction>

$output_instruction$
```

$output_instruction$ is replaced with:

```
Give your final response in the following form:
<answer>
<intent>FAQ_QUERY or OTHER_QUERY or GIBBERISH or GREETINGS or AFFIRMATION or CHITCHAT
 or MISC</intent>
<text>a concise FAQ-style response or CANNOTANSWER</text>
<passage_id>passage_id or CANNOTANSWER</passage_id>
<confidence>LOW or MID or HIGH</confidence>
</answer>
```

Note

If you decide not to use the default instructions, then whatever output the LLM provides will be returned as-is back to the end user.
The output instructions need to contain <text></text> and <passageId></passageId> tags and instructions for the LLM to return the passageIds to provide the response and source attribution.

Additional context support through session attributes

You can pass additional context to the `AMAZON.QnAIntent` at runtime through the session attribute `x-amz-lex:qna-additional-context`. This allows you to provide supplementary information that the model can use alongside the knowledge store results when generating a response. The additional context is inserted into the prompt template through the `$additional_context$` placeholder.

Example:

```
{"sessionAttributes": {"x-amz-lex:qna-additional-context":"Our support hours are Monday
 through Friday, 8AM-8PM EST"}}
```

Amazon Bedrock Knowledge Base metadata filtering support through session attributes

You can pass the Amazon Bedrock Knowledge Base metadata filters as part of session attribute `x-amz-lex:bkb-retrieval-filter`.

```
            {"sessionAttributes":{"x-amz-lex:bkb-retrieval-filter":"{\"equals\":{\"key
\":\"insurancetype\",\"value\":\"farmers\"}}
```

> **Note**
>
> You need to use the Amazon Bedrock Knowledge Base as the Data store for the QnAIntent to use this filter. For more information, see Metadata filtering

Inference configurations

You can define the inference configurations that will be used when making the call to LLM using session attribute:

- temperature: type Integer
- topP
- maxTokens

Example:

```
        {"sessionAttributes":{"x-amz-lex:llm-text-inference-config":"{\"temperature
\":0,\"topP\":1,\"maxTokens\":200}"}}
```

Bedrock Guardrails support through build-time and session attributes

- By using the Console at Buildtime – Provide the GuardrailsIdentifier and the GuardrailsVersion. Learn more under the Additional Model Configurations section.
- By using Session attributes – You can also define the Guardrails configuration using the session attributes: `x-amz-lex:bedrock-guardrails-identifier` and `x-amz-lex:bedrock-guardrails-version`.

For more information on using Bedrock Guardrails, see Guardrails.

AMAZON.QnAIntent (multiple use support)

You can choose to have multiple AMAZON.QnAIntents within a locale. Amazon Lex V2 support up to 5 AMAZON.QnAIntents within a bot locale.

AMAZON.QnAIntent can be triggered if one of the following cases is true:

- If a bot locale contains only 1 AMAZON.QnAIntent and that intent does not contain sample utterances, then it is activated when an utterance is not classified into any of the other intents present in the bot. This intent is activated when an utterance is not classified into any of the other intents present in the bot. Note that this intent will not be activated for missed utterances when eliciting a slot value.

  > **Note**
  >
  > If the response from the FM is unsatisfactory or the call to the FM fails, Amazon Lex V2 then invokes the `AMAZON.FallbackIntent`.

- If AMAZON.QnAIntent does contain sample utterances, then it is only activated when Amazon Lex V2 recognizes that the user wants to initiate that intent based on user input.

  > **Note**
  >
  > If the response from the FM is unsatisfactory or the call to the FM fails, Amazon Lex V2 invokes the failure next step, defined in the fulfillment block.

> **Note**
>
> If `botLocale` has more than 1 AMAZON.QnAIntent, then each AMAZON.QnAIntent needs to have at least 1 sample utterance.

AMAZON.QinConnectIntent

> **Note**
>
> To use generative AI capabilities using Amazon Q In Connect, you must complete the following pre-requisites:

1. Navigate to the Amazon Connect console and create your instance, if you don't have one already, see Get started with Amazon Connect.
2. Enable Amazon Q in Connect for your instance, see Enable Amazon Q in Connect for your instance.

AMAZON.QinConnectIntent responds to customer questions by using the LLM-enhanced evolution of Amazon Connect Wisdom that delivers real-time recommendations to help contact center customers and agents resolve customer issues quickly and accurately. This intent is activated when an utterance is not classified into any of the other intents present in the bot. Note that this intent will not be activated for missed utterances when eliciting a slot value. Once recognized, the AMAZON.QinConnectIntent, uses the specified Q in Connect domain to search the configured Amazon Bedrock Knowledge Base and respond to the customer question.

Note

- You cannot use AMAZON.QinConnectIntent along with AMAZON.QnAIntent in the same bot locale.
- If you select another language besides U.S. English, you must customize the self-service prompts (`SELF_SERVICE_PRE_PROCESSING` and `SELF_SERVICE_ANSWER_GENERATION`) to respond in the specified language. For more information on how to customize your prompt, see Customize Amazon Q in Connect.

If you select this intent, you need to configure the following fields and then select **Save Intent** to add the intent to the bot.

- Amazon Q In Connect Configuration - Provide the Amazon Resource Name (ARN) of the Amazon Q in Connect assistant. Assistant ARN Pattern: `^arn:[a-z-]*?:wisdom:[a-z0-9-]*?:[0-9]{12}:[a-z-]*?/[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}(?:/[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}){0,2}$`.

The responses from the QinConnectIntent will be stored into the request attributes as shown below:

- `x-amz-lex:q-in-connect-response` – The response from QinConnectIntent to the question or utterance.

Session Attributes Returned from QinConnectIntent

Interaction with the QinConnect intent provides additional data about conversation through session attributes.

1. `x-amz-lex:q-in-connect:session-arn` – An unique identifier for the session created with Amazon Q In Connect during the conversation.
2. `x-amz-lex:q-in-connect:conversation-status` – The current status of the conversation with QinConnect assistant or domain. There are three values possible for this status:
   - `CLOSED`
   - `READY`
   - `PROCESSING`
3. `x-amz-lex:q-in-connect:conversation-status-reason` – Provides the reason for the current status reported with above attribute. The possible reasons are as follows:
   - SUCCESS – Indicates customer has nothing left to ask and question has been answered successfully.
   - `FAILED` – Indicates a failure while answering the customer's question. These are mostly due to failures to understand the customer's question.
   - `REJECTED` – Indicates that the assistant is rejecting to answer customer question, and recommending that the question be handled outside of the bot interaction, such as talking to person or agent, to get more information.

> **Note**
>
> When a bot with QinConnectIntent is invoked during customer interactions driven by an Amazon Connect instance, your session arn needs to be created and passed from the Amazon Connect instance. To create a session, Amazon Connect Flows could be configured with Amazon Q in Connect step.

Limitations

- You cannot use AMAZON.QinConnectIntent along with intents without specific utterances such as AMAZON.QnAIntent, AMAZON.BedrockAgentIntent in the same bot locale.
- When a bot with QinConnectIntent is invoked during a customer interactions driven by an Amazon Connect instance, your session arn needs to be created and passed from the Amazon Connect instance. To create a session, Amazon Connect Flows could be configured with Amazon Q In Connect step.
- There can be no more than one AMAZON.QinConnectIntent per bot locale.
- The Amazon Q in Connect domain used with AMAZON.QinConnectIntent must be in the same AWS Region as the Amazon Lex V2 bot.

Permissions

If the QinConnect Intent is used in an Amazon Lex V2 bot, and the bot is using a Service Linked Role (SLR), the Amazon Lex V2 service has the permissions to update the appropriate policies on the role to integrate it with the Q in Connect assistant. If the bot is using a custom IAM role, then the user would need to manually add these permissions to their IAM role.

The Service Linked Role will get updated with the following permissions if the QinConnect intent gets added. A new policy will be added for QinConnect access:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Sid": "QInConnectAssistantPolicy",
            "Action": [
                "wisdom:CreateSession",
                "wisdom:GetAssistant"
            ],
            "Resource": [
                "arn:aws:wisdom:*:accountId:assistant/assistantId",
                "arn:aws:wisdom:*:accountId:assistant/assistantId/*"
            ]
        },
        {
            "Effect": "Allow",
            "Sid": "QInConnectSessionsPolicy",
```

```
            "Action": [
                "wisdom:SendMessage",
                "wisdom:GetNextMessage"
            ],
            "Resource": [
                "arn:aws:wisdom:*:accountId:session/assistantId/*"
            ]
        },
        {
            "Sid": "QInConnectKmsCMKPolicy",
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:GenerateDataKey"
            ],
            "Resource": [
                "arn:aws:kms:region:accountId:key/keyId"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "accountId",
                    "kms:ViaService": "wisdom.region.amazonaws.com",
                    "kms:EncryptionContext:aws:wisdom:assistant:arn":
 ["arn:aws:wisdom:region:accountId:assistant/assistantId"]
                }
            }
        }
    ]
}
```

> **Note**
>
> The `QInConnectKmsCMKPolicy` statement is only required if you are using a customer managed KMS key with the Amazon Q in Connect assistant.

Trust Policy

```
{
    "Effect": "Allow",
    "Sid": "LexV2InternalTrustPolicy",
    "Principal": {
```

```
            "Service": "lexv2.aws.internal"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "accountId"
            },
            "ArnLike": {
                "aws:SourceArn": "arn:aws:lex:*:accountId:bot-alias/botId/*"
            }
        }
}
```

AMAZON.RepeatIntent

Responds to words and phrases that enable the user to repeat the previous message. Your application needs to use a Lambda function to save the previous intent information in session variables, or you need to use the GetSession operation to get the previous intent information.

Common utterances:

- repeat
- say that again
- repeat that

AMAZON.ResumeIntent

Responds to words and phrases the enable the user to resume a previously paused intent. You Lambda function or application must manage the information required to resume the previous intent.

Common utterances:

- resume
- continue
- keep going

AMAZON.StartOverIntent

Responds to words and phrases that enable the user to stop processing the current intent and start over from the beginning. You can use your Lambda function or the `PutSession` operation to elicit the first slot value again.

Common utterances:

- start over
- restart
- start again

AMAZON.StopIntent

Responds to words and phrases that indicate that the user wants to stop processing the current intent and end the interaction with a bot. Your Lambda function or application should clear any existing attributes and slot type values and then end the interaction.

Common utterances:

- stop
- off
- shut up

Adding slot types

Slot types define the values that users can supply for your intent variables. You define slot types for each language so that the values are specific to that language. For example, for a slot type that lists paint colors, you could include the value "red" in English, "rouge" in French, and "rojo" in Spanish.

This topic describes how to create custom slot types that provide values for your intent's slots. You can also use built-in slot types for standard values. For example, you can use the built-in slot type `AMAZON.Country` for a list of countries in the world.

To create a slot type

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of bots, choose the bot that you want to add the language to, then choose **Conversation structure** and then **All languages**.
3. Choose the language to add the slot type to, then choose **Slot types**.
4. Choose **Add slot type**, give your slot type a name, and then choose **Add**.
5. In the slot type editor, add the details of your slot type.

    - **Slot value resolution** – Determines how slot values are resolved. If you choose **Expand values**, Amazon Lex V2 uses the values as representative values for training. If you use **Restrict to slot values**, the allowed values for the slot are restricted to the ones that you provide.
    - **Slot type values** – The values for the slot. If you chose **Restrict to slot values**, you can add synonyms for the value. For example, for the value "football" you can add the synonym "soccer." If the user enters "soccer" in a conversation with your bot, the actual value of the slot is "football."
    - **Use slot values as custom vocabulary** – Enable this option to help improve recognition of slot values and synonyms in audio conversations. Don't enable this option when the slot values are common terms, such as "yes," "no," "one," "two, "three," etc.
6. Choose **Save slot type**.

Amazon Lex V2 offers the following slot types:

Topics

Built-in slot types

Amazon Lex supports built-in slot types that define how data in the slot is recognized and handled. You can create slots of these types in your intents. This eliminates the need to create enumeration

values for commonly used slot data such as date, time, and location. Built-in slot types do not have versions.

Slot Type	Short Description	Supported Locales	
AMAZON.AlphaNumeric	Recognizes words made up of letters and numbers.		
AMAZON.City	Recognizes words that represent a city.	All locales	
AMAZON.Confirmation	Recognizes words that mean 'Yes', 'No', 'Maybe', and 'Don't know' and converts them to a standard (Yes/No/Maybe/Don't know) format.	All locales	
AMAZON.Country	Recognizes words that represent a country.	All locales	
AMAZON.Currency	Recognizes words that represent a currency value and converts them into a standard currency abbreviation and value.	All locales	
AMAZON.Date	Recognizes words that represent a date and converts them to a standard format.	All locales	

Slot Type	Short Description	Supported Locales
AMAZON.Duration	Recognizes words that represent duration and converts them to a standard format.	All locales
AMAZON.EmailAddress	Recognizes words that represent an email address and converts them into a standard email address.	All locales
AMAZON.FirstName	Recognizes words that represent a first name.	All locales
AMAZON.LastName	Recognizes words that represent a last name.	All locales
AMAZON.Number	Recognizes numeric words and converts them into digits.	All locales
AMAZON.Percentage	Recognizes words that represent a percentage and converts them to a number and a percent sign (%).	All locales

Slot Type	Short Description	Supported Locales
AMAZON.PhoneNumber	Recognizes words that represent a phone number and converts them into a numeric string.	All locales
AMAZON.State	Recognizes words that represent a state.	All locales
AMAZON.StreetName	Recognizes words that represent a street name.	All locales
AMAZON.Time	Recognizes words that indicate times and converts them into a time format.	All locales
AMAZON.UKPostalCode	Recognizes words that represent a UK post code and converts them to a standard form.	English (British) (en-GB) only
AMAZON.FreeFormInput	Recognizes strings that consist of any words or characters.	All locales

AMAZON.AlphaNumeric

Recognizes strings made up of letters and numbers, such as **APQ123**.

You can use the `AMAZON.AlphaNumeric` slot type for strings that contain:

- Alphabetical characters, such as **ABC**

- Numeric characters, such as **123**
- A combination of alphanumeric characters, such as **ABC123**

The `AMAZON.AlphaNumeric` slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter letters. For more information, see Capturing slot values with spelling styles during the conversation.

You can add a regular expression to the `AMAZON.AlphaNumeric` slot type to validate values entered for the slot. For example, you can use a regular expression to validate:

- Canadian postal codes
- Driver's license numbers
- Vehicle identification numbers

Use a standard regular expression. Amazon Lex V2 supports the following characters in the regular expression:

- A-Z, a-z
- 0-9

Amazon Lex V2 also supports Unicode characters in regular expressions. The form is `\u`*`Unicode`*. Use four digits to represent Unicode characters. For example, `[\u0041-\u005A]` is equivalent to [A-Z].

The following regular expression operators are not supported:

- Infinite repeaters: *, +, or {x,} with no upper bound.
- Wild card (.)

The maximum length of the regular expression is 300 characters. The maximum length of a string stored in an `AMAZON.AlphaNumeric` slot type that uses a regular expression is 30 characters.

The following are some example regular expressions.

- Alphanumeric strings, such as **APQ123** or **APQ1**: `[A-Z]{3}[0-9]{1,3}` or a more constrained `[A-DP-T]{3} [1-5]{1,3}`

- US Postal Service Priority Mail International format, such as **CP123456789US**: `CP[0-9]{9}US`
- Bank routing numbers, such as **123456789**: `[0-9]{9}`

To set the regular expression for a slot type, use the console or the CreateSlotType operation. The regular expression is validated when you save the slot type. If the expression isn't valid, Amazon Lex V2 returns an error message.

When you use a regular expression in a slot type, Amazon Lex V2 checks input to slots of that type against the regular expression. If the input matches the expression, the value is accepted for the slot. If the input does not match, Amazon Lex V2 prompts the user to repeat the input.

AMAZON.City

Provides a list of local and world cities. The slot type recognizes common variations of city names. Amazon Lex V2 doesn't convert from a variation to an official name.

Examples:

- New York
- Reykjavik
- Tokyo
- Versailles

AMAZON.Confirmation

This slot type recognizes input phrases that corresponds to 'Yes', 'No', 'Maybe', and 'Don't know' phrases and words for Amazon Lex V2 and converts it one of the four values. It can be used to capture confirmation or acknowledgement from the user. Based on the final resolved value, you can create conditions to design multiple conversation paths.

For example:

if {confirmation} = "Yes", fulfill the intent

else, elicit another slot

Examples:

- Yes: Yeah, Yep, Ok, Sure, I have it, I can agree...

- No: Nope, Negative, Naw, Forget it, I'll decline, No way...
- Maybe: It's possible, Perhaps, Sometimes, I might, That could be right...
- Don't know: Dunno, Unknown, No idea, Not sure about it, Who knows...

As of August 17th, 2023, if there is an existing custom slot type named "Confirmation", the name must be changed to avoid conflict with the built-in slot Confirmation. In the left side navigation in the Lex console, go to the slot type (for an existing custom slot type named Confirmation) and update slot type name. The new slot type name must not be “Confirmation,” which is a reserved keyword for the built-in confirmation slot type.

AMAZON.Country

The names of countries around the world. Examples:

- Australia
- Germany
- Japan
- United States
- Uruguay

AMAZON.Currency

Converts words that represent a currency into a standard ISO 4217 alphabetic currency code and a number. Amazon Lex V2 recognizes currencies but does not convert from one currency to another.

For more information, see Currency codes - ISO 4217 on the International Organization for Standardization (ISO) website.

The currency represented is structured as follows: `{Unit} {Amount}`

- {Unit} refers to the specific currency unit (for example, USD).
- {Amount} denotes the monetary value, formatted to two decimal places (for example, 300.00).

Examples (all examples below are using the en-US locale; different locales may yield different results):

- "3USD": USD 3.00

- "USD300": USD 300.00
- "3 dimes" : USD 0.30
- "$1.56": USD 1.56
- "5c": USD 0.05
- "1 dollar": USD 1.00
- "five fifteen": USD 515.00
- “five dollars fifteen cents”: USD 5.15
- "5 usd and 1/2": USD 5.50

AMAZON.Date

Converts words that represent dates into a date format.

The date is provided to your intent in ISO-8601 date format. The date that your intent receives in the slot can vary depending on the specific phrase uttered by the user.

- Utterances that map to a specific date, such as "today," "now," or "November twenty-fifth," convert to a complete date: `2020-11-25`. This defaults to dates *on or after* the current date.
- Utterances that map to a future week, such as "next week," convert to the date of the last day of the current week. In ISO-8601 format, the week starts on Monday and ends on Sunday. For example, if today is 2020-11-25, "next week" converts to `2020-11-29`. Dates that map to the current or previous week convert to the first day of the week. For example, if today is 2020-11-25, "last week" converts to `2020-11-16`.
- Utterances that map to a future month, but not a specific day, such as "next month," convert to the last day of the month. For example, if today is 2020-11-25, "next month" converts to `2020-12-31`. For dates that map to the current or previous month convert to the first day of the month. For example, if today is 2020-11-25, "this month" maps to `2020-11-01`.
- Utterances that map to a future year, but not a specific month or day, such as "next year," convert to the last day of the following year. For example, if today is 2020-11-25, "next year" converts to `2021-12-31`. For dates that map to the current or previous year convert to the first day of the year. For example, if today is 2020-11-25, "last year" converts to `2019-01-01`.

AMAZON.Duration

Converts words that indicate durations into a numeric duration.

The duration is resolved to a format based on the ISO-8601 duration format, PnYnMnWnDTnHnMnS. The P indicates that this is a duration, the n is a numeric value, and the capital letter following the n is the specific date or time element. For example, P3D means 3 days. A T is used to indicate that the remaining values represent time elements rather than date elements.

Examples:

- "ten minutes": PT10M
- "five hours": PT5H
- "three days": P3D
- "forty five seconds": PT45S
- "eight weeks": P8W
- "seven years": P7Y
- "five hours ten minutes": PT5H10M
- "two years three hours ten minutes": P2YT3H10M

AMAZON.EmailAddress

Recognizes words that represent an email address provided as username@domain. Addresses can include the following special characters in a user name: underscore (_), hyphen (-), period (.), and the plus sign (+).

The AMAZON.EmailAddress slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter email addresses. For more information, see Capturing slot values with spelling styles during the conversation.

AMAZON.FirstName

Commonly used first names. This slot type recognizes formal names, informal nicknames, and names consisting of more than one word. The name sent to your intent is the value sent by the user. Amazon Lex V2 doesn't convert from the nickname to the formal name.

For first names that sound alike but are spelled differently, Amazon Lex V2 sends your intent a single common form.

The AMAZON.FirstName slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter names. For more information, see Capturing slot values with spelling styles during the conversation.

Examples:

- Emily
- John
- Sophie
- Anil Kumar

AMAZON.FirstName also returns a list of closely related names based on the original value. You can use the list of resolved values to recover from typos, confirm the name with the user, or perform a database look-up for valid names in your user directory.

For example, the input "John" may result in returning additional related names such as "John J" and "John-Paul".

The following shows the response format for the `AMAZON.FirstName` built-in slot type:

```
"value": {
    "originalValue": "John",
    "interpretedValue": "John",
    "resolvedValues": [
        "John",
        "John J.",
        "John-Paul"
    ]
}
```

AMAZON.LastName

Commonly used last names. For names that sound alike that are spelled differently, Amazon Lex V2 sends your intent a single common form.

The `AMAZON.LastName` slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter names. For more information, see Capturing slot values with spelling styles during the conversation.

Examples:

- Brosky
- Dasher

- Evers
- Parres
- Welt

AMAZON.LastName also returns a list of closely related names based on the original value. You can use the list of resolved values to recover from typos, confirm the name with the user, or perform a database look-up for valid names in your user directory.

For example, the input "Smith" may result in returning additional related names such as "Smyth" and "Smithe".

The following shows the response format for the `AMAZON.LastName` built-in slot type:

```
"value": {
    "originalValue": "Smith",
    "interpretedValue": "Smith",
    "resolvedValues": [
        "Smith",
        "Smyth",
        "Smithe"
    ]
}
```

AMAZON.Number

Converts words or numbers that express a number into digits, including decimal numbers. The following table shows how the `AMAZON.Number` slot type captures numeric words.

Input	Response
one hundred twenty three point four five	123.45
one hundred twenty three dot four five	123.45
point four two	0.42
point forty two	0.42
232.998	232.998

Input	Response
50	50
-15	-15
minus 15	-15
minus fifteen point two four five	-15.245

AMAZON.Percentage

Converts words and symbols that represent a percentage into a numeric value with a percent sign (%).

If the user enters a number without a percent sign or the word "percent," the slot value is set to the number. The following table shows how the `AMAZON.Percentage` slot type captures percentages.

Input	Response
50 percent	50%
0.4 percent	0.4%
23.5%	23.5%
twenty five percent	25%

AMAZON.PhoneNumber

Converts the numbers or words that represent a phone number into a string format without punctuation as follows.

Type	Description	Input	Result
International number with leading plus (+) sign	11-digit number with leading plus sign.	+61 7 4445 1061	+61744431061
		+1 (509) 555-1212	+15095551212

Type	Description	Input	Result
International number without leading plus (+) sign	11-digit number without leading plus sign	1 (509) 555-1212 61 7 4445 1061	15095551212 61744451061
National number	10-digit number without international code	(03) 5115 4444 (509) 555-1212	0351154444 5095551212
Local number	phone number without an internati onal code or an area code	555-1212	5551212

AMAZON.State

The names of geographical and political regions within countries.

Examples:

- Bavaria
- Fukushima Prefecture
- Pacific Northwest
- Queensland
- Wales

AMAZON.StreetName

The names of streets within a typical street address. This includes just the street name, not the house number.

Examples:

- Canberra Avenue
- Front Street
- Market Road

AMAZON.Time

Converts words that represent times into time values. `AMAZON.Time` can resolve exact times, ambiguous values, and time ranges. The slot value can resolve to the following time ranges:

- AM
- PM
- MO (morning)
- AF (afternoon)
- EV (evening)
- NI (night)

When a user enters an ambiguous time, Amazon Lex V2 uses the `slots` attribute of a Lambda event to pass resolutions for the ambiguous times to your Lambda function. For example, if your bot prompts the user for a delivery time, the user can respond by saying "10 o'clock." This time is ambiguous. It means either 10:00 AM or 10:00 PM. In this case, the value in the `interpretedValue` field is `null`, and the `resolvedValues` field contains the two possible resolutions of the time. Amazon Lex V2 inputs the following into the Lambda function:

```
"slots": {
    "deliveryTime": {
    "value": {
        "originalValue": "10 o'clock",
        "interpretedValue": null,
        "resolvedValues": [
            "10:00", "22:00"
        ]
    }
}
```

When the user responds with an unambiguous time, Amazon Lex V2 sends the time to your Lambda function in the `interpretedValue` field of the `slots` attribute of the Lambda event. For example, if your user responds to the prompt for a delivery time with "10:00 AM," Amazon Lex V2 inputs the following into the Lambda function:

```
"slots": {
    "deliveryTime": {
```

```
        "value": {
        "originalValue": "10 AM",
        "interpretedValue": 10:00,
        "resolvedValues": [
            "10:00"
        ]
    }
}
```

When the user responds to a prompt for a delivery time with "in the morning," Amazon Lex V2 inputs the following into the Lambda function:

```
"slots": {
    "deliveryTime": {
    "value": {
        "originalValue": "morning",
        "interpretedValue": "MO",
        "resolvedValues": [
            "MO"
        ]
    }
}
```

For more information about the data sent from Amazon Lex V2 to a Lambda function, see AWS Lambda input event format for Lex V2.

AMAZON.UKPostalCode

Converts words that represent a UK postal code to a standard format for postal codes in the United Kingdom. The `AMAZON.UKPostalCode` slot type validates and resolves the post code to a set of standardized formats, but it doesn't check to make sure that the post code is valid. Your application must validate the post code.

The `AMAZON.UKPostalCode` slot type is available only in the English (UK) (en-GB) locale.

The `AMAZON.UKPostalCode` slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter letters. For more information, see Capturing slot values with spelling styles during the conversation.

The slot type recognizes only the valid post code formats listed below, used in the United Kingdom. The valid formats are ("A" representis a letter, and "9" represents a digit):

- AA9A 9AA
- A9A 9AA
- A9 9AA
- A99 9AA
- AA9 9AA
- AA99 9AA

For text input, the user can enter any mix of upper and lower case letters. The user can use or omit the space in the post code. The resolved value will always include the space in the proper location for the post code.

For spoken input, the user can speak the individual characters, or they can use double letter pronunciations, such as "double A" or "double 9". They can also use double-digit pronunciations, such as "ninety-nine" for "99".

> **Note**
>
> Not all UK postal codes are recognized. Only the formats listed above are supported.

AMAZON.FreeFormInput

`AMAZON.FreeFormInput` can be used to capture free form input from the end user. It recognizes strings that consist of words or characters. The resolved value is the entire input utterance.

Example:

Bot: Please provide feedback from your call experience.

User: I got the answers to all of my questions, and I was able to complete the transaction.

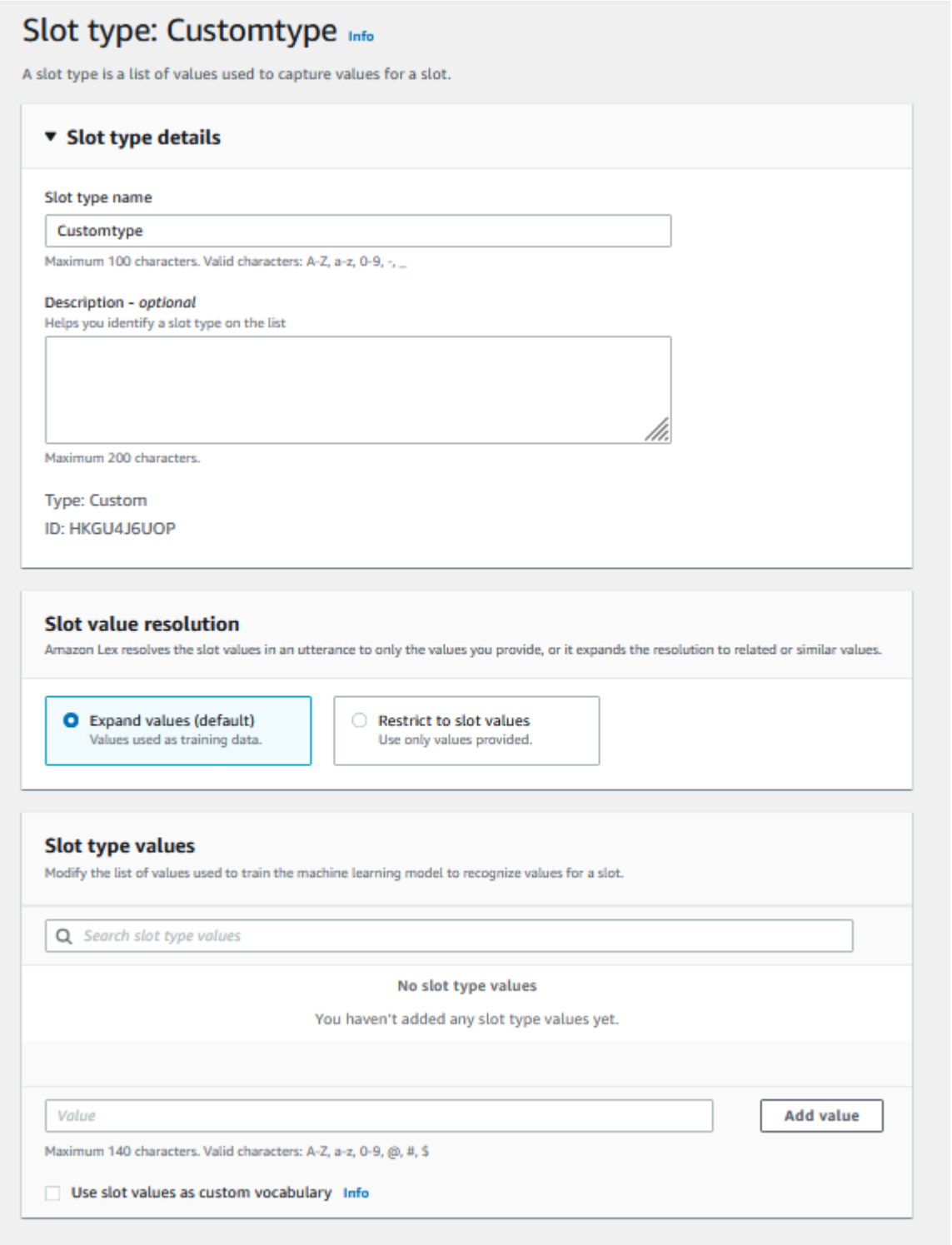
Note:

- `AMAZON.FreeFormInput` can be used to capture free form input as-is from the end user.
- `AMAZON.FreeFormInput` cannot be used in intent sample utterances.
- `AMAZON.FreeFormInput` cannot have slot sample utterances.

- AMAZON.FreeFormInput is only recognized when elicited for.
- AMAZON.FreeFormInput does not support wait and continue.
- AMAZON.FreeFormInput is currently not supported in the Connect Customer Chat channel.
- When a AMAZON.FreeFormInput slot is elicited, FallbackIntent will not be triggered.
- When a AMAZON.FreeFormInput slot is elicited, there will be no intent switch.

Custom slot type

For each intent, you can specify parameters that indicate the information that the intent needs to fulfill the user's request. These parameters, or slots, have a type. A *slot type* is a list of values that Amazon Lex V2 uses to train the machine learning model to recognize values for a slot. For example, you can define a slot type called Genres with values such as "comedy," "adventure," "documentary," and so on. You can define synonyms for a slot type value. For example, you can define the synonyms "funny" and "humorous" for the value "comedy."



You can configure the slot type to expand the slot values. Slot values will be used as training data and the model will resolve the slot to the value provided by the user if it is similar to the slot values and synonyms of those values. This is the default behavior. Amazon Lex V2 maintains a list of possible resolutions for a slot. Each entry in the list provides a *resolved value* that Amazon Lex V2

recognized as additional possibilities for the slot. A resolved value is the best effort to match the slot value. The list contains up to five values.

Alternatively, you can configure the slot type to restrict resolution to the slot values. In this case, the model will resolve a slot value entered by the user to an existing slot value only if it is the same as that slot value or it is a synonym. For example, if the user enters "funny" it will resolve to the slot value "comedy."

When the value entered by the user is a synonym of a slot type value, the model returns that slot type value as the first entry in the list of `resolvedValues`. For example, if the user enters "funny," the model populates the `originalValue` field with the value "funny" and the first entry in the resolvedValues field with "comedy." You can configure the `valueSelectionStrategy` when you create or update a slot type with the CreateSlotType operation so that the slot value is filled with the first value in the resolution list.

Custom slot types support inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter letters. For more information, see Capturing slot values with spelling styles during the conversation.

If you are using a Lambda function, the input event to the function includes a resolution list called `resolvedValues`. The following example shows the slot section of the input to a Lambda function:

```
"slots": {
   "MovieGenre": {
      "value": {
         "originalValue": "funny",
         "interpretedValue": "comedy",
         "resolvedValues": [
            "comedy"
         ]
      }
   }
}
```

For each slot type, you can define a maximum of 10,000 values and synonyms. Each bot can have a total number of 50,000 slot type values and synonyms. For example, you can have 5 slot types, each with 5,000 values and 5,000 synonyms, or you can have 10 slot types, each with 2,500 values and 2,500 synonyms.

A custom slot type should not have the same name as the built-in slot types. For example, a custom slot type should not be named with the reserved keywords of Date, Number, or Confirmation. These keywords are reserved for built-in slot types. For a list of all built-in slot types, see Built-in slot types.

Grammar slot type

> **Important**
>
> Amazon Lex V2 grants access to grammar slot types for each AWS account. If your account doesn't have access, Amazon Lex V2 rejects attempts to add a grammar slot type to a bot. Amazon Lex V2 also rejects builds for locales that include a grammar slot type. To request access for your account, contact AWS Support. If you don't need to author your own grammar, you can use a custom slot type instead.

With the grammar slot type, you can author your own grammar in the XML format per the SRGS specification to collect information in a conversation. Amazon Lex V2 recognizes utterances matched by the rules specified in the grammar. You can also provide semantic interpretation rules using ECMAScript tags within the grammar files. Amazon Lex then returns properties set in the tags as resolved values when a match occurs.

You can only create grammar slot types in the English (Australia), English (UK), and English (US) locales.

There are two parts to a grammar slot type. The first is the grammar itself written using the SRGS specification format. The grammar interprets the utterance from the user. If the utterance is accepted by the grammar it is matched, otherwise it is rejected. If an utterance is matched it is passed on to the script if there is one.

The second is part of a grammar slot type is an optional script written in ECMAScript that transforms the input to the resolved values returned by the slot type. For example, you can use a script to convert spoken numbers to digits. ECMAScript statements are enclosed in the <tag> element.

The following example is in the XML format per the SRGS specification that shows a valid grammar accepted by Amazon Lex V2. It defines a grammar slot type that accepts card numbers and determines if they are for regular or premium accounts. For more information about the acceptable syntax, see the Grammar definition and the Script format topics.

```
<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
         xml:lang="en-US" tag-format="semantics/1.0" root="card_number">

   <rule id="card_number" scope="public">
        <item repeat="0-1">
            card number
        </item>
        <item>
            seven
            <tag>out.value = "7";</tag>
         </item>
         <item>
            <one-of>
               <item>
                  two four one
                  <tag> out.value = out.value + "241"; out.card_type = "premium"; </
tag>
               </item>
               <item>
                  zero zero one
                  <tag> out.value = out.value + "001"; out.card_type = "regular";</tag>
               </item>
            </one-of>
         </item>
   </rule>
</grammar>
```

The above grammar only accepts two types of card numbers: 7241 or 7001. Both of these may be optionally prefixed with “card number”. It also contains ECMAScript tags that can be used for semantic interpretation. With semantic interpretation, the utterance “card number seven two four one” would return following object:

```
{
    "value": "7241",
    "card_type": "premium"
}
```

This object is returned as a JSON-serialized string in the `resolvedValues` object returned by the RecognizeText, RecognizeUtterance, and StartConversation operations.

Adding a grammar slot type

To add a grammar slot type

1. Upload the XML definition of your slot type to an S3 bucket. Make a note of the bucket name and the path to the file.

   > **Note**
   >
   > The maximum file size is 100 KB.

2. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
3. From the left menu, choose **Bots** and then choose the bot to add the grammar slot type to.
4. Choose **View languages**, and then choose the language to add the grammar slot type to.
5. Choose **View slot types**.
6. Choose **Add slot type**, and then choose **Add grammar slot type**.
7. Give the slot type a name, and then choose **Add**.
8. Choose the S3 bucket that contains your definition file and enter the path to the file. Choose **Save slot type**.

Grammar definition

This topic shows the parts of the SRGS specification that Amazon Lex V2 supports. All of the rules are defined in the SRGS specification. For more information, see the Speech recognition grammar specification version 1.0 W3C recommendation.

Topics

- Header declarations
- Supported XML elements
- Tokens
- Rule reference
- Sequences and encapsulation
- Repeats

- Language
- Tags
- Weights

This document includes material copied and derived from the W3C Speech Recognition Grammar Specification Version 1.0 (available at https://www.w3.org/TR/speech-grammar/). Citation information follows:

The SRGS specification document, a W3C Recommendation, is available from the W3C under the following license.

License text

License

By using and/or copying this document, or the W3C document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the W3C document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- A link or URL to the original W3C document.
- The pre-existing copyright notice of the original author, or if it doesn't exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [$date-of-document] World Wide Web Consortium, (MIT, ERCIM, Keio, Beihang). http://www.w3.org/Consortium/Legal/2015/doc-license"
- *If it exists*, the STATUS of the W3C document.

When space permits, inclusion of the full text of this **NOTICE** should be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of W3C documents is granted pursuant to this license, except as follows: To facilitate implementation of the technical specifications set forth in this document, anyone may prepare and distribute derivative works and portions of this document in software, in supporting materials accompanying software, and in documentation of software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

In addition, "Code Components" —Web IDL in sections clearly marked as Web IDL; and W3C-defined markup (HTML, CSS, and so on) and computer programming language code clearly marked as code examples— are licensed under the W3C Software License.

The notice is:

"Copyright © 2015 W3C® (MIT, ERCIM, Keio, Beihang). This software or document includes material copied from or derived from [title and URI of the W3C document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Header declarations

The following table shows the header declarations supported by the grammar slot type. For more information, see Grammar header declarations in the *Speech recognition grammar specification version 1* W3C recommendation.

Declaration	Specification requirement	XML form	Amazon Lex support	Specification
Grammar version	Required	4.3: `version` attribute on `grammar` element	Required	SRGS
XML namespace	Required (XML only)	4.3: `xmlns` attribute on `grammar` element	Required	SRGS
Document type	Required (XML only)	4.3: XML DOCTYPE	Recommended	SRGS
Character encoding	Recommended	4.4: `encoding` attribute in XML declaration	Recommended	SRGS
Language	Required in voice mode Ignored in DTMF mode	4.5: `xml:lang` attribute on `grammar` element	Required in voice mode Ignored in DTMF mode	SRGS
Mode	Optional	4.6: `mode` attribute on `grammar` element	Optional	SRGS
Root rule	Optional	4.7: `root` attribute on `grammar` element	**Required**	SRGS
Tag format	Optional	4.8: `tag-format` attribute	**String literal and ECMAScript are supported**	SRGS, SISR

Declaration	Specification requirement	XML form	Amazon Lex support	Specification
		on `grammar` element		
Base URI	Optional	4.9: `xml:base` attribute on `grammar` element	Optional	SRGS
Pronunciation lexicon	Optional, multiple allowed	4.10: `lexicon` element	**Not supported**	SRGS, PLS
Metadata	Optional, multiple allowed	4.11.1: `meta` element	Required	SRGS
XML metadata	Optional, XML only	4.11.2: `metadata` element	Optional	SRGS
Tag	Optional, multiple allowed	4.12: `tag` element	**Global tags not supported**	SRGS

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
                 "http://www.w3.org/TR/speech-grammar/grammar.dtd">

<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xml:base="http://www.example.com/base-file-path"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US"
         version="1.0"
         mode="voice"
         root="city"
```

```
        tag-format="semantics/1.0">
```

Supported XML elements

Amazon Lex V2 supports the following XML elements for custom grammars:

- `<item>`
- `<token>`
- `<tag>`
- `<one-of>`
- `<rule-ref>`

Tokens

The following table shows the token specifications supported by the grammar slot type. For more information, see Tokens in the *Speech recognition grammar specification version 1* W3C recommendation.

Token type	Example	Supported?
Single unquoted token	hello	Yes
Single unquoted token: non-alphabetic	2	Yes
Single quoted token, no white space	"hello"	Yes, drop double quotes when it only contains a single token
Two tokens delimited by white space	bon voyage	Yes
Four tokens delimited by white space	this is a test	Yes
Single quoted token, including white space	"San Francisco	**No**

Token type	Example	Supported?
Single XML token in <token> tag	<token>San Francisco</token>	**No** (same as single quoted token with white space)

Notes

- *Single quoted token including white space* – The specification requires words enclosed in double quotes be treated as a single token. Amazon Lex V2 treats them as white space delimited tokens.
- *Single XML token in <token>* – The specification requires words delimited by <token> to represent one token. Amazon Lex V2 treats them as white space delimited tokens.
- Amazon Lex V2 throws a validation error when either usage is found in your grammar.

Example

```
<rule id="state" scope="public">
    <one-of>
        <item>FL</item>
        <item>MA</item>
        <item>NY</item>
    </one-of>
</rule>
```

Rule reference

The following table summarizes the various forms of rule reference that are possible within grammar documents. For more information, see Rule reference in the *Speech recognition grammar specification version 1* W3C recommendation.

Reference type	XML form	Supported
2.2.1 Explicit local rule reference	`<ruleref uri="#rulename"/>`	Yes
2.2.2 Explicit reference to a named rule of a grammar identified by a URI	`<ruleref uri="grammarURI#rulename"/>`	**No**

Reference type	XML form	Supported
2.2.2 Implicit reference to the root rule of a grammar identified by a URI	`<ruleref uri="grammarURI"/>`	**No**
2.2.2 Explicit reference to a named rule of a grammar identified by a URI with a media type	`<ruleref uri="grammarURI#rulename" type="media-type"/>`	**No**
2.2.2 Implicit reference to the root rule of a grammar identified by a URI with a media type	`<ruleref uri="grammarURI" type="media-type"/>`	**No**

2.2.3 Special rule definitions	`<ruleref special="NULL"/>`
	`<ruleref special="VOID"/>`
	`<ruleref special="GARBAGE"/>`
		No

Notes

1. Grammar URI is an external URI. For example, `http://grammar.example.com/world-cities.grxml`.
2. Media type can be:
   - `application/srgs+xml`
   - `text/plain`

Example

```
<rule id="city" scope="public">
    <one-of>
```

```
        <item>Boston</item>
        <item>Philadelphia</item>
        <item>Fargo</item>
    </one-of>
</rule>

<rule id="state" scope="public">
    <one-of>
        <item>FL</item>
        <item>MA</item>
        <item>NY</item>
    </one-of>
</rule>

<!-- "Boston MA" -> city = Boston, state = MA -->
<rule id="city_state" scope="public">
    <ruleref uri="#city"/> <ruleref uri="#state"/>
</rule>
```

Sequences and encapsulation

The following example shows the supported sequences. For more information, see Sequences and encapsulation in the *Speech recognition grammar specification version 1* W3C recommendation.

Example

```
<!-- sequence of tokens -->
this is a test

<!--sequence of rule references-->
<ruleref uri="#action"/> <ruleref uri="#object"/>

<!--sequence of tokens and rule references-->
the <ruleref uri="#object"/> is <ruleref uri="#color"/>

<!-- sequence container -->
<item>fly to <ruleref uri="#city"/> </item>
```

Repeats

The following table shows the supported repeated expansions for rules. For more information, see Repeats in the *Speech recognition grammar specification version 1* W3C recommendation.

XML form Example	Behavior	Supported?
repeat="n" repeat="6"	The contained expression is repeated exactly "n" times. "n" must be "0" or a positive integer.	Yes
repeat="m-n" repeat="4-6"	The contained expansion is repeated between "m" and "n" times (inclusive). "m" and "n" must both be "0" or a positive integer, and "m" must be less than or equal to "n".	Yes
repeat="m-" repeat="3-"	The contained expansion is repeated "m" times or more (inclusive). "m" must be "0" or a postive integer. For example, "3-" declares that the contained expansion can occur three, four, five, or more times.	Yes
repeat="0-1"	The contained expansion is optional.	Yes
<item repeat="2-4" repeat-prob="0.8">		**No**

Language

The following discussion applies to language identifiers applied to grammars. For more information, see Language in the *Speech recognition grammar specification version 1* W3C recommendation.

By default a grammar is a single language document with a language identifier provided in the language declaration in the grammar header. All tokens within that grammar, unless otherwise declared, **will be handled according to the grammar's language**. Grammar-level language declarations **are not supported**.

In the following example:

1. The grammar header declaration for the language "en-US" **is supported** by Amazon Lex V2.
2. Item-level language attachment (highlighted in *red*) **is not supported**. Amazon Lex V2 throws a validation error if a language attachment is different from the header declaration.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
                  "http://www.w3.org/TR/speech-grammar/grammar.dtd">

<!-- the default grammar language is US English -->
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0">

  <!--
     single language attachment to tokens
     "yes" inherits US English language
     "oui" is Canadian French language
  -->
  <rule id="yes">
    <one-of>
      <item>yes</item>
      <item xml:lang="fr-CA">oui</item>
    </one-of>
  </rule>

  <!-- Single language attachment to an expansion -->
  <rule id="people1">
    <one-of xml:lang="fr-CA">
      <item>Michel Tremblay</item>
      <item>André Roy</item>
    </one-of>
```

```
  </rule>
</grammar>
```

Tags

The following discussion applies to tags defined for grammars. For more information, see Tags in the *Speech recognition grammar specification version 1* W3C recommendation.

Based on the SRGS specification, tags can be defined in the following ways:

1. As part of a header declaration as described in Header declarations.
2. As part of a *<rule>* definition.

The following tag formats are supported:

- `semantics/1.0` (SISR, ECMAScript)
- `semantics/1.0-literals` (SISR string literals)

The following tag formats are not supported:

- `swi-semantics/1.0` (Nuance proprietary)

Example

```
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xml:base="http://www.example.com/base-file-path"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US"
         version="1.0"
         mode="voice"
         root="city"
         tag-format="semantics/1.0-literals">
    <rule id="no">
        <one-of>
            <item>no</item>
            <item>nope</item>
            <item>no way</item>
        </one-of>
```

```
        <tag>no</tag>
    </rule>
</grammar>
```

Weights

You can add the *weight* attribute to an element. Weight is a positive floating point value that represents the degree to which the phrase in the item is boosted during speech recognition. For more information, see Weights in the Speech recognition grammar specification version 1 W3C recommendation.

Weights must be greater than 0 and less than or equal to 10, and can have only one decimal place. If the weight is greater than 0 and less than 1, the phrase is negatively boosted. If the weight is greater than 1 and less than or equal to 10, the phrase is positively boosted. A weight of 1 is equivalent to giving no weight at all, and there is no boosting for the phrase.

Assigning appropriate weights to items for improving speech recognition performance is a difficult task. Here are some tips you can follow for assigning weights:

- Start with a grammar without item weights assigned.
- Determine which patterns in the speech are frequently misidentified.
- Apply different values for weights until you notice an improvement in the speech recognition performance, and there are no regressions.

Example 1

For example, if you have a grammar for airports, and you observe that *New York* is frequently misidentified as *Newark*, you can positively boost New York by assigning it a weight of 5.

```
<rule> id="airport">
    <one-of>
        <item>
            Boston
            <tag>out="Boston"</tag>
        </item>
        <item weight="5">
            New York
            <tag>out="New York"</tag>
        </item>
        <item>
```

```
            Newark
            <tag>out="Newark"</tag>
        </item>
    </one-of>
</rule>
```

Example 2

For example, you have a grammar for the airline reservation code starting with an English alphabet followed by three digits. The reservation code most likely starts with B or D, but you observe that B is frequently misidentified as P, and D as T. You can positively boost B and D.

```
<rule> id="alphabet">
    <one-of>
        <item>A<tag>out.letters+='A';</tag></item>
        <item weight="3.5">B<tag>out.letters+='B';</tag></item>
        <item>C<tag>out.letters+='C';</tag></item>
        <item weight="2.9">D<tag>out.letters+='D';</tag></item>
        <item>E<tag>out.letters+='E';</tag></item>
        <item>F<tag>out.letters+='F';</tag></item>
        <item>G<tag>out.letters+='G';</tag></item>
        <item>H<tag>out.letters+='H';</tag></item>
        <item>I<tag>out.letters+='I';</tag></item>
        <item>J<tag>out.letters+='J';</tag></item>
        <item>K<tag>out.letters+='K';</tag></item>
        <item>L<tag>out.letters+='L';</tag></item>
        <item>M<tag>out.letters+='M';</tag></item>
        <item>N<tag>out.letters+='N';</tag></item>
        <item>O<tag>out.letters+='O';</tag></item>
        <item>P<tag>out.letters+='P';</tag></item>
        <item>Q<tag>out.letters+='Q';</tag></item>
        <item>R<tag>out.letters+='R';</tag></item>
        <item>S<tag>out.letters+='S';</tag></item>
        <item>T<tag>out.letters+='T';</tag></item>
        <item>U<tag>out.letters+='U';</tag></item>
        <item>V<tag>out.letters+='V';</tag></item>
        <item>W<tag>out.letters+='W';</tag></item>
        <item>X<tag>out.letters+='X';</tag></item>
        <item>Y<tag>out.letters+='Y';</tag></item>
        <item>Z<tag>out.letters+='Z';</tag></item>
    </one-of>
</rule>
```

Script format

Amazon Lex V2 supports the following ECMAScript features for defining grammars.

Amazon Lex V2 supports the following ECMAScript features when specifying tags in the grammar. `tag-format` must be sent to `semantics/1.0` when ECMAScript tags are used in the grammar. For more information, see the ECMA-262 ECMAScript 2021 language specification .

```
<grammar version="1.0"
xmlns="http://www.w3.org/2001/06/grammar"
xml:lang="en-US"
tag-format="semantics/1.0"
root="card_number">
```

Topics

- Variable statement
- Expressions
- If statement
- Switch statement
- Function declarations
- Iteration statement
- Block statement
- Comments
- Unsupported statements

This document contains material from the ECMAScript standard (available at https://www.ecma-international.org/publications-and-standards/standards/ecma-262/). The ECMAScript language specification document is available from Ecma International under the following license.

License text

above copyright notice and this Copyright License and Disclaimer are included on all such copies and derivative works. The only derivative works that are permissible under this Copyright License and Disclaimer are:

(i) works which incorporate all or portion of this document for the purpose of providing commentary or explanation (such as an annotated version of the document),

(ii) works which incorporate all or portion of this document for the purpose of incorporating features that provide accessibility,

(iii) translations of this document into languages other than English and into different formats and

(iv) works by making use of this specification in standard conformant products by implementing (for example, by copy and paste wholly or partly) the functionality therein.

However, the content of this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as required to translate it into languages other than English or into a different format.

The official version of an Ecma International document is the English language version on the Ecma International website. In the event of discrepancies between a translated version and the official version, the official version shall govern.

The limited permissions granted above are perpetual and will not be revoked by Ecma International or its successors or assigns. This document and the information contained herein is provided on an "AS IS" basis and ECMA INTERNATIONAL DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE."

Variable statement

A variable statement defines one or more variables.

```
var x = 10;
var x = 10, var y = <expression>;
```

Expressions

You can add expressions strings to perform functions in Amazon Lex V2. This table shows the syntax and examples that can be used in SRGS expressions.

Expression type	Syntax	Example	Supported?
Regular expression literal	String literal containing valid regex special characters	`"^\d\.$"`	**No**
Function	`function functionName(parameters) { functionBody}`	`var x = function calc() { return 10; }`	**No**
Delete	`delete expression`	`delete obj.property;`	**No**
Void	`void expression`	`void (2 == '2');`	**No**
Typeof	`typeof expression`	`typeof 42;`	**No**
Member index	`expression [ expressions ]`	`var fruits = ["apple"]; fruits[0];`	Yes
Member dot	`expression . identifier`	`out.value`	yes
Arguments	`expression (arguments)`	`new Date('1994-10-11')`	Yes
Post increment	`expression++`	`var x=10; x++;`	Yes
Post decrement	`expression--`	`var x=10; x--;`	Yes

Expression type	Syntax	Example	Supported?
Pre increment	`++expression`	`var x=10; ++x;`	Yes
Pre decrement	`--expression`	`var x=10; --x;`	Yes
Unary plus / Unary minus	`+expression / -expression`	`+x / -x;`	Yes
Bit not	`~ expression`	`const a = 5; consol e.log( ~a );`	Yes
Logical not	`! expression`	`!(a > 0 || b > 0)`	Yes
Multiplicative	`expression ('*' | '/' | '%') expression`	`(x + y) * (a / b)`	Yes
Additive	`expression ('+' | '-') expression`	`(a + b) - (a - (a + b))`	Yes
Bit shift	`expression ('<<' | '>>' | '>>>') expression`	`(a >> b) >>> c`	Yes
Relative	`expression ('<' | '>' | '<=' | '>=') expression`	`if (a > b) { ... }`	Yes
In	`expression in expression`	`fruits[0] in otherFruits;`	Yes

Expression type	Syntax	Example	Supported?
Equality	`expression ('==' | '!=' | '===' | '!==') expression`	`if (a == b) { ... }`	Yes
Bit and / xor / or	`expression ('&' | '^' | '|') expression`	`a & b / a ^ b / a | b`	Yes
Logical and / or	`expression ('&&' | '||') expression`	`if (a && (b ||c)) { ...}`	Yes
Ternary	`expression ? expression : expression`	`a > b ? obj.prop : 0`	Yes
Assignment	`expression = expression`	`out.value = "string";`	Yes
Assignment operator	`expression ('*=' | '/=' | '+=' | '-=' | '%=') expression`	`a *= 10;`	Yes
Assignment bitwise operator	`expression ('<<=' | '>>=' | '>>>=' | '&=' | '^=' | '|=') expression`	`a <<= 10;`	Yes

Expression type	Syntax	Example	Supported?
Identifier	`identifierSequence` where *identifierSequence* is a sequence of valid characters	`fruits=[10, 20, 30];`	Yes
Null literal	`null`	`x = null;`	Yes
Boolean literal	`true | false`	`x = true;`	Yes
String literal	`'string' / "string"`	`a = 'hello', b = "world";`	Yes
Decimal literal	`integer [.] digits [exponent]`	`111.11 e+12`	Yes
Hex literal	`0 (x | X)[0-9a-fA-F]`	`0x123ABC`	Yes
Octal literal	`O [0-7]`	`"O51"`	Yes
Array literal	`[ expression, ... ]`	`v = [a, b, c];`	Yes
Object literal	`{property: value, ...}`	`out = {value: 1, flag: false};`	Yes
Parenthesized	`( expressions )`	`x + (x + y)`	Yes

If statement

You can add if statements to perform functions in Amazon Lex V2. This example shows the syntax that can be used in SRGS expressions.

```
if (expressions) {
    statements;
} else {
    statements;
}
```

Note: In the preceding example, `expressions` and `statements` must be one of the supported ones from this document.

Switch statement

You can add switch statements to perform functions in Amazon Lex V2. This example shows the syntax that can be used in SRGS expressions.

```
switch (expression) {
    case (expression):
     statements
     .
     .
     .
   default:
      statements
}
```

Note: In the preceding example, `expressions` and `statements` must be one of the supported ones from this document.

Function declarations

You can add function declarations to perform functions in Amazon Lex V2. This example shows the syntax that can be used in SRGS expressions.

```
function functionIdentifier([parameterList, ...]) {
     <function body>
}
```

Iteration statement

Iteration statements can be any one of the following:

```
// Do..While statement
do {
   statements
} while (expressions)


// While Loop
while (expressions) {
   statements
}

// For Loop
for ([initialization]; [condition]; [final-expression])
   statement

// For..In
for (variable in object) {
  statement
}
```

Block statement

You can add statement blocks to perform functions in Amazon Lex V2. This example shows the syntax that can be used in SRGS expressions.

```
{
   statements
}

// Example
{
    x = 10;
   if (x > 10) {
     console.log("greater than 10");
   }
}
```

Note: In the preceding example, `statements` provided in the block must be one of the supported ones from this document.

Comments

You can add comments in Amazon Lex V2. This example shows the syntax that can be used in SRGS expressions.

```
// Single Line Comments
"// <comment>"

// Multineline comments
/**
<comment>
**/
```

Unsupported statements

Amazon Lex V2 doesn't support the following ECMAScript features.

Topics

- Empty statement
- Continue statement
- Break statement
- Return statement
- Throw statement
- Try statement
- Debugger statement
- Labeled statement
- Class declaration

Empty statement

The empty statement is used to provide no statement. The following is the syntax for an empty statement:

```
;
```

Continue statement

The continue statement without a label is supported with the Iteration statement. The continue statement with a label isn't supported.

```
// continue with label
// this allows the program to jump to a
// labelled statement (see labelled statement below)
continue <label>;
```

Break statement

The break statement without a label is supported with the Iteration statement. The break statement with a label isn't supported.

```
// break with label
// this allows the program to break out of a
// labelled statement (see labelled statement below)
break <label>;
```

Return statement

```
return expression;
```

Throw statement

The throw statement is used to throw a user-defined exception.

```
throw expression;
```

Try statement

```
try {
  statements
}
catch (expression) {
  statements
}
finally {
  statements
}
```

Debugger statement

The debugger statement is used to invoke debugging functionality provided by the environment.

```
debugger;
```

Labeled statement

The labeled statement can be used with `break` or `continue` statements.

```
label:
   statements


// Example
let str = '';

loop1:
for (let i = 0; i < 5; i++) {
  if (i === 1) {
    continue loop1;
  }
  str = str + i;
}

console.log(str);
```

Class declaration

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Industry grammars

Industry grammars are a set of XML files to use with the grammar slot type. You can use these to quickly deliver a consistent end-user experience as you migrate interactive voice response work flows to Amazon Lex V2. You can select from a range of pre-built grammars across three domains:

financial services, insurance, and telecom. There is also a generic set of grammars that you can use as a starting point for your own grammars.

The grammars contain the rules to collect the information and the ECMAScript tags for semantic interpretation.

Grammars for financial services (download)

The following grammars are supported for financial services: account and routing numbers, credit card and loan numbers, credit score, account opening and closing dates, and Social Security number.

Account number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">


        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: My account number is A B C 1 2 3 4
                Output: ABC1234

            Scenario 2:
                Input: My account number is 1 2 3 4 A B C
                Output: 1234ABC

            Scenario 3:
                Input: Hmm My account number is 1 2 3 4 A B C 1
                Output: 123ABC1
        -->

        <rule id="main" scope="public">
```

```
            <tag>out=""</tag>
            <item><ruleref uri="#alphanumeric"/><tag>out +=
rules.alphanumeric.alphanum;</tag></item>
            <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
rules.digits.numbers</tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">account number is</item>
                <item repeat="0-1">Account Number</item>
                <item repeat="0-1">Here is my Account Number </item>
                <item repeat="0-1">Yes, It is</item>
                <item repeat="0-1">Yes It is</item>
                <item repeat="0-1">Yes It's</item>
                <item repeat="0-1">My account Id is</item>
                <item repeat="0-1">This is the account Id</item>
                <item repeat="0-1">account Id</item>
            </one-of>
        </rule>

        <rule id="hesitation">
          <one-of>
             <item>Hmm</item>
             <item>Mmm</item>
             <item>My</item>
          </one-of>
        </rule>

        <rule id="alphanumeric" scope="public">
            <tag>out.alphanum=""</tag>
            <item><ruleref uri="#alphabets"/><tag>out.alphanum +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum +=
rules.digits.numbers</tag></item>
        </rule>

        <rule id="alphabets">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.letters=""</tag>
            <tag>out.firstOccurence=""</tag>
```

```
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
            <item repeat="1-">
                <one-of>
                    <item>A<tag>out.letters+='A';</tag></item>
                    <item>B<tag>out.letters+='B';</tag></item>
                    <item>C<tag>out.letters+='C';</tag></item>
                    <item>D<tag>out.letters+='D';</tag></item>
                    <item>E<tag>out.letters+='E';</tag></item>
                    <item>F<tag>out.letters+='F';</tag></item>
                    <item>G<tag>out.letters+='G';</tag></item>
                    <item>H<tag>out.letters+='H';</tag></item>
                    <item>I<tag>out.letters+='I';</tag></item>
                    <item>J<tag>out.letters+='J';</tag></item>
                    <item>K<tag>out.letters+='K';</tag></item>
                    <item>L<tag>out.letters+='L';</tag></item>
                    <item>M<tag>out.letters+='M';</tag></item>
                    <item>N<tag>out.letters+='N';</tag></item>
                    <item>O<tag>out.letters+='O';</tag></item>
                    <item>P<tag>out.letters+='P';</tag></item>
                    <item>Q<tag>out.letters+='Q';</tag></item>
                    <item>R<tag>out.letters+='R';</tag></item>
                    <item>S<tag>out.letters+='S';</tag></item>
                    <item>T<tag>out.letters+='T';</tag></item>
                    <item>U<tag>out.letters+='U';</tag></item>
                    <item>V<tag>out.letters+='V';</tag></item>
                    <item>W<tag>out.letters+='W';</tag></item>
                    <item>X<tag>out.letters+='X';</tag></item>
                    <item>Y<tag>out.letters+='Y';</tag></item>
                    <item>Z<tag>out.letters+='Z';</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.numbers=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.numbers+=0;</tag></item>
                    <item>1<tag>out.numbers+=1;</tag></item>
                    <item>2<tag>out.numbers+=2;</tag></item>
                    <item>3<tag>out.numbers+=3;</tag></item>
                    <item>4<tag>out.numbers+=4;</tag></item>
```

```
                    <item>5<tag>out.numbers+=5;</tag></item>
                    <item>6<tag>out.numbers+=6;</tag></item>
                    <item>7<tag>out.numbers+=7;</tag></item>
                    <item>8<tag>out.numbers+=8;</tag></item>
                    <item>9<tag>out.numbers+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Routing number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My routing number is 1 2 3 4 5 6 7 8 9
                 Output: 123456789

             Scenario 2:
                 Input: routing number 1 2 3 4 5 6 7 8 9
                 Output: 123456789

         -->

        <rule id="digits">
            <tag>out=""</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
```

```
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
       <one-of>
          <item repeat="0-1">My routing number</item>
          <item repeat="0-1">Routing number of</item>
          <item repeat="0-1">The routing number is</item>
       </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
           <item>Hmm</item>
           <item>Mmm</item>
           <item>My</item>
        </one-of>
      </rule>

    <rule id="singleDigit">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.digit=""</tag>
        <item repeat="16">
            <one-of>
                <item>0<tag>out.digit+=0;</tag></item>
                <item>zero<tag>out.digit+=0;</tag></item>
                <item>1<tag>out.digit+=1;</tag></item>
                <item>one<tag>out.digit+=1;</tag></item>
                <item>2<tag>out.digit+=2;</tag></item>
                <item>two<tag>out.digit+=2;</tag></item>
                <item>3<tag>out.digit+=3;</tag></item>
                <item>three<tag>out.digit+=3;</tag></item>
                <item>4<tag>out.digit+=4;</tag></item>
                <item>four<tag>out.digit+=4;</tag></item>
                <item>5<tag>out.digit+=5;</tag></item>
                <item>five<tag>out.digit+=5;</tag></item>
                <item>6<tag>out.digit+=6;</tag></item>
                <item>six<tag>out.digit+=5;</tag></item>
                <item>7<tag>out.digit+=7;</tag></item>
                <item>seven<tag>out.digit+=7;</tag></item>
                <item>8<tag>out.digit+=8;</tag></item>
                <item>eight<tag>out.digit+=8;</tag></item>
                <item>9<tag>out.digit+=9;</tag></item>
                <item>nine<tag>out.digit+=9;</tag></item>
            </one-of>
        </item>
    </rule>
```

```
</grammar>
```

Credit card number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My credit card number is 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7
                 Output: 1234567891234567

             Scenario 2:
                 Input: card number 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7
                 Output: 1234567891234567

         -->

        <rule id="digits">
            <tag>out=""</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">My credit card number is</item>
              <item repeat="0-1">card number</item>
           </one-of>
        </rule>

        <rule id="hesitation">
```

```
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

        <rule id="singleDigit">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.digit=""</tag>
            <item repeat="16">
                <one-of>
                    <item>0<tag>out.digit+=0;</tag></item>
                    <item>zero<tag>out.digit+=0;</tag></item>
                    <item>1<tag>out.digit+=1;</tag></item>
                    <item>one<tag>out.digit+=1;</tag></item>
                    <item>2<tag>out.digit+=2;</tag></item>
                    <item>two<tag>out.digit+=2;</tag></item>
                    <item>3<tag>out.digit+=3;</tag></item>
                    <item>three<tag>out.digit+=3;</tag></item>
                    <item>4<tag>out.digit+=4;</tag></item>
                    <item>four<tag>out.digit+=4;</tag></item>
                    <item>5<tag>out.digit+=5;</tag></item>
                    <item>five<tag>out.digit+=5;</tag></item>
                    <item>6<tag>out.digit+=6;</tag></item>
                    <item>six<tag>out.digit+=5;</tag></item>
                    <item>7<tag>out.digit+=7;</tag></item>
                    <item>seven<tag>out.digit+=7;</tag></item>
                    <item>8<tag>out.digit+=8;</tag></item>
                    <item>eight<tag>out.digit+=8;</tag></item>
                    <item>9<tag>out.digit+=9;</tag></item>
                    <item>nine<tag>out.digit+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Loan ID

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
```

```
                            http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">


       <!-- Test Cases

       Grammar will support the following inputs:

           Scenario 1:
               Input: My loan Id is A B C 1 2 3 4
               Output: ABC1234
       -->

       <rule id="main" scope="public">
           <tag>out=""</tag>
           <item><ruleref uri="#alphanumeric"/><tag>out +=
rules.alphanumeric.alphanum;</tag></item>
           <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out +=
rules.alphabets.letters;</tag></item>
           <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
rules.digits.numbers</tag></item>
       </rule>

       <rule id="text">
           <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
               <item repeat="0-1">my loan number is</item>
               <item repeat="0-1">The loan number</item>
               <item repeat="0-1">The loan is </item>
               <item repeat="0-1">The number is</item>
               <item repeat="0-1">loan number</item>
               <item repeat="0-1">loan number of</item>
               <item repeat="0-1">loan Id is</item>
               <item repeat="0-1">My loan Id is</item>
           </one-of>
       </rule>

       <rule id="hesitation">
         <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
```

```
            <item>My</item>
          </one-of>
        </rule>

        <rule id="alphanumeric" scope="public">
            <tag>out.alphanum=""</tag>
            <item><ruleref uri="#alphabets"/><tag>out.alphanum +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum +=
rules.digits.numbers</tag></item>
        </rule>

        <rule id="alphabets">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.letters=""</tag>
            <tag>out.firstOccurence=""</tag>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
            <item repeat="1-">
                <one-of>
                    <item>A<tag>out.letters+='A';</tag></item>
                    <item>B<tag>out.letters+='B';</tag></item>
                    <item>C<tag>out.letters+='C';</tag></item>
                    <item>D<tag>out.letters+='D';</tag></item>
                    <item>E<tag>out.letters+='E';</tag></item>
                    <item>F<tag>out.letters+='F';</tag></item>
                    <item>G<tag>out.letters+='G';</tag></item>
                    <item>H<tag>out.letters+='H';</tag></item>
                    <item>I<tag>out.letters+='I';</tag></item>
                    <item>J<tag>out.letters+='J';</tag></item>
                    <item>K<tag>out.letters+='K';</tag></item>
                    <item>L<tag>out.letters+='L';</tag></item>
                    <item>M<tag>out.letters+='M';</tag></item>
                    <item>N<tag>out.letters+='N';</tag></item>
                    <item>O<tag>out.letters+='O';</tag></item>
                    <item>P<tag>out.letters+='P';</tag></item>
                    <item>Q<tag>out.letters+='Q';</tag></item>
                    <item>R<tag>out.letters+='R';</tag></item>
                    <item>S<tag>out.letters+='S';</tag></item>
                    <item>T<tag>out.letters+='T';</tag></item>
                    <item>U<tag>out.letters+='U';</tag></item>
                    <item>V<tag>out.letters+='V';</tag></item>
                    <item>W<tag>out.letters+='W';</tag></item>
                    <item>X<tag>out.letters+='X';</tag></item>
```

```
                    <item>Y<tag>out.letters+='Y';</tag></item>
                    <item>Z<tag>out.letters+='Z';</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.numbers=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.numbers+=0;</tag></item>
                    <item>1<tag>out.numbers+=1;</tag></item>
                    <item>2<tag>out.numbers+=2;</tag></item>
                    <item>3<tag>out.numbers+=3;</tag></item>
                    <item>4<tag>out.numbers+=4;</tag></item>
                    <item>5<tag>out.numbers+=5;</tag></item>
                    <item>6<tag>out.numbers+=6;</tag></item>
                    <item>7<tag>out.numbers+=7;</tag></item>
                    <item>8<tag>out.numbers+=8;</tag></item>
                    <item>9<tag>out.numbers+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Credit score

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
```

```
                Input: The number is fifteen
                Output: 15

             Scenario 2:
                Input: My credit score is fifteen
                Output: 15
         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <one-of>
              <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></
item>
              <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></
item>
              <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
            </one-of>
        </rule>

        <rule id="text">
           <one-of>
              <item repeat="0-1">Credit score is</item>
              <item repeat="0-1">Last digits are</item>
              <item repeat="0-1">The number is</item>
              <item repeat="0-1">That's</item>
              <item repeat="0-1">It is</item>
              <item repeat="0-1">My credit score is</item>
           </one-of>
        </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>0<tag>out=0;</tag></item>
                <item>1<tag>out=1;</tag></item>
                <item>2<tag>out=2;</tag></item>
                <item>3<tag>out=3;</tag></item>
                <item>4<tag>out=4;</tag></item>
                <item>5<tag>out=5;</tag></item>
                <item>6<tag>out=6;</tag></item>
                <item>7<tag>out=7;</tag></item>
                <item>8<tag>out=8;</tag></item>
                <item>9<tag>out=9;</tag></item>
```

```
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
            <item>nineteen<tag>out=19;</tag></item>
            <item>10<tag>out=10;</tag></item>
            <item>11<tag>out=11;</tag></item>
            <item>12<tag>out=12;</tag></item>
            <item>13<tag>out=13;</tag></item>
            <item>14<tag>out=14;</tag></item>
            <item>15<tag>out=15;</tag></item>
            <item>16<tag>out=16;</tag></item>
            <item>17<tag>out=17;</tag></item>
            <item>18<tag>out=18;</tag></item>
            <item>19<tag>out=19;</tag></item>
        </one-of>
    </rule>

    <rule id="above_twenty">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>twenty<tag>out=20;</tag></item>
            <item>thirty<tag>out=30;</tag></item>
            <item>forty<tag>out=40;</tag></item>
```

```
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
                <item>20<tag>out=20;</tag></item>
                <item>30<tag>out=30;</tag></item>
                <item>40<tag>out=40;</tag></item>
                <item>50<tag>out=50;</tag></item>
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>

</grammar>
```

Account opening date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: I opened account on July Two Thousand and Eleven
                 Output: 07/11

             Scenario 2:
                 Input: I need account number opened on July Two Thousand and Eleven
                 Output: 07/11
```

```
         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + "/";</tag></item>
                <one-of>
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">I opened account on </item>
              <item repeat="0-1">I need account number opened on </item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>
        <rule id="months">
           <item repeat="0-1"><ruleref uri="#text"/></item>
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="01";</tag></item>
               <item>february<tag>out.mon+="02";</tag></item>
               <item>march<tag>out.mon+="03";</tag></item>
               <item>april<tag>out.mon+="04";</tag></item>
```

```
            <item>may<tag>out.mon+="05";</tag></item>
            <item>june<tag>out.mon+="06";</tag></item>
            <item>july<tag>out.mon+="07";</tag></item>
            <item>august<tag>out.mon+="08";</tag></item>
            <item>september<tag>out.mon+="09";</tag></item>
            <item>october<tag>out.mon+="10";</tag></item>
            <item>november<tag>out.mon+="11";</tag></item>
            <item>december<tag>out.mon+="12";</tag></item>
            <item>jan<tag>out.mon+="01";</tag></item>
            <item>feb<tag>out.mon+="02";</tag></item>
            <item>aug<tag>out.mon+="08";</tag></item>
            <item>sept<tag>out.mon+="09";</tag></item>
            <item>oct<tag>out.mon+="10";</tag></item>
            <item>nov<tag>out.mon+="11";</tag></item>
            <item>dec<tag>out.mon+="12";</tag></item>
        </one-of>
    </rule>

    <rule id="digits">
        <one-of>
            <item>zero<tag>out=0;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
```

```
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand<!--<tag>out=2000;</tag>--></item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Automatic pay date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">
```

```
         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: I want to schedule auto pay for twenty five Dollar
                 Output: $25

             Scenario 2:
                 Input: Setup automatic payments for twenty five dollars
                 Output: $25

         -->

        <rule id="main" scope="public">
            <tag>out="$"</tag>
            <one-of>
                <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
                <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</
tag></item>
            </one-of>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">I want to schedule auto pay for</item>
              <item repeat="0-1">Setup automatic payments for twenty five dollars</
item>
              <item repeat="0-1">Auto pay amount of</item>
              <item repeat="0-1">Set it up for</item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

        <rule id="digits">
```

```
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.num = 0;</tag>
        <one-of>
            <item>0<tag>out.num+=0;</tag></item>
            <item>1<tag>out.num+=1;</tag></item>
            <item>2<tag>out.num+=2;</tag></item>
            <item>3<tag>out.num+=3;</tag></item>
            <item>4<tag>out.num+=4;</tag></item>
            <item>5<tag>out.num+=5;</tag></item>
            <item>6<tag>out.num+=6;</tag></item>
            <item>7<tag>out.num+=7;</tag></item>
            <item>8<tag>out.num+=8;</tag></item>
            <item>9<tag>out.num+=9;</tag></item>
            <item>one<tag>out.num+=1;</tag></item>
            <item>two<tag>out.num+=2;</tag></item>
            <item>three<tag>out.num+=3;</tag></item>
            <item>four<tag>out.num+=4;</tag></item>
            <item>five<tag>out.num+=5;</tag></item>
            <item>six<tag>out.num+=6;</tag></item>
            <item>seven<tag>out.num+=7;</tag></item>
            <item>eight<tag>out.num+=8;</tag></item>
            <item>nine<tag>out.num+=9;</tag></item>
        </one-of>
        <item repeat="0-1"><ruleref uri="#currency"/></item>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.teen = 0;</tag>
        <one-of>
            <item>ten<tag>out.teen+=10;</tag></item>
            <item>eleven<tag>out.teen+=11;</tag></item>
            <item>twelve<tag>out.teen+=12;</tag></item>
            <item>thirteen<tag>out.teen+=13;</tag></item>
            <item>fourteen<tag>out.teen+=14;</tag></item>
            <item>fifteen<tag>out.teen+=15;</tag></item>
            <item>sixteen<tag>out.teen+=16;</tag></item>
            <item>seventeen<tag>out.teen+=17;</tag></item>
            <item>eighteen<tag>out.teen+=18;</tag></item>
            <item>nineteen<tag>out.teen+=19;</tag></item>
        </one-of>
        <item repeat="0-1"><ruleref uri="#currency"/></item>
    </rule>
```

```
        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.tens = 0;</tag>
            <one-of>
                <item>twenty<tag>out.tens+=20;</tag></item>
                <item>thirty<tag>out.tens+=30;</tag></item>
                <item>forty<tag>out.tens+=40;</tag></item>
                <item>fifty<tag>out.tens+=50;</tag></item>
                <item>sixty<tag>out.tens+=60;</tag></item>
                <item>seventy<tag>out.tens+=70;</tag></item>
                <item>eighty<tag>out.tens+=80;</tag></item>
                <item>ninety<tag>out.tens+=90;</tag></item>
                <item>hundred<tag>out.tens+=100;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens +=
 rules.digits.num;</tag></item>
        </rule>

        <rule id="currency">
            <one-of>
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>


        <rule id="sub_hundred">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></
item>
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh +=
 rules.above_twenty.tens;</tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></
item>
            </one-of>
         </rule>
```

```
        <rule id="subThousands">
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</
tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
</grammar>
```

Credit card expiration date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="dateCardExpiration"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="dateCardExpiration" scope="public">
            <tag>out="";</tag>
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months;</
tag></item>
            <item repeat="1"><ruleref uri="#year"/><tag>out += " " + rules.year.yr;</
tag></item>
        </rule>

        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: My card expiration date is july eleven
                Output: 07 2011

            Scenario 2:
                Input: My card expiration date is may twenty six
```

```
        Output: 05 2026

-->

<rule id="text">
   <item repeat="0-1"><ruleref uri="#hesitation"/></item>
   <one-of>
      <item repeat="0-1">My card expiration date is </item>
      <item repeat="0-1">Expiration date is </item>
   </one-of>
</rule>

<rule id="hesitation">
  <one-of>
     <item>Hmm</item>
     <item>Mmm</item>
     <item>My</item>
  </one-of>
</rule>

<rule id="months">
   <item repeat="0-1"><ruleref uri="#text"/></item>
   <one-of>
       <item>january<tag>out="01";</tag></item>
       <item>february<tag>out="02";</tag></item>
       <item>march<tag>out="03";</tag></item>
       <item>april<tag>out="04";</tag></item>
       <item>may<tag>out="05";</tag></item>
       <item>june<tag>out="06";</tag></item>
       <item>july<tag>out="07";</tag></item>
       <item>august<tag>out="08";</tag></item>
       <item>september<tag>out="09";</tag></item>
       <item>october<tag>out="10";</tag></item>
       <item>november<tag>out="11";</tag></item>
       <item>december<tag>out="12";</tag></item>
       <item>jan<tag>out="01";</tag></item>
       <item>feb<tag>out="02";</tag></item>
       <item>aug<tag>out="08";</tag></item>
       <item>sept<tag>out="09";</tag></item>
       <item>oct<tag>out="10";</tag></item>
       <item>nov<tag>out="11";</tag></item>
       <item>dec<tag>out="12";</tag></item>
       <item>1<tag>out="01";</tag></item>
       <item>2<tag>out="02";</tag></item>
```

```
                <item>3<tag>out="03";</tag></item>
                <item>4<tag>out="04";</tag></item>
                <item>5<tag>out="05";</tag></item>
                <item>6<tag>out="06";</tag></item>
                <item>7<tag>out="07";</tag></item>
                <item>8<tag>out="08";</tag></item>
                <item>9<tag>out="09";</tag></item>
                <item>ten<tag>out="10";</tag></item>
                <item>eleven<tag>out="11";</tag></item>
                <item>twelve<tag>out="12";</tag></item>
            </one-of>
        </rule>

         <rule id="digits">
             <item repeat="0-1"><ruleref uri="#text"/></item>
             <one-of>
                 <item>0<tag>out=0;</tag></item>
                 <item>1<tag>out=1;</tag></item>
                 <item>2<tag>out=2;</tag></item>
                 <item>3<tag>out=3;</tag></item>
                 <item>4<tag>out=4;</tag></item>
                 <item>5<tag>out=5;</tag></item>
                 <item>6<tag>out=6;</tag></item>
                 <item>7<tag>out=7;</tag></item>
                 <item>8<tag>out=8;</tag></item>
                 <item>9<tag>out=9;</tag></item>
                 <item>one<tag>out=1;</tag></item>
                 <item>two<tag>out=2;</tag></item>
                 <item>three<tag>out=3;</tag></item>
                 <item>four<tag>out=4;</tag></item>
                 <item>five<tag>out=5;</tag></item>
                 <item>six<tag>out=6;</tag></item>
                 <item>seven<tag>out=7;</tag></item>
                 <item>eight<tag>out=8;</tag></item>
                 <item>nine<tag>out=9;</tag></item>
             </one-of>
         </rule>

         <rule id="year">
           <tag>out.yr="20"</tag>
           <one-of>
               <item><ruleref uri="#teens"/><tag>out.yr += rules.teens;</tag></item>
               <item><ruleref uri="#above_twenty"/><tag>out.yr += rules.above_twenty;</
tag></item>
```

```
      </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
            <item>nineteen<tag>out=19;</tag></item>
            <item>10<tag>out=10;</tag></item>
            <item>11<tag>out=11;</tag></item>
            <item>12<tag>out=12;</tag></item>
            <item>13<tag>out=13;</tag></item>
            <item>14<tag>out=14;</tag></item>
            <item>15<tag>out=15;</tag></item>
            <item>16<tag>out=16;</tag></item>
            <item>17<tag>out=17;</tag></item>
            <item>18<tag>out=18;</tag></item>
            <item>19<tag>out=19;</tag></item>
        </one-of>
    </rule>

    <rule id="above_twenty">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>twenty<tag>out=20;</tag></item>
            <item>thirty<tag>out=30;</tag></item>
            <item>forty<tag>out=40;</tag></item>
            <item>fifty<tag>out=50;</tag></item>
            <item>sixty<tag>out=60;</tag></item>
            <item>seventy<tag>out=70;</tag></item>
            <item>eighty<tag>out=80;</tag></item>
            <item>ninety<tag>out=90;</tag></item>
            <item>20<tag>out=20;</tag></item>
            <item>30<tag>out=30;</tag></item>
            <item>40<tag>out=40;</tag></item>
            <item>50<tag>out=50;</tag></item>
```

```
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Statement date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: Show me statements from July Five Two Thousand and Eleven
                 Output: 07/5/11

             Scenario 2:
                 Input: Show me statements from July Sixteen Two Thousand and Eleven
                 Output: 07/16/11

             Scenario 3:
                 Input: Show me statements from July Thirty Two Thousand and Eleven
                 Output: 07/30/11
         -->

         <rule id="main" scope="public">
             <tag>out=""</tag>
             <item>
```

```
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + "/";</tag></item>
                <one-of>
                  <item><ruleref uri="#digits"/><tag>out += rules.digits + "/";</
tag></item>
                  <item><ruleref uri="#teens"/><tag>out += rules.teens+ "/";</tag></
item>
                  <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
 "/";</tag></item>
                </one-of>
                <one-of>
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
             </item>
        </rule>

        <rule id="text">
          <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">I want to see bank statements from </item>
              <item repeat="0-1">Show me statements from</item>
           </one-of>
        </rule>

        <rule id="hesitation">
           <one-of>
              <item>Hmm</item>
              <item>Mmm</item>
              <item>My</item>
           </one-of>
         </rule>

        <rule id="months">
           <tag>out.mon=""</tag>
        <item repeat="0-1"><ruleref uri="#text"/></item>
           <one-of>
             <item>january<tag>out.mon+="01";</tag></item>
```

```
            <item>february<tag>out.mon+="02";</tag></item>
            <item>march<tag>out.mon+="03";</tag></item>
            <item>april<tag>out.mon+="04";</tag></item>
            <item>may<tag>out.mon+="05";</tag></item>
            <item>june<tag>out.mon+="06";</tag></item>
            <item>july<tag>out.mon+="07";</tag></item>
            <item>august<tag>out.mon+="08";</tag></item>
            <item>september<tag>out.mon+="09";</tag></item>
            <item>october<tag>out.mon+="10";</tag></item>
            <item>november<tag>out.mon+="11";</tag></item>
            <item>december<tag>out.mon+="12";</tag></item>
            <item>jan<tag>out.mon+="01";</tag></item>
            <item>feb<tag>out.mon+="02";</tag></item>
            <item>aug<tag>out.mon+="08";</tag></item>
            <item>sept<tag>out.mon+="09";</tag></item>
            <item>oct<tag>out.mon+="10";</tag></item>
            <item>nov<tag>out.mon+="11";</tag></item>
            <item>dec<tag>out.mon+="12";</tag></item>
        </one-of>
    </rule>

    <rule id="digits">
        <one-of>
            <item>zero<tag>out=0;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
```

```
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand</item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
 </grammar>
```

Transaction date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:
```

```
             Scenario 1:
                 Input: My last incorrect transaction date is july twenty three
                 Output: 07/23

             Scenario 2:
                 Input: My last incorrect transaction date is july fifteen
                 Output: 07/15

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></
item>
                <one-of>
                    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
                    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
                    <item><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">My last incorrect transaction date is</item>
              <item repeat="0-1">It is</item>
           </one-of>
        </rule>
        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

        <rule id="months">
           <item repeat="0-1"><ruleref uri="#text"/></item>
           <tag>out.mon=""</tag>
           <one-of>
```

```
            <item>january<tag>out.mon+="01";</tag></item>
            <item>february<tag>out.mon+="02";</tag></item>
            <item>march<tag>out.mon+="03";</tag></item>
            <item>april<tag>out.mon+="04";</tag></item>
            <item>may<tag>out.mon+="05";</tag></item>
            <item>june<tag>out.mon+="06";</tag></item>
            <item>july<tag>out.mon+="07";</tag></item>
            <item>august<tag>out.mon+="08";</tag></item>
            <item>september<tag>out.mon+="09";</tag></item>
            <item>october<tag>out.mon+="10";</tag></item>
            <item>november<tag>out.mon+="11";</tag></item>
            <item>december<tag>out.mon+="12";</tag></item>
            <item>jan<tag>out.mon+="01";</tag></item>
            <item>feb<tag>out.mon+="02";</tag></item>
            <item>aug<tag>out.mon+="08";</tag></item>
            <item>sept<tag>out.mon+="09";</tag></item>
            <item>oct<tag>out.mon+="10";</tag></item>
            <item>nov<tag>out.mon+="11";</tag></item>
            <item>dec<tag>out.mon+="12";</tag></item>
        </one-of>
    </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>0<tag>out=0;</tag></item>
            <item>1<tag>out=1;</tag></item>
            <item>2<tag>out=2;</tag></item>
            <item>3<tag>out=3;</tag></item>
            <item>4<tag>out=4;</tag></item>
            <item>5<tag>out=5;</tag></item>
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>first<tag>out=01;</tag></item>
            <item>second<tag>out=02;</tag></item>
            <item>third<tag>out=03;</tag></item>
            <item>fourth<tag>out=04;</tag></item>
            <item>fifth<tag>out=05;</tag></item>
            <item>sixth<tag>out=06;</tag></item>
            <item>seventh<tag>out=07;</tag></item>
            <item>eighth<tag>out=08;</tag></item>
            <item>ninth<tag>out=09;</tag></item>
```

```
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
            <item>nineteen<tag>out=19;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
            <item>eleventh<tag>out=11;</tag></item>
            <item>twelveth<tag>out=12;</tag></item>
            <item>thirteenth<tag>out=13;</tag></item>
            <item>fourteenth<tag>out=14;</tag></item>
            <item>fifteenth<tag>out=15;</tag></item>
            <item>sixteenth<tag>out=16;</tag></item>
            <item>seventeenth<tag>out=17;</tag></item>
            <item>eighteenth<tag>out=18;</tag></item>
            <item>nineteenth<tag>out=19;</tag></item>
        </one-of>
    </rule>

    <rule id="above_twenty">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>twenty<tag>out=20;</tag></item>
            <item>thirty<tag>out=30;</tag></item>
```

```
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Transfer amount

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: I want to transfer twenty five Dollar
                 Output: $25

             Scenario 2:
                 Input: transfer twenty five dollars
                 Output: $25

         -->

        <rule id="main" scope="public">
            <tag>out="$"</tag>
            <one-of>
                <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
                <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</
tag></item>
            </one-of>
        </rule>

        <rule id="text">
```

```
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
       <one-of>
          <item repeat="0-1">I want to transfer</item>
          <item repeat="0-1">transfer</item>
          <item repeat="0-1">make a transfer for</item>
       </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
           <item>Hmm</item>
           <item>Mmm</item>
           <item>My</item>
        </one-of>
      </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.num = 0;</tag>
        <one-of>
            <item>0<tag>out.num+=0;</tag></item>
            <item>1<tag>out.num+=1;</tag></item>
            <item>2<tag>out.num+=2;</tag></item>
            <item>3<tag>out.num+=3;</tag></item>
            <item>4<tag>out.num+=4;</tag></item>
            <item>5<tag>out.num+=5;</tag></item>
            <item>6<tag>out.num+=6;</tag></item>
            <item>7<tag>out.num+=7;</tag></item>
            <item>8<tag>out.num+=8;</tag></item>
            <item>9<tag>out.num+=9;</tag></item>
            <item>one<tag>out.num+=1;</tag></item>
            <item>two<tag>out.num+=2;</tag></item>
            <item>three<tag>out.num+=3;</tag></item>
            <item>four<tag>out.num+=4;</tag></item>
            <item>five<tag>out.num+=5;</tag></item>
            <item>six<tag>out.num+=6;</tag></item>
            <item>seven<tag>out.num+=7;</tag></item>
            <item>eight<tag>out.num+=8;</tag></item>
            <item>nine<tag>out.num+=9;</tag></item>
        </one-of>
        <item repeat="0-1"><ruleref uri="#currency"/></item>
    </rule>

    <rule id="teens">
```

```
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.teen = 0;</tag>
            <one-of>
                <item>ten<tag>out.teen+=10;</tag></item>
                <item>eleven<tag>out.teen+=11;</tag></item>
                <item>twelve<tag>out.teen+=12;</tag></item>
                <item>thirteen<tag>out.teen+=13;</tag></item>
                <item>fourteen<tag>out.teen+=14;</tag></item>
                <item>fifteen<tag>out.teen+=15;</tag></item>
                <item>sixteen<tag>out.teen+=16;</tag></item>
                <item>seventeen<tag>out.teen+=17;</tag></item>
                <item>eighteen<tag>out.teen+=18;</tag></item>
                <item>nineteen<tag>out.teen+=19;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.tens = 0;</tag>
            <one-of>
                <item>twenty<tag>out.tens+=20;</tag></item>
                <item>thirty<tag>out.tens+=30;</tag></item>
                <item>forty<tag>out.tens+=40;</tag></item>
                <item>fifty<tag>out.tens+=50;</tag></item>
                <item>sixty<tag>out.tens+=60;</tag></item>
                <item>seventy<tag>out.tens+=70;</tag></item>
                <item>eighty<tag>out.tens+=80;</tag></item>
                <item>ninety<tag>out.tens+=90;</tag></item>
                <item>hundred<tag>out.tens+=100;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens +=
rules.digits.num;</tag></item>
        </rule>

        <rule id="currency">
            <one-of>
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>
```

```
        <rule id="sub_hundred">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></
item>
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh +=
 rules.above_twenty.tens;</tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></
item>
            </one-of>
         </rule>

        <rule id="subThousands">
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</
tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
</grammar>
```

Social Security number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <rule id="main" scope="public">
```

```
            <tag>out=""</tag>
            <ruleref uri="#digits"/><tag>out += rules.digits.numbers;</tag>
        </rule>

        <rule id="digits">
            <tag>out.numbers=""</tag>
            <item repeat="1-12">
                <one-of>
                    <item>0<tag>out.numbers+=0;</tag></item>
                    <item>1<tag>out.numbers+=1;</tag></item>
                    <item>2<tag>out.numbers+=2;</tag></item>
                    <item>3<tag>out.numbers+=3;</tag></item>
                    <item>4<tag>out.numbers+=4;</tag></item>
                    <item>5<tag>out.numbers+=5;</tag></item>
                    <item>6<tag>out.numbers+=6;</tag></item>
                    <item>7<tag>out.numbers+=7;</tag></item>
                    <item>8<tag>out.numbers+=8;</tag></item>
                    <item>9<tag>out.numbers+=9;</tag></item>
                    <item>zero<tag>out.numbers+=0;</tag></item>
                    <item>one<tag>out.numbers+=1;</tag></item>
                    <item>two<tag>out.numbers+=2;</tag></item>
                    <item>three<tag>out.numbers+=3;</tag></item>
                    <item>four<tag>out.numbers+=4;</tag></item>
                    <item>five<tag>out.numbers+=5;</tag></item>
                    <item>six<tag>out.numbers+=6;</tag></item>
                    <item>seven<tag>out.numbers+=7;</tag></item>
                    <item>eight<tag>out.numbers+=8;</tag></item>
                    <item>nine<tag>out.numbers+=9;</tag></item>
                    <item>dash</item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Grammars for insurance (download)

The following grammars are supported for insurance domain: claim and policy numbers, driver's license and license plate numbers, expiration dates, start dates and renewal dates, claim and policy amounts.

Claim ID

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My claim number is One Five Four Two
                 Output: 1542

             Scenario 2:
                 Input: Claim number One Five Four Four
                 Output: 1544

         -->

        <rule id="digits">
            <tag>out=""</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My claim number is</item>
                <item repeat="0-1">Claim number</item>
                <item repeat="0-1">This is for claim</item>
            </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
```

```
          </rule>

        <rule id="singleDigit">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.digit=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.digit+=0;</tag></item>
                    <item>zero<tag>out.digit+=0;</tag></item>
                    <item>1<tag>out.digit+=1;</tag></item>
                    <item>one<tag>out.digit+=1;</tag></item>
                    <item>2<tag>out.digit+=2;</tag></item>
                    <item>two<tag>out.digit+=2;</tag></item>
                    <item>3<tag>out.digit+=3;</tag></item>
                    <item>three<tag>out.digit+=3;</tag></item>
                    <item>4<tag>out.digit+=4;</tag></item>
                    <item>four<tag>out.digit+=4;</tag></item>
                    <item>5<tag>out.digit+=5;</tag></item>
                    <item>five<tag>out.digit+=5;</tag></item>
                    <item>6<tag>out.digit+=6;</tag></item>
                    <item>six<tag>out.digit+=5;</tag></item>
                    <item>7<tag>out.digit+=7;</tag></item>
                    <item>seven<tag>out.digit+=7;</tag></item>
                    <item>8<tag>out.digit+=8;</tag></item>
                    <item>eight<tag>out.digit+=8;</tag></item>
                    <item>9<tag>out.digit+=9;</tag></item>
                    <item>nine<tag>out.digit+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Policy ID

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">
```

```
        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: My policy number is A B C 1 2 3 4
                Output: ABC1234

            Scenario 2:
                Input: This is the policy number 1 2 3 4 A B C
                Output: 1234ABC

            Scenario 3:
                Input: Hmm My policy number is 1 2 3 4 A B C 1
                Output: 123ABC1
        -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item><ruleref uri="#alphanumeric"/><tag>out +=
rules.alphanumeric.alphanum;</tag></item>
            <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
rules.digits.numbers</tag></item>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My policy number is</item>
                <item repeat="0-1">This is the policy number</item>
                <item repeat="0-1">Policy number</item>
                <item repeat="0-1">Yes, It is</item>
                <item repeat="0-1">Yes It is</item>
                <item repeat="0-1">Yes It's</item>
                <item repeat="0-1">My policy Id is</item>
                <item repeat="0-1">This is the policy Id</item>
                <item repeat="0-1">Policy Id</item>
            </one-of>
        </rule>
```

```
        <rule id="hesitation">
          <one-of>
             <item>Hmm</item>
             <item>Mmm</item>
             <item>My</item>
          </one-of>
        </rule>

        <rule id="thanks">
            <one-of>
               <item>Thanks</item>
               <item>I think</item>
            </one-of>
          </rule>

        <rule id="alphanumeric" scope="public">
            <tag>out.alphanum=""</tag>
            <item><ruleref uri="#alphabets"/><tag>out.alphanum +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum +=
rules.digits.numbers</tag></item>
        </rule>

        <rule id="alphabets">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.letters=""</tag>
            <tag>out.firstOccurence=""</tag>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
            <item repeat="1-">
                <one-of>
                    <item>A<tag>out.letters+='A';</tag></item>
                    <item>B<tag>out.letters+='B';</tag></item>
                    <item>C<tag>out.letters+='C';</tag></item>
                    <item>D<tag>out.letters+='D';</tag></item>
                    <item>E<tag>out.letters+='E';</tag></item>
                    <item>F<tag>out.letters+='F';</tag></item>
                    <item>G<tag>out.letters+='G';</tag></item>
                    <item>H<tag>out.letters+='H';</tag></item>
                    <item>I<tag>out.letters+='I';</tag></item>
                    <item>J<tag>out.letters+='J';</tag></item>
                    <item>K<tag>out.letters+='K';</tag></item>
                    <item>L<tag>out.letters+='L';</tag></item>
```

```
                    <item>M<tag>out.letters+='M';</tag></item>
                    <item>N<tag>out.letters+='N';</tag></item>
                    <item>O<tag>out.letters+='O';</tag></item>
                    <item>P<tag>out.letters+='P';</tag></item>
                    <item>Q<tag>out.letters+='Q';</tag></item>
                    <item>R<tag>out.letters+='R';</tag></item>
                    <item>S<tag>out.letters+='S';</tag></item>
                    <item>T<tag>out.letters+='T';</tag></item>
                    <item>U<tag>out.letters+='U';</tag></item>
                    <item>V<tag>out.letters+='V';</tag></item>
                    <item>W<tag>out.letters+='W';</tag></item>
                    <item>X<tag>out.letters+='X';</tag></item>
                    <item>Y<tag>out.letters+='Y';</tag></item>
                    <item>Z<tag>out.letters+='Z';</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.numbers=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.numbers+=0;</tag></item>
                    <item>1<tag>out.numbers+=1;</tag></item>
                    <item>2<tag>out.numbers+=2;</tag></item>
                    <item>3<tag>out.numbers+=3;</tag></item>
                    <item>4<tag>out.numbers+=4;</tag></item>
                    <item>5<tag>out.numbers+=5;</tag></item>
                    <item>6<tag>out.numbers+=6;</tag></item>
                    <item>7<tag>out.numbers+=7;</tag></item>
                    <item>8<tag>out.numbers+=8;</tag></item>
                    <item>9<tag>out.numbers+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Driver's license number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My drivers license number is One Five Four Two
                 Output: 1542

             Scenario 2:
                 Input: driver license number One Five Four Four
                 Output: 1544

         -->

        <rule id="digits">
            <tag>out=""</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My drivers license number is</item>
                <item repeat="0-1">My drivers license id is</item>
                <item repeat="0-1">Driver license number</item>
            </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>
```

```
        <rule id="singleDigit">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.digit=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.digit+=0;</tag></item>
                    <item>zero<tag>out.digit+=0;</tag></item>
                    <item>1<tag>out.digit+=1;</tag></item>
                    <item>one<tag>out.digit+=1;</tag></item>
                    <item>2<tag>out.digit+=2;</tag></item>
                    <item>two<tag>out.digit+=2;</tag></item>
                    <item>3<tag>out.digit+=3;</tag></item>
                    <item>three<tag>out.digit+=3;</tag></item>
                    <item>4<tag>out.digit+=4;</tag></item>
                    <item>four<tag>out.digit+=4;</tag></item>
                    <item>5<tag>out.digit+=5;</tag></item>
                    <item>five<tag>out.digit+=5;</tag></item>
                    <item>6<tag>out.digit+=6;</tag></item>
                    <item>six<tag>out.digit+=5;</tag></item>
                    <item>7<tag>out.digit+=7;</tag></item>
                    <item>seven<tag>out.digit+=7;</tag></item>
                    <item>8<tag>out.digit+=8;</tag></item>
                    <item>eight<tag>out.digit+=8;</tag></item>
                    <item>9<tag>out.digit+=9;</tag></item>
                    <item>nine<tag>out.digit+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

License plate number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">
```

```
        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: my license plate is A B C D 1 2
                Output: ABCD12

            Scenario 2:
                Input: license plate number A B C 1 2 3 4
                Output: ABC1234

            Scenario 3:
                Input: my plates say A F G K 9 8 7 6 Thanks
                Output: AFGK9876
        -->

        <rule id="main" scope="public">
            <tag>out.licenseNum=""</tag>
            <item><ruleref uri="#alphabets"/><tag>out.licenseNum +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">my license plate is</item>
                <item repeat="0-1">license plate number</item>
                <item repeat="0-1">my plates say</item>
            </one-of>
        </rule>

        <rule id="hesitation">
          <one-of>
             <item>Hmm</item>
             <item>Mmm</item>
             <item>My</item>
          </one-of>
        </rule>

        <rule id="thanks">
            <one-of>
               <item>Thanks</item>
```

```
            <item>I think</item>
          </one-of>
        </rule>

      <rule id="alphabets">
          <item repeat="0-1"><ruleref uri="#text"/></item>
          <tag>out.letters=""</tag>
          <tag>out.firstOccurence=""</tag>
          <item repeat="3-4">
              <one-of>
                  <item>A<tag>out.letters+='A';</tag></item>
                  <item>B<tag>out.letters+='B';</tag></item>
                  <item>C<tag>out.letters+='C';</tag></item>
                  <item>D<tag>out.letters+='D';</tag></item>
                  <item>E<tag>out.letters+='E';</tag></item>
                  <item>F<tag>out.letters+='F';</tag></item>
                  <item>G<tag>out.letters+='G';</tag></item>
                  <item>H<tag>out.letters+='H';</tag></item>
                  <item>I<tag>out.letters+='I';</tag></item>
                  <item>J<tag>out.letters+='J';</tag></item>
                  <item>K<tag>out.letters+='K';</tag></item>
                  <item>L<tag>out.letters+='L';</tag></item>
                  <item>M<tag>out.letters+='M';</tag></item>
                  <item>N<tag>out.letters+='N';</tag></item>
                  <item>O<tag>out.letters+='O';</tag></item>
                  <item>P<tag>out.letters+='P';</tag></item>
                  <item>Q<tag>out.letters+='Q';</tag></item>
                  <item>R<tag>out.letters+='R';</tag></item>
                  <item>S<tag>out.letters+='S';</tag></item>
                  <item>T<tag>out.letters+='T';</tag></item>
                  <item>U<tag>out.letters+='U';</tag></item>
                  <item>V<tag>out.letters+='V';</tag></item>
                  <item>W<tag>out.letters+='W';</tag></item>
                  <item>X<tag>out.letters+='X';</tag></item>
                  <item>Y<tag>out.letters+='Y';</tag></item>
                  <item>Z<tag>out.letters+='Z';</tag></item>
              </one-of>
          </item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
       </rule>

       <rule id="digits">
           <item repeat="0-1"><ruleref uri="#text"/></item>
```

```
            <tag>out.numbers=""</tag>
            <item repeat="2-4">
                <one-of>
                    <item>0<tag>out.numbers+=0;</tag></item>
                    <item>1<tag>out.numbers+=1;</tag></item>
                    <item>2<tag>out.numbers+=2;</tag></item>
                    <item>3<tag>out.numbers+=3;</tag></item>
                    <item>4<tag>out.numbers+=4;</tag></item>
                    <item>5<tag>out.numbers+=5;</tag></item>
                    <item>6<tag>out.numbers+=6;</tag></item>
                    <item>7<tag>out.numbers+=7;</tag></item>
                    <item>8<tag>out.numbers+=8;</tag></item>
                    <item>9<tag>out.numbers+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Credit card expiration date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="dateCardExpiration"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="dateCardExpiration" scope="public">
            <tag>out=""</tag>
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months;</
tag></item>
            <item repeat="1"><ruleref uri="#year"/><tag>out += " " + rules.year.yr;</
tag></item>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <!-- Test Cases

        Grammar will support the following inputs:
```

```
        Scenario 1:
            Input: My card expiration date is july eleven
            Output: 07 2011

        Scenario 2:
            Input: My card expiration date is may twenty six
            Output: 05 2026

    -->

    <rule id="text">
       <item repeat="0-1"><ruleref uri="#hesitation"/></item>
       <one-of>
          <item repeat="0-1">My card expiration date is </item>
       </one-of>
    </rule>

    <rule id="hesitation">
      <one-of>
         <item>Hmm</item>
         <item>Mmm</item>
         <item>My</item>
      </one-of>
    </rule>

    <rule id="thanks">
        <one-of>
           <item>Thanks</item>
           <item>I think</item>
        </one-of>
      </rule>

    <rule id="months">
       <item repeat="0-1"><ruleref uri="#text"/></item>
       <one-of>
           <item>january<tag>out="01";</tag></item>
           <item>february<tag>out="02";</tag></item>
           <item>march<tag>out="03";</tag></item>
           <item>april<tag>out="04";</tag></item>
           <item>may<tag>out="05";</tag></item>
           <item>june<tag>out="06";</tag></item>
           <item>july<tag>out="07";</tag></item>
           <item>august<tag>out="08";</tag></item>
           <item>september<tag>out="09";</tag></item>
```

```
            <item>october<tag>out="10";</tag></item>
            <item>november<tag>out="11";</tag></item>
            <item>december<tag>out="12";</tag></item>
            <item>jan<tag>out="01";</tag></item>
            <item>feb<tag>out="02";</tag></item>
            <item>aug<tag>out="08";</tag></item>
            <item>sept<tag>out="09";</tag></item>
            <item>oct<tag>out="10";</tag></item>
            <item>nov<tag>out="11";</tag></item>
            <item>dec<tag>out="12";</tag></item>
            <item>1<tag>out="01";</tag></item>
            <item>2<tag>out="02";</tag></item>
            <item>3<tag>out="03";</tag></item>
            <item>4<tag>out="04";</tag></item>
            <item>5<tag>out="05";</tag></item>
            <item>6<tag>out="06";</tag></item>
            <item>7<tag>out="07";</tag></item>
            <item>8<tag>out="08";</tag></item>
            <item>9<tag>out="09";</tag></item>
            <item>ten<tag>out="10";</tag></item>
            <item>eleven<tag>out="11";</tag></item>
            <item>twelve<tag>out="12";</tag></item>
        </one-of>
    </rule>

     <rule id="digits">
         <item repeat="0-1"><ruleref uri="#text"/></item>
         <one-of>
             <item>0<tag>out=0;</tag></item>
             <item>1<tag>out=1;</tag></item>
             <item>2<tag>out=2;</tag></item>
             <item>3<tag>out=3;</tag></item>
             <item>4<tag>out=4;</tag></item>
             <item>5<tag>out=5;</tag></item>
             <item>6<tag>out=6;</tag></item>
             <item>7<tag>out=7;</tag></item>
             <item>8<tag>out=8;</tag></item>
             <item>9<tag>out=9;</tag></item>
             <item>one<tag>out=1;</tag></item>
             <item>two<tag>out=2;</tag></item>
             <item>three<tag>out=3;</tag></item>
             <item>four<tag>out=4;</tag></item>
             <item>five<tag>out=5;</tag></item>
             <item>six<tag>out=6;</tag></item>
```

```
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="year">
          <tag>out.yr="20"</tag>
          <one-of>
              <item><ruleref uri="#teens"/><tag>out.yr += rules.teens;</tag></item>
              <item><ruleref uri="#above_twenty"/><tag>out.yr += rules.above_twenty;</
tag></item>
          </one-of>
        </rule>

        <rule id="teens">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
                <item>10<tag>out=10;</tag></item>
                <item>11<tag>out=11;</tag></item>
                <item>12<tag>out=12;</tag></item>
                <item>13<tag>out=13;</tag></item>
                <item>14<tag>out=14;</tag></item>
                <item>15<tag>out=15;</tag></item>
                <item>16<tag>out=16;</tag></item>
                <item>17<tag>out=17;</tag></item>
                <item>18<tag>out=18;</tag></item>
                <item>19<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
```

```
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
                <item>20<tag>out=20;</tag></item>
                <item>30<tag>out=30;</tag></item>
                <item>40<tag>out=40;</tag></item>
                <item>50<tag>out=50;</tag></item>
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Policy expiration date, day/month/year

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My policy expired on July Five Two Thousand and Eleven
                 Output: 07/5/11

             Scenario 2:
```

```
                Input: My policy will expire on July Sixteen Two Thousand and Eleven
                Output: 07/16/11

            Scenario 3:
                Input: My policy expired on July Thirty Two Thousand and Eleven
                Output: 07/30/11
        -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item>
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + "/";</tag></item>
                <one-of>
                  <item><ruleref uri="#digits"/><tag>out += rules.digits + "/";</
tag></item>
                  <item><ruleref uri="#teens"/><tag>out += rules.teens+ "/";</tag></
item>
                  <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
 "/";</tag></item>
                </one-of>
                <one-of>
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
             </item>
        </rule>

        <rule id="text">
          <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">My policy expired on</item>
              <item repeat="0-1">My policy will expire on</item>
           </one-of>
        </rule>

        <rule id="hesitation">
           <one-of>
```

```
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
          </one-of>
        </rule>

      <rule id="months">
         <tag>out.mon=""</tag>
  <item repeat="0-1"><ruleref uri="#text"/></item>
         <one-of>
           <item>january<tag>out.mon+="01";</tag></item>
           <item>february<tag>out.mon+="02";</tag></item>
           <item>march<tag>out.mon+="03";</tag></item>
           <item>april<tag>out.mon+="04";</tag></item>
           <item>may<tag>out.mon+="05";</tag></item>
           <item>june<tag>out.mon+="06";</tag></item>
           <item>july<tag>out.mon+="07";</tag></item>
           <item>august<tag>out.mon+="08";</tag></item>
           <item>september<tag>out.mon+="09";</tag></item>
           <item>october<tag>out.mon+="10";</tag></item>
           <item>november<tag>out.mon+="11";</tag></item>
           <item>december<tag>out.mon+="12";</tag></item>
           <item>jan<tag>out.mon+="01";</tag></item>
           <item>feb<tag>out.mon+="02";</tag></item>
           <item>aug<tag>out.mon+="08";</tag></item>
           <item>sept<tag>out.mon+="09";</tag></item>
           <item>oct<tag>out.mon+="10";</tag></item>
           <item>nov<tag>out.mon+="11";</tag></item>
           <item>dec<tag>out.mon+="12";</tag></item>
         </one-of>
      </rule>

       <rule id="digits">
           <one-of>
               <item>zero<tag>out=0;</tag></item>
               <item>one<tag>out=1;</tag></item>
               <item>two<tag>out=2;</tag></item>
               <item>three<tag>out=3;</tag></item>
               <item>four<tag>out=4;</tag></item>
               <item>five<tag>out=5;</tag></item>
               <item>six<tag>out=6;</tag></item>
               <item>seven<tag>out=7;</tag></item>
               <item>eight<tag>out=8;</tag></item>
               <item>nine<tag>out=9;</tag></item>
```

```
            </one-of>
        </rule>

        <rule id="teens">
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand</item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
 </grammar>
```

Policy renewal date, month/year

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
```

```
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: I renewed my policy on July Two Thousand and Eleven
                 Output: 07/11

             Scenario 2:
                 Input: My policy will renew on July Two Thousand and Eleven
                 Output: 07/11

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + "/";</tag></item>
                <one-of>
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My policy will renew on</item>
```

```
            <item repeat="0-1">My policy was renewed on</item>
            <item repeat="0-1">Renew policy on</item>
            <item repeat="0-1">I renewed my policy on</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
           <item>Hmm</item>
           <item>Mmm</item>
           <item>My</item>
        </one-of>
      </rule>

    <rule id="months">
       <item repeat="0-1"><ruleref uri="#text"/></item>
       <tag>out.mon=""</tag>
       <one-of>
           <item>january<tag>out.mon+="01";</tag></item>
           <item>february<tag>out.mon+="02";</tag></item>
           <item>march<tag>out.mon+="03";</tag></item>
           <item>april<tag>out.mon+="04";</tag></item>
           <item>may<tag>out.mon+="05";</tag></item>
           <item>june<tag>out.mon+="06";</tag></item>
           <item>july<tag>out.mon+="07";</tag></item>
           <item>august<tag>out.mon+="08";</tag></item>
           <item>september<tag>out.mon+="09";</tag></item>
           <item>october<tag>out.mon+="10";</tag></item>
           <item>november<tag>out.mon+="11";</tag></item>
           <item>december<tag>out.mon+="12";</tag></item>
           <item>jan<tag>out.mon+="01";</tag></item>
           <item>feb<tag>out.mon+="02";</tag></item>
           <item>aug<tag>out.mon+="08";</tag></item>
           <item>sept<tag>out.mon+="09";</tag></item>
           <item>oct<tag>out.mon+="10";</tag></item>
           <item>nov<tag>out.mon+="11";</tag></item>
           <item>dec<tag>out.mon+="12";</tag></item>
       </one-of>
    </rule>

    <rule id="digits">
        <one-of>
            <item>zero<tag>out=0;</tag></item>
            <item>one<tag>out=1;</tag></item>
```

```
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="teens">
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand<!--<tag>out=2000;</tag>--></item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
```

```
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Policy start date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: I bought my policy on july twenty three
                 Output: 07/23

             Scenario 2:
                 Input: My policy started on july fifteen
                 Output: 07/15

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></
item>
                <one-of>
                    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
                    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
```

```
                    <item><ruleref uri="#above_twenty"/><tag>out+=
rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
             <one-of>
                <item repeat="0-1">I bought my policy on</item>
                <item repeat="0-1">I bought policy on</item>
                <item repeat="0-1">My policy started on</item>
             </one-of>
          </rule>

          <rule id="hesitation">
             <one-of>
                <item>Hmm</item>
                <item>Mmm</item>
                <item>My</item>
             </one-of>
           </rule>

        <rule id="months">
           <item repeat="0-1"><ruleref uri="#text"/></item>
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="01";</tag></item>
               <item>february<tag>out.mon+="02";</tag></item>
               <item>march<tag>out.mon+="03";</tag></item>
               <item>april<tag>out.mon+="04";</tag></item>
               <item>may<tag>out.mon+="05";</tag></item>
               <item>june<tag>out.mon+="06";</tag></item>
               <item>july<tag>out.mon+="07";</tag></item>
               <item>august<tag>out.mon+="08";</tag></item>
               <item>september<tag>out.mon+="09";</tag></item>
               <item>october<tag>out.mon+="10";</tag></item>
               <item>november<tag>out.mon+="11";</tag></item>
               <item>december<tag>out.mon+="12";</tag></item>
               <item>jan<tag>out.mon+="01";</tag></item>
               <item>feb<tag>out.mon+="02";</tag></item>
               <item>aug<tag>out.mon+="08";</tag></item>
               <item>sept<tag>out.mon+="09";</tag></item>
               <item>oct<tag>out.mon+="10";</tag></item>
```

```
            <item>nov<tag>out.mon+="11";</tag></item>
            <item>dec<tag>out.mon+="12";</tag></item>
        </one-of>
    </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>0<tag>out=0;</tag></item>
            <item>1<tag>out=1;</tag></item>
            <item>2<tag>out=2;</tag></item>
            <item>3<tag>out=3;</tag></item>
            <item>4<tag>out=4;</tag></item>
            <item>5<tag>out=5;</tag></item>
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>first<tag>out=01;</tag></item>
            <item>second<tag>out=02;</tag></item>
            <item>third<tag>out=03;</tag></item>
            <item>fourth<tag>out=04;</tag></item>
            <item>fifth<tag>out=05;</tag></item>
            <item>sixth<tag>out=06;</tag></item>
            <item>seventh<tag>out=07;</tag></item>
            <item>eighth<tag>out=08;</tag></item>
            <item>ninth<tag>out=09;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
```

```
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
                <item>tenth<tag>out=10;</tag></item>
                <item>eleventh<tag>out=11;</tag></item>
                <item>twelveth<tag>out=12;</tag></item>
                <item>thirteenth<tag>out=13;</tag></item>
                <item>fourteenth<tag>out=14;</tag></item>
                <item>fifteenth<tag>out=15;</tag></item>
                <item>sixteenth<tag>out=16;</tag></item>
                <item>seventeenth<tag>out=17;</tag></item>
                <item>eighteenth<tag>out=18;</tag></item>
                <item>nineteenth<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Claim amount

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
```

```
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: I want to make a claim of one hundre ten dollars
                 Output: $110

             Scenario 2:
                 Input: Requesting claim of Two hundred dollars
                 Output: $200

         -->

        <rule id="main" scope="public">
            <tag>out="$"</tag>
            <one-of>
                <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
                <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</
tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
           <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">I want to place a claim for</item>
              <item repeat="0-1">I want to make a claim of</item>
              <item repeat="0-1">I assess damage of</item>
              <item repeat="0-1">Requesting claim of</item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>
```

```
  <rule id="thanks">
      <one-of>
         <item>Thanks</item>
         <item>I think</item>
      </one-of>
    </rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
        <item>three<tag>out.num+=3;</tag></item>
        <item>four<tag>out.num+=4;</tag></item>
        <item>five<tag>out.num+=5;</tag></item>
        <item>six<tag>out.num+=6;</tag></item>
        <item>seven<tag>out.num+=7;</tag></item>
        <item>eight<tag>out.num+=8;</tag></item>
        <item>nine<tag>out.num+=9;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
```

```
                <item>fifteen<tag>out.teen+=15;</tag></item>
                <item>sixteen<tag>out.teen+=16;</tag></item>
                <item>seventeen<tag>out.teen+=17;</tag></item>
                <item>eighteen<tag>out.teen+=18;</tag></item>
                <item>nineteen<tag>out.teen+=19;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.tens = 0;</tag>
            <one-of>
                <item>twenty<tag>out.tens+=20;</tag></item>
                <item>thirty<tag>out.tens+=30;</tag></item>
                <item>forty<tag>out.tens+=40;</tag></item>
                <item>fifty<tag>out.tens+=50;</tag></item>
                <item>sixty<tag>out.tens+=60;</tag></item>
                <item>seventy<tag>out.tens+=70;</tag></item>
                <item>eighty<tag>out.tens+=80;</tag></item>
                <item>ninety<tag>out.tens+=90;</tag></item>
                <item>hundred<tag>out.tens+=100;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens +=
 rules.digits.num;</tag></item>
        </rule>

        <rule id="currency">
            <one-of>
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>


        <rule id="sub_hundred">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></
item>
```

```
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh +=
 rules.above_twenty.tens;</tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></
item>
            </one-of>
         </rule>

        <rule id="subThousands">
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</
tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
</grammar>
```

Premium amount

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Premium amounts
             Scenario 1:
                 Input: The premium for one hundre ten dollars
                 Output: $110
```

```
             Scenario 2:
                 Input: RPremium amount of Two hundred dollars
                 Output: $200

         -->

        <rule id="main" scope="public">
            <tag>out="$"</tag>
            <one-of>
                <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
                <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</
tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
           <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">A premium of</item>
              <item repeat="0-1">Premium amount of</item>
              <item repeat="0-1">The premium for</item>
              <item repeat="0-1">Insurance premium for</item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

          <rule id="thanks">
              <one-of>
                 <item>Thanks</item>
                 <item>I think</item>
              </one-of>
            </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
```

```
            <tag>out.num = 0;</tag>
            <one-of>
                <item>0<tag>out.num+=0;</tag></item>
                <item>1<tag>out.num+=1;</tag></item>
                <item>2<tag>out.num+=2;</tag></item>
                <item>3<tag>out.num+=3;</tag></item>
                <item>4<tag>out.num+=4;</tag></item>
                <item>5<tag>out.num+=5;</tag></item>
                <item>6<tag>out.num+=6;</tag></item>
                <item>7<tag>out.num+=7;</tag></item>
                <item>8<tag>out.num+=8;</tag></item>
                <item>9<tag>out.num+=9;</tag></item>
                <item>one<tag>out.num+=1;</tag></item>
                <item>two<tag>out.num+=2;</tag></item>
                <item>three<tag>out.num+=3;</tag></item>
                <item>four<tag>out.num+=4;</tag></item>
                <item>five<tag>out.num+=5;</tag></item>
                <item>six<tag>out.num+=6;</tag></item>
                <item>seven<tag>out.num+=7;</tag></item>
                <item>eight<tag>out.num+=8;</tag></item>
                <item>nine<tag>out.num+=9;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="teens">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.teen = 0;</tag>
            <one-of>
                <item>ten<tag>out.teen+=10;</tag></item>
                <item>eleven<tag>out.teen+=11;</tag></item>
                <item>twelve<tag>out.teen+=12;</tag></item>
                <item>thirteen<tag>out.teen+=13;</tag></item>
                <item>fourteen<tag>out.teen+=14;</tag></item>
                <item>fifteen<tag>out.teen+=15;</tag></item>
                <item>sixteen<tag>out.teen+=16;</tag></item>
                <item>seventeen<tag>out.teen+=17;</tag></item>
                <item>eighteen<tag>out.teen+=18;</tag></item>
                <item>nineteen<tag>out.teen+=19;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="above_twenty">
```

```
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.tens = 0;</tag>
            <one-of>
                <item>twenty<tag>out.tens+=20;</tag></item>
                <item>thirty<tag>out.tens+=30;</tag></item>
                <item>forty<tag>out.tens+=40;</tag></item>
                <item>fifty<tag>out.tens+=50;</tag></item>
                <item>sixty<tag>out.tens+=60;</tag></item>
                <item>seventy<tag>out.tens+=70;</tag></item>
                <item>eighty<tag>out.tens+=80;</tag></item>
                <item>ninety<tag>out.tens+=90;</tag></item>
                <item>hundred<tag>out.tens+=100;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens +=
 rules.digits.num;</tag></item>
        </rule>

        <rule id="currency">
            <one-of>
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>


        <rule id="sub_hundred">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></
item>
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh +=
 rules.above_twenty.tens;</tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></
item>
            </one-of>
         </rule>

        <rule id="subThousands">
```

```
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</
tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
</grammar>
```

Policy quantity

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: The number is one
                 Output: 1

             Scenario 2:
                 Input: I want policy for ten
                 Output: 10

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <one-of>
              <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></
item>
```

```
              <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></
item>
              <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
           <one-of>
              <item repeat="0-1">I want policy for</item>
              <item repeat="0-1">I want to order policy for</item>
              <item repeat="0-1">The number is</item>
           </one-of>
        </rule>

        <rule id="thanks">
            <one-of>
               <item>Thanks</item>
               <item>I think</item>
            </one-of>
          </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>0<tag>out=0;</tag></item>
                <item>1<tag>out=1;</tag></item>
                <item>2<tag>out=2;</tag></item>
                <item>3<tag>out=3;</tag></item>
                <item>4<tag>out=4;</tag></item>
                <item>5<tag>out=5;</tag></item>
                <item>6<tag>out=6;</tag></item>
                <item>7<tag>out=7;</tag></item>
                <item>8<tag>out=8;</tag></item>
                <item>9<tag>out=9;</tag></item>
                <item>one<tag>out=1;</tag></item>
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
```

```
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
            <item>nineteen<tag>out=19;</tag></item>
            <item>10<tag>out=10;</tag></item>
            <item>11<tag>out=11;</tag></item>
            <item>12<tag>out=12;</tag></item>
            <item>13<tag>out=13;</tag></item>
            <item>14<tag>out=14;</tag></item>
            <item>15<tag>out=15;</tag></item>
            <item>16<tag>out=16;</tag></item>
            <item>17<tag>out=17;</tag></item>
            <item>18<tag>out=18;</tag></item>
            <item>19<tag>out=19;</tag></item>
        </one-of>
    </rule>

    <rule id="above_twenty">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>twenty<tag>out=20;</tag></item>
            <item>thirty<tag>out=30;</tag></item>
            <item>forty<tag>out=40;</tag></item>
            <item>fifty<tag>out=50;</tag></item>
            <item>sixty<tag>out=60;</tag></item>
            <item>seventy<tag>out=70;</tag></item>
            <item>eighty<tag>out=80;</tag></item>
            <item>ninety<tag>out=90;</tag></item>
            <item>20<tag>out=20;</tag></item>
            <item>30<tag>out=30;</tag></item>
            <item>40<tag>out=40;</tag></item>
```

```
                <item>50<tag>out=50;</tag></item>
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>

</grammar>
```

Grammars for telecom (download)

The following grammars are supported for telecom: Phone number, serial number, SIM number, US Zip code, credit card expiration date, plan start, renewal and expiration dates, service start date, equipment quantity and bill amount.

Phone number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support 10-12 digits number and here are couple of examples of
 valid inputs:

             Scenario 1:
                 Input: Mmm My phone number is two zero one two five two six seven
 eight five
                 Output: 2012526785

             Scenario 2:
                 Input: My phone number is two zero one two five two six seven eight
 five
```

```
                Output: 2012526785

         -->

        <rule id="digits">
            <tag>out=""</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">My phone number is</item>
              <item repeat="0-1">Phone number is</item>
              <item repeat="0-1">It is</item>
              <item repeat="0-1">Yes, it's</item>
              <item repeat="0-1">Yes, it is</item>
              <item repeat="0-1">Yes it is</item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

        <rule id="singleDigit">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.digit=""</tag>
            <item repeat="10-12">
                <one-of>
                    <item>0<tag>out.digit+=0;</tag></item>
                    <item>zero<tag>out.digit+=0;</tag></item>
                    <item>1<tag>out.digit+=1;</tag></item>
                    <item>one<tag>out.digit+=1;</tag></item>
                    <item>2<tag>out.digit+=2;</tag></item>
                    <item>two<tag>out.digit+=2;</tag></item>
                    <item>3<tag>out.digit+=3;</tag></item>
                    <item>three<tag>out.digit+=3;</tag></item>
                    <item>4<tag>out.digit+=4;</tag></item>
```

```
                    <item>four<tag>out.digit+=4;</tag></item>
                    <item>5<tag>out.digit+=5;</tag></item>
                    <item>five<tag>out.digit+=5;</tag></item>
                    <item>6<tag>out.digit+=6;</tag></item>
                    <item>six<tag>out.digit+=5;</tag></item>
                    <item>7<tag>out.digit+=7;</tag></item>
                    <item>seven<tag>out.digit+=7;</tag></item>
                    <item>8<tag>out.digit+=8;</tag></item>
                    <item>eight<tag>out.digit+=8;</tag></item>
                    <item>9<tag>out.digit+=9;</tag></item>
                    <item>nine<tag>out.digit+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Serial number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My serial number is 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6
                 Output: 123456789123456

             Scenario 2:
                 Input: Device Serial number 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6
                 Output: 123456789123456

         -->

        <rule id="digits">
```

```
            <tag>out=""</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My serial number is</item>
                <item repeat="0-1">Device Serial number</item>
                <item repeat="0-1">The number is</item>
                <item repeat="0-1">The IMEI number is</item>
            </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

        <rule id="singleDigit">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.digit=""</tag>
            <item repeat="15">
                <one-of>
                    <item>0<tag>out.digit+=0;</tag></item>
                    <item>zero<tag>out.digit+=0;</tag></item>
                    <item>1<tag>out.digit+=1;</tag></item>
                    <item>one<tag>out.digit+=1;</tag></item>
                    <item>2<tag>out.digit+=2;</tag></item>
                    <item>two<tag>out.digit+=2;</tag></item>
                    <item>3<tag>out.digit+=3;</tag></item>
                    <item>three<tag>out.digit+=3;</tag></item>
                    <item>4<tag>out.digit+=4;</tag></item>
                    <item>four<tag>out.digit+=4;</tag></item>
                    <item>5<tag>out.digit+=5;</tag></item>
                    <item>five<tag>out.digit+=5;</tag></item>
                    <item>6<tag>out.digit+=6;</tag></item>
                    <item>six<tag>out.digit+=5;</tag></item>
                    <item>7<tag>out.digit+=7;</tag></item>
                    <item>seven<tag>out.digit+=7;</tag></item>
```

```
                    <item>8<tag>out.digit+=8;</tag></item>
                    <item>eight<tag>out.digit+=8;</tag></item>
                    <item>9<tag>out.digit+=9;</tag></item>
                    <item>nine<tag>out.digit+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

SIM number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">


        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: My SIM number is A B C 1 2 3 4
                Output: ABC1234

            Scenario 2:
                Input: My SIM number is 1 2 3 4 A B C
                Output: 1234ABC

            Scenario 3:
                Input: My SIM number is 1 2 3 4 A B C 1
                Output: 123ABC1
        -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item><ruleref uri="#alphanumeric"/><tag>out +=
 rules.alphanumeric.alphanum;</tag></item>
```

```
            <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
rules.digits.numbers</tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My SIM number is</item>
                <item repeat="0-1">SIM number is</item>
            </one-of>
        </rule>

        <rule id="hesitation">
          <one-of>
             <item>Hmm</item>
             <item>Mmm</item>
             <item>My</item>
          </one-of>
        </rule>

        <rule id="alphanumeric" scope="public">
            <tag>out.alphanum=""</tag>
            <item><ruleref uri="#alphabets"/><tag>out.alphanum +=
rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum +=
rules.digits.numbers</tag></item>
        </rule>

        <rule id="alphabets">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.letters=""</tag>
            <tag>out.firstOccurence=""</tag>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
            <item repeat="1-">
                <one-of>
                    <item>A<tag>out.letters+='A';</tag></item>
                    <item>B<tag>out.letters+='B';</tag></item>
                    <item>C<tag>out.letters+='C';</tag></item>
                    <item>D<tag>out.letters+='D';</tag></item>
                    <item>E<tag>out.letters+='E';</tag></item>
                    <item>F<tag>out.letters+='F';</tag></item>
```

```
                    <item>G<tag>out.letters+='G';</tag></item>
                    <item>H<tag>out.letters+='H';</tag></item>
                    <item>I<tag>out.letters+='I';</tag></item>
                    <item>J<tag>out.letters+='J';</tag></item>
                    <item>K<tag>out.letters+='K';</tag></item>
                    <item>L<tag>out.letters+='L';</tag></item>
                    <item>M<tag>out.letters+='M';</tag></item>
                    <item>N<tag>out.letters+='N';</tag></item>
                    <item>O<tag>out.letters+='O';</tag></item>
                    <item>P<tag>out.letters+='P';</tag></item>
                    <item>Q<tag>out.letters+='Q';</tag></item>
                    <item>R<tag>out.letters+='R';</tag></item>
                    <item>S<tag>out.letters+='S';</tag></item>
                    <item>T<tag>out.letters+='T';</tag></item>
                    <item>U<tag>out.letters+='U';</tag></item>
                    <item>V<tag>out.letters+='V';</tag></item>
                    <item>W<tag>out.letters+='W';</tag></item>
                    <item>X<tag>out.letters+='X';</tag></item>
                    <item>Y<tag>out.letters+='Y';</tag></item>
                    <item>Z<tag>out.letters+='Z';</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="digits">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.numbers=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.numbers+=0;</tag></item>
                    <item>1<tag>out.numbers+=1;</tag></item>
                    <item>2<tag>out.numbers+=2;</tag></item>
                    <item>3<tag>out.numbers+=3;</tag></item>
                    <item>4<tag>out.numbers+=4;</tag></item>
                    <item>5<tag>out.numbers+=5;</tag></item>
                    <item>6<tag>out.numbers+=6;</tag></item>
                    <item>7<tag>out.numbers+=7;</tag></item>
                    <item>8<tag>out.numbers+=8;</tag></item>
                    <item>9<tag>out.numbers+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

US Zip code

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support 5 digits code and here are couple of examples of valid
 inputs:

             Scenario 1:
                 Input: Mmmm My zipcode is umm One Oh Nine Eight Seven
                 Output: 10987

             Scenario 2:
                 Input: My zipcode is One Oh Nine Eight Seven
                 Output: 10987

         -->

        <rule id="digits">
            <tag>out="";</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">My zipcode is</item>
              <item repeat="0-1">Zipcode is</item>
              <item repeat="0-1">It is</item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
```

```
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
          </one-of>
        </rule>

      <rule id="singleDigit">
          <item repeat="0-1"><ruleref uri="#text"/></item>
          <tag>out.digit=""</tag>
          <item repeat="5">
              <one-of>
                  <item>0<tag>out.digit+=0;</tag></item>
                  <item>zero<tag>out.digit+=0;</tag></item>
                  <item>Oh<tag>out.digit+=0;</tag></item>
                  <item>1<tag>out.digit+=1;</tag></item>
                  <item>one<tag>out.digit+=1;</tag></item>
                  <item>2<tag>out.digit+=2;</tag></item>
                  <item>two<tag>out.digit+=2;</tag></item>
                  <item>3<tag>out.digit+=3;</tag></item>
                  <item>three<tag>out.digit+=3;</tag></item>
                  <item>4<tag>out.digit+=4;</tag></item>
                  <item>four<tag>out.digit+=4;</tag></item>
                  <item>5<tag>out.digit+=5;</tag></item>
                  <item>five<tag>out.digit+=5;</tag></item>
                  <item>6<tag>out.digit+=6;</tag></item>
                  <item>six<tag>out.digit+=5;</tag></item>
                  <item>7<tag>out.digit+=7;</tag></item>
                  <item>seven<tag>out.digit+=7;</tag></item>
                  <item>8<tag>out.digit+=8;</tag></item>
                  <item>eight<tag>out.digit+=8;</tag></item>
                  <item>9<tag>out.digit+=9;</tag></item>
                  <item>nine<tag>out.digit+=9;</tag></item>
              </one-of>
          </item>
      </rule>
</grammar>
```

Credit card expiration date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
```

```
                              http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="dateCardExpiration"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="dateCardExpiration" scope="public">
            <tag>out=""</tag>
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months;</
tag></item>
            <item repeat="1"><ruleref uri="#year"/><tag>out += " " + rules.year.yr;</
tag></item>
        </rule>

        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: My card expiration date is july eleven
                Output: 07 2011

            Scenario 2:
                Input: My card expiration date is may twenty six
                Output: 05 2026

        -->

        <rule id="text">
           <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">My card expiration date is </item>
           </one-of>
        </rule>

        <rule id="hesitation">
          <one-of>
             <item>Hmm</item>
             <item>Mmm</item>
             <item>My</item>
          </one-of>
        </rule>

        <rule id="months">
```

```
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>january<tag>out="01";</tag></item>
                <item>february<tag>out="02";</tag></item>
                <item>march<tag>out="03";</tag></item>
                <item>april<tag>out="04";</tag></item>
                <item>may<tag>out="05";</tag></item>
                <item>june<tag>out="06";</tag></item>
                <item>july<tag>out="07";</tag></item>
                <item>august<tag>out="08";</tag></item>
                <item>september<tag>out="09";</tag></item>
                <item>october<tag>out="10";</tag></item>
                <item>november<tag>out="11";</tag></item>
                <item>december<tag>out="12";</tag></item>
                <item>jan<tag>out="01";</tag></item>
                <item>feb<tag>out="02";</tag></item>
                <item>aug<tag>out="08";</tag></item>
                <item>sept<tag>out="09";</tag></item>
                <item>oct<tag>out="10";</tag></item>
                <item>nov<tag>out="11";</tag></item>
                <item>dec<tag>out="12";</tag></item>
                <item>1<tag>out="01";</tag></item>
                <item>2<tag>out="02";</tag></item>
                <item>3<tag>out="03";</tag></item>
                <item>4<tag>out="04";</tag></item>
                <item>5<tag>out="05";</tag></item>
                <item>6<tag>out="06";</tag></item>
                <item>7<tag>out="07";</tag></item>
                <item>8<tag>out="08";</tag></item>
                <item>9<tag>out="09";</tag></item>
                <item>ten<tag>out="10";</tag></item>
                <item>eleven<tag>out="11";</tag></item>
                <item>twelve<tag>out="12";</tag></item>
            </one-of>
        </rule>

         <rule id="digits">
             <item repeat="0-1"><ruleref uri="#text"/></item>
             <one-of>
                 <item>0<tag>out=0;</tag></item>
                 <item>1<tag>out=1;</tag></item>
                 <item>2<tag>out=2;</tag></item>
                 <item>3<tag>out=3;</tag></item>
                 <item>4<tag>out=4;</tag></item>
```

```
                <item>5<tag>out=5;</tag></item>
                <item>6<tag>out=6;</tag></item>
                <item>7<tag>out=7;</tag></item>
                <item>8<tag>out=8;</tag></item>
                <item>9<tag>out=9;</tag></item>
                <item>one<tag>out=1;</tag></item>
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="year">
          <tag>out.yr="20"</tag>
          <one-of>
              <item><ruleref uri="#teens"/><tag>out.yr += rules.teens;</tag></item>
              <item><ruleref uri="#above_twenty"/><tag>out.yr += rules.above_twenty;</
tag></item>
          </one-of>
        </rule>

        <rule id="teens">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
                <item>10<tag>out=10;</tag></item>
                <item>11<tag>out=11;</tag></item>
                <item>12<tag>out=12;</tag></item>
                <item>13<tag>out=13;</tag></item>
                <item>14<tag>out=14;</tag></item>
```

```
                <item>15<tag>out=15;</tag></item>
                <item>16<tag>out=16;</tag></item>
                <item>17<tag>out=17;</tag></item>
                <item>18<tag>out=18;</tag></item>
                <item>19<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
                <item>20<tag>out=20;</tag></item>
                <item>30<tag>out=30;</tag></item>
                <item>40<tag>out=40;</tag></item>
                <item>50<tag>out=50;</tag></item>
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Plan expiration date, day/month/year

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
```

```
        tag-format="semantics/1.0">

        <!-- Test Cases

        Grammar will support the following inputs:

            Scenario 1:
                Input: My plan expires on July Five Two Thousand and Eleven
                Output: 07/5/11

            Scenario 2:
                Input: My plan will expire on July Sixteen Two Thousand and Eleven
                Output: 07/16/11

            Scenario 3:
                Input: My plan will expire on July Thirty Two Thousand and Eleven
                Output: 07/30/11
        -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item>
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + "/";</tag></item>
                <one-of>
                  <item><ruleref uri="#digits"/><tag>out += rules.digits + "/";</
tag></item>
                  <item><ruleref uri="#teens"/><tag>out += rules.teens+ "/";</tag></
item>
                  <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
 "/";</tag></item>
                </one-of>
                <one-of>
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>
```

```
<rule id="text">
  <item repeat="0-1"><ruleref uri="#hesitation"/></item>
  <one-of>
       <item repeat="0-1">My plan expires on</item>
       <item repeat="0-1">My plan expired on</item>
       <item repeat="0-1">My plan will expire on</item>
   </one-of>
</rule>

<rule id="hesitation">
   <one-of>
      <item>Hmm</item>
      <item>Mmm</item>
      <item>My</item>
   </one-of>
 </rule>

<rule id="months">
   <tag>out.mon=""</tag>
<item repeat="0-1"><ruleref uri="#text"/></item>

   <one-of>
     <item>january<tag>out.mon+="01";</tag></item>
     <item>february<tag>out.mon+="02";</tag></item>
     <item>march<tag>out.mon+="03";</tag></item>
     <item>april<tag>out.mon+="04";</tag></item>
     <item>may<tag>out.mon+="05";</tag></item>
     <item>june<tag>out.mon+="06";</tag></item>
     <item>july<tag>out.mon+="07";</tag></item>
     <item>august<tag>out.mon+="08";</tag></item>
     <item>september<tag>out.mon+="09";</tag></item>
     <item>october<tag>out.mon+="10";</tag></item>
     <item>november<tag>out.mon+="11";</tag></item>
     <item>december<tag>out.mon+="12";</tag></item>
     <item>jan<tag>out.mon+="01";</tag></item>
     <item>feb<tag>out.mon+="02";</tag></item>
     <item>aug<tag>out.mon+="08";</tag></item>
     <item>sept<tag>out.mon+="09";</tag></item>
     <item>oct<tag>out.mon+="10";</tag></item>
     <item>nov<tag>out.mon+="11";</tag></item>
     <item>dec<tag>out.mon+="12";</tag></item>
   </one-of>
</rule>
```

```
        <rule id="digits">
            <one-of>
                <item>zero<tag>out=0;</tag></item>
                <item>one<tag>out=1;</tag></item>
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="teens">
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand</item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
```

```
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
 </grammar>
```

Plan renewal date, month/year

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My plan will renew on July Two Thousand and Eleven
                 Output: 07/11

             Scenario 2:
                 Input: Renew plan on July Two Thousand and Eleven
                 Output: 07/11

         -->

        <rule id="main" scope="public">
            <tag>out="";</tag>
            <item repeat="1-10">
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + "/";</tag></item>
                <one-of>
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
```

```
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My plan will renew on</item>
                <item repeat="0-1">My plan was renewed on</item>
                <item repeat="0-1">Renew plan on</item>
            </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
            </one-of>
          </rule>

        <rule id="months">
           <item repeat="0-1"><ruleref uri="#text"/></item>
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="01";</tag></item>
               <item>february<tag>out.mon+="02";</tag></item>
               <item>march<tag>out.mon+="03";</tag></item>
               <item>april<tag>out.mon+="04";</tag></item>
               <item>may<tag>out.mon+="05";</tag></item>
               <item>june<tag>out.mon+="06";</tag></item>
               <item>july<tag>out.mon+="07";</tag></item>
               <item>august<tag>out.mon+="08";</tag></item>
               <item>september<tag>out.mon+="09";</tag></item>
               <item>october<tag>out.mon+="10";</tag></item>
               <item>november<tag>out.mon+="11";</tag></item>
               <item>december<tag>out.mon+="12";</tag></item>
               <item>jan<tag>out.mon+="01";</tag></item>
```

```
                <item>feb<tag>out.mon+="02";</tag></item>
                <item>aug<tag>out.mon+="08";</tag></item>
                <item>sept<tag>out.mon+="09";</tag></item>
                <item>oct<tag>out.mon+="10";</tag></item>
                <item>nov<tag>out.mon+="11";</tag></item>
                <item>dec<tag>out.mon+="12";</tag></item>
            </one-of>
        </rule>

        <rule id="digits">
            <one-of>
                <item>zero<tag>out=0;</tag></item>
                <item>one<tag>out=1;</tag></item>
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="teens">
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand</item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
```

```
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Plan start date, month/day

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My plan will start on july twenty three
                 Output: 07/23

             Scenario 2:
```

```
                 Input: My plan will start on july fifteen
                 Output: 07/15

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></
item>
                <one-of>
                    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
                    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
                    <item><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
             <one-of>
                <item repeat="0-1">My plan started on</item>
                <item repeat="0-1">My plan will start on</item>
                <item repeat="0-1">I paid it on</item>
                <item repeat="0-1">I paid bill for</item>
             </one-of>
          </rule>

          <rule id="hesitation">
             <one-of>
                <item>Hmm</item>
                <item>Mmm</item>
                <item>My</item>
             </one-of>
           </rule>

        <rule id="months">
           <item repeat="0-1"><ruleref uri="#text"/></item>
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="01";</tag></item>
               <item>february<tag>out.mon+="02";</tag></item>
               <item>march<tag>out.mon+="03";</tag></item>
```

```
            <item>april<tag>out.mon+="04";</tag></item>
            <item>may<tag>out.mon+="05";</tag></item>
            <item>june<tag>out.mon+="06";</tag></item>
            <item>july<tag>out.mon+="07";</tag></item>
            <item>august<tag>out.mon+="08";</tag></item>
            <item>september<tag>out.mon+="09";</tag></item>
            <item>october<tag>out.mon+="10";</tag></item>
            <item>november<tag>out.mon+="11";</tag></item>
            <item>december<tag>out.mon+="12";</tag></item>
            <item>jan<tag>out.mon+="01";</tag></item>
            <item>feb<tag>out.mon+="02";</tag></item>
            <item>aug<tag>out.mon+="08";</tag></item>
            <item>sept<tag>out.mon+="09";</tag></item>
            <item>oct<tag>out.mon+="10";</tag></item>
            <item>nov<tag>out.mon+="11";</tag></item>
            <item>dec<tag>out.mon+="12";</tag></item>
        </one-of>
    </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>0<tag>out=0;</tag></item>
            <item>1<tag>out=1;</tag></item>
            <item>2<tag>out=2;</tag></item>
            <item>3<tag>out=3;</tag></item>
            <item>4<tag>out=4;</tag></item>
            <item>5<tag>out=5;</tag></item>
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>first<tag>out=01;</tag></item>
            <item>second<tag>out=02;</tag></item>
            <item>third<tag>out=03;</tag></item>
            <item>fourth<tag>out=04;</tag></item>
            <item>fifth<tag>out=05;</tag></item>
            <item>sixth<tag>out=06;</tag></item>
            <item>seventh<tag>out=07;</tag></item>
            <item>eighth<tag>out=08;</tag></item>
            <item>ninth<tag>out=09;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
```

```
                <item>four<tag>out=4;</tag></item>
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="teens">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>tenth<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
                <item>tenth<tag>out=10;</tag></item>
                <item>eleventh<tag>out=11;</tag></item>
                <item>twelveth<tag>out=12;</tag></item>
                <item>thirteenth<tag>out=13;</tag></item>
                <item>fourteenth<tag>out=14;</tag></item>
                <item>fifteenth<tag>out=15;</tag></item>
                <item>sixteenth<tag>out=16;</tag></item>
                <item>seventeenth<tag>out=17;</tag></item>
                <item>eighteenth<tag>out=18;</tag></item>
                <item>nineteenth<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
```

```
        </rule>
</grammar>
```

Service start date, month/day

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: My plan starts on july twenty three
                 Output: 07/23

             Scenario 2:
                 Input: I want to activate on july fifteen
                 Output: 07/15

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></
item>
                <one-of>
                    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
                    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
                    <item><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>
```

```
        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
             <one-of>
                <item repeat="0-1">My plan starts on</item>
                <item repeat="0-1">I want to start my plan on</item>
                <item repeat="0-1">Activation date of</item>
                <item repeat="0-1">Start activation on</item>
                <item repeat="0-1">I want to activate on</item>
                <item repeat="0-1">Activate plan starting</item>
                <item repeat="0-1">Starting</item>
                <item repeat="0-1">Start on</item>
             </one-of>
          </rule>

          <rule id="hesitation">
             <one-of>
                <item>Hmm</item>
                <item>Mmm</item>
                <item>My</item>
             </one-of>
           </rule>

        <rule id="months">
           <item repeat="0-1"><ruleref uri="#text"/></item>
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="01";</tag></item>
               <item>february<tag>out.mon+="02";</tag></item>
               <item>march<tag>out.mon+="03";</tag></item>
               <item>april<tag>out.mon+="04";</tag></item>
               <item>may<tag>out.mon+="05";</tag></item>
               <item>june<tag>out.mon+="06";</tag></item>
               <item>july<tag>out.mon+="07";</tag></item>
               <item>august<tag>out.mon+="08";</tag></item>
               <item>september<tag>out.mon+="09";</tag></item>
               <item>october<tag>out.mon+="10";</tag></item>
               <item>november<tag>out.mon+="11";</tag></item>
               <item>december<tag>out.mon+="12";</tag></item>
               <item>jan<tag>out.mon+="01";</tag></item>
               <item>feb<tag>out.mon+="02";</tag></item>
               <item>aug<tag>out.mon+="08";</tag></item>
               <item>sept<tag>out.mon+="09";</tag></item>
               <item>oct<tag>out.mon+="10";</tag></item>
               <item>nov<tag>out.mon+="11";</tag></item>
```

```
            <item>dec<tag>out.mon+="12";</tag></item>
        </one-of>
    </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>0<tag>out=0;</tag></item>
            <item>1<tag>out=1;</tag></item>
            <item>2<tag>out=2;</tag></item>
            <item>3<tag>out=3;</tag></item>
            <item>4<tag>out=4;</tag></item>
            <item>5<tag>out=5;</tag></item>
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>first<tag>out=01;</tag></item>
            <item>second<tag>out=02;</tag></item>
            <item>third<tag>out=03;</tag></item>
            <item>fourth<tag>out=04;</tag></item>
            <item>fifth<tag>out=05;</tag></item>
            <item>sixth<tag>out=06;</tag></item>
            <item>seventh<tag>out=07;</tag></item>
            <item>eighth<tag>out=08;</tag></item>
            <item>ninth<tag>out=09;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
```

```
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
                <item>tenth<tag>out=10;</tag></item>
                <item>eleventh<tag>out=11;</tag></item>
                <item>twelveth<tag>out=12;</tag></item>
                <item>thirteenth<tag>out=13;</tag></item>
                <item>fourteenth<tag>out=14;</tag></item>
                <item>fifteenth<tag>out=15;</tag></item>
                <item>sixteenth<tag>out=16;</tag></item>
                <item>seventeenth<tag>out=17;</tag></item>
                <item>eighteenth<tag>out=18;</tag></item>
                <item>nineteenth<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Equipment quantity

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">
```

```
         <!-- Test Cases

         Grammar will support the following inputs:

             Scenario 1:
                 Input: The number is one
                 Output: 1

             Scenario 2:
                 Input: It is ten
                 Output: 10

         -->

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <one-of>
              <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></
item>
              <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></
item>
              <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">It is</item>
              <item repeat="0-1">The number is</item>
              <item repeat="0-1">Order</item>
              <item repeat="0-1">I want to order</item>
              <item repeat="0-1">Total equipment</item>
           </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
               <item>Hmm</item>
               <item>Mmm</item>
               <item>My</item>
```

```
        </one-of>
      </rule>

    <rule id="thanks">
        <one-of>
           <item>Thanks</item>
           <item>I think</item>
        </one-of>
      </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>0<tag>out=0;</tag></item>
            <item>1<tag>out=1;</tag></item>
            <item>2<tag>out=2;</tag></item>
            <item>3<tag>out=3;</tag></item>
            <item>4<tag>out=4;</tag></item>
            <item>5<tag>out=5;</tag></item>
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
```

```
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
                <item>10<tag>out=10;</tag></item>
                <item>11<tag>out=11;</tag></item>
                <item>12<tag>out=12;</tag></item>
                <item>13<tag>out=13;</tag></item>
                <item>14<tag>out=14;</tag></item>
                <item>15<tag>out=15;</tag></item>
                <item>16<tag>out=16;</tag></item>
                <item>17<tag>out=17;</tag></item>
                <item>18<tag>out=18;</tag></item>
                <item>19<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
                <item>20<tag>out=20;</tag></item>
                <item>30<tag>out=30;</tag></item>
                <item>40<tag>out=40;</tag></item>
                <item>50<tag>out=50;</tag></item>
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>

</grammar>
```

Bill amount

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <!-- Test Cases

         Grammar will support the following inputs:

                 Input: I want to make a payment of one hundred ten dollars
                 Output: $110

         -->

        <rule id="main" scope="public">
            <tag>out="$"</tag>
            <one-of>
                <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
                <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</
tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
           <item repeat="0-1"><ruleref uri="#hesitation"/></item>
           <one-of>
              <item repeat="0-1">I want to make a payment for</item>
              <item repeat="0-1">I want to make a payment of</item>
              <item repeat="0-1">Pay a total of</item>
              <item repeat="0-1">Paying</item>
              <item repeat="0-1">Pay bill for </item>
           </one-of>
        </rule>

        <rule id="hesitation">
```

```
        <one-of>
           <item>Hmm</item>
           <item>Mmm</item>
           <item>My</item>
        </one-of>
      </rule>

      <rule id="thanks">
          <one-of>
             <item>Thanks</item>
             <item>I think</item>
          </one-of>
        </rule>

    <rule id="digits">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.num = 0;</tag>
        <one-of>
            <item>0<tag>out.num+=0;</tag></item>
            <item>1<tag>out.num+=1;</tag></item>
            <item>2<tag>out.num+=2;</tag></item>
            <item>3<tag>out.num+=3;</tag></item>
            <item>4<tag>out.num+=4;</tag></item>
            <item>5<tag>out.num+=5;</tag></item>
            <item>6<tag>out.num+=6;</tag></item>
            <item>7<tag>out.num+=7;</tag></item>
            <item>8<tag>out.num+=8;</tag></item>
            <item>9<tag>out.num+=9;</tag></item>
            <item>one<tag>out.num+=1;</tag></item>
            <item>two<tag>out.num+=2;</tag></item>
            <item>three<tag>out.num+=3;</tag></item>
            <item>four<tag>out.num+=4;</tag></item>
            <item>five<tag>out.num+=5;</tag></item>
            <item>six<tag>out.num+=6;</tag></item>
            <item>seven<tag>out.num+=7;</tag></item>
            <item>eight<tag>out.num+=8;</tag></item>
            <item>nine<tag>out.num+=9;</tag></item>
        </one-of>
        <item repeat="0-1"><ruleref uri="#currency"/></item>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.teen = 0;</tag>
```

```
            <one-of>
                <item>ten<tag>out.teen+=10;</tag></item>
                <item>eleven<tag>out.teen+=11;</tag></item>
                <item>twelve<tag>out.teen+=12;</tag></item>
                <item>thirteen<tag>out.teen+=13;</tag></item>
                <item>fourteen<tag>out.teen+=14;</tag></item>
                <item>fifteen<tag>out.teen+=15;</tag></item>
                <item>sixteen<tag>out.teen+=16;</tag></item>
                <item>seventeen<tag>out.teen+=17;</tag></item>
                <item>eighteen<tag>out.teen+=18;</tag></item>
                <item>nineteen<tag>out.teen+=19;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="above_twenty">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.tens = 0;</tag>
            <one-of>
                <item>twenty<tag>out.tens+=20;</tag></item>
                <item>thirty<tag>out.tens+=30;</tag></item>
                <item>forty<tag>out.tens+=40;</tag></item>
                <item>fifty<tag>out.tens+=50;</tag></item>
                <item>sixty<tag>out.tens+=60;</tag></item>
                <item>seventy<tag>out.tens+=70;</tag></item>
                <item>eighty<tag>out.tens+=80;</tag></item>
                <item>ninety<tag>out.tens+=90;</tag></item>
                <item>hundred<tag>out.tens+=100;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens +=
rules.digits.num;</tag></item>
        </rule>

        <rule id="currency">
            <one-of>
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>
```

```
        <rule id="sub_hundred">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></
item>
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh +=
 rules.above_twenty.tens;</tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></
item>
            </one-of>
         </rule>

        <rule id="subThousands">
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</
tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
</grammar>
```

Generic grammars (download)

We provide the following generic grammars: alphanumeric, currency, date (mm/dd/yy), numbers, greeting, hesitation, and agent.

Alphanumeric

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">
```

```
       <!-- Test Cases

           Scenario 1:
               Input: A B C 1 2 3 4
               Output: ABC1234

           Scenario 2:
               Input: 1 2 3 4 A B C
               Output: 1234ABC

           Scenario 3:
               Input: 1 2 3 4 A B C 1
               Output: 123ABC1
       -->

       <rule id="main" scope="public">
           <tag>out=""</tag>
           <item><ruleref uri="#alphanumeric"/><tag>out +=
rules.alphanumeric.alphanum;</tag></item>
           <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out +=
rules.alphabets.letters;</tag></item>
           <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
rules.digits.numbers</tag></item>
       </rule>

       <rule id="alphanumeric" scope="public">
           <tag>out.alphanum=""</tag>
           <item><ruleref uri="#alphabets"/><tag>out.alphanum +=
rules.alphabets.letters;</tag></item>
           <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum +=
rules.digits.numbers</tag></item>
       </rule>

       <rule id="alphabets">
           <tag>out.letters=""</tag>
           <tag>out.firstOccurence=""</tag>
           <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
           <item repeat="1-">
               <one-of>
                   <item>A<tag>out.letters+='A';</tag></item>
                   <item>B<tag>out.letters+='B';</tag></item>
```

```
                <item>C<tag>out.letters+='C';</tag></item>
                <item>D<tag>out.letters+='D';</tag></item>
                <item>E<tag>out.letters+='E';</tag></item>
                <item>F<tag>out.letters+='F';</tag></item>
                <item>G<tag>out.letters+='G';</tag></item>
                <item>H<tag>out.letters+='H';</tag></item>
                <item>I<tag>out.letters+='I';</tag></item>
                <item>J<tag>out.letters+='J';</tag></item>
                <item>K<tag>out.letters+='K';</tag></item>
                <item>L<tag>out.letters+='L';</tag></item>
                <item>M<tag>out.letters+='M';</tag></item>
                <item>N<tag>out.letters+='N';</tag></item>
                <item>O<tag>out.letters+='O';</tag></item>
                <item>P<tag>out.letters+='P';</tag></item>
                <item>Q<tag>out.letters+='Q';</tag></item>
                <item>R<tag>out.letters+='R';</tag></item>
                <item>S<tag>out.letters+='S';</tag></item>
                <item>T<tag>out.letters+='T';</tag></item>
                <item>U<tag>out.letters+='U';</tag></item>
                <item>V<tag>out.letters+='V';</tag></item>
                <item>W<tag>out.letters+='W';</tag></item>
                <item>X<tag>out.letters+='X';</tag></item>
                <item>Y<tag>out.letters+='Y';</tag></item>
                <item>Z<tag>out.letters+='Z';</tag></item>
            </one-of>
        </item>
    </rule>

    <rule id="digits">
        <tag>out.numbers=""</tag>
        <item repeat="1-10">
            <one-of>
                <item>0<tag>out.numbers+=0;</tag></item>
                <item>1<tag>out.numbers+=1;</tag></item>
                <item>2<tag>out.numbers+=2;</tag></item>
                <item>3<tag>out.numbers+=3;</tag></item>
                <item>4<tag>out.numbers+=4;</tag></item>
                <item>5<tag>out.numbers+=5;</tag></item>
                <item>6<tag>out.numbers+=6;</tag></item>
                <item>7<tag>out.numbers+=7;</tag></item>
                <item>8<tag>out.numbers+=8;</tag></item>
                <item>9<tag>out.numbers+=9;</tag></item>
            </one-of>
        </item>
```

```
        </rule>
</grammar>
```

Currency

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="main" scope="public">
            <tag>out="$"</tag>
            <one-of>
                <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
                <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</
tag></item>
            </one-of>
        </rule>

        <rule id="digits">
            <tag>out.num = 0;</tag>
            <one-of>
                <item>0<tag>out.num+=0;</tag></item>
                <item>1<tag>out.num+=1;</tag></item>
                <item>2<tag>out.num+=2;</tag></item>
                <item>3<tag>out.num+=3;</tag></item>
                <item>4<tag>out.num+=4;</tag></item>
                <item>5<tag>out.num+=5;</tag></item>
                <item>6<tag>out.num+=6;</tag></item>
                <item>7<tag>out.num+=7;</tag></item>
                <item>8<tag>out.num+=8;</tag></item>
                <item>9<tag>out.num+=9;</tag></item>
                <item>one<tag>out.num+=1;</tag></item>
                <item>two<tag>out.num+=2;</tag></item>
                <item>three<tag>out.num+=3;</tag></item>
                <item>four<tag>out.num+=4;</tag></item>
                <item>five<tag>out.num+=5;</tag></item>
```

```
                <item>six<tag>out.num+=6;</tag></item>
                <item>seven<tag>out.num+=7;</tag></item>
                <item>eight<tag>out.num+=8;</tag></item>
                <item>nine<tag>out.num+=9;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="teens">
            <tag>out.teen = 0;</tag>
            <one-of>
                <item>ten<tag>out.teen+=10;</tag></item>
                <item>eleven<tag>out.teen+=11;</tag></item>
                <item>twelve<tag>out.teen+=12;</tag></item>
                <item>thirteen<tag>out.teen+=13;</tag></item>
                <item>fourteen<tag>out.teen+=14;</tag></item>
                <item>fifteen<tag>out.teen+=15;</tag></item>
                <item>sixteen<tag>out.teen+=16;</tag></item>
                <item>seventeen<tag>out.teen+=17;</tag></item>
                <item>eighteen<tag>out.teen+=18;</tag></item>
                <item>nineteen<tag>out.teen+=19;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>

        <rule id="above_twenty">
            <tag>out.tens = 0;</tag>
            <one-of>
                <item>twenty<tag>out.tens+=20;</tag></item>
                <item>thirty<tag>out.tens+=30;</tag></item>
                <item>forty<tag>out.tens+=40;</tag></item>
                <item>fifty<tag>out.tens+=50;</tag></item>
                <item>sixty<tag>out.tens+=60;</tag></item>
                <item>seventy<tag>out.tens+=70;</tag></item>
                <item>eighty<tag>out.tens+=80;</tag></item>
                <item>ninety<tag>out.tens+=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens +=
rules.digits.num;</tag></item>
        </rule>

        <rule id="currency">
            <one-of>
```

```
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>


        <rule id="sub_hundred">
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></
item>
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh +=
 rules.above_twenty.tens;</tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></
item>
            </one-of>
         </rule>

        <rule id="subThousands">
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</
tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
        </rule>
</grammar>
```

Date, dd/mm

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
```

```
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <one-of>
                    <item><ruleref uri="#digits"/><tag>out += rules.digits + " ";</
tag></item>
                    <item><ruleref uri="#teens"/><tag>out += rules.teens+ " ";</tag></
item>
                    <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
 " ";</tag></item>
                </one-of>
                <item><ruleref uri="#months"/><tag>out = out + rules.months;</tag></
item>
            </item>
        </rule>

        <rule id="months">
           <one-of>
             <item>january<tag>out="january";</tag></item>
             <item>february<tag>out="february";</tag></item>
             <item>march<tag>out="march";</tag></item>
             <item>april<tag>out="april";</tag></item>
             <item>may<tag>out="may";</tag></item>
             <item>june<tag>out="june";</tag></item>
             <item>july<tag>out="july";</tag></item>
             <item>august<tag>out="august";</tag></item>
             <item>september<tag>out="september";</tag></item>
             <item>october<tag>out="october";</tag></item>
             <item>november<tag>out="november";</tag></item>
             <item>december<tag>out="december";</tag></item>
             <item>jan<tag>out="january";</tag></item>
             <item>feb<tag>out="february";</tag></item>
             <item>aug<tag>out="august";</tag></item>
             <item>sept<tag>out="september";</tag></item>
             <item>oct<tag>out="october";</tag></item>
             <item>nov<tag>out="november";</tag></item>
             <item>dec<tag>out="december";</tag></item>
           </one-of>
       </rule>

        <rule id="digits">
```

```
        <one-of>
            <item>0<tag>out=0;</tag></item>
            <item>1<tag>out=1;</tag></item>
            <item>2<tag>out=2;</tag></item>
            <item>3<tag>out=3;</tag></item>
            <item>4<tag>out=4;</tag></item>
            <item>5<tag>out=5;</tag></item>
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>first<tag>out=1;</tag></item>
            <item>second<tag>out=2;</tag></item>
            <item>third<tag>out=3;</tag></item>
            <item>fourth<tag>out=4;</tag></item>
            <item>fifth<tag>out=5;</tag></item>
            <item>sixth<tag>out=6;</tag></item>
            <item>seventh<tag>out=7;</tag></item>
            <item>eighth<tag>out=8;</tag></item>
            <item>ninth<tag>out=9;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
```

```
                <item>nineteen<tag>out=19;</tag></item>
                <item>tenth<tag>out=10;</tag></item>
                <item>eleventh<tag>out=11;</tag></item>
                <item>twelveth<tag>out=12;</tag></item>
                <item>thirteenth<tag>out=13;</tag></item>
                <item>fourteenth<tag>out=14;</tag></item>
                <item>fifteenth<tag>out=15;</tag></item>
                <item>sixteenth<tag>out=16;</tag></item>
                <item>seventeenth<tag>out=17;</tag></item>
                <item>eighteenth<tag>out=18;</tag></item>
                <item>nineteenth<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>
</grammar>
```

Date, mm/yy

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <item repeat="1-10">
                <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + " ";</tag></item>
                <one-of>
```

```
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="months">
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="january";</tag></item>
               <item>february<tag>out.mon+="february";</tag></item>
               <item>march<tag>out.mon+="march";</tag></item>
               <item>april<tag>out.mon+="april";</tag></item>
               <item>may<tag>out.mon+="may";</tag></item>
               <item>june<tag>out.mon+="june";</tag></item>
               <item>july<tag>out.mon+="july";</tag></item>
               <item>august<tag>out.mon+="august";</tag></item>
               <item>september<tag>out.mon+="september";</tag></item>
               <item>october<tag>out.mon+="october";</tag></item>
               <item>november<tag>out.mon+="november";</tag></item>
               <item>december<tag>out.mon+="december";</tag></item>
               <item>jan<tag>out.mon+="january";</tag></item>
               <item>feb<tag>out.mon+="february";</tag></item>
               <item>aug<tag>out.mon+="august";</tag></item>
               <item>sept<tag>out.mon+="september";</tag></item>
               <item>oct<tag>out.mon+="october";</tag></item>
               <item>nov<tag>out.mon+="november";</tag></item>
               <item>dec<tag>out.mon+="december";</tag></item>
           </one-of>
       </rule>

        <rule id="digits">
            <one-of>
                <item>zero<tag>out=0;</tag></item>
                <item>one<tag>out=1;</tag></item>
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
```

```
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="teens">
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

       <!--  <rule id="singleDigit">
            <item><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
        </rule> -->

        <rule id="thousands">
            <!-- <item>
                <ruleref uri="#digits"/>
                <tag>out = (1000 * rules.digits);</tag>
                thousand
            </item> -->
            <item>two thousand<tag>out=2000;</tag></item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
```

```
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>

</grammar>
```

Date, dd/mm/yyyy

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <rule id="main" scope="public">
             <tag>out="";</tag>
             <item repeat="1-10">
                 <one-of>
                   <item><ruleref uri="#digits"/><tag>out += rules.digits + " ";</
tag></item>
                   <item><ruleref uri="#teens"/><tag>out += rules.teens+ " ";</tag></
item>
                   <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
 " ";</tag></item>
                 </one-of>
                 <item repeat="1"><ruleref uri="#months"/><tag>out = out +
 rules.months.mon + " ";</tag></item>
                 <one-of>
```

```
                    <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</
tag></item>
                    <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
 rules.digits;</tag></item>
                    <item repeat="0-1"><ruleref uri="#teens"/><tag>out +=
 rules.teens;</tag></item>
                    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="months">
           <tag>out.mon=""</tag>
           <one-of>
               <item>january<tag>out.mon+="january";</tag></item>
               <item>february<tag>out.mon+="february";</tag></item>
               <item>march<tag>out.mon+="march";</tag></item>
               <item>april<tag>out.mon+="april";</tag></item>
               <item>may<tag>out.mon+="may";</tag></item>
               <item>june<tag>out.mon+="june";</tag></item>
               <item>july<tag>out.mon+="july";</tag></item>
               <item>august<tag>out.mon+="august";</tag></item>
               <item>september<tag>out.mon+="september";</tag></item>
               <item>october<tag>out.mon+="october";</tag></item>
               <item>november<tag>out.mon+="november";</tag></item>
               <item>december<tag>out.mon+="december";</tag></item>
               <item>jan<tag>out.mon+="january";</tag></item>
               <item>feb<tag>out.mon+="february";</tag></item>
               <item>aug<tag>out.mon+="august";</tag></item>
               <item>sept<tag>out.mon+="september";</tag></item>
               <item>oct<tag>out.mon+="october";</tag></item>
               <item>nov<tag>out.mon+="november";</tag></item>
               <item>dec<tag>out.mon+="december";</tag></item>
           </one-of>
       </rule>

        <rule id="digits">
            <one-of>
                <item>zero<tag>out=0;</tag></item>
                <item>one<tag>out=1;</tag></item>
                <item>two<tag>out=2;</tag></item>
                <item>three<tag>out=3;</tag></item>
                <item>four<tag>out=4;</tag></item>
```

```
                <item>five<tag>out=5;</tag></item>
                <item>six<tag>out=6;</tag></item>
                <item>seven<tag>out=7;</tag></item>
                <item>eight<tag>out=8;</tag></item>
                <item>nine<tag>out=9;</tag></item>
            </one-of>
        </rule>

        <rule id="teens">
            <one-of>
                <item>ten<tag>out=10;</tag></item>
                <item>eleven<tag>out=11;</tag></item>
                <item>twelve<tag>out=12;</tag></item>
                <item>thirteen<tag>out=13;</tag></item>
                <item>fourteen<tag>out=14;</tag></item>
                <item>fifteen<tag>out=15;</tag></item>
                <item>sixteen<tag>out=16;</tag></item>
                <item>seventeen<tag>out=17;</tag></item>
                <item>eighteen<tag>out=18;</tag></item>
                <item>nineteen<tag>out=19;</tag></item>
            </one-of>
        </rule>

        <rule id="thousands">
            <item>two thousand<tag>out=2000;</tag></item>
            <item repeat="0-1">and</item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</tag></
item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
 rules.above_twenty;</tag></item>
        </rule>

        <rule id="above_twenty">
            <one-of>
                <item>twenty<tag>out=20;</tag></item>
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
```

```
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>

 </grammar>
```

Numbers, digits

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="digits">
            <tag>out="";</tag>
            <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</
tag></item>
        </rule>

        <rule id="singleDigit">
            <tag>out.digit="";</tag>
            <item repeat="1-10">
                <one-of>
                    <item>0<tag>out.digit+=0;</tag></item>
                    <item>zero<tag>out.digit+=0;</tag></item>
                    <item>1<tag>out.digit+=1;</tag></item>
                    <item>one<tag>out.digit+=1;</tag></item>
                    <item>2<tag>out.digit+=2;</tag></item>
                    <item>two<tag>out.digit+=2;</tag></item>
                    <item>3<tag>out.digit+=3;</tag></item>
                    <item>three<tag>out.digit+=3;</tag></item>
                    <item>4<tag>out.digit+=4;</tag></item>
                    <item>four<tag>out.digit+=4;</tag></item>
                    <item>5<tag>out.digit+=5;</tag></item>
                    <item>five<tag>out.digit+=5;</tag></item>
                    <item>6<tag>out.digit+=6;</tag></item>
                    <item>six<tag>out.digit+=6;</tag></item>
```

```
                    <item>7<tag>out.digit+=7;</tag></item>
                    <item>seven<tag>out.digit+=7;</tag></item>
                    <item>8<tag>out.digit+=8;</tag></item>
                    <item>eight<tag>out.digit+=8;</tag></item>
                    <item>9<tag>out.digit+=9;</tag></item>
                    <item>nine<tag>out.digit+=9;</tag></item>
                </one-of>
            </item>
        </rule>
</grammar>
```

Numbers, ordinal

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

        <rule id="main" scope="public">
            <tag>out=""</tag>
            <one-of>
              <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></
item>
              <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></
item>
              <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
 rules.above_twenty;</tag></item>
            </one-of>
        </rule>

        <rule id="digits">
            <one-of>
                <item>0<tag>out=0;</tag></item>
                <item>1<tag>out=1;</tag></item>
                <item>2<tag>out=2;</tag></item>
                <item>3<tag>out=3;</tag></item>
                <item>4<tag>out=4;</tag></item>
                <item>5<tag>out=5;</tag></item>
```

```
            <item>6<tag>out=6;</tag></item>
            <item>7<tag>out=7;</tag></item>
            <item>8<tag>out=8;</tag></item>
            <item>9<tag>out=9;</tag></item>
            <item>one<tag>out=1;</tag></item>
            <item>two<tag>out=2;</tag></item>
            <item>three<tag>out=3;</tag></item>
            <item>four<tag>out=4;</tag></item>
            <item>five<tag>out=5;</tag></item>
            <item>six<tag>out=6;</tag></item>
            <item>seven<tag>out=7;</tag></item>
            <item>eight<tag>out=8;</tag></item>
            <item>nine<tag>out=9;</tag></item>
        </one-of>
    </rule>

    <rule id="teens">
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
            <item>nineteen<tag>out=19;</tag></item>
            <item>10<tag>out=10;</tag></item>
            <item>11<tag>out=11;</tag></item>
            <item>12<tag>out=12;</tag></item>
            <item>13<tag>out=13;</tag></item>
            <item>14<tag>out=14;</tag></item>
            <item>15<tag>out=15;</tag></item>
            <item>16<tag>out=16;</tag></item>
            <item>17<tag>out=17;</tag></item>
            <item>18<tag>out=18;</tag></item>
            <item>19<tag>out=19;</tag></item>
        </one-of>
    </rule>

    <rule id="above_twenty">
        <one-of>
            <item>twenty<tag>out=20;</tag></item>
```

```
                <item>thirty<tag>out=30;</tag></item>
                <item>forty<tag>out=40;</tag></item>
                <item>fifty<tag>out=50;</tag></item>
                <item>sixty<tag>out=60;</tag></item>
                <item>seventy<tag>out=70;</tag></item>
                <item>eighty<tag>out=80;</tag></item>
                <item>ninety<tag>out=90;</tag></item>
                <item>20<tag>out=20;</tag></item>
                <item>30<tag>out=30;</tag></item>
                <item>40<tag>out=40;</tag></item>
                <item>50<tag>out=50;</tag></item>
                <item>60<tag>out=60;</tag></item>
                <item>70<tag>out=70;</tag></item>
                <item>80<tag>out=80;</tag></item>
                <item>90<tag>out=90;</tag></item>
            </one-of>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
        </rule>

</grammar>
```

Agent

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <rule id="main" scope="public">
             <tag>out=""</tag>
             <ruleref uri="#text"/><tag>out = rules.text</tag>
         </rule>

         <rule id="text">
           <one-of>
              <item>Can I talk to the agent<tag>out="You will be trasnfered to the
 agent in a while"</tag></item>
```

```
              <item>talk to an agent<tag>out="You will be trasnfered to the agent in a
 while"</tag></item>
           </one-of>
         </rule>
</grammar>
```

Greeting

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

         <rule id="main" scope="public">
             <tag>out=""</tag>
             <ruleref uri="#text"/><tag>out = rules.text</tag>
         </rule>

         <rule id="text">
           <one-of>
              <item>hey<tag>out="Greeting"</tag></item>
              <item>hi<tag>out="Greeting"</tag></item>
              <item>Hi<tag>out="Greeting"</tag></item>
              <item>Hey<tag>out="Greeting"</tag></item>
              <item>Hello<tag>out="Greeting"</tag></item>
              <item>hello<tag>out="Greeting"</tag></item>
           </one-of>
         </rule>
</grammar>
```

Hesitation

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
```

```
          root="main"
          mode="voice"
          tag-format="semantics/1.0">

          <rule id="main" scope="public">
              <tag>out=""</tag>
              <ruleref uri="#text"/><tag>out = rules.text</tag>
          </rule>

          <rule id="text">
            <one-of>
               <item>Hmm<tag>out="Waiting for your input"</tag></item>
               <item>Mmm<tag>out="Waiting for your input"</tag></item>
               <item>Can you please wait<tag>out="Waiting for your input"</tag></item>
            </one-of>
          </rule>
</grammar>
```

Composite slot type

A composite slot is a combination of two or more slots that capture multiple pieces of information in a single user input. For example, you can configure the bot to elicit the location by requesting for the “city and state or zipcode”. In contrast, when the conversation is configured to use separate slot types resulting in a rigid conversational experience (“What is the city?” followed by “What is the zipcode?”). With a composite slot, you can capture all the information through a single slot. A composite slot is a combination of slots called subslots, such as city, state, and zip code.

You can use a combination of available Amazon Lex slot types (built-ins) and your own slots (custom slots). You can design logical expressions to capture information within the required subslots. For example: city and state or zipcode.

The composite slot type is only available in en-US and locales with limited feature support via Generative AI.

Creating a composite slot type

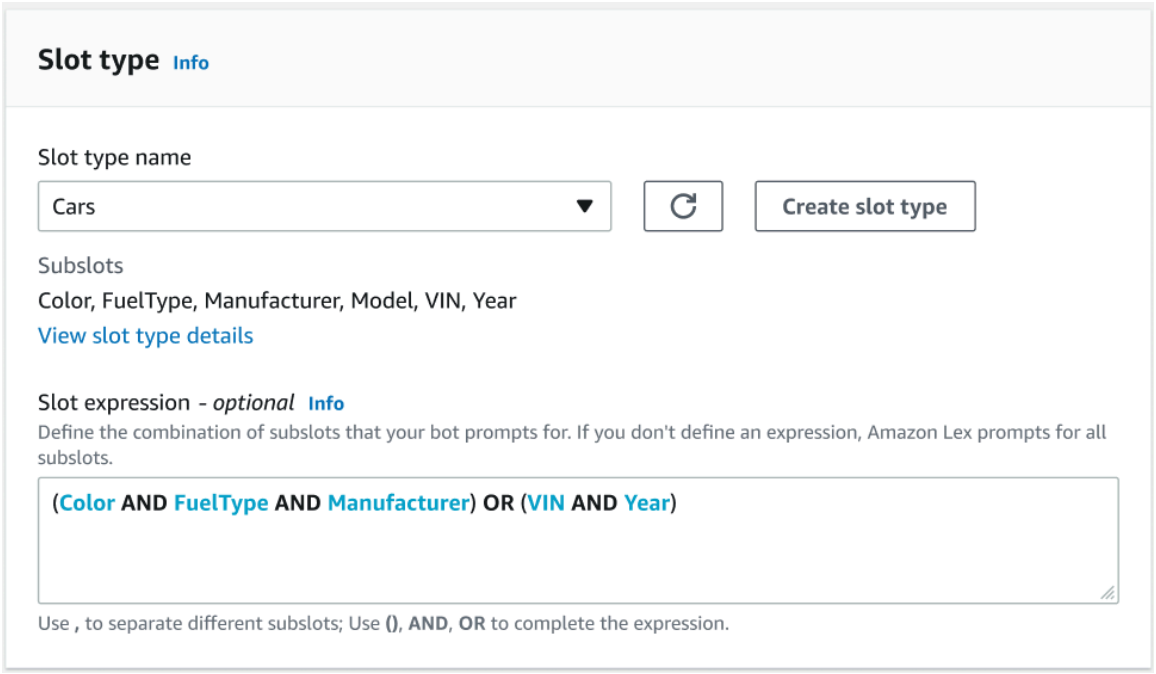
To use subslots within a composite slot, you must first configure the composite slot type. To do so, use the adding a slot type console steps or the API operation. After you have chosen the name and a description for the composite slot type, you have to provide information for subslots. For more information on adding a slot type, see Adding slot types

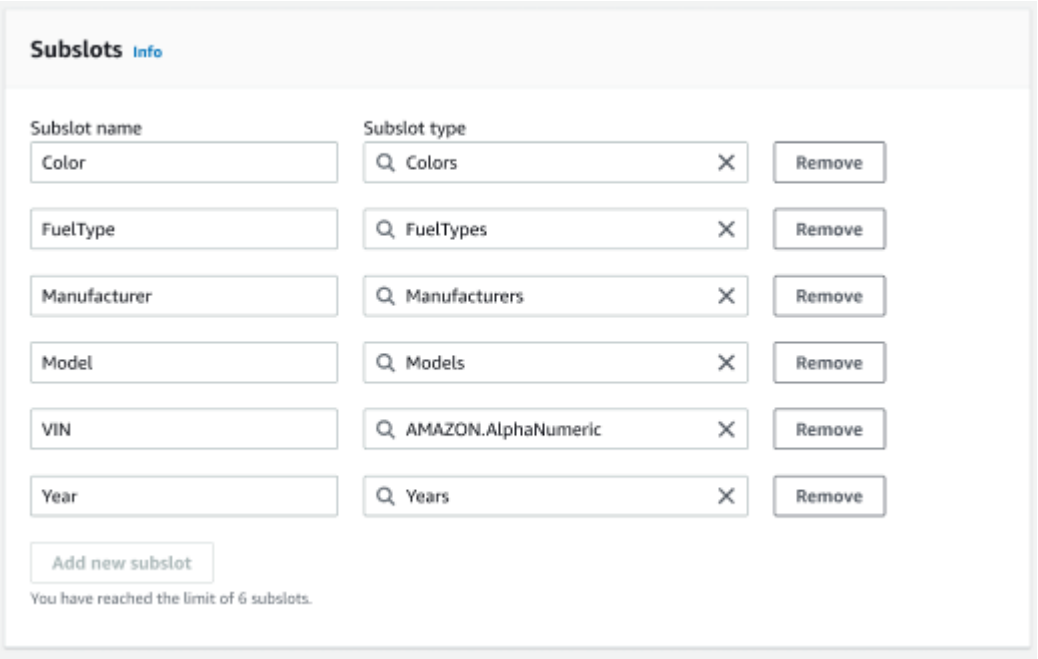
Subslots

A composite slot type requires configuration of the underlying slots, called subslots. If you would like to elicit multiple pieces of information from a customer in one request, configure a combination of subslots. For example: city, state, and zipcode. You can add up to 6 subslots for a composite slot.

Slots of singular slot types may be used to add subslots to the composite slot type. However, you cannot use a composite slot type as a slot type for a subslot.

The following images are an illustration of a composite slot “Car”, which is a combination of subslots: Color, FuelType, Manufacturer, Model, VIN, and Year.





Expression builder

To drive fulfillment of a composite slot, you can optionally use the expression builder. With the expression builder, you can design a logical slot expression to capture the required subslot values in the desired order. As part of the boolean expression, you can use operators such as AND and OR. Based on the designed expression, when the required subslots are fulfilled, the composite slot is considered fulfilled.

Using a composite slot type

For some intents, you might want to capture different slots as part of a single slot. For example, a car maintenance scheduling bot might have an intent with the following utterance:

```
My car is a {car}
```

The intent expects that the {car} composite slot contains a list of the slots, comprising details of the car. For example, "2021 White Toyota Camry".

The composite slot differs from a multi-valued slot. The composite slot is comprised of multiple slots, each with its own value. Whereas, a multi-valued slot is a singular slot that can contain a list of values. For more information on multi-values slots see, Using multiple values in a slot

For a composite slot, Amazon Lex returns a value for each subslot in the response to the `RecognizeText` or `RecognizeUtterance` operation. The following is the slot information

returned for the utterance: "I want to schedule a service for my “2021 White Toyota Camry" from the CarService bot.

```
"slots": {
    "CarType": {
        "value": {
            "originalValue": "White Toyota Camry 2021",
            "interpretedValue": "White Toyota Camry 2021",
            "resolvedValues": [
                "white Toyota Camry 2021"
            ]
        },
        "subSlots": {
            "Color": {
                "value": {
                    "originalValue": "White",
                    "interpretedValue": "White",
                    "resolvedValues": [
                        "white"
                    ]
                },
                "shape": "Scalar"
            },
            "Manufacturer": {
                "value": {
                    "originalValue": "Toyota",
                    "interpretedValue": "Toyota",
                    "resolvedValues": [
                        "Toyota"
                    ]
                },
                "shape": "Scalar"
            },
            "Model": {
                "value": {
                    "originalValue": "Camry",
                    "interpretedValue": "Camry",
                    "resolvedValues": [
                       "Camry"
                    ]
                },
                "shape": "Scalar"
            },
```

```
            "Year": {
                "value": {
                    "originalValue": "2021",
                    "interpretedValue": "2021",
                    "resolvedValues": [
                        "2021"
                    ]
                },
                "shape": "Scalar"
            }
        }
    },
    ...
}
```

A composite slot can be elicited for in the first turn or the n-th turn of a conversation. Based on the input values supplied, the composite slot can elicit for the remaining required subslots.

Composite slots always return a value for each subslot. When the utterance does not contain a recognizable value for a given subslot, there is no response returned for that particular subslot.

Composite slots work with both text and voice input.

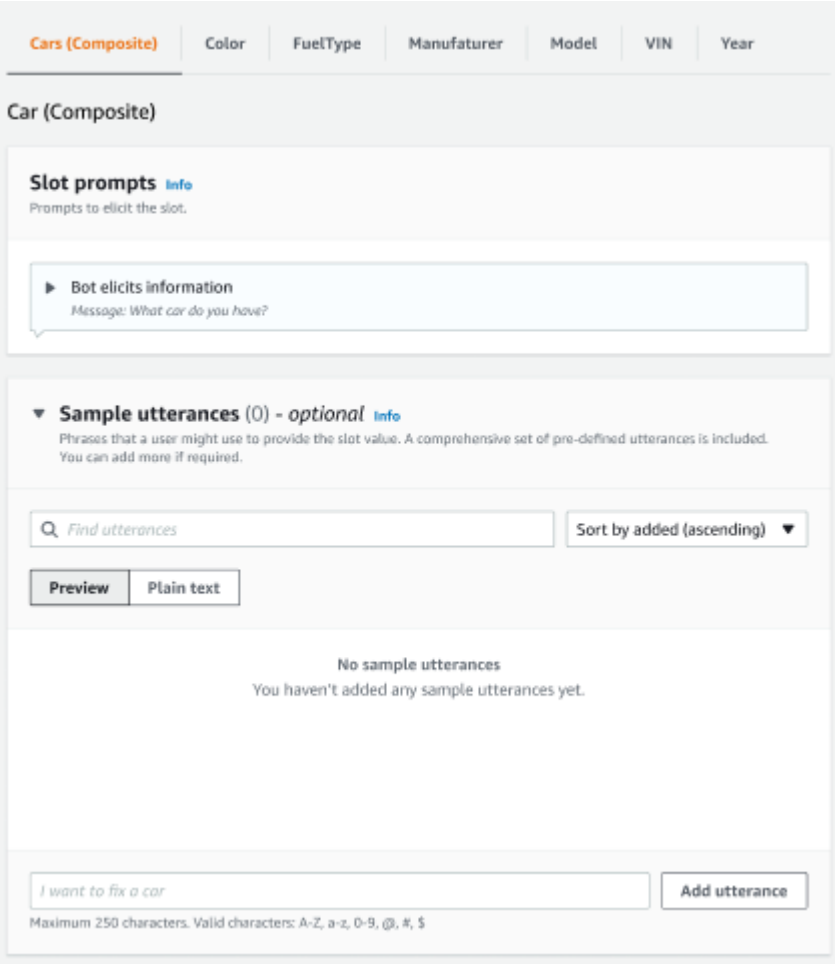
When adding a slot to an intent, a composite slot is only available as a custom slot type.

You can use Composite slots in prompts. For example, you can set the confirmation prompt for an intent.

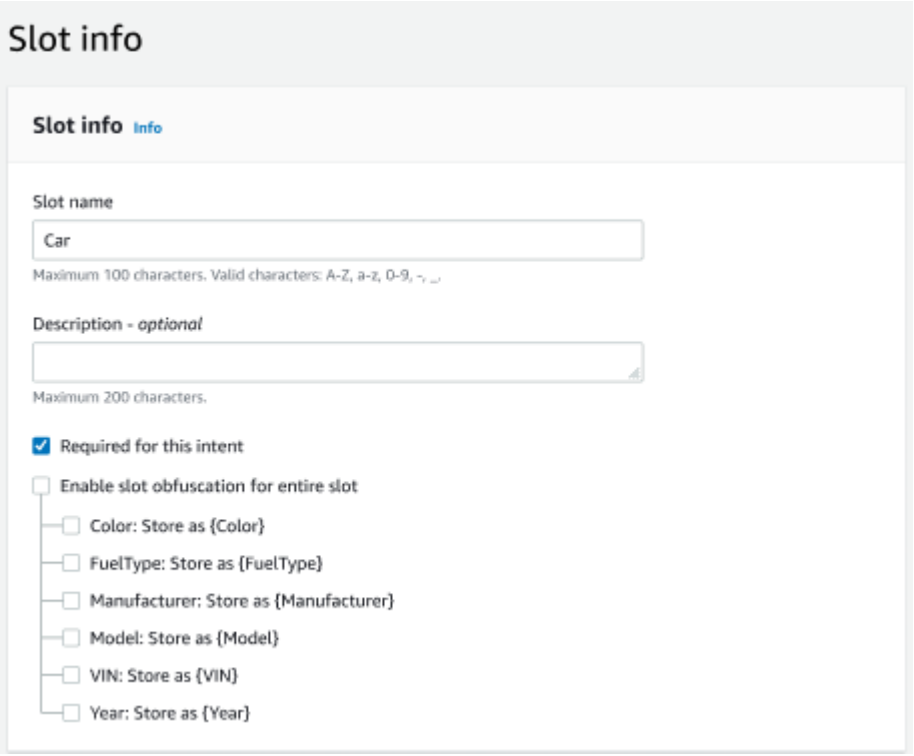
`Would you like me to schedule service for your 2021 White Toyota Camry?`

When Amazon Lex sends the prompt to the user, it sends "Would you like me to schedule service for your 2021 White Toyota Camry?”

Each subslot is configured as a slot. You can add slot prompts to elicit the subslot and sample utterances. You can enable wait and continue for a subslot as well as default values. For more information, see Using default slot values in intents for your Lex V2 bot



You can use slot obfuscation to mask the whole composite slot in conversation logs. Please note that slot obfuscation is applied at the composite slot level and when enabled, the values for subslots belonging to a composite slot are obfuscated. When you obfuscate slot values, the value of each of the slot values is replaced with the name of the slot. For more information, see Obscuring slot values in conversation logs from Lex V2.



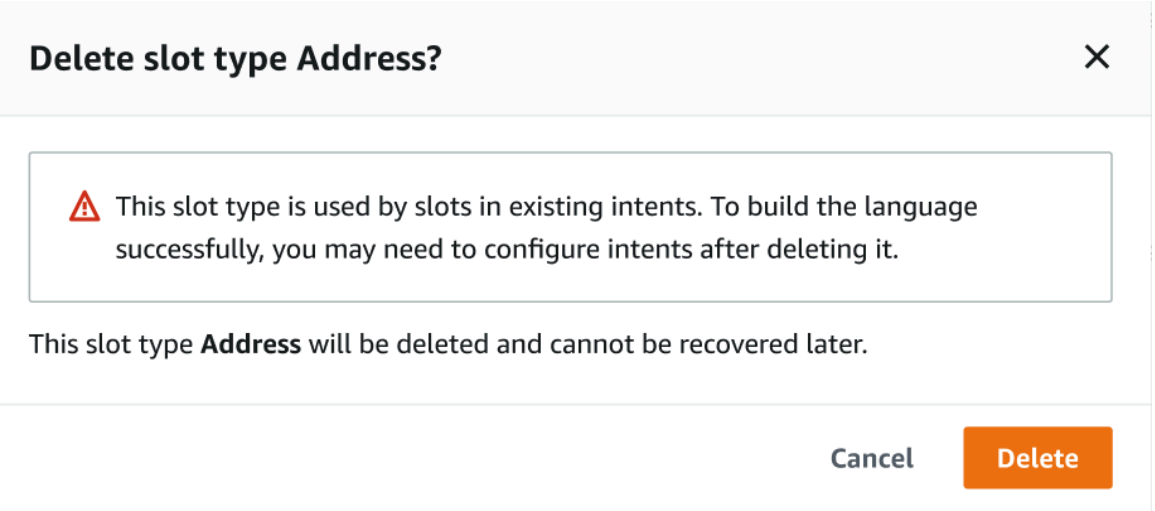
Editing a composite slot type

You can edit a subslot from within the composite slot configuration in order to modify subslot name and slot type. However, when a composite slot is in use by an intent, you will have to edit the intents before modifying the subslot.

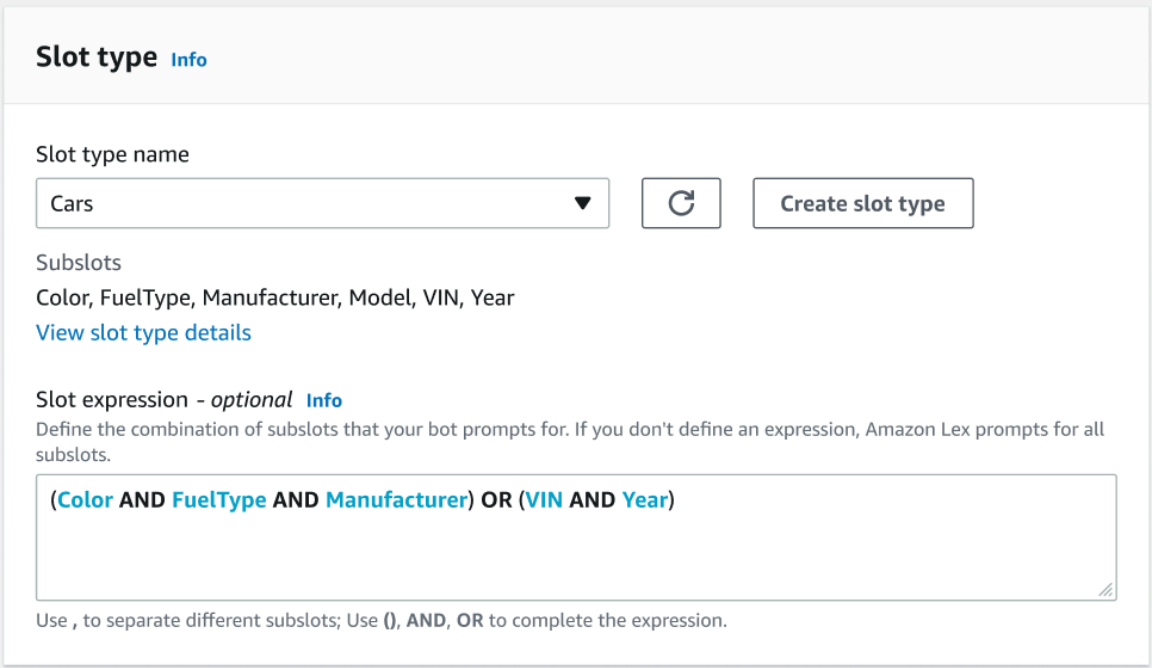
> Existing intents use this slot type. To build the language successfully, you may need to configure those intents after editing sub slots.

Deleting a composite slot type

You can delete a subslot from within the composite slot configuration. Please note that when a subslot is in use within an intent, the subslots are still removed from that intent.



The slot expression in the expression builder provides an alert to inform about the deleted subslots.



Testing a bot using the console

The Amazon Lex V2 console contains a test window that you can use to test the interaction with your bot. You use the test window to have a test conversation with your bot and to see the responses that your application receives from the bot.

There are two types of testing that you can perform with your bot. The first, express testing, enables you to test your bot with the exact phrases that you used for creating the bot. For

example, if you added the utterance "I want to pick up flowers" to your intent, you can test the bot using that exact phrase.

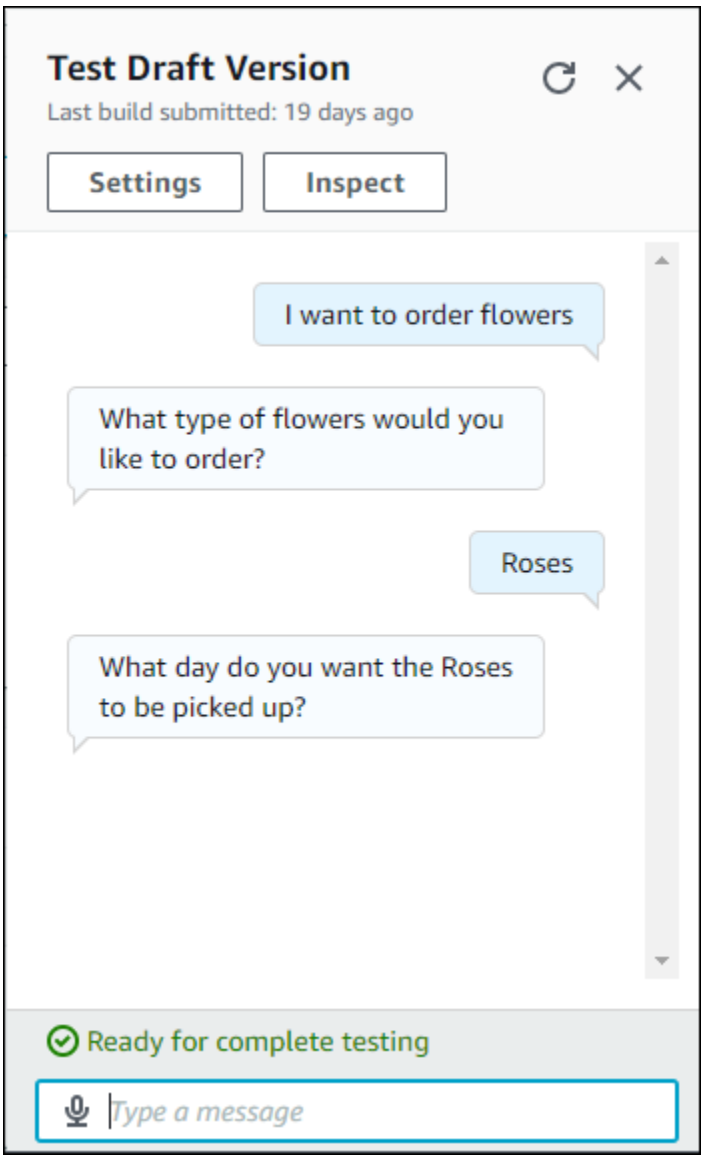
The second type, complete testing, enables you to test your bot using phrases related to the utterances that you configured. For example, you can use the phrase "Can I order flowers" to start a conversation with your bot.

You test a bot using a specific alias and language. If you are testing the development version of the bot, you use the `TestBotAlias` alias for testing.

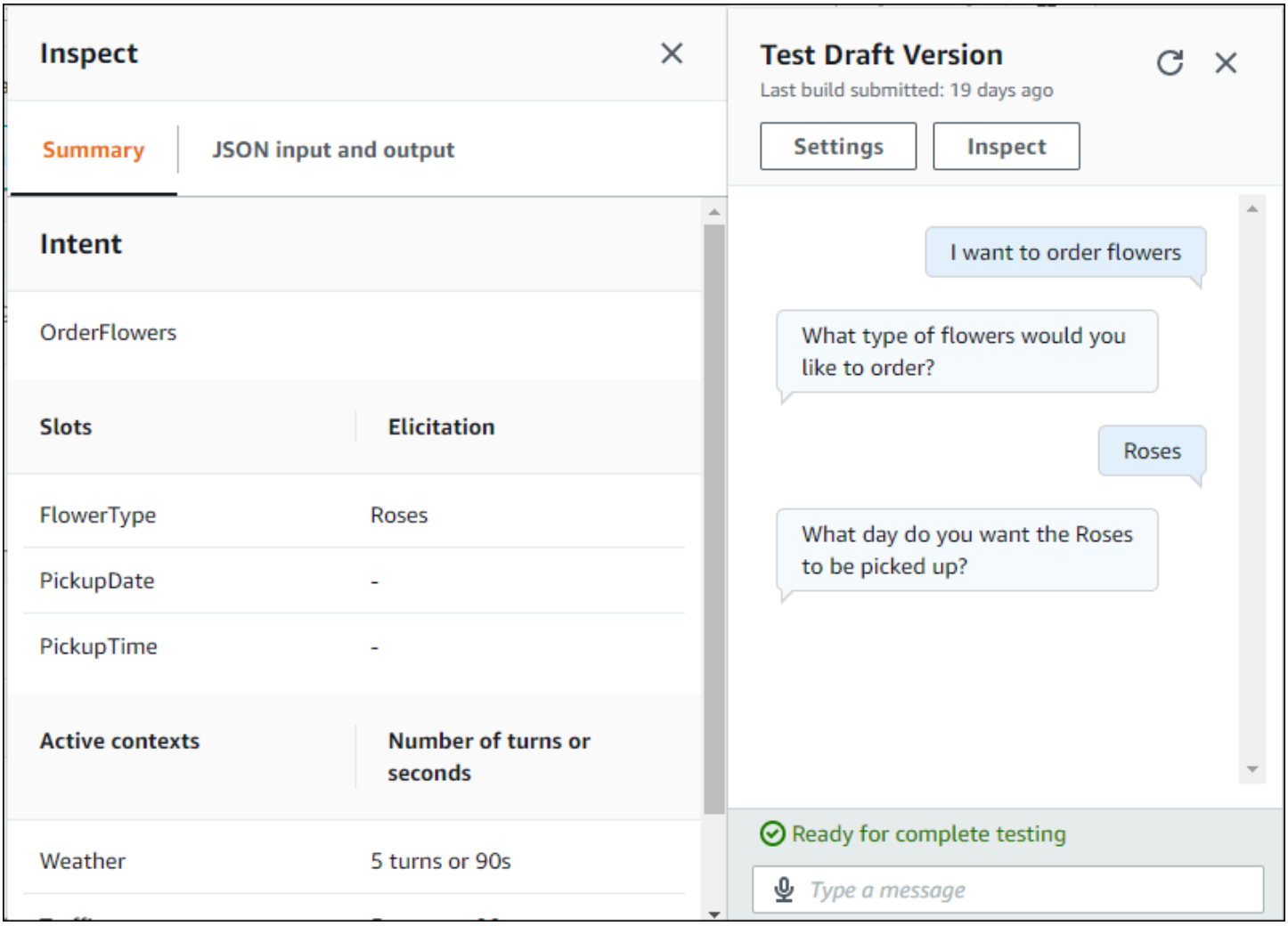
To open the test window

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot to test from the list of bots.
3. From the left menu, choose **Aliases**.
4. From the list of aliases, choose the alias to test.
5. From **Languages**, choose the radio button of the language to test, and then choose **Test**.

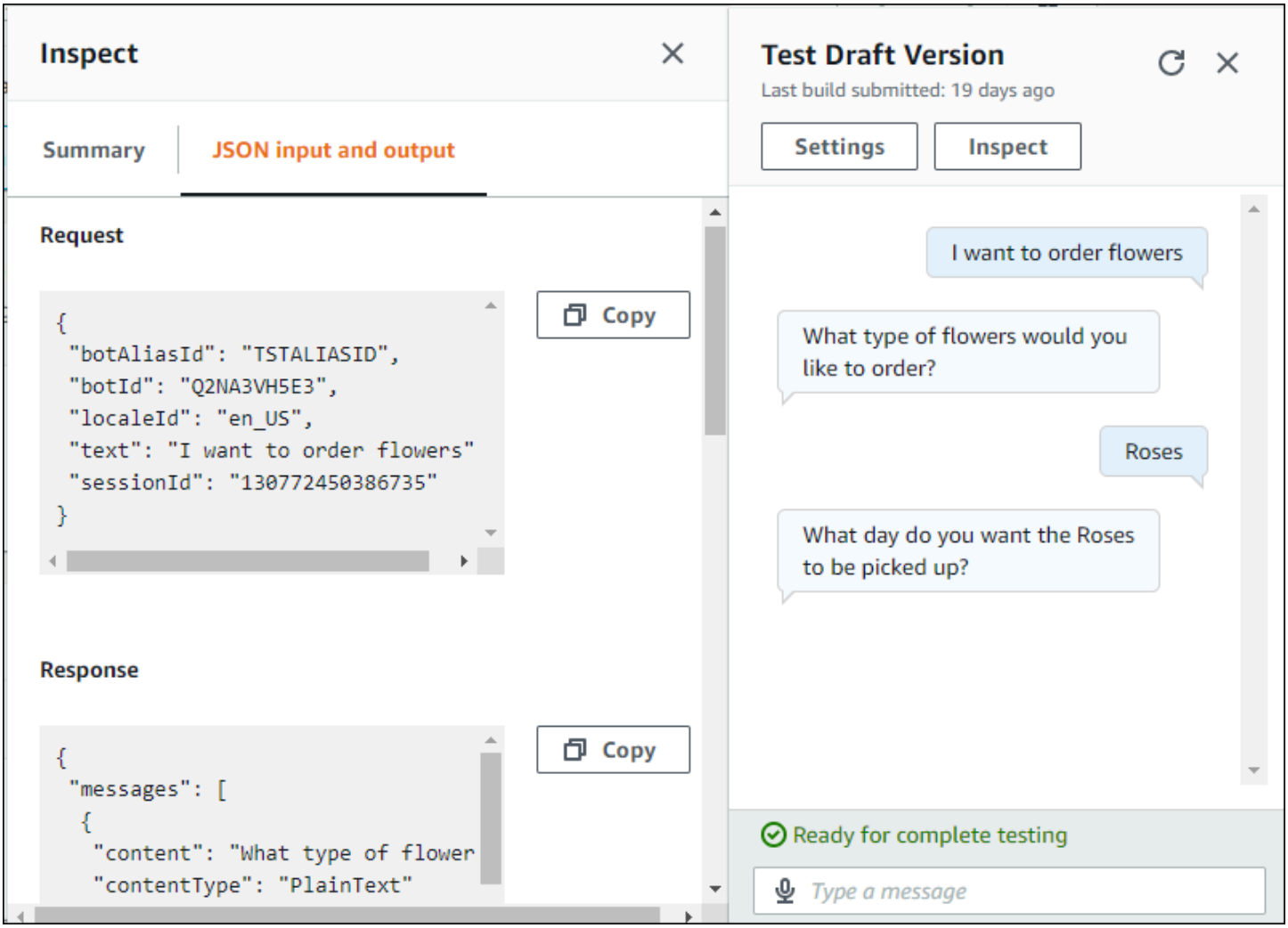
After you choose **Test**, the test window opens in the console. You can use the test window to interact with your bot, as shown in the following graphic.



In addition to the conversation, you can also choose **Inspect** in the test window to see the responses returned from the bot. The first view shows you a summary of the information returned from your bot to the test window.



You can also use the test inspection window to see the JSON structures that are sent between the bot and the test window. You can see both the request from the test window and the response from Amazon Lex V2.


Inspect
Summary
JSON input and output
Request
{
"botAliasId": "TSTALIASID",
"botId": "Q2NA3VH5E3",
"localeId": "en_US",
"text": "I want to order flowers"
"sessionId": "130772450386735"
}
Copy
Response
{
"messages": [
{
"content": "What type of flower
"contentType": "PlainText"
Copy
Test Draft Version
Last build submitted: 19 days ago
Settings
Inspect
I want to order flowers
What type of flowers would you like to order?
Roses
What day do you want the Roses to be picked up?
Ready for complete testing
Type a message

Optimize Lex V2 bot creation and performance by using generative AI

Take advantage of Amazon Bedrock's generative AI capabilities to automate and speed up your Amazon Lex V2 bot building process. You can carry out the following processes with the help of Amazon Bedrock.

> **Note**
>
> These features uses generative AI. As you use the service, remember that it may give inaccurate or inappropriate responses. For more information, refer to AWS Responsible AI Policy.
> Powered by Amazon Bedrock: AWS implements automated abuse detection. Because the Amazon Lex V2 generative AI features are built on Amazon Bedrock, users inherit the controls implemented in Amazon Bedrock to enforce safety, security, and the responsible use of AI.

- Create new bots and populate them with relevant intents and slot types efficiently using natural language description.
- Automatically generate sample utterances for your bot's intents.
- Improve your bots' slot resolution performance.
- Create an intent to help answer your customer's questions.
- Use Amazon Bedrock Agents and Amazon Bedrock Knowledge Bases to help answer your customer's questions.

You can activate generative AI capabilities for Amazon Lex V2 either through the console or the API.

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.

2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

Using the console

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. Select the bot and the locale in the bot for which you want to turn on generative AI capabilities.
3. In the **Generative AI configurations** section, select **Configure**.
4. Toggle the **Enabled** button for each feature that you want to activate. Select the model and version that you want to use for that feature. Enabling a feature may incur additional charges. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing. To learn more about a feature, select the corresponding topic from the list below. Select **Save** after you turn on the features that you want to activate. A green success banner appears to confirm that the capabilities are turned on.

Using the API

1. To enable generative AI capabilities for a new bot, use the CreateBot operation to create a new bot.
2. Send a CreateBotLocale request, modifying the `generativeAISettings` object as necessary. If you are enabling the capabilities for an existing bot, send a UpdateBotLocale request instead.

    a. To enable usage of the descriptive bot builder, modify the `descriptiveBotBuilder` object. Specify the foundation model to use in the `modelArn` field and set the `enabled` value to `True`.

    b. To enable slot resolution improvement, modify the `slotResolutionImprovement` object. Specify the foundation model to use in the `modelArn` field and set the `enabled` value to `True`.

    c. To enable sample utterance generation, modify the `sampleUtteranceGeneration` object. Specify the foundation model to use in the `modelArn` field and set the `enabled` value to `True`.

Topics

Use a description to build a bot in Lex V2 with the descriptive bot builder

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
> 2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

The descriptive bot builder lets you take advantage of Amazon Bedrock's access to large language models to improve the efficiency of the bot creation process. You provide a prompt using natural language that includes the purpose of the bot and the actions that it should perform. Amazon Lex V2 harnesses Amazon Bedrock's capabilities to generate relevant intents and slot types for your bot based on your description. Once you choose the intents and slot types that you want to keep, you can then iterate upon the bot to modify it to your specific use-case. The descriptive bot builder saves you time by letting you avoid having to manually create intents and slot types for the bot.

The descriptive bot builder is available in the English locales (see the locales that begin with en_ in the table in Languages and locales supported by Amazon Lex V2).

Before you create your bot, do the following.

1. Check that your role has the correct permissions by reviewing the steps at Permissions needed to create a bot with natural language description in Lex V2.
2. Decide on the description to use. You can refer to Example bot descriptions for descriptive bot builder for sample bot descriptions.

Create a bot by using natural language to describe what the bot should be able to do. Amazon Lex V2 invoke Amazon Bedrock models to generate intents and slot types that fit your bot's use case. You can create the bot with either the console or the API.

Console

Create a bot using the descriptive bot builder

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the **Bots** page, select **Create bot**.
3. For the **Creation method**, choose **Descriptive Bot Builder - GenAI**.
4. Give your bot a name and optional description, configure the IAM permissions, and choose whether your bot is subject to COPPA or not. Then select **Next**.
5. Select a language to create the bot in, a voice for the bot, and a confidence threshold for intent classification (for more information, see Using intent confidence scores to improve intent selection with Lex V2.
6. Under **Descriptive Bot Builder - GenAI**, provide a description for the bot you want to create. Your description should be both *detailed* and *precise* to help generate appropriate and sufficient intents for your bot. Include a list of actions to improve the intent creation process.
7. Select a model provider and model under **Select model**.
8. To create the bot in another locale, choose **Add another language**. When you are finished adding languages, select **Done**. Amazon Lex V2 creates your bot and the descriptive bot builder generates intents and slots for it. When the locale has been generated, the banner turns from blue to green. Select **Review** to see the generated intents and slot types.

> **Note**
>
> The descriptive bot builder is currently only available in English locales. However, you can copy a bot to a non-English locale after creating it.

Review the generated intents and slot types and add them to your bot

1. If there are enough intents and slot types that are applicable for your bot's use-case, you can review the generated intents.

   a. Review the **Generated intents**.

      i. Choose a checkbox next to an intent to remove it from the list of intents to add to the bot.

      ii. Choose an intent name to view the **Sample utterances** and **Slots** generated for the intent.

      iii. By default, all the utterances and slots are selected. Choose a checkbox to remove that item from the intent. Select **Add to selection** to keep the checked items in the intent.

   b. Review the **Generated slot types**.

      i. Choose a checkbox next to a slot type to remove it from the list of intents to add to the bot.

      ii. You can add values to a slot type after you have added it to the bot

2. When you're satisfied with your intents and slot types, select **Add intents and slot types** at the top of the page to add the intents and slot types to your bot.
3. When the resources are finished being added, a green success banner appears. Go to **Intents** and **Slot types** to edit the generated ones and to add more values.
4. If the **Generated intents** and **Generated slot types** are mostly inapplicable to the bot you want to create, carry out the following steps.

   a. Select **New generation** in the **Descriptive bot builder details** section.

   b. Rewrite the prompt and select **Re-generate** to generate new intents and slot types. The results differ if you use a different model.

> **Important**
>
> There is no guarantee that the same intents and slots will be generated. You are charged each time you regenerate the intents and slot types.

API

Create the bot using natural language description

When you use the descriptive bot builder through the API, it creates a bot definition in a .zip file in an Amazon S3 bucket. You download this file and import the bot definition into Amazon Lex V2 to create your bot.

1. Send a CreateBot request to create a new bot. Then send a CreateBotLocale request to create a locale for the bot.
2. Send a StartBotResourceGeneration request, specifying the ID, version, and locale of the bot. You can use `DRAFT` for the bot version. Provide your prompt in the `generationInputPrompt` field. Your description should be both *detailed* and *precise* to help generate appropriate and sufficient intents for your bot. Include a list of actions to improve the intent creation process.
3. Take note of the `generationId` in the response.
4. Send a DescribeBotResourceGeneration request using the `generationId` you received in the `StartBotResourceGeneration` response. Include the bot ID, version, and locale.
5. If the `generationStatus` in the `DescribeBotResourceGeneration` response is `Complete`, the `generatedBotLocaleUrl` field will also be populated. Use this Amazon S3 URI to download the bot definition by following the steps at Downloading an object.

Check the generated bot definition and import it

1. Use the Amazon S3 URI from the `generationStatus` in the `DescribeBotResourceGeneration` response to download the bot definition by following the steps at Downloading an object.
2. You can directly modify the generated content for your bot's specific use-case by editing the file. You can also send another `StartBotResourceGeneration` request to regenerate intents and slots.

> **Important**
>
> There is no guarantee that the same intents and slots will be generated. You are charged each time you regenerate the intents and slot types.

3. To import the bot definition, follow the steps at Importing bots in Lex V2.
4. After importing, you can modify the generated intents and slots by using the UpdateIntent, UpdateSlot, and UpdateSlotType operations.

To list metadata about all the generated items for a bot locale, use the ListBotResourceGenerations operation. Use any of the returned `generationId` values in a `DescribeBotResourcGeneration` request to retrieve the Amazon S3 URI for a generated bot definition.

Topics

- Example bot descriptions for descriptive bot builder
- Permissions needed to create a bot with natural language description in Lex V2

Example bot descriptions for descriptive bot builder

Here are some helpful example bot descriptions you can use with descriptive bot builder in Amazon Lex V2.

Industry	Example prompt
Financial services	We are a financial card service that helps users perform with tasks when they receive a new card such as activate card, email or mail a pin, verify a new card (using a zip code). We also help them with tasks associated with their existing card, such as enquire about credit card benefits, report a lost card, request a new card, reset a card pin, or pay a card bill.

Industry	Example prompt
Food services	I want a bot to help customers order food (using item ID, quantity, size), check order status, and cancel an order. Use Order ID for indexing orders.
Airline	We are an airline domain that helps users book flight tickets, check details of a reservati on, obtain receipt for a booked flight, enquire flight status, reschedule booked flights, elicit flight details, and cancel booked flights. You can also generate additional intents if they help support functions in the domain descripti on.
Insurance	Objective: We are an insurance company that sells car, home and annuity insurance policies. I want a bot that can check claim status, file a claim, make policy payments and cancel a policy. We use policy_id and last 4 of SSN for account identification and validation.I expect the bot to have at least the following intents and slots: authentication - policy_id, last4SSNpolicy type: car, home, annuitypolicy status: check balance, check due date, check coveragemake a payment: one-time payment, installments, amount

Industry	Example prompt
Vehicle management	We are building a Towed Cars Lookup bot that helps drivers in a city with whose car has been towed to find where the car is located. This bot should ask the address or location of where the automobile was towed from, and details about the vehicle such as license plate and make, model, and year of car. The bot should reply with location of the towed car parking lot, and hours of operation.
Travel	I am a travel agent and I want a bot to help my customers book a trip to Disney. Disney has several parks all over the world to choose from, and also has hotels, dining, and special entertainment that can be reserved. Users of the bot should be able to modify or cancel their booking. Bookings must include at a minimum the park, dates, and hotel. Including dining or entertainment is optional and can be added or changed later.

Permissions needed to create a bot with natural language description in Lex V2

- To access this feature on Amazon Lex V2 console, ensure your console role has `bedrock:ListFoundationModels` and `bedrock:ListInferenceProfiles` permissions.
- The IAM role associated with the bot should have `bedrock:InvokeModel` permission. When you enable the feature with the Amazon Lex console the policy will get automatically added to the bot role provided your bot is using a service-linked role generated by Amazon Lex.

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/model-id"
            ]
        }
    ]
}
```

Use utterance generation to generate sample utterances for intent recognition

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
> 2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

Use utterance generation to automate the creation of sample utterances for your intent. Instead of manually inputting sample utterances, Amazon Lex V2 generates sample utterances for you based off the intent name, description, and existing sample utterances, so that you can reduce the time and effort you spend in discovering and writing your own sample utterances. After Amazon Lex V2 generates utterances, you can edit and delete the utterances. Use this tool to expedite the creation of sample utterances for the intent recognition process.

To allow utterance generation, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI to activate generative AI capabilities.

To access this feature on Amazon Lex V2 console, ensure your console role has `bedrock:ListFoundationModels`, `bedrock:ListInferenceProfiles`, and `bedrock:InvokeModel` permissions.

You can generate utterances with either the console or the API.

Console

1. Navigate to the **Sample utterances** section of any intent in your bot (in the Visual conversation builder, it is in the **Start** block).
2. Select the **Generate utterances** button to generate 5 sample utterances. If your intent has over 25 sample utterances, the **Generate utterances** button becomes disabled.
3. Generated utterances are displayed with a green banner that differentiates the generated utterances from the existing utterances.
4. Hover over an utterance to display the options to edit, delete, and sort the generated utterances.

API

1. Send a GenerateBotElement request, filling in the intent and bot ID, version, and locale for which you want to generate sample utterances.
2. The response returns a list of SampleUtterance objects, each of which contains a generated utterance.
3. To add the utterances to the intent, send an UpdateIntent request and add the utterances to the `sampleUtterances` field.

Using assisted slot resolution to clarify slot values in Amazon Lex V2

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
> 2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

You can improve the accuracy of some built-in slots in your bot's conversation flow by using assisted slot resolution. Assisted slot resolution uses Amazon Bedrock large language models (LLMs) to improve recognition of some built-in slots, which results in an improved interpretation of customer responses during slot elicitation. For utterances that could not be resolved normally, Amazon Lex V2 will attempt to resolve them a second time using Amazon Bedrock.

Assisted slot resolution allows you to use the power of Amazon Bedrock foundation models to improve the accuracy of the following built-in slots:

- `AMAZON.Alphanumeric` without regex support
- `AMAZON.City`
- `AMAZON.Country`
- `AMAZON.Date`
- `AMAZON.Number`
- `AMAZON.PhoneNumber`
- `AMAZON.Confirmation`

You can enable assisted slot resolution for any intent that uses the above listed built-in slots. Assisted slot resolution does not apply to custom slots or Amazon built-in slots not listed above.

You can gather data on the accuracy improvements after you enable assisted slot resolution in your Amazon Lex V2 bot by using conversation logs and metrics.

- Conversation logs - Interpretations will have the `interpretationSource` as `Bedrock`, if Amazon Bedrock was used to resolve the slot.
- CloudWatch metrics - Metrics will be published under the dimentions listed in CloudWatch metric. To learn more, see Monitoring Amazon Lex with Amazon CloudWatch.

To use the descriptive bot builder, ensure that your IAM role has the proper permissions by following the steps at Permissions needed in Lex V2 for assisted slot resolution.

Topics

- Permissions needed in Lex V2 for assisted slot resolution

Examples of assisted slot resolution used in Lex V2

Below are some examples where assisted slot resolution is able to intelligently resolve user utterances into a value.

AMAZON.Number

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Travel	AMAZON.Number	numberOfNightsStayed	How many nights did you stay for the trip?	A whole week, 7 nights.	7
Banking	AMAZON.Number	numberOfPeopleOnTheAccount	How many people are on the account?	Me and my wife.	2
Travel	AMAZON.Number	numberOfStops	How many stops?	Once in Japan. Once in LA.	2

AMAZON.AlphaNumeric

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Car Rental	AMAZON.Alphanumeric	transactionId	What is your transaction id?	I believe it was alpha whiskey echo eight three four nine romeo juliet.	AWE8349RJ

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Travel	AMAZON.Alphanumeric	confirmationCode	What is the confirmation number for your reservation?	The confirmation number is BLT2UE.	BLT2UE

AMAZON.Date

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value	currentDate
Car Rental	AMAZON.Date	dueDate	When is the rental agreement due to expire?	The lease is up on the 1st of next month.	2023-12-01	2023-11-09
Travel	AMAZON.Date	returnDate	When are you returning?	Later today around 7.	2023-11-09	2023-11-09

AMAZON.PhoneNumber

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Insurance	AMAZON.PhoneNumber	policyHolder	What is the phone number of the policy holder?	The phone number for the policy holder is 123-456-7890.	1234567890

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Retail	AMAZON.PhoneNumber	phoneLookup	What is your phone number so I can find your account?	I think it's under 413-570-9617, let me double check.	4135709617

AMAZON.Country

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Travel	AMAZON.Country	nativeCountry	What is your country of origin?	I am Indian.	India
Banking	AMAZON.Country	countryItinerary	What countries will you be traveling to with your debit card?	I will be travelling to New Delhi.	India

AMAZON.City

Vertical	Slot Type	Intent	Question	Response	Resolved Value
Insurance	AMAZON.City	policyHolderCity	What city does the policy holder reside in?	I live in Springfield.	Springfield

Vertical	Slot Type	Intent	Question	Response	Resolved Value
Travel	AMAZON.City	destinationCity	Which city are you traveling to?	I'm traveling to Tokyo.	Tokyo

AMAZON.Confirmation

Vertical	slotType	slotName	slotPrompt	utterance	Resolved Value
Insurance	AMAZON.Confirmation	policyExpired	Has the insurance policy expired?	Yes, unfortunately it has expired.	Yes
Banking	AMAZON.Confirmation	hasInvestments	Do you have any investments?	I haven't invested in anything yet.	No

Enable assisted slot resolution in the Generative AI configuration screen

You can enable assisted slot resolution for supported built-in slots by navigating to the Generative AI screen.

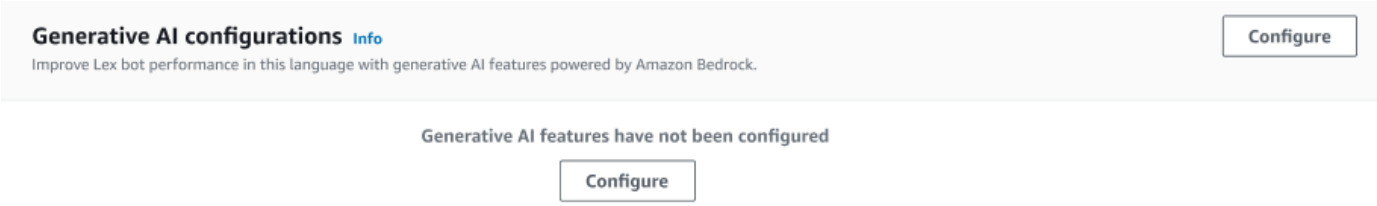
If the slot is an supported built-in slot, you will have the option to activate the assisted slot resolution at the slot level.

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the navigation pane under **Bots**, select the bot you want to use for assisted slot resolution.
3. Select the language **English (US)**) for the bot you want to enable.
4. Go to the **Generative AI configuration** section on the screen.

5. Select **Go to Amazon Bedrock** to sign up and enable the feature, if the feature has not been enabled.

> **Note**
>
> If you do not have access to Amazon Bedrock foundation models, you should see **Go to Amazon Bedrock**. Click on **Go to Amazon Bedrock** to go to the Amazon Bedrock page where you can sign up for access to foundation models. Assisted slot resolution currently supports Anthropic Claude. We suggest using Anthropic Claude for best results.

6. If you already have access to Amazon Bedrock Foundation models, you should see a **Configure** button. Click on this button to go the generative AI configuration page to activate generative AI features in Lex.



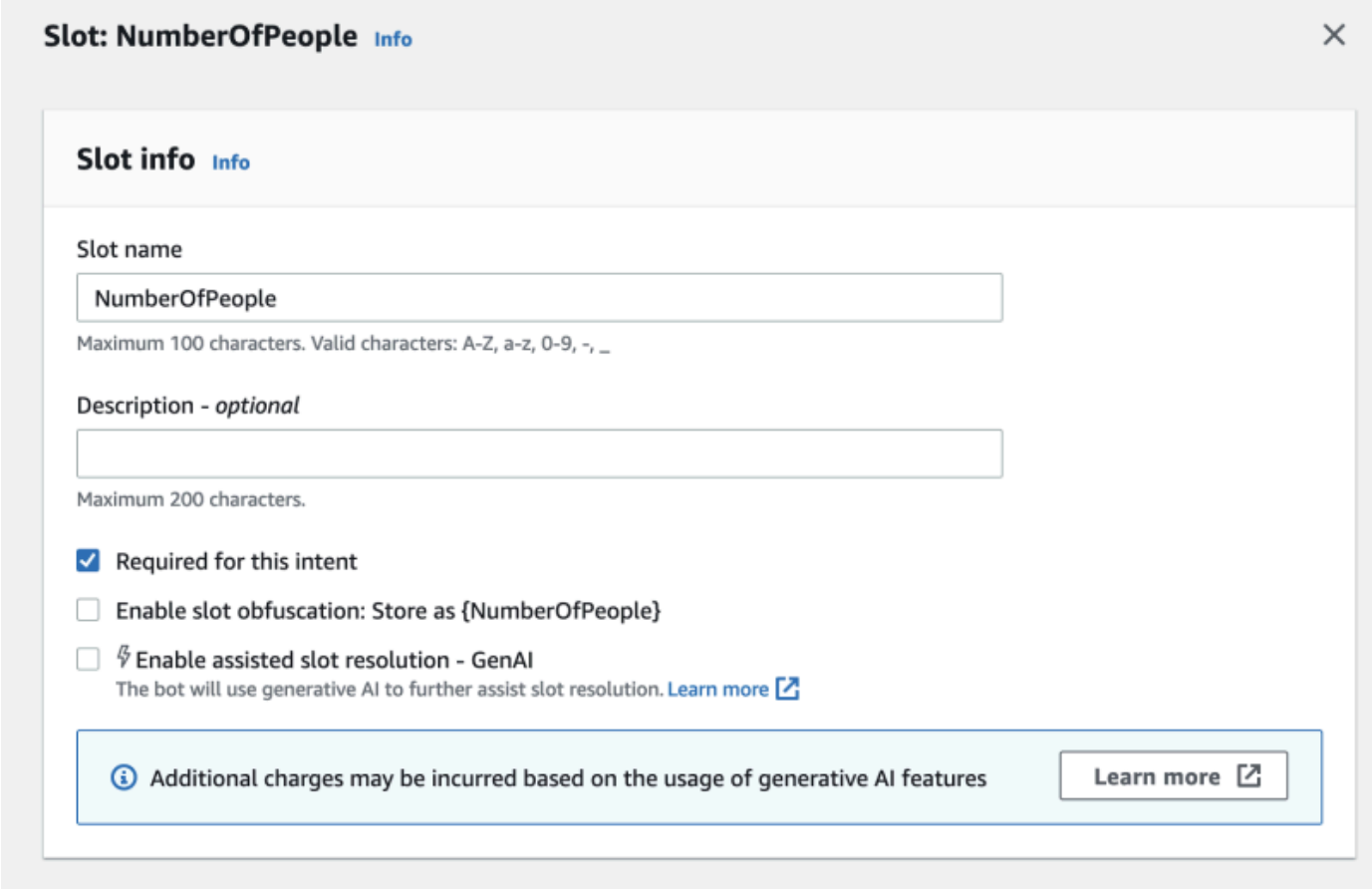
7. In the upper right corner of the box, move the slider to the right to choose the **Enabled** setting.
8. Choose the **Enable** button to activate assisted slot resolution for the selected slots.
9. You can disable assisted slot resolution by selecting the slots from the list and selecting the **Disable** button.

Enable assisted slot resolution in the slot settings in Lex V2

You can enable assisted slot resolution for supported built-in slots by navigating to the slot level for each intent that has slots. The slots must be one of the supported built-in slots listed above to have the option of activating assisted slot resolution. If the slot does not have the option to activate assisted slot resolution, the option will be greyed out.

> **Note**
>
> You must first activate the assisted slot resolution feature on the Generative AI panel in order to activate the feature for individual slots.

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the navigation pane under **Bots**, select the bot you want to use for the assisted slot resolution.
3. Under All languages, select **English (US)** to expand the list.
4. In the left side panel, choose **Intents** to view a list of intents in the bot you selected.
5. In the **Intents** screen, choose the intent that contains the slots you want to modify.
6. Select the intent name to view the slots for that intent.
7. Select the **Advanced Options** button in the **Slots** section.
8. Select the check box for **Enable assisted slot resolution** to enable the feature.



9. Choose the **Update Slot** button in the bottom right corner of the screen. This will activate assisted slot resolution for the slots you have chosen.

You can enable assisted slot resolution for supported built-in slots by making API calls.

- Follow the steps at Optimize Lex V2 bot creation and performance by using generative AI to enable assisted slot resolution for your bot locale.
- Send an UpdateSlot request, specifying the slot for which you want to enable assisted slot resolution. In the `slotResolutionSetting` field, set the `slotResolutionStrategy` value as `EnhancedFallback`. To create a new slot with assisted slot resolution enabled, send an CreateSlot request instead.

Permissions needed in Lex V2 for assisted slot resolution

- To access this feature on Amazon Lex V2 console, ensure your console role has `bedrock:ListFoundationModels` and `bedrock:ListInferenceProfiles` permissions.
- The IAM role associated with the bot should have `bedrock:InvokeModel` permission. When you enable the feature with the Amazon Lex V2 console the policy will get automatically added to the bot role provided your bot is using a service-linked role generated by Amazon Lex V2.

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": [
                "arn:aws:bedrock:us-east-1::foundation-model/modelId"
            ]
        }
    ]
}
```

Using BedrockAgentIntent to use a Amazon Bedrock Agent in Amazon Lex V2

Note

Before you can take advantage of the generative AI features, you must fulfill the following prerequisites

1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

You can take advantage of Amazon Bedrock Agents to handle complex workloads requested by customers without having to go through a comprehensive task definition process. Amazon Lex V2 offers a built-in `AMAZON.BedrockAgentIntent` that you can add to your bot. This intent harnesses generative AI capabilities from Amazon Bedrock by recognizing customer requests, analyzing them, reasoning them, and finally responding. It also has the capability to ask any follow-up questions in order to achieve the task needed (for example, image you defined a retail agent that can check customer's order status. When the customer asks for order status, agent first requests `customerId` or associated `emailId` to retrieve the details, and finally responds with correct order status). You can also decide to integrate your AMAZON.BedrockAgentIntent with a Bedrock Knowledge Base to directly answer any customer queries.

Ensure that your IAM role has the proper permissions to access the AMAZON.BedrockAgentIntent by following the steps at Permissions for the AMAZON.BedrockAgentIntent

To take advantage of the AMAZON.BedrockAgentIntent you must have set up one of the following knowledge stores.

- Amazon Bedrock Agents – For more information, see Creating Bedrock Agents.
- Amazon Bedrock Knowledge Base – For more information, see Building a Knowledge Base.

To use the AMAZON.BedrockAgentIntent, ensure that your IAM role has the proper permissions by following the steps at Permissions needed in Lex V2 for Bedrock Agent Intent.

Topics

Enable Bedrock Agent Intent in the Generative AI configuration screen

You can enable Bedrock Agent Intent by navigating to the Generative AI screen.

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the navigation pane under **Bots**, select the bot you want to use for Bedrock Agent Intent.
3. Select the language for the bot you want to enable.
4. Go to the **Generative AI configuration** section on the screen, and select **Configure**.
5. In the BedrockAgentIntent configuration section, select **Create BedrockAgent intent**.

Enable Bedrock Agent Intent by adding a built-in intent to your bot

You can enable Bedrock Agent Intent by adding a built-in intent to your Amazon Lex V2 bot.

> **Note**
>
> You must first activate the Bedrock Agent Intent feature on the Generative AI panel in order to activate the feature for individual bots.

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the navigation pane under **Bots**, select the bot you want to use for the Bedrock Agent Intent.
3. Under All languages, select **English (US)** to expand the list.
4. Select **Add intent** and choose **Use built-in intent** from the dropdown menu.
5. For more details about configurations for the AMAZON.BedrockAgentIntent, see AMAZON.BedrockAgentIntent.

Permissions needed in Lex V2 for Bedrock Agent Intent

- To access this feature on Amazon Lex V2 console, ensure your console role has `bedrock:ListFoundationModels` and `bedrock:ListInferenceProfiles` permissions.
- The IAM role associated with the bot should have permissions required for `AMAZON.BedrockAgentIntent`.

  The bot role should have permissions for calling `bedrock:InvokeAgent` if using Amazon Bedrock Agents, and `bedrock:InvokeModel` and `bedrock:RetrieveAndGenerate` if using Amazon Bedrock Knowledge Bases in the intent. You should also attach a statement for each of the agents that you specify in your bots' `AMAZON.BedrockAgentIntent` (see the Permissions to access Amazon Bedrock Agent or Permissions to access Amazon Bedrock Knowledge Bases statement in the policy below).

  When you enable the feature with the Amazon Lex console the policy will get automatically added to the bot role provided your bot is using a service-linked role generated by Amazon Lex.

Sample request with session attributes

The following example shows how to invoke the AMAZON.BedrockAgentIntent and demonstrates the session and request attributes that are populated in the response. These attributes contain the Bedrock Agent's response data and can be used to access the agent's output, Amazon Bedrock Knowledge Base sources, and action group invocation details.

```
{
    "sessionId": "user-session-123",
    "messages": [{
        "content": "Your order #12345 is currently being processed and will ship within
 2-3 business days. You will receive a tracking number via email once it ships.",
        "contentType": "PlainText"
    }],
    "sessionState": {
        "sessionAttributes": {
            "x-amz-lex:bedrock-agent-search-response": "Your order #12345 is currently
 being processed and will ship within 2-3 business days. You will receive a tracking
 number via email once it ships.",
            "x-amz-lex:bedrock-knowledge-base-search-response-source": "[{\"title\":
 \"Order Processing Guide\", \"uri\": \"s3://knowledge-base/orders/processing.pdf\",
 \"excerpt\": \"Standard orders typically ship within 2-3 business days...\"}]",
```

```
            "x-amz-lex:bedrock-agent-action-group-invocation-input":
 "{\"actionGroupName\": \"OrderLookup\", \"function\": \"getOrderStatus\", \"parameters
\": {\"orderId\": \"12345\", \"customerId\": \"67890\"}}",
            "x-amz-lex:bedrock-agent-knowledge-base-lookup-input": "{\"knowledgeBaseId
\": \"KB123456\", \"query\": \"order status processing time\", \"numberOfResults\": 3}"
        },
        "intent": {
            "name": "BedrockAgentIntent",
            "slots": {},
            "state": "Fulfilled",
            "confirmationState": "None"
        },
        "dialogAction": {
            "type": "ElicitIntent"
        }
    },
    "interpretations": [{
        "intent": {
            "name": "FallbackIntent",
            "slots": {}
        },
        "interpretationSource": "Lex"
    }],
    "requestAttributes": {
        "x-amz-lex:channels:platform": "Web",
        "x-amz-lex:accept-content-types": "PlainText",
        "x-amz-lex:bedrock-agent-search-response": "Your order #12345 is currently
 being processed and will ship within 2-3 business days. You will receive a tracking
 number via email once it ships.",
        "x-amz-lex:bedrock-knowledge-base-search-response-source": "[{\"title\":
 \"Order Processing Guide\", \"uri\": \"s3://knowledge-base/orders/processing.pdf\",
 \"excerpt\": \"Standard orders typically ship within 2-3 business days...\"}]",
        "x-amz-lex:bedrock-agent-action-group-invocation-input": "{\"actionGroupName
\": \"OrderLookup\", \"function\": \"getOrderStatus\", \"parameters\": {\"orderId\":
 \"12345\", \"customerId\": \"67890\"}}",
        "x-amz-lex:bedrock-agent-knowledge-base-lookup-input": "{\"knowledgeBaseId\":
 \"KB123456\", \"query\": \"order status processing time\", \"numberOfResults\": 3}"
    }
}
```

In this example, the session attributes show how the BedrockAgentIntent populates response data including the agent's answer, Amazon Bedrock Knowledge Base sources used, action group

invocations, and Amazon Bedrock Knowledge Base lookup details that were used to generate the response.

For more information, see AMAZON.BedrockAgentIntent.

Improve intent classification and slot resolution in Lex V2 with assisted NLU

Assisted NLU is a feature that uses Large Language Models (LLMs) to improve Amazon Lex V2's intent classification and slot resolution capabilities. It enhances accuracy while staying within your bot's configured intents and slots. The feature does not generate or modify any bot content. This feature helps to improve the overall accuracy of the NLU system, resulting in a more seamless and effective conversational experience for users.

The assisted NLU feature is available in English, Spanish, Portuguese, Catalan, French, Italian, and German locales. Specifically, it supports locales that begin with `en_`, `es_`, `pt_` (pt_BR, pt_PT), `ca_` (ca_ES), `fr_` (fr_CA, fr_FR), `it_` (it_IT), `de_` (de_AT, de_DE), `zh_` (zh_CN, zh_HK), `ja_JP`, and `ko_KR`. For the complete list of supported locales, see the table in Languages and locales supported by Amazon Lex V2.

Use assisted NLU to improve intent classification and slot resolution. Amazon Lex V2 invokes Amazon Bedrock models to help classify intents and resolve slot types that fit your bot's use case. You can enable assisted NLU for your bot with the console.

Assisted NLU Mode

In Primary mode, Lex will default to utilizing the LLM as the primary means of processing the user input to determine the user Intent, as well as to fill slot values.

In Fallback mode, Lex will use the LLM for determining user intent if the confidence score determined by NLU is lower than the configured threshold or otherwise routing to the FallbackIntent, as well as to determine slot values from user inputs if the traditional NLU does not capture a value.

Console

Using assisted NLU with your Amazon Lex V2 bot

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.

2. In the **Bots** page, select the bot you want to use with assisted NLU.
3. On the **Bot Locale** page, click on **Configure** under the **Assisted NLU** section.
4. Under the Runtime generative AI features section, you can see the Assisted NLU feature. Use the toggle button beside it to enable LLM Assisted NLU feature. You can then select Primary or Fallback mode, and click **Save**.
5. Verify that the LLM Assisted NLU feature is enabled under Assisted NLU section in the Bot Locale page.
6. Build the bot to see the changes are reflected in your bot in Runtime.
7. Once the bot build is completed, you can use the test panel in the console or run a test set to see the improvements after enabling LLM Assisted NLU feature.

Guidance to improve the accuracy of your bot when using the LLM Assisted NLU Feature

The following best practices can help you maximize the effectiveness of the Assisted NLU feature:

1. **Make Intent Names Self-Explanatory** — Use names that immediately convey the action or purpose of the intent. For example, if you're creating an intent for booking flights, simply call it "BookFlight".
2. **Keep Names Clean and Simple** — Avoid adding prefixes, suffixes, or unnecessary words to your intent and slot names. Extra elements like "Dev" or "Test" can confuse the LLM and make the purpose less clear.
3. **Provide Detailed Descriptions** — For each custom intent and slot, include a brief but informative description. This helps explain its specific use and context, making it easier for both humans and the LLM to understand its purpose.

> **Note**
>
> When you enable this feature, your data might be processed across AWS Regions. For more information on Cross-Region Inference, see https://docs.aws.amazon.com/bedrock/latest/userguide/cross-region-inference.html.

> **Important**
>
> Enable this feature in a draft version of the bot. Test it before using it in a production alias.

Disabling Assisted NLU

To disable the Assisted NLU feature, follow these steps:

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the **Bots** page, select your bot.
3. On the **Bot Locale** page, click on **Configure** under the **Assisted NLU** section.
4. Under the Runtime generative AI features section, toggle off the Assisted NLU feature, and click **Save**.
5. Build the bot to apply the changes.

Resolve ambiguous user inputs with Intent Disambiguation

Intent Disambiguation is an improvement to Assisted NLU that helps resolve ambiguous user inputs when multiple intents could match. When enabled, the system presents clarifying questions to users, helping them specify their exact intent for improved conversation accuracy. The system uses a large language model (LLM) that analyzes the intent names and descriptions as context, and based on the ambiguity of the user utterance, returns the most likely matching intents. The LLM evaluates whether the user input clearly matches a single intent or multiple intents and is ambiguous enough to require disambiguation, then provides the candidate intents.

The Intent Disambiguation feature is available in English, Spanish, Portuguese, Catalan, French, Italian, German, Chinese, Japanese, and Korean locales. Specifically, it supports locales that begin with `en_`, `es_`, `pt_` (pt_BR, pt_PT), `ca_` (ca_ES), `fr_` (fr_CA, fr_FR), `it_` (it_IT), `de_` (de_AT, de_DE), `zh_` (zh_CN, zh_HK), `ja_JP`, and `ko_KR`. For the complete list of supported locales, see the table in Languages and locales supported by Amazon Lex V2.

You can configure the following options for Intent Disambiguation:

Number of Intent Options

Configure the maximum number of intents (2-5) to present to users when disambiguation is needed. This setting determines how many intent options will be shown to users when the system detects ambiguous input. The default value is 3, which provides a good balance between giving users sufficient options while keeping the selection manageable.

Disambiguation Message

Provide a custom message that will be displayed before presenting the disambiguation options to users. This message helps set the context for users and can be customized to match your bot's tone and brand. If not specified, a default message will be used.

Intent Display Names

Configure user-friendly display names for your intents to improve the disambiguation experience. This is recommended when your intent names are technical or not suitable for display to end users. Display names will be shown to users during disambiguation instead of the technical intent name.

Console

Using Intent Disambiguation with your Amazon Lex V2 bot

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the **Bots** page, select the bot you want to use with Intent Disambiguation.
3. On the **Bot Locale** page, click on **Configure** under the **Assisted NLU** section.
4. Enable Assisted NLU and select either Primary or Fallback mode (Intent Disambiguation works with both modes).
5. In the **Intent Disambiguation** section within the Assisted NLU configuration, use the toggle button to enable the Intent Disambiguation feature.
6. Configure the following optional settings:
   - **Number of Intent options:** Select the maximum number of intents (2-5) to present to users during disambiguation. Default is 3.
   - **Disambiguation Message:** Provide a custom message that will be displayed when presenting intent options. If not specified, a default message will be used.
7. Click **Save** to apply the configuration.
8. Optionally configure Intent Display Names for better user experience:
   a. Navigate to each intent in your bot that you want to configure.
   b. In the Intent Editor Page, locate the **Display name** field.
   c. Enter a user-friendly name that will be shown to users during disambiguation instead of the Intent name.
9. Build the bot to see the changes reflected in your bot at runtime.

Guidance to improve the effectiveness of your bot when using the Intent Disambiguation Feature

The following best practices can help you maximize the effectiveness of the Intent Disambiguation feature:

1. **Clear Intent Names and Descriptions:** Ensure intent names and descriptions are clean, clear and do not overlap with other intents, as these are the main inputs provided to the LLM for disambiguation.
2. **Descriptive Display Names:** If current Intent names are technical, use descriptive display names that clearly communicate the intent's purpose. Display names should resemble or match the intent names.
3. **Appropriate Maximum Intents:** Set max number of intent options based on your preference and testing.
4. **Custom Messages:** Create concise disambiguation messages that acknowledge the user's input and lead to the intent options.
5. **Test Scenarios:** Test with ambiguous utterances to ensure the disambiguation experience feels natural with intent names or intent display names and custom messages, and verify that the correct intent options are being presented during disambiguation prompt.

> **Important**
>
> Enable this feature in a draft version of the bot. Test it before using it in a production alias.

Disabling Intent Disambiguation

To disable the Intent Disambiguation feature, follow these steps:

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the **Bots** page, select your bot.
3. On the **Bot Locale** page, click on **Configure** under the **Assisted NLU** section.
4. In the **Intent Disambiguation** section within the Assisted NLU configuration, toggle off the Intent Disambiguation feature, and click **Save**.

5. Build the bot to apply the changes.

Optimize Bot using AI-powered Bot Analyzer

Analyze your Amazon Lex V2 bot configuration against AWS best practices using AI-powered recommendations. Bot Analyzer uses Amazon Bedrock's generative AI capabilities to identify configuration issues and provide actionable guidance to improve intent classification and slot resolution performance.

Bot Analyzer automatically evaluates your bot's intent configurations and provides recommendations to:

- **Improve intent separation** - Identify and resolve generic intents that group multiple concepts
- **Eliminate intent overlap** - Detect similar meanings and phrasings between intents that cause routing errors
- **Optimize slot usage** - Recommend proper use of slots to combine similar intents and improve entity extraction
- **Enhance utterance quality** - Analyze sample utterance coverage and diversity for better intent classification

Prerequisites

Before using Bot Analyzer, ensure:

- Your bot locale is built successfully
- Your bot version for analysis is `DRAFT`
- Your bot locale is one of the supported English locales: `en_AU`, `en_GB`, `en_IN`, `en_US`, `en_ZA`

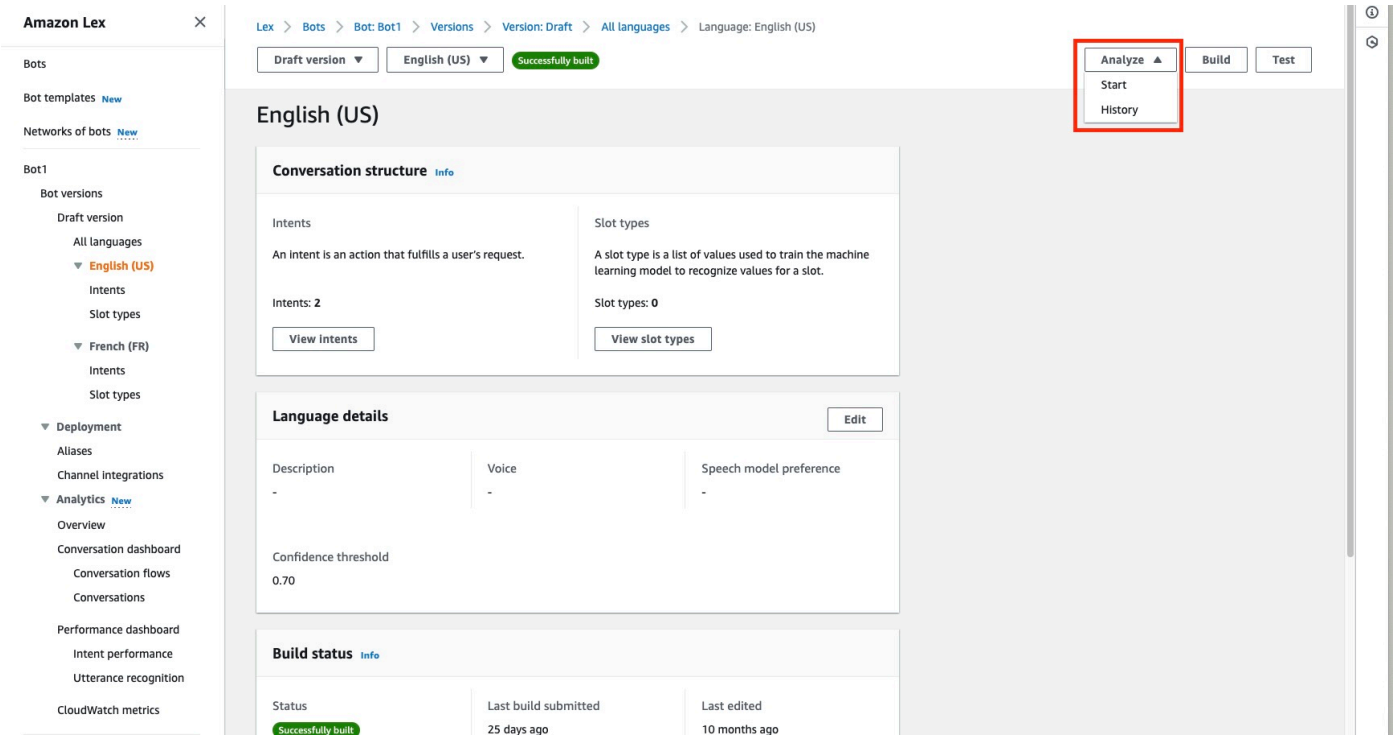
You can use Bot Analyzer with either the console or the API.

Console

Start an analysis

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.

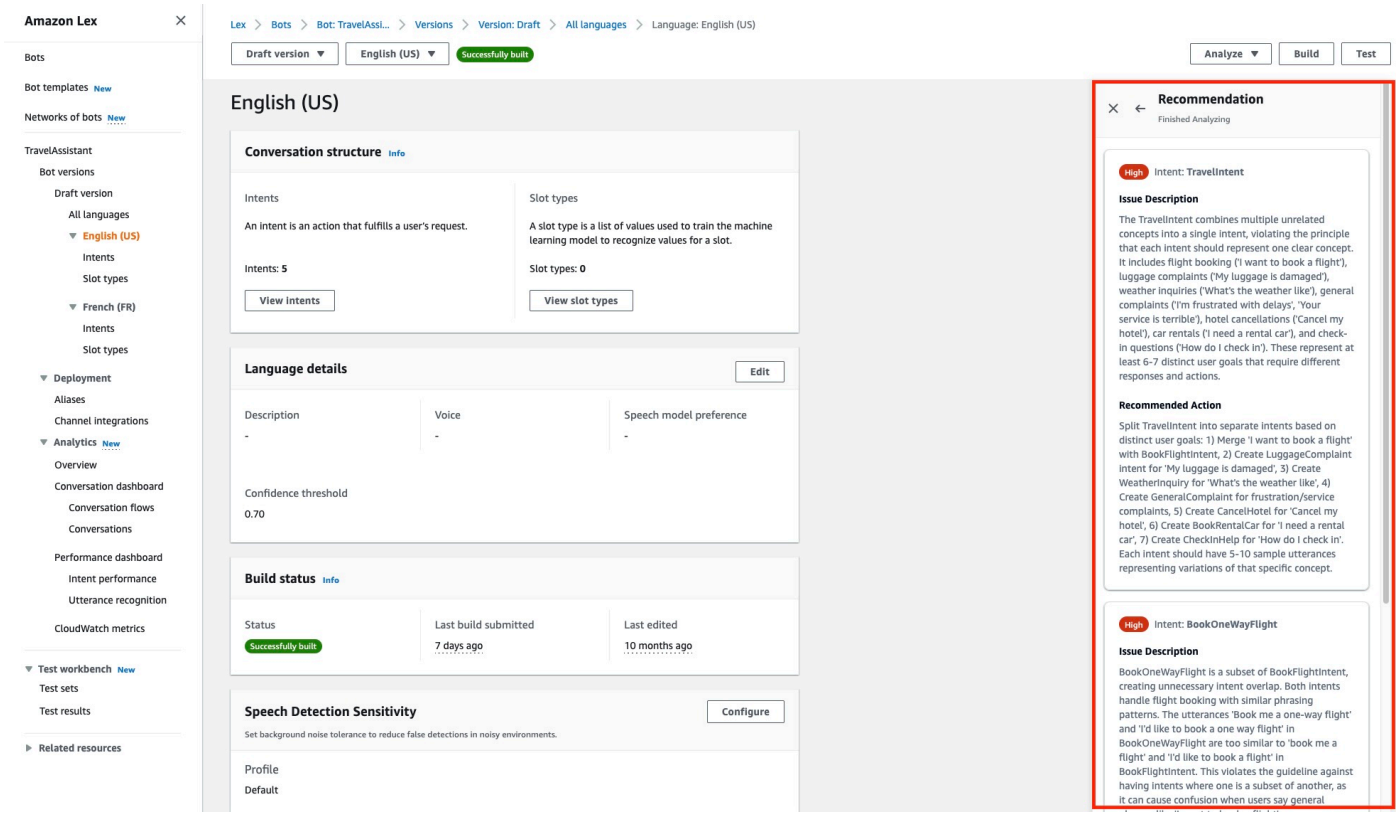
2. Select your bot and navigate to the bot locale you want to analyze.
3. In the bot locale editor, click the **Analyze** dropdown menu.
4. Select **Start** to begin the analysis.



The analysis typically completes within minutes. During analysis, the **Start** button changes to **Stop Analyzing** if you need to cancel.

View Recommendations

Once analysis completes, recommendations appear in the **Recommendations** panel on the right side.



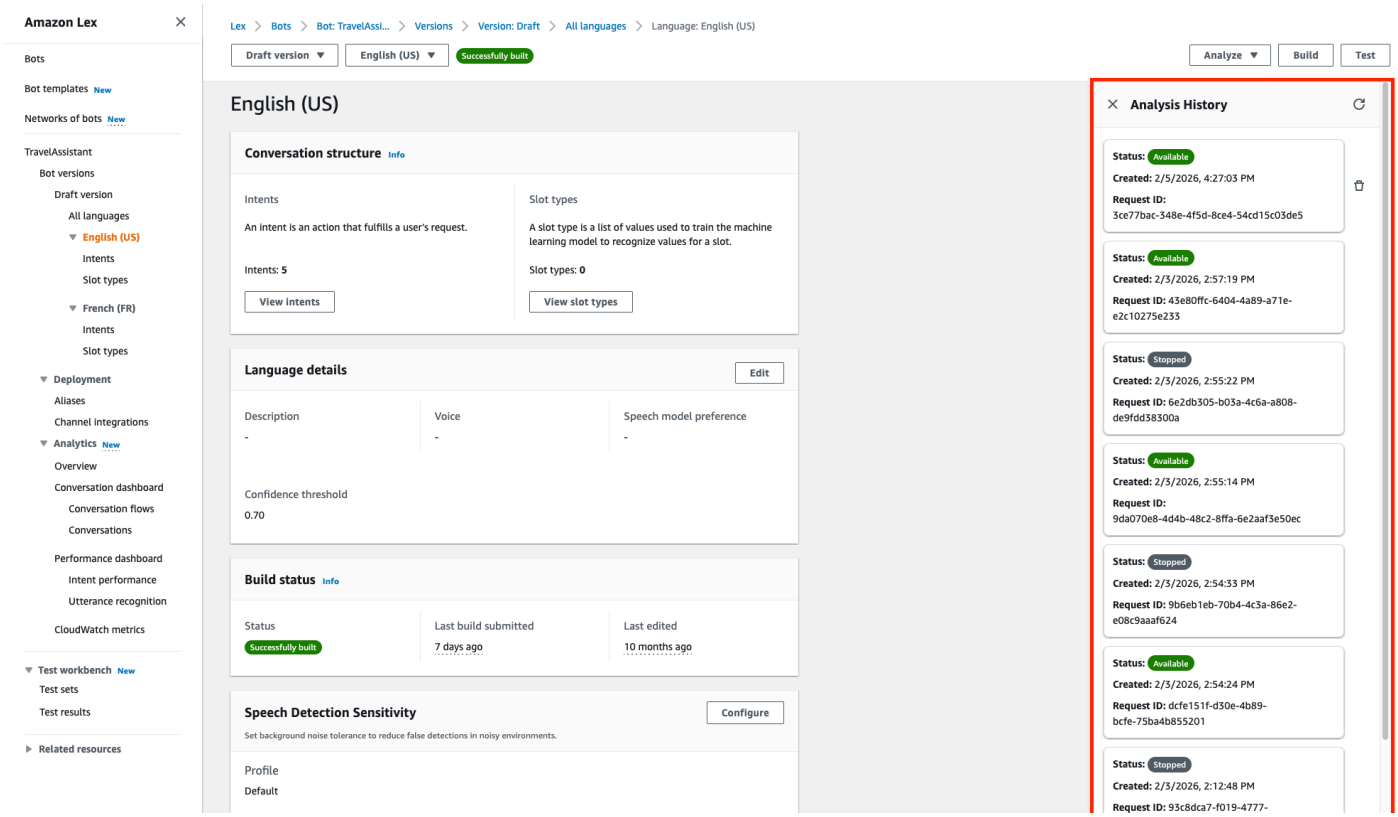
Each recommendation includes:

- **Priority** - High, Medium, or Low severity
- **Issue Location** - The specific intent affected
- **Issue Description** - What configuration problem was detected
- **Proposed Fix** - Actionable steps to resolve the issue

View analysis history

To see previous analyses:

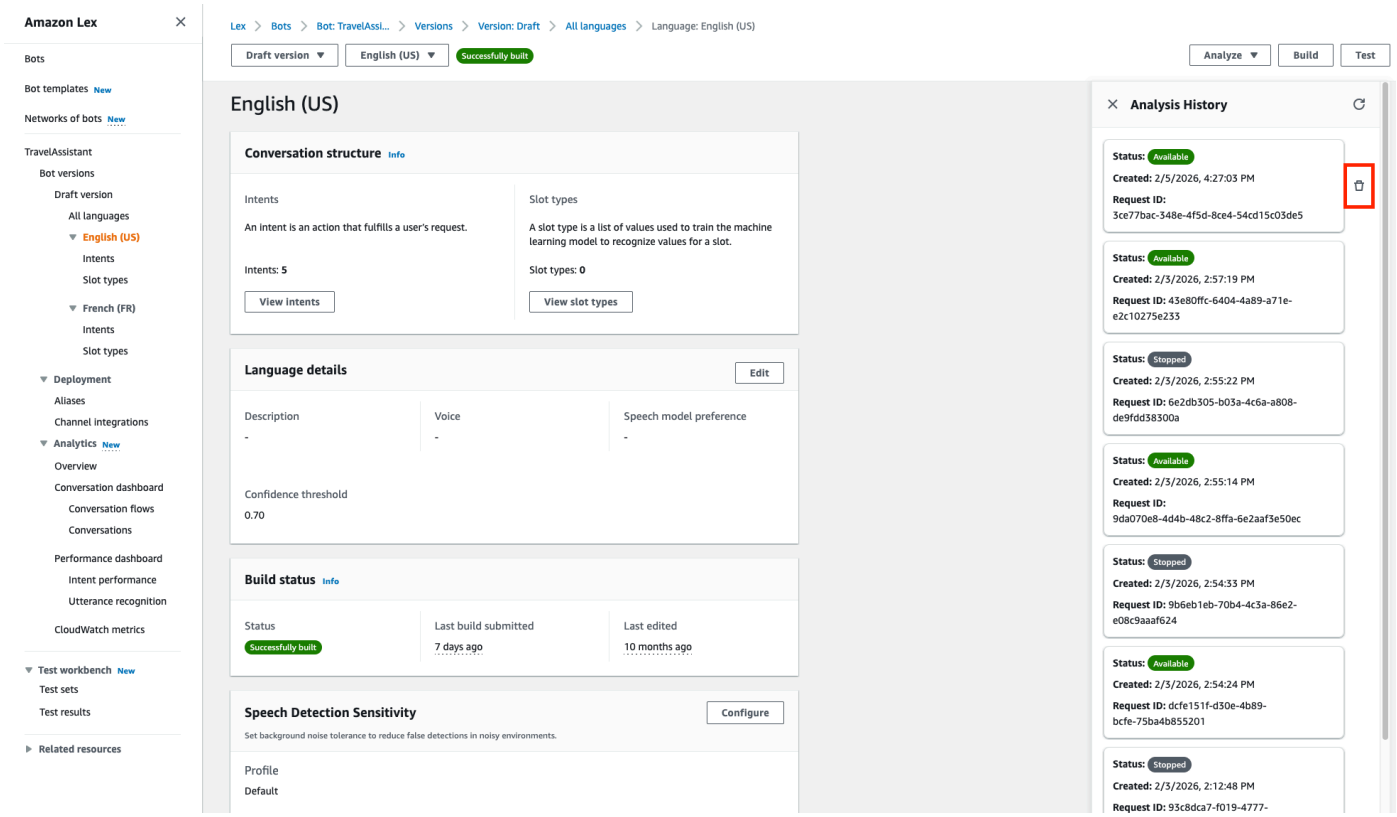
1. Click the **Analyze** dropdown menu.
2. Select **History**.
3. The **Analysis History** panel displays past analysis requests with their status and timestamps.



Delete Recommendations

To remove analysis results:

1. Click the **Analyze** dropdown menu.
2. Select **Delete**.
3. Confirm deletion of the current recommendations.



API

Start an analysis

Send a `StartBotAnalyzer` request to initiate analysis for your bot locale. The response returns an HTTP 202 status with a `botAnalyzerRequestId`. Take note of this ID and you'll need it to check the analysis status and retrieve recommendations.

Check analysis status and retrieve recommendations

Send a `DescribeBotAnalyzerRecommendation` request using the `botAnalyzerRequestId` from the previous step. Include the `botId` in the request path.

If the `botAnalyzerStatus` in the response is `Available`, the analysis is complete and the `botAnalyzerRecommendationList` field will be populated with recommendations. Each recommendation includes:

- `issueLocation` - The location where the issue was detected
- `priority` - High, Medium, or Low severity
- `issueDescription` - Details about the configuration problem
- `proposedFix` - Actionable guidance to resolve the issue

Stop an ongoing analysis

If you need to cancel an analysis in progress, send a `StopBotAnalyzer` request with the `botId` and `botAnalyzerRequestId`.

View analysis history

To retrieve a list of previous analyses for a bot locale, send a `ListBotAnalyzerHistory` request. Specify the `botId` and `localeId` to see all past analysis requests with their status and timestamps.

Delete recommendations

To remove analysis results, send a `DeleteBotAnalyzerRecommendation` request with the `botId` and `botAnalyzerRequestId`. This permanently deletes the recommendations associated with that analysis.

> **Note**
>
> Recommendations are automatically deleted after 15 days.

Related Topics

- Best practices for creating Amazon Lex interaction models
- Improve intent classification and slot resolution in Lex V2 with assisted NLU
- Using assisted slot resolution to clarify slot values in Amazon Lex V2

AMAZON.QnAIntent

> **Note**
>
> Before you can take advantage of the generative AI features, you must fulfill the following prerequisites
>
> 1. For information about pricing for using Amazon Bedrock, see Amazon Bedrock pricing.
> 2. Turn on the generative AI capabilities for your bot locale. To do so, follow the steps at Optimize Lex V2 bot creation and performance by using generative AI.

You can take advantage of Amazon Bedrock FMs to help answer customer questions in a bot conversation. Amazon Lex V2 offers a built-in `AMAZON.QnAIntent` that you can add to your bot. This intent harnesses generative AI capabilities from Amazon Bedrock by recognizing customer questions and searching for an answer from the following knowledge stores (for example, **`Can you provide me details on the baggage limits for my international flight?`**). This feature reduces the need to configure questions and answers using task-oriented dialogue within Amazon Lex V2 intents. This intent also recognizes follow-up questions (for example, **`What about domestic flight?`**) based on the conversation history and provides the answer accordingly.

Ensure that your IAM role has the proper permissions to access the `AMAZON.QnAIntent` by following the steps at Permissions for the AMAZON.QnAIntent.

To take advantage of the `AMAZON.QnAIntent` you must have set up one of the following knowledge stores.

- Amazon OpenSearch Service database – For more information, see Creating and managing Amazon OpenSearch Service domains.
- Amazon Kendra index – For more information, see Creating an index.
- Amazon Bedrock knowledge base – For more information, see Building a knowledge base.

You can set up the `AMAZON.QnAIntent` in one of two ways:

To set up using Generative AI configurations

1. In the Amazon Lex V2 console, select **Bots** from the left navigation pane and choose the bot for which you want to add the intent from the **Bots** section.
2. From the left navigation pane, select the language for which you want to add the intent.
3. In the **Generative AI configurations** section, select **Configure**.
4. In the **QnA configurations** section, select **Create QnA intent**.

To set up by adding a built-in intent to your bot

1. In the Amazon Lex V2 console, select **Bots** from the left navigation pane and choose the bot for which you want to add the intent from the **Bots** section.
2. From the left navigation pane, select **Intents** under the language for which you want to add the intent.

3. Select **Add intent** and choose **Use built-in intent** from the dropdown menu.
4. For more details about configurations for the `AMAZON.QnAIntent`, see AMAZON.QnAIntent.

> **Note**
>
> The `AMAZON.QnAIntent` is activated when an utterance is not classified into any of the other intents present in the bot. This intent is activated when an utterance is not classified into any of the other intents present in the bot. Note that this intent will not be activated for missed utterances when eliciting a slot value. Once recognized, the `AMAZON.QnAIntent` uses the specified Amazon Bedrock model to search the configured knowledge base and respond to the customer question.

Topics

Permissions for the AMAZON.QnAIntent

To access this feature on Amazon Lex V2 console, ensure your console role has `bedrock:ListFoundationModels` and `bedrock:ListInferenceProfiles` permissions.

The IAM role associated with the bot should have the following permissions required for `AMAZON.QnAIntent`. The bot role should have permissions for calling `bedrock:InvokeModel`. You should also attach a statement for each data stores that you specify in your bots' `AMAZON.QnAIntent` (see the `Permissions to access Amazon Kendra index`, `Permissions to access OpenSearch Service index`, and `Permissions to access knowledge base in Amazon Bedrock` statements in the policy below). When you enable the feature with the Amazon Lex console, the policies will automatically get added to the bot role provided your bot is using a service-linked role generated by Amazon Lex.

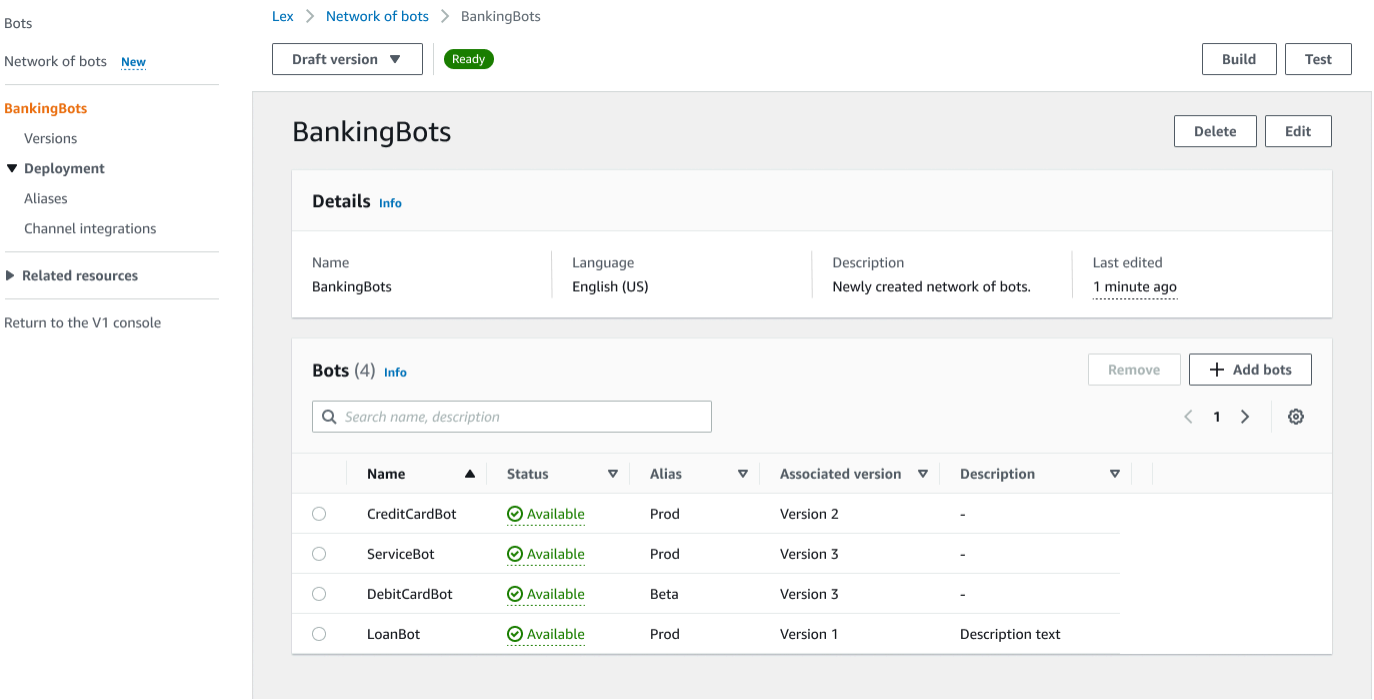
Creating a network of bots for your Lex V2 bots

Network of Bots enables enterprises to deliver a unified user experience across multiple bots. With Network of Bots, enterprises can add multiple bots to a single network to enable flexible and independent bot lifecycle management. The network exposes a single unified interface to the end-user and routes the request to the appropriate bot based on user input.

Teams can collaborate to create a network of bots to meet various business needs by maintaining and adding bots to the network as improved bots are deployed to production. Developers can simplify and speed up deployment and improvements by integrating multiple bots into a single network.

Network of bots is currently only available in the en-US language.

> **Note**
>
> Currently, a network of bots is limited to one account. You cannot add bots from other accounts.



Create a network of bots for your Lex V2 bots

Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home. Choose **Network of bots** from the side menu. You must have built at least one bot in order to create a network of bots.

Step 1: Configure network of bot settings

1. In the **Details** section, enter the name of your network and give it an optional description.
2. In the **IAM permissions** section, choose an AWS Identity and Access Management (IAM) role that provides Amazon Lex V2 permission to access other AWS services, such as Amazon CloudWatch. You can have Amazon Lex V2 create the role, or you can choose an existing role with CloudWatch permissions. See Identity and access management for Amazon Lex V2 for more information.
3. In the **Children's Online Privacy Protection Act (COPPA)** section, choose the appropriate response. See DataPrivacy for more information.
4. In the **Idle session timeout** section, choose the duration that Amazon Lex V2 keeps a session with a user open. Amazon Lex V2 maintains session variables for the duration of the session so that your bot can resume a conversation with the same variables. See Setting the session timeout for more information.
5. In the **Add language settings** section, choose a voice for your bot to interact with users. You can type a phrase in **Voice sample** and select **Play** to listen to the voice.
6. In the **Advanced settings** section, optionally add tags that help identify the bot. Tags can be used to control access and monitor resources. See Tagging resources for more information.
7. Choose **Next** to create the bot network and move to adding bots.

Step 2: Add bots

1. In the **Bots** section, select **+ Add bots**.
2. An **Add bots** modal will pop up. Choose a bot to add from the **Bot** dropdown menu and the alias of the bot that you want to use from the **Alias** dropdown menu.

   The alias must point to a numbered version of the bot and not to the draft version. You may add up to 5 bots. A bot may be added to up to 25 different networks.

3. Select **+ Add bot** to add more bots to your network. To remove a bot, select **Remove** next to the bot that you want to remove. When you are done adding bots, choose **Save** to close the modal.
4. Select **Save** to finish creating your network.

Manage your network of bots for your Lex V2 bots

After creating your network of bots, you will be taken to a page where you can manage and build your network. Or you can reach this page by selecting **Network of bots** in the side menu, and choosing the name of the network to manage.

1. To edit information for your network, select **Edit** above the **Details** section. To delete the network, select **Delete** above the **Details** section.
2. In the **Bots** section, you can add more bots by selecting **+ Add bots**. You can also add bots if you navigate to the **Bots** page in the side menu in the Amazon Lex V2 console. Toggle the radio button next to the bot you want to add, and select **Add to a network of bots** from the **Actions** dropdown menu.

   From the **Network of bots** dropdown menu in the modal that pops up, choose the network to which you want to add the bot. Then choose the alias of the bot that you want to use from the **Bot alias** dropdown menu. Select **Add** to add the bot to the network that you chose.
3. You can remove bots from your network by toggling the radio button next to a bot and choosing **Remove**.
4. When you are done configuring your network, select **Build** in the upper-right to build your network. It may take a few minutes to build. If the build is successful, a green success banner appears at the top of the page.
5. Once the network is built, you can select **Test** in the upper-right for a chat window to appear in the bottom right corner. You can use this chat window to converse with your network's bots and make sure the conversation flows and transitions are configured correctly.

> **Note**
>
> If you add, remove, or update bots within your network, you must rebuild the network.

Versions of your network of bots for Lex V2

You can create different versions of your network of bots. To manage versions, choose your network from the side menu in the Amazon Lex V2 console and select **Versions**.

1. Select **Create Version** to make a new version of your network of bots. You can add an optional description. Choose **Create** to create the version.
2. When you toggle the radio button next to a version of your network of bots, you can select **Associate alias with version** to point an alias to this version.
3. To manage a version of your network, select the name of the version in the **Versions** section. In the following page, you can edit details of the version and manage the bots within the version and its associated alias.

Aliases for your network of bots for Lex V2

You can use aliases to deploy your networks. To manage aliases, choose your network from the side menu in the Amazon Lex V2 console and select **Aliases**.

1. Select **Create alias** to make a new alias.
2. Give the alias a name and an optional **Description** in the **Alias details** section. You can choose a version to associate the alias with the **Associate with a version** section and add tags in the **Tags** section. Choose **Create** to create the alias.
3. To manage an alias for your network, select the name of the alias in the **Aliases** section. In the following page, you can edit details of the alias and manage its tags, channel integrations, and resource-based policy. You can also view the history of its association with versions of the network.

Channel integrations for your Lex V2 network of bots

To integrate your network of bots with a messaging platform, choose your network of bots from the side menu in the Amazon Lex V2 console. Then select **Channel integrations**.

1. Select **Add channel** to integrate your network with a new channel.
2. In the **Platform** section, choose the platform that you want to deploy your bot to in **Select platform**. An IAM role will be created. Choose a key from the dropdown menu under **KMS key** to protect your information.

3. In the **Integration configuration** channel, enter the **Name** and an optional **Description**. Choose an **Alias** from the dropdown menu.
4. Get your account SID and authentication token from the platform and fill out the **Account SID** and **Authentication token** fields. See Integrating your bots for more information.
5. Select **Create** to complete the channel integration.

> **Note**
>
> Network of bots is currently not available in Connect Customer voice or chat.

Deploying bots from Lex V2 for your production environment

After creating and testing your bot, it is ready for deployment to interact with your customers. In this section, learn to create *versions* of your bot when you have made an update. Use *aliases* to point to different versions of your bot when they are ready for deployment. Learn how to integrate your bots with messaging platforms, mobile applications, and websites.

Topics

Versioning and aliases with your Lex V2 bot

Amazon Lex V2 supports creating versions and aliases of bots and bot networks so that you can control the implementation that your client applications use. A version acts as a numbered snapshot of your work. You can point an alias to the version of your bot that you want to be available to your customers. In between creating versions, you can continue to update the `Draft` version of your bot without affecting the user experience.

Versions

Amazon Lex V2 supports creating versions of bots so that you can control the implementation that your client applications use. A *version* is a numbered snapshot of your work that you can create for use in different parts of your workflow, such as development, beta deployment, and production.

The Draft version of your Lex V2 bot

When you create an Amazon Lex V2 bot there is only one version, the `Draft` version.

`Draft` is the working copy of your bot. You can update only the `Draft` version and until you create your first version, `Draft` is the only version of the bot that you have.

The `Draft` version of your bot is associated with the `TestBotAlias`. The `TestBotAlias` should only be used for manual testing. Amazon Lex V2 limits the number of runtime requests that you can make to the `TestBotAlias` alias of the bot.

Creating a version for your Lex V2 bot

When you version an Amazon Lex V2 bot you create a numbered snapshot of the bot so that you can use the bot as it existed when the version was made. Once you've created a numeric version it will stay the same while you continue to work on the draft version of your application.

When you create a version, you can choose the locales to include in the version. You don't need to choose all of the locales in a bot. Also, when you create a version you can choose a locale from a previous version. For example, if you have three versions of a bot, you can choose one locale from the `Draft` version and one from version two when you create version four.

If you delete a locale from the `Draft` version, it is not deleted from a numbered version.

If a bot version is not used for six months, Amazon Lex V2 will mark the version inactive. When a version is inactive, you can't use runtime operations with the bot. To make the bot active, rebuild all the languages associated with the version.

Updating an Amazon Lex V2 bot

You can update only the `Draft` version of an Amazon Lex V2 bot. Versions can't be changed. You can create a new version any time after you update a resource in the console or with the CreateBotVersion operation.

Deleting an Amazon Lex V2 bot or version

Amazon Lex V2 supports deleting a bot or version using the console or one of the API operations:

- DeleteBot
- DeleteBotVersion

Aliases for your Lex V2 bot

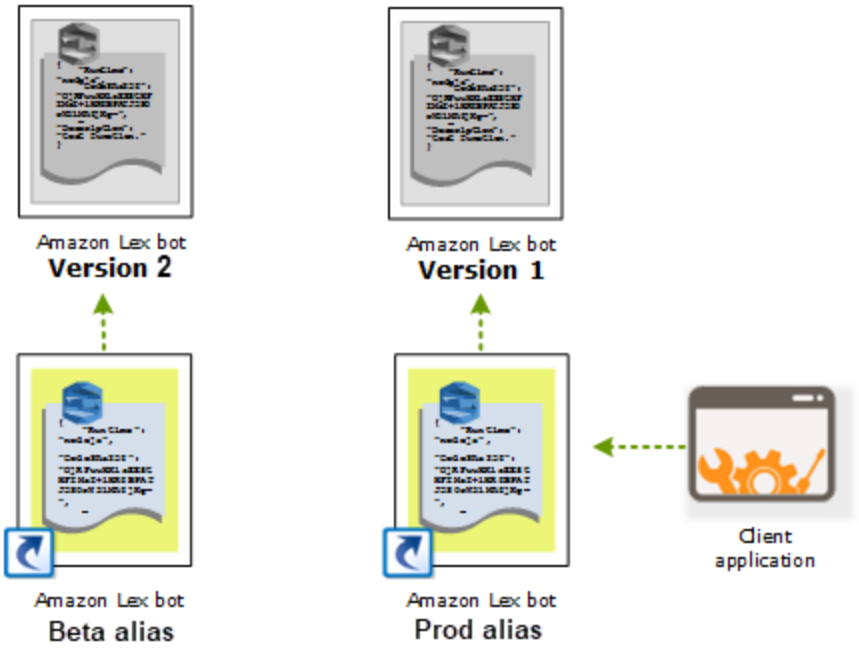
Amazon Lex V2 bots support aliases. An *alias* is a pointer to a specific version of a bot. With an alias, you can easily update the version that your client applications are using. For example, you can point an alias to version 1 of your bot. When you are ready to update the bot, you create version 2

and change the alias to point to the new version. Because your applications use the alias instead of a specific version, all of your clients get the new functionality without needing to be updated.

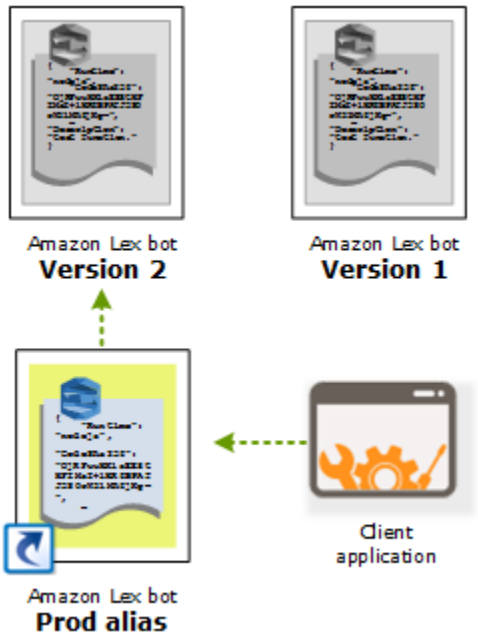
An alias is a pointer to a specific version of an Amazon Lex V2 bot. Use an alias to allow client applications to use a specific version of the bot without requiring the application to track which version that is.

When you create a bot, Amazon Lex V2 creates an alias called `TestBotAlias` that you can use for testing your bot. The `TestBotAlias` alias is always associated with the `Draft` version of your bot. You should only use the `TestBotAlias` alias for testing, Amazon Lex V2 limits the number of runtime requests that you can make to the alias.

The following example shows two versions of an Amazon Lex V2 bot, version 1 and version 2. Each of these bot versions has an associated alias, BETA and PROD, respectively. Client applications use the PROD alias to access the bot.



When you create a second version of the bot, you can update the alias to point to the new version of the bot using the console or the UpdateBotAlias operation. When you change the alias, all of your client applications use the new version. If there is a problem with the new version, you can roll back to the previous version by simply changing the alias to point to that version.



When you set up your client applications to call the Amazon Lex Runtime V2 APIs to let customers interact with your bot, you use the alias that points the version that you want your customers to use.

> **Note**
>
> Although you can test the `Draft` version of a bot in the console, we recommend that when you integrate a bot with your client application, you first create a version and create an alias that points to that version. Use the alias in your client application for the reasons explained in this section. When you update an alias, Amazon Lex V2 will use the current version for all in-progress sessions. New sessions use the new version.

Using a Java application to interact with an Amazon Lex V2 bot

The AWS SDK for Java 2.0 provides an interface that you can use from your Java applications to interact with your bots. Use the SDK for Java to build client applications for users.

The following application interacts with the OrderFlowers bot that you created in Exercise 1: Create a chatbot from a template. It uses the `LexRuntimeV2Client` from the SDK for Java SDK to call the RecognizeText operation to conduct a conversation with the bot.

The output from the conversation looks like this:

```
User : I would like to order flowers
Bot  : What type of flowers would you like to order?
User : 1 dozen roses
Bot  : What day do you want the dozen roses to be picked up?
User : Next Monday
Bot  : At what time do you want the dozen roses to be picked up?
User : 5 in the evening
Bot  : Okay, your dozen roses will be ready for pickup by 17:00 on 2021-01-04.  Does
 this sound okay?
User : Yes
Bot  : Thanks.
```

For the JSON structures that are sent between the client application and the Amazon Lex V2 bot, see Exercise 2: Review the conversation flow.

To run the application, you must provide the following information:

- botId – The identifier assigned to the bot when you created it. You can see the bot ID in the Amazon Lex V2 console on the bot **Settings** page.
- botAliasId – The identifier assigned to the bot alias when you created it. You can see the bot alias ID in the Amazon Lex V2 console on the **Aliases** page. If you can't see the alias ID in the list, choose the gear icon on the upper right and turn on **Alias ID**.
- localeId – The identifier of the locale that you used for your bot. For a list of locales, see Languages and locales supported by Amazon Lex V2.
- accessKey and secretKey – The authentication keys for your account. If you don't have a set of keys, create them using the AWS Identity and Access Management console.
- sessionId – An identifier for the session with the Amazon Lex V2 bot. In this case, the code uses a random UUID.
- region – If your bot is not in the US East (N. Virginia) Region, make sure that you change the Region.

The applications uses a function called `getRecognizeTextRequest` to create individual requests to the bot. The function builds a request with the required parameters to send to Amazon Lex V2.

```
package com.lex.recognizetext.sample;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;
```

```
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lexruntimev2.LexRuntimeV2Client;
import software.amazon.awssdk.services.lexruntimev2.model.RecognizeTextRequest;
import software.amazon.awssdk.services.lexruntimev2.model.RecognizeTextResponse;

import java.net.URISyntaxException;
import java.util.UUID;


/**
 * This is a sample application to interact with a bot using RecognizeText API.
 */
public class OrderFlowersSampleApplication {

    public static void main(String[] args) throws URISyntaxException,
 InterruptedException {
        String botId = "";
        String botAliasId = "";
        String localeId = "en_US";
        String accessKey = "";
        String secretKey = "";
        String sessionId = UUID.randomUUID().toString();
        Region region = Region.US_EAST_1; // pick an appropriate region

        AwsBasicCredentials awsCreds = AwsBasicCredentials.create(accessKey,
 secretKey);
        AwsCredentialsProvider awsCredentialsProvider =
 StaticCredentialsProvider.create(awsCreds);

        LexRuntimeV2Client lexV2Client = LexRuntimeV2Client
                .builder()
                .credentialsProvider(awsCredentialsProvider)
                .region(region)
                .build();

        // utterance 1
        String userInput = "I would like to order flowers";
        RecognizeTextRequest recognizeTextRequest = getRecognizeTextRequest(botId,
 botAliasId, localeId, sessionId, userInput);
        RecognizeTextResponse recognizeTextResponse =
 lexV2Client.recognizeText(recognizeTextRequest);
```

```
        System.out.println("User : " + userInput);
        recognizeTextResponse.messages().forEach(message -> {
            System.out.println("Bot : " + message.content());
        });

        // utterance 2
        userInput = "1 dozen roses";
        recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
        recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

        System.out.println("User : " + userInput);
        recognizeTextResponse.messages().forEach(message -> {
            System.out.println("Bot : " + message.content());
        });

        // utterance 3
        userInput = "next monday";
        recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
        recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

        System.out.println("User : " + userInput);
        recognizeTextResponse.messages().forEach(message -> {
            System.out.println("Bot : " + message.content());
        });

        // utterance 4
        userInput = "5 in evening";
        recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
        recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

        System.out.println("User : " + userInput);
        recognizeTextResponse.messages().forEach(message -> {
            System.out.println("Bot : " + message.content());
        });

        // utterance 5
        userInput = "Yes";
        recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
        recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);
```

```
        System.out.println("User : " + userInput);
        recognizeTextResponse.messages().forEach(message -> {
            System.out.println("Bot : " + message.content());
        });
    }

    private static RecognizeTextRequest getRecognizeTextRequest(String botId, String
 botAliasId, String localeId, String sessionId, String userInput) {
        RecognizeTextRequest recognizeTextRequest = RecognizeTextRequest.builder()
                .botAliasId(botAliasId)
                .botId(botId)
                .localeId(localeId)
                .sessionId(sessionId)
                .text(userInput)
                .build();
        return recognizeTextRequest;
    }
}
```

Use Global Resiliency to deploy bots to other Regions

Global Resiliency lets you replicate a bot in a secondary region. The secondary region can be made active with the automatic replication of the user's bot in both regions. You will have a backup region in case of a regional outage. Once Global Resiliency is active, new bots created are replicated in a second AWS region.

> **Note**
>
> This feature is available only for Amazon Connect and Amazon Lex V2 instances created in the US East (N. Virginia) and US West (Oregon) Regions, and the Europe West (London) and Europe Central (Frankfurt) Regions.
> To obtain access to this feature, contact your Amazon Connect Solutions Architect or Technical Account Manager.

After activating this feature, you can automate the replication of Amazon Lex V2 bots and their resources, versions, and aliases across a paired AWS region in near real-time. With this feature, you can monitor the version number of the original and replica bot to ensure that the

bot replica remains in sync with the original bot. When you enable replication, you can activate the pre-determined AWS region you want the bot to be replicated in (regions are based on pre-determined pairs). Any updates to the source bot in the source region is automatically updated to the replicated bot in the second region.

> **Note**
>
> When Global Resiliency is enabled for a bot, all existing aliases and their associated versions get replicated in the replica region. Versions which are not associated to an alias before enabling replication are replicated when they get associated to an alias. Every version and alias created after enabling replication, is automatically replicated. Users can use `ListBotVersionReplicas` and `ListBotAliasReplicas` to review the status of replication of each individual version and alias. Bot mutations are uni-directional from bot to its replica. Users cannot modify the replica bot because it is always kept in sync with the bot.

Additional information about using Global Resiliency:

- Global Resiliency currently only works with pre-determined pairs of regions.

us-east-1	us-west-2
eu-west-2	eu-central-1

- When Global Resiliency is enabled for a bot, all existing aliases and their associated versions get replicated in the replica region. Versions which are not associated to an alias before enabling replication, are replicated when they are associated to an alias. Every version and alias created after enabling replication, gets automatically replicated. Users can use `ListBotVersionReplicas` and `ListBotAliasReplicas` to know the status of replication for each individual version and alias. Bot mutations are uni-directional from bot to its replica. Users cannot modify the replica bot, because it is always kept in sync with the bot.
- Any Alias can be associated with any version. If the version is not replicated already, it will be replicated during the association with the Alias.

Limitations:

- Global Resiliency does not replicate bots created with slots that use LLM such as CFAQ and Utterance Generation.
- Global Resiliency does not replicate a Network of Bots, but any bot that is part of the Network of Bots can still be individually replicated.

Topics

Permissions to replicate bots and manage bot replicas in Lex V2

If an IAM role has the AmazonLexFullAccess policy attached, it can create and manage bot replicas.

If you prefer to create a role with minimal permissions for Global Resiliency, use the following policy, which contains the following statements.

- Permissions to access the Amazon Lex V2 service-linked role for bot replication.
- Permissions to allow Amazon Lex V2 to create a service-linked role for bot replication on your behalf.
- Permissions to call the bot replication APIs.

You can restrict permissions further by modifying them as follows.

- Replace * with specific bot or bot alias IDs to limit the permissions to specific bots or bot aliases.
- Use a subset of the `lex BotReplica` actions to restrict the role to specific actions.

For an example, see Allow users to create and view bot replicas, but not to delete them.

Deploying Global Resiliency with your Lex V2 bot

Global Resiliency lets you replicate a bot in a secondary region. The secondary region can be made active with the automatic replication of the user's bot in both regions. You will have a backup region in case of a regional outage. Once Global Resiliency is active, new bots created are replicated in a second AWS region.

Global Resiliency information panel displays details about your deployments

You can access the following information in the Global Resiliency panel:

- **Source details** – Information about your bot's source region, replica type, replication enabled date, and last created version. Use this information to track iterations of your bot.
- **Replication details** – After creating your bot replica, you can track the replicated region, replica type, last version synced date, and last replicated version. Use this information to track the sync of your bot replica.
- **Source region** – The region where Global Resiliency is enabled. You can make changes in the source region to replicate the bot in both regions.
- **Replica type** – Indicates if the bot is read only or able to read and write based on the region.
- **Replica region** – The secondary region that is used to replicate your source bot for Global Resiliency. Global Resiliency currently only works with IAD/PDX and LDN/FRA regional pairs.
- **Replication enabled date** – The date and time the bot replica was enabled.
- **Last created version** – The last bot version associated with the replica in the source region.

Enabling Global Resiliency for your Lex V2 bots

Before activating Global Resiliency in the Amazon Lex V2 console, you must ensure that the user that enables bot replication has permission to create Service Linked Roles (SLR). Global Resiliency will use these FAS credentials to create a SLR in the enabled account when CreateReplica is invoked. For more information on setting up the SLR for Global Resiliency in Amazon Lex V2, see AWS managed policy: AmazonLexFullAccess .

> **Note**
>
> This feature is available only for Amazon Connect and Amazon Lex V2 instances created in the US East (N. Virginia) and US West (Oregon) Regions, and the Europe West (London) and Europe Central (Frankfurt) Regions.
> To obtain access to this feature, contact your Amazon Connect Solutions Architect or Technical Account Manager.

Activate Global Resiliency and set up bot replication for a second region:

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot you want to replicate from the Bots navigation on the left side navigation panel.
3. Choose Deployment > Global Resiliency.
4. Select the **Create replica** button on the upper right corner of the window to create a draft version of your bot.

> **Note**
>
> Check to make certain you do not have any bots in the secondary region with the same name as the bot you want to replicate. (Your bot must be uniquely named).

5. Go to **Global Resiliency**, Click **Create Replica** - This action creates a draft version of your bot. (you do not need to go back to the Global Resiliency tab except to review status or see details of future builds).

> **Note**
>
> You can also create an Alias bot for replication in Global Resiliency by going to Alias and selecting **Create New Alias** for Global Resiliency enabled bot. All aliases will be replicated, even if created before replication was enabled.

6. Go to **Alias** - **Create New Alias** for the Global Resiliency enabled bot. All aliases will be replicated, even if they were created before replication was enabled.
7. Go to **Version** - **Create New Version** for the Global Resiliency enabled bot. Any version associated with an alias will be replicated, even if it was created before replication was enabled.

> **Note**
>
> Customers still have full control of managing their resource based policies and tags for replicated bots. Lambda functions and CloudWatch Logs Groups will need to be deployed in both regions with the same identifiers. Users will not have to associate the lambda function again in the replica region.

Disabling Global Resiliency

You can disable Global Resiliency at any time by selecting the **Disable Global Resiliency** button. This action stops your source bot and any aliases and versions associated with it from being replicated in other regions.

Using APIs with Global Resiliency for your Lex V2 bots

You can make API calls in Global Resiliency using the following APIs. Additional information about Global Resiliency APIs and Amazon Lex V2 can be found in the Amazon Lex V2 API Guide.

- **CreateBotReplica**

  Enable Global Resiliency and creates a replicated bot. Requires `replicaRegion`.

  For more information, see CreateBotReplica in the Amazon Lex V2 API Guide.
- **DeleteBotReplica**

  Disable Global Resiliency and delete the replicated bot. Requires `replicaRegion` and `botId`.

  For more information, see DeleteBotReplica in the Amazon Lex V2 API Guide.
- **ListBotReplicas**

  List the replicated bots in the secondary zone. Requires `botId`.

  For more information, see ListBotReplicas in the Amazon Lex V2 API Guide.
- **DescribeBotReplica**

  Summary of information for the replicated bot. Requires `replicaRegion` and `botId`.

  For more information, see DescribeBotReplica in the Amazon Lex V2 API Guide.

Integrating an Amazon Lex V2 bot with a messaging platform

This section explains how to integrate Amazon Lex V2 bots with the Facebook, Slack, and Twilio messaging platforms. If you don't already have an Amazon Lex V2 bot, create one. In this topic, we assume that you are using the bot that you created in Exercise 1: Create a chatbot from a template. However, you can use any bot.

> **Note**
>
> When storing your Facebook, Slack, or Twilio configurations, Amazon Lex V2 uses an AWS KMS key to encrypt information. The first time that you create a channel to one of these messaging platforms, Amazon Lex V2 creates a default customer managed key (`aws/lex`) in your AWS account or you can select your own customer managed key. Amazon Lex V2 supports only symmetric keys. For more information, see the AWS Key Management Service Developer Guide.

When a messaging platform sends a request to Amazon Lex V2 it includes platform-specific information as a request attribute to you Lambda function. Use this attribute to customize the way that your bot behaves. For more information, see Setting request attributes for your Lex V2 bot.

Common request attributes for messaging platforms

Attribute	Description
x-amz-lex:channels:platform	One of the following values: • `Facebook` • `Slack` • `Twilio`

Integrating an Amazon Lex V2 bot with Facebook Messenger

You can host your Amazon Lex V2 bot in Facebook Messenger. When you do, Facebook users can interact with your bot to fulfill intents.

Before you start, you need to sign up for a Facebook developer account at https://developers.facebook.com.

You need to perform the following steps:

Topics

- Step 4: Test the integration with Facebook Messenger

Step 1: Create a Facebook application

On the Facebook developer portal, create a Facebook application and a Facebook page.

To create a Facebook application

1. Open https://developers.facebook.com/apps
2. Choose **Create App**.
3. In the **Create an App** page, choose **Business**, then choose **Next**.
4. For the **Add on app name**, **App contact email**, and **Business Account** fields, make the appropriate choices for your app. Choose **Create App** to continue.
5. From **Add Products to Your App**, choose **Set Up** from the **Messenger** tile.
6. In the **Access Tokens** section, choose **Add or Remove pages**.
7. Choose a page to use with your app, then choose **Next**.
8. For **What is app allowed to do**, leave the defaults then choose **Done**.
9. On the confirmation page, choose **OK**.
10. In the **Access Tokens** section, choose **Generate Token**, then copy the token. You enter this token in the Amazon Lex V2 console.
11. From the left menu, choose **Settings** and then choose **Basic**.
12. For **App Secret**, choose **Show** and then copy the secret. You enter this token in the Amazon Lex V2 console.

Next step

Step 2: Integrate Facebook Messenger with the Amazon Lex V2 bot

Step 2: Integrate Facebook Messenger with the Amazon Lex V2 bot

In this step you link your Amazon Lex V2 bot with Facebook.

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of bots, choose the Amazon Lex V2 bot that you created.

3. In the left menu, choose **Channel integrations** and then choose **Add channel**.
4. In **Create channel**, do the following:

   a. For **Platform**, choose **Facebook**.

   b. For **Identity policies**, choose the AWS KMS key to protect channel information. The default key is provided by Amazon Lex V2.

   c. For **Integration configuration**, give the channel a name and an optional description. Choose the alias that points to the version of the bot to use, and choose the language that the channel supports.

   d. For **Additional configuration**, enter the following:

      - **Alias** – A string that identifies the app that is calling Amazon Lex V2. You can use any string. Record this string, you enter it in the Facebook developer console.
      - **Page access token** – The page access token that you copied from the Facebook developer console.
      - **App secret key** – The secret key that you copied from the Facebook developer console.

   e. Choose **Create**

   f. Amazon Lex V2 shows the list of channels for your bot. From the list, choose the channel that you just created.

   g. From **Callback URL**, record the callback URL. You enter this URL in the Facebook developer console.

Next step

Step 3: Complete Facebook integration with your Lex V2 bot

Step 3: Complete Facebook integration with your Lex V2 bot

In this step, use the Facebook developer console to complete integration with Amazon Lex V2.

To complete Facebook Messenger integration

1. Open https://developers.facebook.com/apps
2. From the list of apps, choose the app that you are integrating with Facebook Messenger.
3. In the left menu, choose **Messenger**, then choose **Settings**.
4. In the **Webhooks** section:

a. Choose **Add Callback URL**.

b. In **Edit Callback URL**, enter the following:

- **Callback URL** – Enter the callback URL that you recorded from the Amazon Lex V2 console.
- **Verify Token** – Enter the alias that you entered in the Amazon Lex V2 console.

c. Choose **Verify and Save**.

d. Choose **Add subscriptions** under **Webhooks** next to your page.

e. In the window that pops up, choose `messages` and then click **Save**.

Next step

Step 4: Test the integration with Facebook Messenger

Step 4: Test the integration with Facebook Messenger

You can now start a conversation from Facebook Messenger with your Amazon Lex V2 bot.

To test the integration between Facebook Messenger and an Amazon Lex V2 bot

1. Open the Facebook page that you associated with your bot in step 1.
2. In the Messenger window, use the test utterances provided in Exercise 1: Create a chatbot from a template.

Integrating an Amazon Lex V2 bot with Slack

This topic provides instructions for integrating an Amazon Lex V2 bot with the Slack messaging application. You perform the following steps:

Topics

Step 1: Sign up for Slack and create a Slack team

Sign up for a Slack account and create a Slack team. For instructions, see Using Slack. In the next section you create a Slack application, which any Slack team can install.

Next step

Step 2: Create a Slack application

Step 2: Create a Slack application

In this section, you do the following:

1. Create a Slack application in the Slack API Console.
2. Configure the application to add interactive messaging to your bot.

At the end of this section, you get application credentials (Client ID, Client Secret, and Verification Token). In the next step, you use this information to integrate the bot in the Amazon Lex V2 console.

To create a Slack application

1. Sign in to the Slack API Console at https://api.slack.com .
2. Create an application.

   After you have successfully created the application, Slack displays the **Basic Information** page for the application.
3. Configure the application features as follows:

   - In the left menu, choose **Interactivity & Shortcuts**.
     - Choose the toggle to turn interactive components on.
     - In the **Request URL** box, specify any valid URL. For example, you can use **`https://slack.com`**.

> **Note**
>
> For now, enter any valid URL to get the verification token that you need in the next step. You will update this URL after you add the bot channel association in the Amazon Lex console.

- Choose **Save Changes**.

4. In the left menu, in **Settings**, choose **Basic Information**. Record the following application credentials:

   - Client ID
   - Client Secret
   - Verification Token

Next step

Step 3: Integrate the Slack application with the Amazon Lex V2 bot

Step 3: Integrate the Slack application with the Amazon Lex V2 bot

In this section, integrate the Slack application you created with the Amazon Lex V2 bot you created by using Channel integrations.

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of bots, choose the Amazon Lex V2 bot that you created.
3. In the left menu, choose **Channel integrations** and then choose **Add channel**.
4. In **Create channel**, do the following:

   a. For **Platform**, choose **Slack**.

   b. For **Identity policies**, choose the AWS KMS key to protect channel information. The default key is provided by Amazon Lex V2.

   c. For **Integration configuration**, give the channel a name and an optional description. Choose the alias that points to the version of the bot to use, and choose the language that the channel supports.

> **Note**
>
> If your bot is available in multiple languages, you must create a different channel and a different application for each language.

d. For **Additional configuration**, enter the following:

- **Client ID** – enter the client ID from Slack.
- **Client secret** – enter the client secret from Slack.
- **Verification token** – enter the verification token from Slack.
- **Success page URL** – The URL of the page that Slack should open when the user is authenticated. Typically you leave this blank.

5. Choose **Create** to create the channel.
6. Amazon Lex V2 shows the list of channels for your bot. From the list, choose the channel that you just created.
7. From **Callback URL**, record the endpoint and the OAuth endpoint.

Next step

Step 4: Complete Slack integration with your Lex V2 bot

Step 4: Complete Slack integration with your Lex V2 bot

In this section, use the Slack API console to complete integration with the Slack application.

1. Sign in to the Slack API console at https://api.slack.com. Choose the app that you created in Step 2: Create a Slack application.
2. Update the **OAuth & Permissions** feature as follows:

   a. In the left menu, choose **OAuth & Permissions**.

   b. In the **Redirect URLs** section, add the OAuth endpoint that Amazon Lex provided in the preceding step. Choose **Add**, and then choose **Save URLs**.

   c. In the **Bot Token Scopes** section, add two permissions with the **Add an OAuth Scope** button. Filter the list with the following text:

   - `chat:write`

- **`team:read`**

3. Update the **Interactivity & Shortcuts** feature by updating the **Request URL** value to the endpoint that Amazon Lex provided in the preceding step. Enter the endpoint that you saved in step 3, and then choose **Save Changes**.
4. Subscribe to the **Event Subscriptions** feature as follows:
   - Enable events by choosing the **On** option.
   - Set the **Request URL** value to the endpoint that Amazon Lex provided in the preceding step.
   - In the **Subscribe to Bot Events** section, select **Add Bot User Event** and add the **`message.im`** bot event to enable direct messaging between the end user and the Slack bot.
   - Save the changes.
5. Enable sending messages from the messages tab as follows:
   - From the left menu, choose **App Home**.
   - In the **Show Tabs** section, choose **Allow users to send Slash commands and messages from the messages tab.**
6. Choose **Manage Distribution** under **Settings**. Choose **Add to Slack** to install the application. If you are authenticated to multiple workspaces, first choose the correct workspace in the upper right-hand corner from the drop-down list. Next, select **Allow** to authorize the bot to respond to messages.

> **Note**
>
> If you make any changes to your Slack application settings later, you must redo this substep.

Next step

Step 5: Test the integration between your Lex V2 bot and Slack

Step 5: Test the integration between your Lex V2 bot and Slack

Now use a browser window to test the integration of Slack with your Amazon Lex V2 bot.

To test your Slack application

1. Launch Slack. From the left menu, in the **Direct Messages** section, choose your bot. If you don't see your bot, choose the plus icon (+) next to **Direct Messages** to search for it.
2. Engage in a chat with your Slack application. Your bot responds to messages.

   If you created the bot using Exercise 1: Create a chatbot from a template, you can use the example conversations from that exercise.

Integrating an Amazon Lex V2 bot with Twilio SMS

This topic provides instructions for integrating an Amazon Lex V2 bot with the Twilio simple message service (SMS). You perform the following steps:

Topics

Step 1: Create a Twilio SMS account

Sign up for a Twilio account and record the following account information:

- **ACCOUNT SID**
- **AUTH TOKEN**

For sign-up instructions, see https://www.twilio.com/console .

Next step

Step 2: Integrate the Twilio message service endpoint with the Amazon Lex V2 bot

Step 2: Integrate the Twilio message service endpoint with the Amazon Lex V2 bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of bots, choose the Amazon Lex V2 bot that you created.
3. In the left menu, choose **Channel integrations** and then choose **Add channel**.
4. In **Create channel**, do the following:
    a. For **Platform**, choose **Twilio**.
    b. For **Identity policies**, choose the AWS KMS key to protect channel information. The default key is provided by Amazon Lex V2.
    c. For **Integration configuration**, give the channel a name and an optional description. Choose the alias that points to the version of the bot to use, and choose the language that the channel supports.
    d. For **Additional configuration**, enter the account SID and authentication token from the Twilio dashboard.
5. Choose **Create**.
6. From the list of channels, choose the channel that you just created.
7. Copy the **Callback URL**.

Next step

Step 3: Complete Twilio integration between your Lex V2 bot and Twilio

Step 3: Complete Twilio integration between your Lex V2 bot and Twilio

Use the Twilio console to complete the integration of your Amazon Lex V2 bot with Twilio SMS.

1. Open the Twilio console at https://www.twilio.com/console .
2. From the left menu, choose **All Products & Services**, then choose **Phone Number**.
3. If you have a phone number, choose it. If you don't have a phone number, choose **Buy a Number** to get one.
4. In the **Messaging** section, in **A MESSAGE COMES IN**, enter the callback URL from the Amazon Lex V2 console.

5. Choose **Save**.

Next step

Step 4: Test the integration between your Lex V2 bot and Twilio

Step 4: Test the integration between your Lex V2 bot and Twilio

Use your mobile phone to test the integration between Twilio SMS and your bot. Using your mobile phone, send messages to the Twilio number.

If you created the bot using Exercise 1: Create a chatbot from a template, you can use the example conversations from that exercise.

Integrating an Amazon Lex V2 bot with a contact center

You can integrate Amazon Lex V2 bots with your contact centers to enable self-service use-cases using the Amazon Lex V2 streaming API. Use these bots as interactive voice response (IVR) agents on telephony or as a text-based chatbot integrated into your contact center. For more information about the streaming APIs, see Streaming conversations to an Amazon Lex V2 bot.

With streaming APIs, you can enable the following features:

- **Interruptions** ("barge-in") – Callers can interrupt the bot and answer a question before the prompt is complete. For more information, see Enabling your Amazon Lex V2 bot to be interrupted by the user.
- **Wait and continue** – Callers can instruct the bot to wait if they need time for retrieving additional information during a call, such as a credit card number or a booking ID. For more information, see Enabling the Amazon Lex V2 bot to wait for the user to provide more information during a pause.
- **DTMF support** – Callers can provide information via speech or DTMF interchangeably.
- **SSML support** – You can configure Amazon Lex V2 bot prompts using SSML tags for greater control over speech generation from text. For more information, see Generating speech from SSML documents in the *Amazon Polly developer guide*.
- **Configurable timeouts** – You can configure how long to wait for customers to finish speaking before Amazon Lex V2 collects their speech input, such as answering a yes or no question, or

providing a date or credit card number. For more information, see Configuring timeouts for capturing user input with a Lex V2 bot.

- **Fulfillment progress updates** – You can configure the bot to respond with multiple messages based on the fulfillment status during the business logic execution for intent fulfillment. You can set the bot to respond with messages when the fulfillment begins and completes, and provides periodic updates for long running Lambda functions. For more information, see Configuring fulfillment progress updates for your Lex V2 bot.

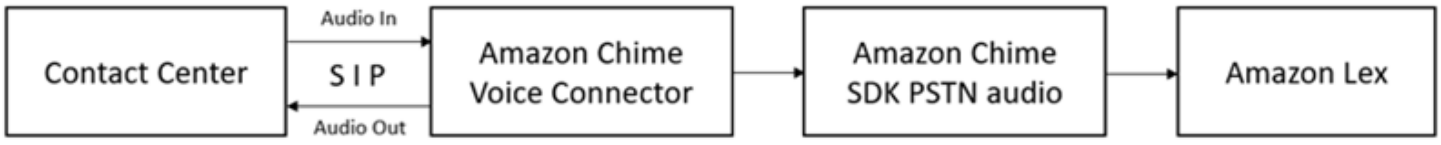
Amazon Chime SDK

Use the Amazon Chime SDK to add real-time audio, video, screen sharing, and messaging capabilities to your web or mobile applications. The Amazon Chime SDK provides public switched telephone network (PSTN) audio service so that you can build custom telephony applications with an AWS Lambda function.

Amazon Chime PSTN audio is integrated with Amazon Lex V2. You can use this integration to access Amazon Lex V2 bots as interactive voice response (IVR) systems in contact centers for audio interactions. Use this to integrate Amazon Lex V2 using PSTN audio services in the following scenarios.

Contact center integrations—You can use the Amazon Chime Voice Connector and Amazon Chime SDK PSTN audio service to access Amazon Lex V2 bots. Use them in any contact center application that uses the session initiation protocol (SIP) for voice communications. This integration adds natural language voice conversation experiences to your existing on-premises or cloud-based contact center with SIP support. For a list of supported contact center platforms, see Amazon Chime voice connector resources.

The following diagram shows the integration between a contact center using SIP and Amazon Lex V2.



Direct telephony support—You can build customized IVR solutions to directly access Amazon Lex V2 bots using a phone number provisioned in the Amazon Chime SDK.

For more information, see the following topics in the *Amazon Chime SDK guide*.

- SIP integration using an Amazon Chime voice connector
- Using the Amazon Chime SDK PSTN audio service
- Integrating Amazon Chime PSTN audio with Amazon Lex V2

When the Amazon Chime SDK sends a request to Amazon Lex V2, it includes platform-specific information to your Lambda function and conversation logs. Use this information to determine the contact center application that is sending traffic to your bot.

Common request attribute	Value
x-amz-lex:channels:platform	`Amazon Chime SDK PSTN Audio`

Connect Customer

Connect Customer is an omnichannel cloud contact center. You can set up a contact center in a few steps, add agents located anywhere, and start engaging with your customers. For more information, see Get started with Connect Customer in the *Connect Customer administrator guide*.

You can create personalized experiences for your customers using omnichannel communications. For example, you can offer chat and voice contact based on customer preference and estimated wait times. Meanwhile agents can handle all customers from just one interface. For example, they can chat with customers, and create or respond to tasks as they are routed to them.

You can use Connect Customer for audio interactions with your customers, or Connect Customer Chat for text-only interactions.

For more information, see the following topics in the *Connect Customer administrator guide*.

- What is Connect Customer
- Add an Amazon Lex V2 bot
- Connect Customer get customer input contact block

When a contact center sends a request to Amazon Lex V2, it includes platform-specific information as a request attribute to your Lambda function and conversation logs. Use this information to determine which contact center application is sending traffic to your bot.

Common request attributes for Amazon Connect

Attribute	Value
x-amz-lex:channels:platform	One of the following values: • `Connect` • `Connect Chat`

Genesys Cloud

Genesys Cloud is a suite of cloud services for enterprise communication, collaboration, and contact center management. Genesys Cloud is built on top of AWS and uses a distributed cloud environment that provides secure access to organizations around the work.

For more information, see the following pages on the Genesys Cloud website.

- About Genesys Cloud contact center
- About the Amazon Lex V2 integration

When a contact center sends a request to Amazon Lex V2 it includes platform-specific information as a request attribute to your Lambda function and conversation logs. Use this information to determine which contact center application is sending traffic to your bot.

Common request attributes for Genesys Cloud

Attribute	Value
x-amz-lex:channels:platform	• `Genesys Cloud`

Learn more

- Power your contact center with Amazon Lex and Genesys Cloud

Understanding Amazon Lex V2 bot conversations

After you build a bot, you integrate your client application with the Amazon Lex V2 runtime operations to hold conversations with your bot. When a user starts a conversation with your bot, Amazon Lex V2 creates a *session*. A session encapsulates the information exchanged between your application and the bot. For more information, see Understanding Amazon Lex V2 bot sessions.

A typical conversation involves a back and forth flow between the user and a bot. For example:

```
User : I'd like to make an appointment
Bot : What type of appointment would you like to schedule?
User : dental
Bot : When should I schedule your dental appointment?
User : Tomorrow
Bot : At what time do you want to schedule the dental appointment on 2021-01-01?
User : 9 am
Bot : 09:00 is available, should I go ahead and book your appointment?
User : Yes
Bot : Thank you. Your appointment has been set successfully.
```

Use the RecognizeText or RecognizeUtterance API operations to manage the conversations yourself. Use the StartConversation API operation to let Amazon Lex V2 manage the conversation for you.

To manage the conversation, you must send user utterances to the bot until the conversation reaches a logical end. The current conversation is captured in session state. The session state is updated after each user utterance. The session state contains the current state of the conversation and is returned by the bot in a response to each user utterance.

A conversation can be in any of the following states:

- **ElicitIntent** – Indicates that the bot has not yet determined the user's intent.
- **ElicitSlot** – Indicates that the bot has detected the user's intent and is gathering the required information to fulfill the intent.
- **ConfirmIntent** – Indicates that the bot is waiting for the user to confirm that the information collected is correct.
- **Closed** – Indicates that the user's intent is complete and that the conversation with the bot reached a logical end.

A user can specify a new intent after the first intent is completed. For more information, see Conversation context with your Lex V2 bots.

An intent can have the one of the following states:

- **InProgress** – Indicates that the bot is gathering information necessary to complete the intent. This is in conjunction with the `ElicitSlot` conversation state.
- **Waiting** – Indicates that the user requested the bot to wait when the bot asked for information for a specific slot.
- **Fulfilled** – Indicates that the business logic in a Lambda function associated with the intent ran successfully.
- **ReadyForFulfillment** – Indicates that the bot gathered all of the information required to fulfill the intent and that the client application can run fulfillment business logic.
- **Failed** – Indicates that an intent has failed.

See the following topics to learn how to use Amazon Lex V2 APIs to manage conversation context and sessions between your bot and users.

Topics

- Conversation context with your Lex V2 bots
- Understanding Amazon Lex V2 bot sessions

Conversation context with your Lex V2 bots

Conversation context is information that the user, your client application, or a Lambda function provides to a Amazon Lex V2 bot to fulfill an intent. Conversation context includes slot data that the user provides, request attributes set by the client application, and session attributes that the client application and Lambda functions create.

Topics

- Setting intent context for your Lex V2 bot
- Using default slot values in intents for your Lex V2 bot
- Setting session attributes for your Lex V2 bot
- Setting request attributes for your Lex V2 bot
- Setting the session timeout

- Sharing information between intents with your Lex V2 bot
- Setting complex attributes in your Lex V2 bot

Setting intent context for your Lex V2 bot

You can have Amazon Lex V2 trigger intents based on *context*. A *context* is a state variable that can be associated with an intent when you define a bot. You configure the contexts for an intent when you create the intent using the console or using the CreateIntent operation. You can only use context in the English (US) (en-US) locale.

There are two types of relationships for contexts, output contexts and input contexts. An *output context* becomes active when an associated intent is fulfilled. An output context is returned to your application in the response from the RecognizeText or RecognizeUtterance operation, and it is set for the current session. After a context is activated, it stays active for the number of turns or time limit configured when the context was defined.

An *input context* specifies conditions under which an intent can be recognized. An intent can only be recognized during a conversation when all of its input contexts are active. An intent with no input contexts is always eligible for recognition.

Amazon Lex V2 automatically manages the lifecycle of contexts that are activated by fulfilling intents with output contexts. You can also set active contexts in a call to the `RecognizeText` or `RecognizeUtterance` operation.

You can also set the context of a conversation using the Lambda function for the intent. The output context from Amazon Lex V2 is sent to the Lambda function input event. The Lambda function can send contexts in its response. For more information, see Integrating an AWS Lambda function into your Amazon Lex V2 bot.

For example, suppose you have an intent to book a rental car that is configured to return an output context called "book_car_fulfilled". When the intent is fulfilled, Amazon Lex V2 sets the output context variable "book_car_fulfilled". Since "book_car_fulfilled" is an active context, an intent with the "book_car_fulfilled" context set as an input context is now considered for recognition, as long as a user utterance is recognized as an attempt to elicit that intent. You can use this for intents that only make sense after booking a car, such as emailing a receipt or modifying a reservation.

Output context of intents for your Lex V2 bot

Amazon Lex V2 makes an intent's output contexts active when the intent is fulfilled. You can use the output context to control the intents eligible to follow up the current intent.

Each context has a list of parameters that are maintained in the session. The parameters are the slot values for the fulfilled intent. You can use these parameters to pre-populate slot values for other intents. For more information,see Using default slot values in intents for your Lex V2 bot.

You configure the output context when you create an intent with the console or with the CreateIntent operation. You can configure an intent with more than one output context. When the intent is fulfilled, all of the output contexts are activated and returned in the RecognizeText or RecognizeUtterance response.

When you define an output context you also define its *time to live*, the length of time or number of turns that the context is included in responses from Amazon Lex V2. A *turn* is one request from your application to Amazon Lex V2. Once the number of turns or the time has expired, the context is no longer active.

Your application can use the output context as needed. For example, your application can use the output context to:

- Change the behavior of the application based on the context. For example, a travel application could have a different action for the context "book_car_fulfilled" than "rental_hotel_fulfilled."
- Return the output context to Amazon Lex V2 as the input context for the next utterance. If Amazon Lex V2 recognizes the utterance as an attempt to elicit an intent, it uses the context to limit the intents that can be returned to ones with the specified context.

Input context of intents for your Lex V2 bot

You set an input context to limit the points in the conversation where the intent is recognized. Intents without an input context are always eligible to be recognized.

You set the input contexts that an intent responds to using the console or the `CreateIntent` operation. An intent can have more than one input context.

For an intent with more than one input context, all contexts must be active to trigger the intent. You can set an input context when you call the RecognizeText, RecognizeUtterance, or PutSession operation.

You can configure the slots in an intent to take default values from the current active context. Default values are used when Amazon Lex V2 recognizes a new intent but doesn't receive a slot value. You specify the context name and slot name in the form `#context-name.parameter-name` when you define the slot. For more information, see Using default slot values in intents for your Lex V2 bot.

Using default slot values in intents for your Lex V2 bot

When you use a default value, you specify a source for a slot value to be filled for new intents when no slot is provided by the user's input. This source can be previous dialog, request or session attributes, or a fixed value that you set at build-time.

You can use the following as the source for your default values.

- Previous dialog (contexts) – #context-name.parameter-name
- Session attributes – [attribute-name]
- Request attributes – <attribute-name>
- Fixed value – Any value that doesn't match the previous

When you use the CreateIntent operation to add slots to an intent, you can add a list of default values. Default values are used in the order that they are listed. For example, suppose you have an intent with a slot with the following definition:

```
"slots": [
    {
        "botId": "string",
        "defaultValueSpec": {
            "defaultValueList": [
                {
                    "defaultValue": "#book-car-fulfilled.startDate"
                },
                {
                    "defaultValue": "[reservationStartDate]"
                }
            ]
        },
        Other slot configuration settings
    }
]
```

When the intent is recognized, the slot named "reservation-start-date" has its value set to one of the following.

1. If the "book-car-fulfilled" context is active, the value of the "startDate" parameter is used as the default value.
2. If the "book-car-fulfilled" context is not active, or if the "startDate" parameter is not set, the value of the "reservationStartDate" session attribute is used as the default value.
3. If neither of the first two default values are used, then the slot doesn't have a default value and Amazon Lex V2 will elicit a value as usual.

If a default value is used for the slot, the slot is not elicited even if it is required.

Setting session attributes for your Lex V2 bot

Session attributes contain application-specific information that is passed between a bot and a client application during a session. Amazon Lex V2 passes session attributes to all Lambda functions configured for a bot. If a Lambda function adds or updates session attributes, Amazon Lex V2 passes the new information back to the client application.

Use session attributes in your Lambda functions to initialize a bot and to customize prompts and response cards. For example:

- Initialization — In a pizza ordering bot, the client application passes the user's location as a session attribute in the first call to the RecognizeText or RecognizeUtterance operation. For example, `"Location": "111 Maple Street"`. The Lambda function uses this information to find the closest pizzeria to place the order.
- Personalize prompts — Configure prompts and response cards to refer to session attributes. For example, "Hey [FirstName], what toppings would you like?" If you pass the user's first name as a session attribute (`{"FirstName": "Vivian"}`), Amazon Lex substitutes the name for the placeholder. It then sends a personalized prompt to the user, "Hey Vivian, which toppings would you like?"

Session attributes persist for the duration of the session. Amazon Lex V2 stores them in an encrypted data store until the session ends. The client can create session attributes in a request by calling either the RecognizeText or RecognizeUtterance operation with the `sessionAttributes` field set to a value. A Lambda function can create a session attribute in a response. After the client

or a Lambda function creates a session attribute, the stored attribute value is used any time that the client application doesn't include `sessionAttribute` field in a request to Amazon Lex V2.

For example, suppose you have two session attributes, `{"x": "1", "y": "2"}`. If the client calls the `RecognizeText` or `RecognizeUtterance` operation without specifying the `sessionAttributes` field, Amazon Lex V2 calls the Lambda function with the stored session attributes (`{"x": 1, "y": 2}`). If the Lambda function doesn't return session attributes, Amazon Lex V2 returns the stored session attributes to the client application.

If either the client application or a Lambda function passes session attributes, Amazon Lex V2 updates the stored session attributes. Passing an existing value, such as `{"x": 2}`, updates the stored value. If you pass a new set of session attributes, such as `{"z": 3}`, the existing values are removed and only the new value is kept. When an empty map, `{}`, is passed, stored values are erased.

To send session attributes to Amazon Lex V2, you create a string-to-string map of the attributes. The following shows how to map session attributes:

```
{
   "attributeName": "attributeValue",
   "attributeName": "attributeValue"
}
```

For the `RecognizeText` operation, you insert the map into the body of the request using the `sessionAttributes` field of the `sessionState` structure, as follows:

```
"sessionState": {
    "sessionAttributes": {
        "attributeName": "attributeValue",
        "attributeName": "attributeValue"
    }
}
```

For the `RecognizeUtterance` operation, you base64 encode the map, and then send it as part of the `x-amz-lex-session-state` header.

If you are sending binary or structured data in a session attribute, you must first transform the data to a simple string. For more information, see Setting complex attributes in your Lex V2 bot.

Predefined session attributes

Amazon Lex provides predefined session attributes. These attributes manage how Amazon Lex processes information sent to your bot. The attributes persist for the entire session. All predefined attributes are in the `x-amz-lex:` namespace.

Increasing code hook Lambda timeout

To increase the code hook Lambda timeout value in your bot, which is 30 seconds by default, use the `x-amz-lex:codehook-timeout-ms` session attribute. Set the session attribute value in milliseconds. The maximum timeout is 120 seconds.

For example, if a user sets "x-amz-lex:codehook-timeout-ms": "90000", the code hook Lambda timeout will be 90 seconds.

Setting request attributes for your Lex V2 bot

Request attributes contain request-specific information and apply only to the current request. A client application sends this information to Amazon Lex V2. Use request attributes to pass information that doesn't need to persist for the entire session. You can create your own request attributes or you can use predefined attributes. To send request attributes, use the `x-amz-lex-request-attributes` header in a RecognizeUtterance or the `requestAttributes` field in a RecognizeText request. Because request attributes don't persist across requests like session attributes do, they are not returned in `RecognizeUtterance` or `RecognizeText` responses.

> **Note**
>
> To send information that persists across requests, use session attributes.

Setting user-defined request attributes for each Lex V2 bot request

A *user-defined request attribute* is data that you send to your bot in each request. You send the information in the `amz-lex-request-attributes` header of a `RecognizeUtterance` request or in the `requestAttributes` field of a `RecognizeText` request.

To send request attributes to Amazon Lex V2, you create a string-to-string map of the attributes. The following shows how to map request attributes:

```
{
```

```
   "attributeName": "attributeValue",
   "attributeName": "attributeValue"
}
```

For the `PostText` operation, you insert the map into the body of the request using the `requestAttributes` field, as follows:

```
"requestAttributes": {
   "attributeName": "attributeValue",
   "attributeName": "attributeValue"
}
```

For the `PostContent` operation, you base64 encode the map, and then send it as the `x-amz-lex-request-attributes` header.

If you are sending binary or structured data in a request attribute, you must first transform the data to a simple string. For more information, see Setting complex attributes in your Lex V2 bot.

Amazon Lex V2 provides predefined request attributes for managing the way that it processes information sent to your bot. The attributes do not persist for the entire session, so you must send the predefined attributes in each request. All predefined attributes are in the `x-amz-lex:` namespace.

In addition to the following predefined attributes, Amazon Lex provides predefined attributes for messaging platforms. For a list of those attributes, see Deploying an Amazon Lex Bot on a Messaging Platform.

Setting the Response Type

If you have two client applications that have different capabilities, you may need to limit the format of messages in a response. For example, you might want to restrict messages sent to a Web client to plain text, but enable a mobile client to use both plain text and Speech Synthesis Markup Language (SSML). To set the format of messages returned by the `PostContent` and `PostText` operations, use the x-amz-lex:accept-content-types" request attribute.

You can set the attribute to any combination of the following message types:

- `PlainText` — The message contains plain UTF-8 text.
- `SSML` — The message contains text formatted for voice output.

- `CustomPayload` — The message contains a custom format that you have created for your client. You can define the payload to meet the needs of your application.

Amazon Lex V2 returns only messages with the specified type in the Message field of the response. You can set more than one value by separating values with a comma. If you are using message groups, every message group must contain at least one message of the specified type. Otherwise, you get a `NoUsableMessageException` error. For more information, see Message Groups.

Setting predefined request attributes in your Lex V2 bot

Amazon Lex V2 provides predefined request attributes for managing the way that it processes information sent to your bot. The attributes do not persist for the entire session, so you must send the predefined attributes in each request. All predefined attributes are in the `x-amz-lex:` namespace.

Disabling intent switches in your Lex V2 bot

To control whether users can switch between intents during intent confirmation or slot elicitation, use the `x-amz-lex:intent-switch` request attribute. When set to `DISABLE`, this attribute prevents users from triggering a different intent while they are in the middle of completing the current intent flow.

For example, if a user is in the process of booking a flight and is being prompted for flight details, then utterances such as “check weather” or “book hotel” - which might normally trigger other intents - will be ignored, ensuring the conversation remains focused on the current booking process.

Setting the session timeout

Amazon Lex retains context information—slot data and session attributes—until a conversation session ends. To control how long a session lasts for a bot, set the session timeout. By default, session duration is 5 minutes, but you can specify any duration between 0 and 1,440 minutes (24 hours).

For example, suppose that you create a `ShoeOrdering` bot that supports intents such as `OrderShoes` and `GetOrderStatus`. When Amazon Lex detects that the user's intent is to order shoes, it asks for slot data. For example, it asks for shoe size, color, brand, etc. If the user provides some of the slot data but doesn't complete the shoe purchase, Amazon Lex remembers all of the

slot data and session attributes for the entire session. If the user returns to the session before it expires, they can provide the remaining slot data, and complete the purchase.

In the Amazon Lex V2 console, you set the session timeout when you create a bot. With the AWS command line interface (AWS CLI) or API, you set the timeout when you create a bot with the CreateBot operation by setting the idleSessionTTLInSeconds field.

Sharing information between intents with your Lex V2 bot

Amazon Lex V2 supports sharing information between intents. To share between intents, use output contexts or session attributes.

To use output contexts, you define an output context when you create or update an intent. When the intent is fulfilled, responses from Amazon Lex V2 contain the context and slot values from the intent as context parameters. You can use these parameters as default values in subsequent intents or in you application code or Lambda functions.

To use session attributes, you set the attributes in your Lambda or application code. For example, a user of the `ShoeOrdering` bot starts by ordering shoes. The bot engages in a conversation with the user, gathering slot data, such as shoe size, color, and brand. When the user places an order, the Lambda function that fulfills the order sets the `orderNumber` session attribute, which contains the order number. To get the status of the order, the user uses the `GetOrderStatus` intent. The bot can ask the user for slot data, such as order number and order date. When the bot has the required information, it returns the status of the order.

If you think that your users might switch intents during the same session, you can design your bot to return the status of the latest order. Instead of asking the user for order information again, you use the `orderNumber` session attribute to share information across intents and fulfill the `GetOrderStatus` intent. The bot does this by returning the status of the last order that the user placed.

Setting complex attributes in your Lex V2 bot

Session and request attributes are string-to-string maps of attributes and values. In many cases, you can use the string map to transfer attribute values between your client application and a bot. In some cases, however, you might need to transfer binary data or a complex structure that can't be easily converted to a string map. For example, the following JSON object represents an array of the three most populous cities in the United States:

```
{
```

```
   "cities": [
      {
         "city": {
            "name": "New York",
            "state": "New York",
            "pop": "8537673"
         }
      },
      {
         "city": {
            "name": "Los Angeles",
            "state": "California",
            "pop": "3976322"
         }
      },
      {
         "city": {
            "name": "Chicago",
            "state": "Illinois",
            "pop": "2704958"
         }
      }
   ]
}
```

This array of data doesn't translate well to a string-to-string map. In such a case, you can transform an object to a simple string so that you can send it to your bot with the RecognizeText and RecognizeUtterance operations.

For example, if you are using JavaScript, you can use the `JSON.stringify` operation to convert an object to JSON, and the `JSON.parse` operation to convert JSON text to a JavaScript object:

```
// To convert an object to a string.
var jsonString = JSON.stringify(object, null, 2);
// To convert a string to an object.
var obj = JSON.parse(JSON string);
```

To send attributes with the `RecognizeUtterance` operation, you must base64 encode the attributes before you add them to the request header, as shown in the following JavaScript code:

```
var encodedAttributes = new Buffer(attributeString).toString("base64");
```

You can send binary data to the `RecognizeText` and `RecognizeUtterance` operations by first converting the data to a base64-encoded string, and then sending the string as the value in the session attributes:

```
"sessionAttributes" : {
   "binaryData": "base64 encoded data"
}
```

Understanding Amazon Lex V2 bot sessions

When a user starts a conversation with your bot, Amazon Lex V2 creates a *session*. The information exchanged between your application and Amazon Lex V2 makes up the session state for the conversation. When you make a request, the session is identified by an identifier that you specify. For more information about the session identifier, see the `sessionId` field in the RecognizeText or RecognizeUtterance operation.

You can modify the session state sent between your application and your bot. For example, you can create and modify session attributes that contain custom information about the session, and you can change the flow of the conversation by setting the dialog context to interpret the next utterance.

There are three ways that you can update session state.

- Pass the session information inline as part of a call to the `RecognizeText` or `RecognizeUtterance` operation.
- Use a Lambda function with the `RecognizeText` or `RecognizeUtterance` operation that is called after each turn of the conversation. For more information, see Integrating an AWS Lambda function into your Amazon Lex V2 bot. The other is to use the Amazon Lex V2 runtime API in your application to make changes to the session state.
- Use operations that enable you to manage session information for a conversation with your bot. The operations are the PutSession operation, the GetSession operation, and the DeleteSession operation. You use these operations to get information about the state of your user's session with your bot, and to have fine-grained control over the state.

Use the `GetSession` operation when you want to get the current state of the session. The operation returns the current state of the session, including the state of the dialog with your user, any session attributes that have been set and slot values for the current intent and any other intents that Amazon Lex V2 has identified as possible intents that match the user's utterance.

The `PutSession` operation enables you to directly manipulate the current session state. You can set the session, including the type of dialog action that the bot will perform next and the messages that Amazon Lex V2 sends to the user. This gives you control over the flow of the conversation with the bot. Set the dialog action `type` field to `Delegate` to have Amazon Lex V2 determine the next action for the bot.

You can use the `PutSession` operation to create a new session with a bot and set the intent that the bot should start with. You can also use the `PutSession` operation to change from one intent to another. When you create a session or change the intent you also can set session state, such as slot values and session attributes. When the new intent is finished, you have the option of restarting the prior intent.

The response from the `PutSession` operation contains the same information as the `RecognizeUtterance` operation. You can use this information to prompt the user for the next piece of information, just as you would with the response from the `RecognizeUtterance` operation.

Use the `DeleteSession` operation to remove an existing session and start over with a new session. For example, when you are testing your bot you can use the `DeleteSession` operation to remove test sessions from your bot.

The session operations work with your fulfillment Lambda functions. For example, if your Lambda function returns `Failed` as the fulfillment state you can use the `PutSession` operation to set the dialog action type to `close` and `fulfillmentState` to `ReadyForFulfillment` to retry the fulfillment step.

Here are some things that you can do with the session operations:

- Have the bot start a conversation instead of waiting for the user.
- Switch intents during a conversation.
- Return to a previous intent.
- Start or restart a conversation in the middle of the interaction.
- Validate slot values and have the bot re-prompt for values that are not valid.

Each of these are described further below.

Starting a new session

If you want to have the bot start the conversation with your user, you can use the `PutSession` operation.

- Create a welcome intent with no slots and a conclusion message that prompts the user to state an intent. For example, "What would you like to order? You can say 'Order a drink' or 'Order a pizza.'"
- Call the `PutSession` operation. Set the intent name to the name of your welcome intent and set the dialog action to `Delegate`.
- Amazon Lex will respond with the prompt from your welcome intent to start the conversation with your user.

Switching intents

You can use the `PutSession` operation to switch from one intent to another. You can also use it to switch back to a previous intent. You can use the `PutSession` operation to set session attributes or slot values for the new intent.

- Call the `PutSession` operation. Set the intent name to the name of the new intent and set the dialog action to `Delegate`. You can also set any slot values or session attributes required for the new intent.
- Amazon Lex will start a conversation with the user using the new intent.

Resuming a prior intent

To resume a prior intent you use the `GetSession` operation to get the state of the intent, perform the needed interaction, and then use the `PutSession` operation to set the intent to its previous dialog state.

- Call the `GetSession` operation. Store the state of the intent.
- Perform another interaction, such as fulfilling a different intent.
- Using the information saved information for the previous intent, call the `PutSession` operation. This will return the user to the previous intent in the same place in the conversation.

In some cases it may be necessary to resume your user's conversation with your bot. For example, say that you have created a customer service bot. Your application determines that the user needs to talk to a customer service representative. After talking to the user, the representative can direct the conversation back to the bot with the information that they collected.

To resume a session, use steps similar to these:

- Your application determines that the user needs to speak to a customer service representative.
- Use the `GetSession` operation to get the current dialog state of the intent.
- The customer service representative talks to the user and resolves the issue.
- Use the `PutSession` operation to set the dialog state of the intent. This may include setting slot values, setting session attributes, or changing the intent.
- The bot resumes the conversation with the user.

Validating slot values

You can validate responses to your bot using your client application. If the response isn't valid, you can use the `PutSession` operation to get a new response from your user. For example, suppose that your flower ordering bot can only sell tulips, roses, and lilies. If the user orders carnations, your application can do the following:

- Examine the slot value returned from the `PostText` or `PostContent` response.
- If the slot value is not valid, call the `PutSession` operation. Your application should clear the slot value, set the `slotToElicit` field, and set the `dialogAction.type` value to `elicitSlot`. Optionally, you can set the `message` and `messageFormat` fields if you want to change the message that Amazon Lex uses to elicit the slot value.

Integrating an AWS Lambda function into your Amazon Lex V2 bot

With AWS Lambda functions, you can extend and better control the behavior of your Amazon Lex V2 bot through custom functions that you define. Amazon Lex V2 uses one Lambda function per bot alias per language instead of one Lambda function for each intent. Before you begin, determine which fields in the input event you want to draw information from and which fields in the response you want to manipulate and return from your Lambda function

To integrate a Lambda function with your Amazon Lex V2 bot, carry out the following steps:

1. Create a function in AWS Lambda using your programming language of choice and write up your script.
2. Make sure that the function returns a structure matching the response format.
3. Deploy the Lambda function.
4. Associate the Lambda function with an Amazon Lex V2 bot alias with the console or API operations.
5. Select the conversation stages at which you want to invoke your Lambda function with the console or API operations.
6. Build your Amazon Lex V2 bot and test that the Lambda function works as intended. Debug your function with the help of Amazon CloudWatch.

Topics

AWS Lambda input event format for Lex V2

The first step in integrating a Lambda function into your Amazon Lex V2 bot is to understand the fields in the Amazon Lex V2 event and to determine the information from these fields that you

want to use when writing your script. The following JSON object shows the general format of an Amazon Lex V2 event passed to a Lambda function:

> **Note**
>
> The input format may change without a corresponding change to the `messageVersion`. Your code shouldn't throw an error if new fields are present.

```
{
    "messageVersion": "1.0",
    "invocationSource": "DialogCodeHook | FulfillmentCodeHook",
    "inputMode": "DTMF | Speech | Text",
    "responseContentType": "audio/mpeg | audio/ogg | audio/pcm | text/plain;
 charset=utf-8",
    "sessionId": string,
    "inputTranscript": string,
    "invocationLabel": string,
    "bot": {
        "id": string,
        "name": string,
        "localeId": string,
        "version": string,
        "aliasId": string,
        "aliasName": string
    },
    "interpretations": [
        {
            "interpretationSource": "Bedrock | Lex",
            "intent": {
                // see Intent for details about the structure
            },
            "nluConfidence": number,
            "sentimentResponse": {
                "sentiment": "MIXED | NEGATIVE | NEUTRAL | POSITIVE",
                "sentimentScore": {
                    "mixed": number,
                    "negative": number,
                    "neutral": number,
                    "positive": number
                }
            }
```

```
        },
        ...
    ],
    "proposedNextState": {
        "dialogAction": {
            "slotToElicit": string,
            "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"
        },
        "intent": {
            // see Intent for details about the structure
        },
        "prompt": {
            "attempt": string
        }
    },
    "requestAttributes": {
        string: string,
        ...
    },
    "sessionState": {
        // see Session state for details about the structure
    },
    "transcriptions": [
        {
            "transcription": string,
            "transcriptionConfidence": number,
            "resolvedContext": {
                "intent": string
            },
            "resolvedSlots": {
                slot name: {
                    // see Slots for details about the structure
                },
                ...
            }
        },
        ...
    ]
}
```

Each field in the input event is described below:

messageVersion

The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function.

> **Note**
>
> You configure this value when you define an intent. In the current implementation, Amazon Lex V2 only supports message version 1.0. Therefore, the console assumes the default value of 1.0 and doesn't show the message version.

invocationSource

The code hook that called the Lambda function. The following values are possible:

`DialogCodeHook` – Amazon Lex V2 called the Lambda function after input from the user.

`FulfillmentCodeHook` – Amazon Lex V2 called the Lambda function after filling all the required slots and the intent is ready for fulfillment.

inputMode

The mode of the user utterance. The possible values are as follows:

`DTMF` – The user input the utterance using a touch-tone keypad (Dual Tone Multi-Frequency).

`Speech` – The user spoke the utterance.

`Text` – The user typed the utterance.

responseContentType

The mode of the bot's response to the user. `text/plain; charset=utf-8` indicates that the last utterance was written, while a value beginning with `audio` indicates that the last utterance was spoken.

sessionId

The alphanumeric session identifier used for the conversation.

inputTranscript

A transcription of the input from the user.

- For text input, this is the text that the user typed. For DTMF input, this is the key that the user input.
- For speech input, this is the text to which Amazon Lex V2 converts the user utterance in order to invoke an intent or fill a slot.

invocationLabel

A value that indicates the response that invoked the Lambda function. You can set invocation labels for the initial response, slots, and confirmation response.

bot

Information about the bot that processed the request, consisting of the following fields:

- **id** – The identifier assigned to the bot when you created it. You can see the bot ID in the Amazon Lex V2 console on the bot **Settings** page.
- **name** – The name that you gave the bot when you created it.
- **localeId** – The identifier of the locale that you used for your bot. For a list of locales, see Languages and locales supported by Amazon Lex V2.
- **version** – The version of the bot that processed the request.
- **aliasId** – The identifier assigned to the bot alias when you created it. You can see the bot alias ID in the Amazon Lex V2 console on the **Aliases** page. If you can't see the alias ID in the list, choose the gear icon on the upper right and turn on **Alias ID**.
- **aliasName** – The name that you gave the bot alias.

interpretations

A list of information about intents that Amazon Lex V2 considers possible matches to the user's utterance. Each item is a structure that provides information about the utterance's match to an intent, with the following format:

```
{
    "intent": {
        // see Intent for details about the structure
```

```
    },
    "interpretationSource": "Bedrock | Lex",
    "nluConfidence": number,
    "sentimentResponse": {
        "sentiment": "MIXED | NEGATIVE | NEUTRAL | POSITIVE",
        "sentimentScore": {
            "mixed": number,
            "negative": number,
            "neutral": number,
            "positive": number
        }
    }
}
```

The fields within the structure are as follows:

- **intent** – A structure containing information about the intent. See Intent for details about the structure.
- **nluConfidence** – A score that indicates how confident Amazon Lex V2 is that the intent matches the user's intent.
- **sentimentResponse** – An analysis of the sentiment of the response, containing the following fields:
  - **sentiment** – Indicates whether the sentiment of the utterance is `POSITIVE`, `NEGATIVE`, `NEUTRAL`, or `MIXED`.
  - **sentimentScore** – A structure mapping each sentiment to a number indicating how confident Amazon Lex V2 is that the utterance conveys that sentiment.
- **interpretationSource** – Indicates whether a slot is resolved by Amazon Lex V2 or Amazon Bedrock.

proposedNextState

If the Lambda function sets the `dialogAction` of the `sessionState` to `Delegate`, this field appears and shows Amazon Lex V2's proposal for the next step in the conversation. Otherwise, the next state depends on the settings that you return in the response from your Lambda function. This structure is only present if both of the following statements are true:

1. The `invocationSource` value is `DialogCodeHook`
2. The predicted `type` of `dialogAction` is `ElicitSlot`.

You can use this information to add `runtimeHints` at the right point in the conversation. See Improving recognition of slot values with runtime hints in the conversation for more information. `proposedNextState` is a structure containing the following fields:

The structure of `proposedNextState` is as follows:

```
"proposedNextState": {
    "dialogAction": {
        "slotToElicit": string,
        "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"
    },
    "intent": {
        // see Intent for details about the structure
    },
    "prompt": {
        "attempt": string
    }
}
```

- **dialogAction** – Contains information about the next step that Amazon Lex V2 proposes. The fields in the structure are as follows:
  - **slotToElicit** – The slot to elicit next as proposed by Amazon Lex V2. This field only appears if the `type` is `ElicitSlot`.
  - **type** – The next step in the conversation as proposed by Amazon Lex V2. The following values are possible:

    `Delegate` – Amazon Lex V2 determines the next action.

    `ElicitIntent` – The next action is to elicit an intent from the user.

    `ElicitSlot` – The next action is to elicit a slot value from the user.

    `Close` – Ends the intent fulfillment process and indicates that there will not be a response from the user.

    `ConfirmIntent` – The next action is to ask the user if the slots are correct and the intent is ready for fulfillment.
- **intent** – The intent that the bot has determined that the user is trying to fulfill. See Intent for details about the structure.

- **prompt** – A structure containing the field `attempt`, which maps to a value that specifies how many times Amazon Lex V2 has prompted the user for the next slot. The possible values are `Initial` for the first attempt and `Retry1`, `Retry2`, `Retry3`, `Retry4`, and `Retry5` for subsequent attempts.

requestAttributes

A structure containing request-specific attributes that the client sends in the request. Use request attributes to pass information that doesn't need to persist for the entire session. If there are no request attributes, the value will be null. For more information, see Setting request attributes for your Lex V2 bot.

sessionState

The current state of the conversation between the user and your Amazon Lex V2 bot. See Session state for details about the structure.

transcriptions

A list of transcriptions that Amazon Lex V2 considers possible matches to the user's utterance. For more information, see Using voice transcription confidence scores to improve conversations with your Lex V2 bot. Each item is an object with the following format, containing information about one possible transcription:

```
{
    "transcription": string,
    "transcriptionConfidence": number,
    "resolvedContext": {
        "intent": string
    },
    "resolvedSlots": {
        slot name: {
            // see Slots for details about the structure
        },
        ...
    }
}
```

The fields are described below:

- **transcription** – A transcription that Amazon Lex V2 considers a possible match to the user's audio utterance.
- **transcriptionConfidence** – A score that indicates how confident Amazon Lex V2 is that the intent matches the user's intent.
- **resolvedContext** – A structure containing the field `intent`, which maps to the intent to which the utterance pertains.
- **resolvedSlots** – A structure whose keys are the names of each slot that is resolved by the utterance. Each slot name maps to a structure containing information about that slot. See Slots for details about the structure.

AWS Lambda response format for Lex V2

The second step in integrating a Lambda function into your Amazon Lex V2 bot is to understand the fields in the Lambda function response and to determine which parameters you want to manipulate. The following JSON object shows the general format of an Lambda response that is returned to Amazon Lex V2:

```
{
    "sessionState": {
        // see Session state for details about the structure
    },
    "messages": [
        {
            "contentType": "CustomPayload | ImageResponseCard | PlainText | SSML",
            "content": string,
            "imageResponseCard": {
                "title": string,
                "subtitle": string,
                "imageUrl": string,
                "buttons": [
                    {
                        "text": string,
                        "value": string
                    },
                    ...
                ]
            }
        },
        ...
```

```
    ],
    "requestAttributes": {
        string: string,
        ...
    }
}
```

Each field in the response is described below:

sessionState

The state of the conversation between the user and your Amazon Lex V2 bot that you want to return. See Session state for details about the structure. This field is always required.

messages

A list of messages that Amazon Lex V2 returns to the customer for the next turn of the conversation. If the `contentType` you provide is `PlainText`, `CustomPayload`, or `SSML`, write the message you want to return to the customer in the `content` field. If the `contentType` you provide is `ImageResponseCard`, give the details of the card in the `imageResponseCard` field. If you don't supply messages, Amazon Lex V2 uses the appropriate message defined when the bot was created.

The `messages` field is required if the `dialogAction.type` is `ElicitIntent` or `ConfirmIntent`.

Each item in the list is a structure in the following format, containing information about a message to return to the user. Here is an example:

```
{
    "contentType": "CustomPayload | ImageResponseCard | PlainText | SSML",
    "content": string,
    "imageResponseCard": {
        "title": string,
        "subtitle": string,
        "imageUrl": string,
        "buttons": [
            {
                "text": string,
                "value": string
            },
```

```
            ...
        ]
    }
}
```

A description for each field is provided below:

- **contentType** – The type of message to use.

  `CustomPayload` – A response string that you can customize to include data or metadata for your application.

  `ImageResponseCard` – An image with buttons that the customer can select. See ImageResponseCard for more information.

  `PlainText` – A plain text string.

  SSML – A string that includes Speech Synthesis Markup Language to customize the audio response.

- **content** – The message to send to the user. Use this field if the message type is `PlainText`, `CustomPayload`, or SSML.
- **imageResponseCard** – Contains the definition of the response card to show to the user. Use this field if the message type is `ImageResponseCard`. Maps to a structure containing the following fields:
  - **title** – The title of the response card.
  - **subtitle** – The prompt for the user to choose a button.
  - **imageUrl** – A link to an image for the card.
  - **buttons** – A list of structures containing information about a button. Each structure contains a `text` field with the text to display and a `value` field with the value to send to Amazon Lex V2 if the customer selects that button. You can include up to three buttons.

requestAttributes

A structure containing request-specific attributes for the response to the customer. See Setting request attributes for your Lex V2 bot for more information. This field is optional.

Required fields in the response

Minimally, the Lambda response must include a `sessionState` object. Within that, provide a `dialogAction` object and specify the `type` field. Depending on the `type` of `dialogAction` that you provide, there may be other required fields for the Lambda response. These requirements are described as follows, alongside minimal working examples:

Delegate

Delegate lets Amazon Lex V2 determine the next step. No other fields are required.

```
{
    "sessionState": {
        "dialogAction": {
            "type": "Delegate"
    }
}
```

ElicitIntent

ElicitIntent prompts the customer to express an intent. You must include at least one message in the `messages` field to prompt elicitation of an intent.

```
{
    "sessionState": {
        "dialogAction": {
            "type": "ElicitIntent"
    },
    "messages": [
        {
            "contentType": PlainText,
            "content": "How can I help you?"
        }
    ]
}
```

ElicitSlot

ElicitSlot prompts the customer to provide a slot value. You must include the name of the slot in the `slotToElicit` field in the `dialogAction` object. You must also include the `name` of the `intent` in the `sessionState` object.

```
{`
    "sessionState": {
        "dialogAction": {
            "slotToElicit": "OriginCity",
            "type": "ElicitSlot"
        },
        "intent": {
            "name": "BookFlight"
        }
    }
}
```

ConfirmIntent

ConfirmIntent confirms the customer's slot values and whether the intent is ready to be fulfilled. You must include the `name` of the `intent` in the `sessionState` object and the `slots` to be confirmed. You must also include at least one message in the `messages` field to ask the user for confirmation of the slot values. Your message should prompt a "yes" or "no" response. If the user responds "yes", Amazon Lex V2 sets the `confirmationState` of the intent to `Confirmed`. If the user responds "no", Amazon Lex V2 sets the `confirmationState` of the intent to `Denied`.

```
{
    "sessionState": {
        "dialogAction": {
            "type": "ConfirmIntent"
        },
        "intent": {
            "name": "BookFlight",
            "slots": {
                "DepartureDate": {
                    "value": {
                        "originalValue": "tomorrow",
                        "interpretedValue": "2023-05-09",
                        "resolvedValues": [
                            "2023-05-09"
                    ]
                 }
                },
                "DestinationCity": {
                    "value": {
                        "originalValue": "sf",
                        "interpretedValue": "sf",
```

```
                        "resolvedValues": [
                            "sf"
                        ]
                    }
                },
                "OriginCity": {
                    "value": {
                        "originalValue": "nyc",
                        "interpretedValue": "nyc",
                        "resolvedValues": [
                            "nyc"
                        ]
                    }
                }
            }
        }
    },
    "messages": [
        {
            "contentType": PlainText,
            "content": "Okay, you want to fly from {OriginCity} to \
            {DestinationCity} on {DepartureDate}. Is that correct?"
        }
    ]
}
```

Close

Close ends the fulfillment process of the intent and indicates that no further responses are expected from the user. You must include the `name` and `state` of the `intent` in the `sessionState` object. The compatible intent states are `Failed`, `Fulfilled`, and `InProgress`.

```
"sessionState": {
    "dialogAction": {
        "type": "Close"
    },
    "intent": {
        "name": "BookFlight",
        "state": "Failed | Fulfilled | InProgress"
    }
}
```

Common structures in an AWS Lambda function for Amazon Lex V2

Within the Lambda response, there are a number of structures that recur. Details about these common structures are provided in this section.

Intent

```
"intent": {
    "confirmationState": "Confirmed | Denied | None",
    "name": string,
    "slots": {
        // see Slots for details about the structure
    },
    "state": "Failed | Fulfilled | FulfillmentInProgress | InProgress |
 ReadyForFulfillment | Waiting",
    "kendraResponse": {
        // Only present when intent is KendraSearchIntent. For details, see
// https://docs.aws.amazon.com/kendra/latest/dg/API_Query.html#API_Query_ResponseSyntax
       }
}
```

The `intent` field is mapped to an object with the following fields:

confirmationState

Indicates whether the user has confirmed the slots for the intent and the intent is ready for fulfillment. The following values are possible:

`Confirmed` – The user confirms that the slot values are correct.

`Denied` – The user indicates that the slot values are incorrect.

`None` – The user has not reached the confirmation stage yet.

name

The name of the intent.

slots

Information about the slots required to fulfill the intent. See Slots for details about the structure.

state

Indicates the fulfillment state for the intent. The following values are possible:

`Failed` – The bot failed to fulfill the intent.

`Fulfilled` – The bot has completed fulfillment of the intent.

`FulfillmentInProgress` – The bot is in the middle of fulfilling the intent.

`InProgress` – The bot is in the middle of eliciting the slot values that are necessary to fulfill the intent.

`ReadyForFulfillment` – The bot has elicited all the slot values for the intent and is ready to fulfill the intent.

`Waiting` – The bot is waiting for a response from the user (limited to streaming conversations).

kendraResponse

Contains information about the results of the Kendra search query. This field only appears if the intent is a `KendraSearchIntent`. See the response syntax in the Query API call for Kendra for more information.

Slots

The `slots` field exists within an `intent` structure and is mapped to a structure whose keys are the names of the slots for that intent. If the slot is not a multi-valued slot (see Using multiple values in a slot for more details), it is mapped to a structure with the following format. Note that the `shape` is `Scalar`.

```
{
    slot name: {
        "shape": "Scalar",
        "value": {
            "originalValue": string,
            "interpretedValue": string,
            "resolvedValues": [
                string,
                ...
            ]
        }
    }
```

```
}
```

If the slot is a multi-valued slot, the object to which it is mapped contains another field called `values`, which is mapped to a list of structures, each containing information about a slot that makes up the multi-valued slot. The format of each object in the list matches that of the object to which a regular slot is mapped. Note that the `shape` is `List`, but the `shape` of the component slots under `values` is `Scalar`.

```
{
    slot name: {
    "shape": "List",
    "value": {
        "originalValue": string,
        "interpretedValue": string,
        "resolvedValues": [
            string,
            ...
        ]
    },
    "values": [
        {
            "shape": "Scalar",
            "value": {
                "originalValue": string,
                "interpretedValue": string,
                "resolvedValues": [
                    string,
                    ...
                ]
            }
        },
        {
            "shape": "Scalar",
            "value": {
                "originalValue": string,
                "interpretedValue": string,
                "resolvedValues": [
                    string,
                    ...
                ]
            }
        },
        ...
```

```
    ]
}
```

The fields in the slot object are described below:

shape

The shape of the slot. This value is `List` if there are multiple values in the slot (see Using multiple values in a slot for more details) and is `Scalar` otherwise.

value

An object containing information about the value that the user provided for a slot and Amazon Lex V2's interpretation, in the following format:

```
{
    "originalValue": string,
    "interpretedValue": string,
    "resolvedValues": [
        string,
        ...
    ]
}
```

The fields are described below:

- **originalValue** – The part of the user's response to the slot elicitation that Amazon Lex V2 determines is relevant to the slot value.
- **interpretedValue** – The value that Amazon Lex V2 determines for the slot, given the user input.
- **resolvedValues** – A list of values that Amazon Lex V2 determines are possible resolutions for the user input.

values

A list of objects containing information about the slots that make up the multi-value slot. The format of each object matches that of a normal slot, with the `shape` and `value` fields described above. `values` only appears if the slot consists of multiple values (see Using multiple values in a slot for more details). The following JSON object shows two component slots:

```
"values": [
    {
```

```
        "shape": "Scalar",
        "value": {
            "originalValue": string,
            "interpretedValue": string,
            "resolvedValues": [
                string,
                ...
            ]
        }
    },
    {
        "shape": "Scalar",
        "value": {
            "originalValue": string,
            "interpretedValue": string,
            "resolvedValues": [
                string,
                ...
            ]
        }
    },
    ...
]
```

Session state

The `sessionState` field is mapped to an object containing information about the state of the conversation with the user. The actual fields that appear in the object depend on the type of dialog action. See Required fields in the response for the required fields in a Lambda response. The format of the `sessionState` object is as follows:

```
"sessionState": {
    "activeContexts": [
        {
            "name": string,
            "contextAttributes": {
                string: string
            },
            "timeToLive": {
                "timeToLiveInSeconds": number,
                "turnsToLive": number
            }
        },
```

```
        ...
    ],
    "sessionAttributes": {
        string: string,
        ...
    },
    "runtimeHints": {
        "slotHints": {
            intent name: {
                slot name: {
                    "runtimeHintValues": [
                        {
                            "phrase": string
                        },
                        ...
                    ]
                },
                ...
            },
            ...
        }
    },
    "dialogAction": {
        "slotElicitationStyle": "Default | SpellByLetter | SpellByWord",
        "slotToElicit": string,
        "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"
    },
    "intent": {
        // see Intent for details about the structure
    },
    "originatingRequestId": string
}
```

The fields are described below:

activeContexts

A list of objects containing information about a context that a user is using in a session. Use contexts to facilitate and control intent recognition. For more information about contexts, see Setting intent context for your Lex V2 bot. Each object is formatted as follows:

```
{
    "name": string,
```

```
    "contextAttributes": {
        string: string
    },
    "timeToLive": {
        "timeToLiveInSeconds": number,
        "turnsToLive": number
    }
}
```

The fields are described below:

- **name** – The name of the context.
- **contextAttributes** – An object containing the names of attributes for the context and the values that they are mapped to.
- **timeToLive** – An object that specifies how long the context remains active. This object can contain one or both of the following fields:
  - **timeToLiveInSeconds** – The number of seconds that the context remains active.
  - **turnsToLive** – The number of turns that the context remains active.

sessionAttributes

A map of key/value pairs representing session-specific context information. For more information, see Setting session attributes for your Lex V2 bot. The object is formatted as follows:

```
{
    string: string,
    ...
}
```

runtimeHints

Provides hints to the phrases that a customer is likely to use for a slot in order to improve audio recognition. The values that you provide in the hints boost audio recognition of those values over similar-sounding words. The format of the `runtimeHints` object is as follows:

```
{
    "slotHints": {
        intent name: {
            slot name: {
                "runtimeHintValues": [
```

```
                    {
                        "phrase": string
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

The `slotHints` field maps to an object whose fields are the names of the intents in the bot. Each intent name maps to an object whose fields are the names of the slots for that intent. Each slot name maps to a structure with a single field, `runtimeHintValues`, which is a list of objects. Each object contains a `phrase` field that maps to a hint.

dialogAction

Determines the next action for Amazon Lex V2 to take. The format of the object is as follows:

```
{
    "slotElicitationStyle": "Default | SpellByLetter | SpellByWord",
    "slotToElicit": string,
    "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"
}
```

The fields are described below:

- **slotElicitationStyle** – Determines how Amazon Lex V2 interprets audio input from the user if the `type` of `dialogAction` is `ElicitSlot`. For more information, see Capturing slot values with spelling styles during the conversation. The following values are possible:

  `Default` – Amazon Lex V2 interprets the audio input in the default manner to fulfill a slot.

  `SpellByLetter` – Amazon Lex V2 listens for the user's spelling of the slot value.

  `SpellByWord` – Amazon Lex V2 listens for the user's spelling of the slot value using words associated with each letter (for example, "a as in apple").
- **slotToElicit** – Defines the slot to elicit from the user if the `type` of `dialogAction` is `ElicitSlot`.

- **type** – Defines the action that the bot should execute. The following values are possible:

  `Delegate` – Lets Amazon Lex V2 determine the next step.

  `ElicitIntent` – Prompts the customer to express an intent.

  `ConfirmIntent` – Confirms the customer's slot values and whether the intent is ready for fulfillment.

  `ElicitSlot` – Prompts the customer to provide a slot value for an intent.

  `Close` – Ends the intent fulfillment process.

intent

See Intent for the structure of the `intent` field.

originatingRequestId

A unique identifier for the request. This field is optional for the Lambda response.

Creating an AWS Lambda function for your Amazon Lex V2 bot

To create a Lambda function for your Amazon Lex V2 bot, access AWS Lambda from your AWS Management Console and create a new function. You can refer to the AWS Lambda developer guide for more details about AWS Lambda.

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose **Functions** in the left sidebar.
3. Select **Create function**.
4. You can select **Author from scratch** to start with minimal code, **Use a blueprint** to select sample code for common use cases from a list, or **Container image** to select a container image to deploy for your function. If you select **Author from scratch**, continue with the following steps:

   a. Give your function a meaningful **Function name** to describe what it does.

   b. Choose a language from the drop down menu under **Runtime** to write your function in.

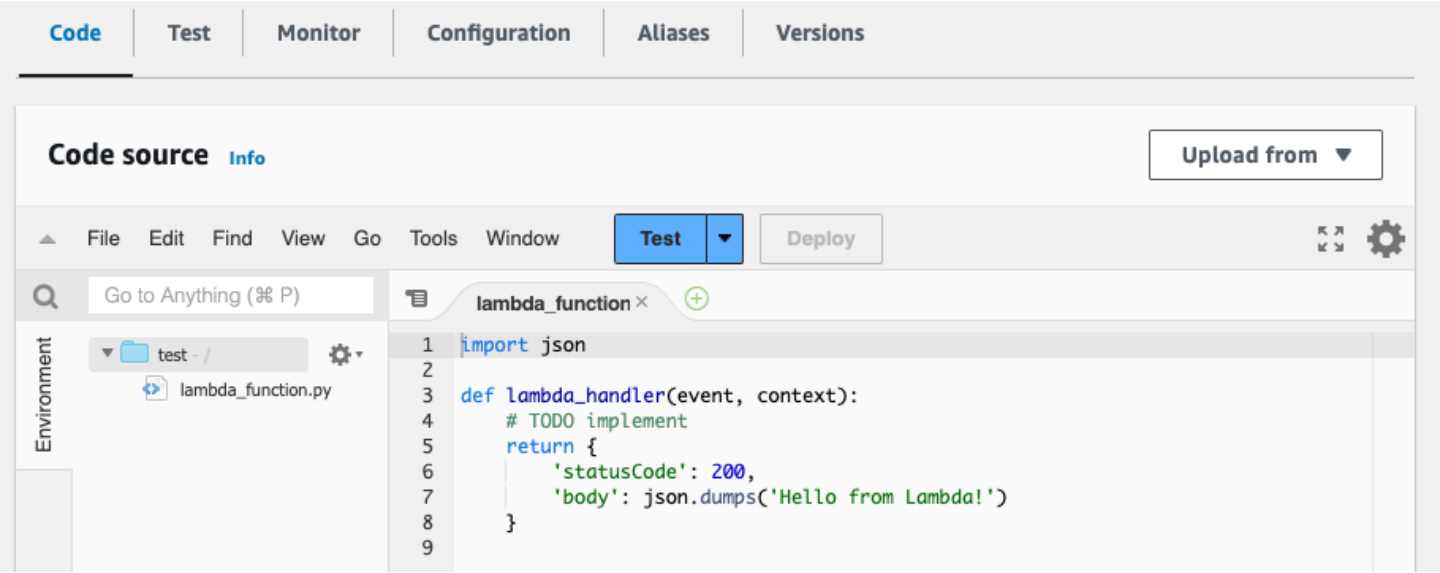
   c. Select an instruction set **Architecture** for your function.

d. By default, Lambda creates a role with basic permissions. To use an existing role or to create a role using AWS policy templates, expand the **Change default execution role** menu and select an option.

e. Expand the **Advanced settings** menu to configure more options.

5. Select **Create function**.

The following image shows what you see when you create a new function from scratch:



The Lambda handler function differs depending on the language you use. It minimally takes an `event` JSON object as an argument. You can see the fields in the `event` that Amazon Lex V2 provides at AWS Lambda input event format for Lex V2. Modify the handler function to ultimately return a `response` JSON object that matches the format described in AWS Lambda response format for Lex V2.

- Once you finish writing your function, select **Deploy** to allow the function to be used.

Remember that you can associate each bot alias with at most one Lambda function. However, you can define as many functions as you need for your bot within the Lambda code and call these functions in the Lambda handler function. For example, while all intents in the same bot alias must call the same Lambda function, you can create a router function that activates a separate function for each intent. The following is a sample router function that you can use or modify for your application:

```
import os
```

```
import json
import boto3

# reuse client connection as global
client = boto3.client('lambda')

def router(event):
    intent_name = event['sessionState']['intent']['name']
    fn_name = os.environ.get(intent_name)
    print(f"Intent: {intent_name} -> Lambda: {fn_name}")
    if (fn_name):
        # invoke lambda and return result
        invoke_response = client.invoke(FunctionName=fn_name, Payload =
 json.dumps(event))
        print(invoke_response)
        payload = json.load(invoke_response['Payload'])
        return payload
    raise Exception('No environment variable for intent: ' + intent_name)

def lambda_handler(event, context):
    print(event)
    response = router(event)
    return response
```

When to use AWS Lambda functions in Amazon Lex V2 bot conversations

You can use Lambda functions at the following points in a conversation with a user:

- In the initial response after the intent is recognized. For example, after the user says they want to order a pizza.
- After eliciting a slot value from the user. For example, after the user tells the bot the size of pizza they want to order.
- Between each retry for eliciting a slot. For example, if the customer doesn't use a recognized pizza size.
- When confirming an intent. For example, when confirming a pizza order.
- To fulfill an intent. For example, to place an order for a pizza.
- After fulfillment of the intent, and before your bot closes the conversation. For example, to switch to an intent to order a drink.

Topics

Attach an AWS Lambda function to a Amazon Lex V2 bot using the console

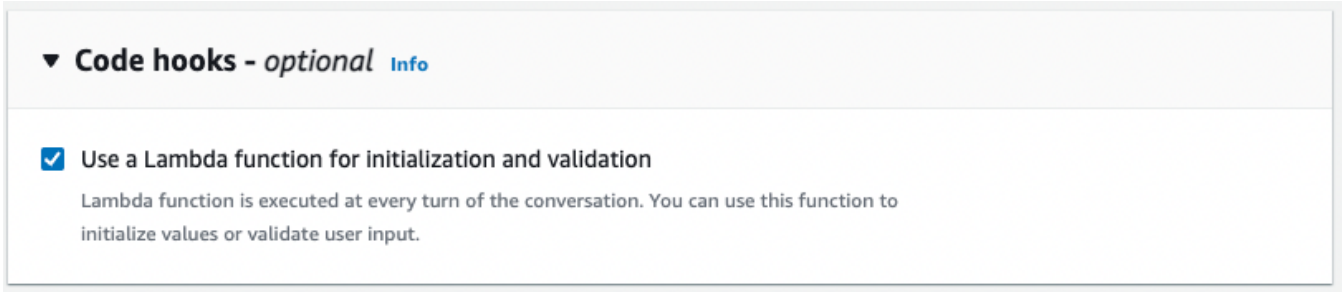
You must first attach a Lambda function to your Amazon Lex V2 bot alias before you can invoke it. You can only attach one Lambda function with each bot alias. Perform these steps to attach the Lambda function using the AWS console.

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Bots** from the left side panel and from the list of bots, choose the name of the bot that you want to attach a Lambda function to.
3. From the left side panel, select **Aliases** under the **Deployment** menu.
4. From the list of aliases, choose the name of the alias that you want to attach a Lambda function to.
5. In the **Languages** panel, select the language that you want a Lambda function to. Select **Manage languages in alias** to add a language if it is not present in the panel.
6. In the **Source** dropdown menu, choose the name of the Lambda function that you want to attach.
7. In the **Lambda function version or alias** dropdown menu, choose the version or alias of the Lambda function that you want to use. Then select **Save**. The same Lambda function is used for all intents in a language supported by the bot.

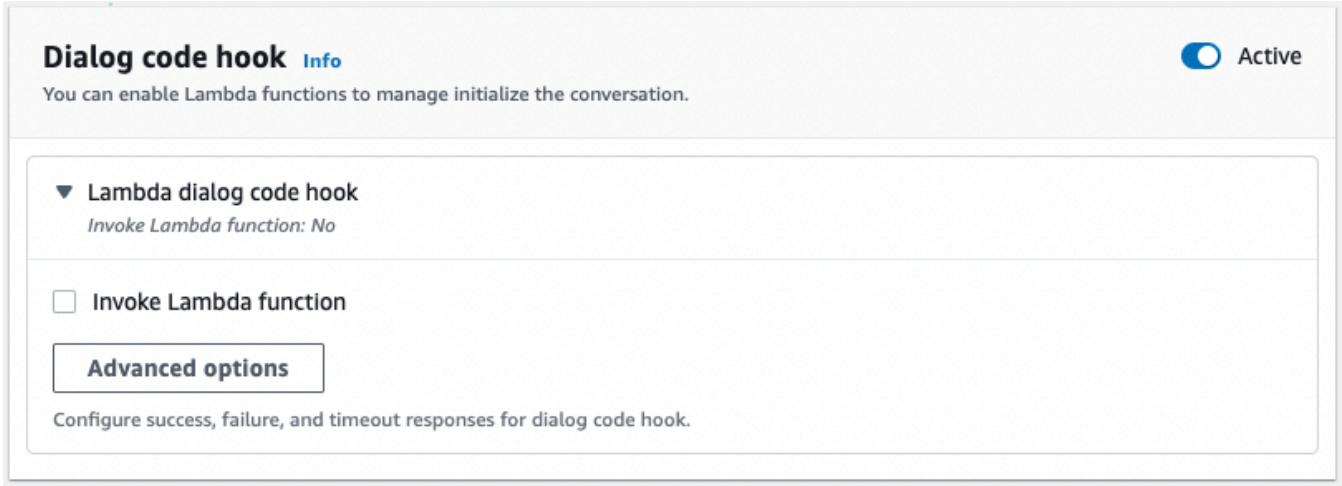
Setting a Amazon Lex V2 intent to invoke a Lambda function using the console

1. After selecting a bot, select **Intents** in the left side menu under the language of the bot for which you want to invoke the Lambda function.
2. Choose the intent in which you want to invoke the Lambda function to open the intent editor.
3. There are two options for setting the Lambda code hook:

    1. To invoke the Lambda function after every step of the conversation, scroll to the **Code hooks** section at the bottom of the intent editor and select the **Use a Lambda function for initialization and validation** check box, as in the following image:



2. Alternatively, use the **Dialog code hook** section in the conversation stages at which to invoke the Lambda function. The **Dialog code hook** section appears as follows:



There are two ways to control how Amazon Lex V2 calls the code hook for a response:

- Toggle the **Active** button to mark it as *active* or *inactive*. When a code hook is *active*, Amazon Lex V2 will call the code hook. When the code hook is *inactive*, Amazon Lex V2 does not run the code hook.
- Expand the **Lambda dialog code hook** section and select the **Invoke Lambda function** check box to mark it as *enabled* or *disabled*. You can only enable or disable a code hook when it is marked active. When it is marked *enabled*, the code hook is run normally. When it is *disabled*, the code hook is not called and Amazon Lex V2 acts as if the code hook returned successfully. To configure responses after the dialog code hook succeeds, fails, or times out, select **Advanced options**

The Lambda code hook can be invoked at the following conversation stages:

- To invoke the function as the **initial response**, scroll to the **Initial Response** section, expand the arrow next to **Response to acknowledge the user's request**, and select **Advanced options**. Find the **Dialog code hook** section at the bottom of the menu that pops up.

- To invoke the function after **slot elicitation**, scroll to the **Slots** section, expand the arrow next to the relevant **Prompt for slot**, and select **Advanced options**. Find the **Dialog code hook** section near the bottom of the menu that pops up, just above **Default values**.

  You can also invoke the function after each elicitation. To do this, expand **Bot elicits information** in the **Slot prompts** section, select **More prompt options**, and select the check box next to **Invoke Lambda code hook after each elicitation**.
- To invoke the function for **intent confirmation**, scroll to the **Confirmation** section, expand the arrow next to **Prompts to confirm the intent**, and select **Advanced options**. Find the **Dialog code hook** section at the bottom of the menu that pops up.
- To invoke the function for **intent fulfillment**, scroll to the **Fulfillment** section. Toggle the **Active** button to set the code hook to *active*. Expand the arrow next to **On successful fulfillment**, and select **Advanced options**. Select the check box next to **Use a Lambda function for fulfillment** under the **Fulfillment Lambda code hook** section to set the code hook to *enabled*.

4. Once you set the conversation stages at which to invoke the Lambda function, **Build** the bot again to test the function.

Attach an AWS Lambda function to a Amazon Lex V2 bot using API operations

You must first attach a Lambda function to your Amazon Lex V2 bot alias before you can invoke it. You can only associate one Lambda function with each bot alias. Perform these steps to attach the Lambda function using API operations.

If you are creating a new bot alias, use the CreateBotAlias operation to attach a Lambda function. To attach a Lambda function to an existing bot alias, use the UpdateBotAlias operation. Modify the `botAliasLocaleSettings` field to contain the correct settings:

```
{
    "botAliasLocaleSettings" : {
        locale: {
            "codeHookSpecification": {
                "lambdaCodeHook": {
                    "codeHookInterfaceVersion": "1.0",
                    "lambdaARN": "arn:aws:lambda:region:account-id:function:function-
name"
```

```
                }
            },
            "enabled": true
        },
        ...
    }
}
```

1. The `botAliasLocaleSettings` field maps to an object whose keys are the locales in which you want to attach the Lambda function. See Supported languages and locales for a list of supported locales and the codes that are valid keys.
2. To find the `lambdaARN` for a Lambda function, open the AWS Lambda console at https://console.aws.amazon.com/lambda/home, select **Functions** in the left sidebar, and select the function to associate with the bot alias. On the right side of the **Function overview**, find the `lambdaARN` under **Function ARN**. It should contain a region, account ID, and the name of the function.
3. To allow Amazon Lex V2 to invoke the Lambda function for the alias, set the `enabled` field to `true`.

Setting a Amazon Lex V2 intent to invoke a Lambda function using API operations

To set up the Lambda function invocation during an intent, use the CreateIntent operation if you are creating a new intent, or the UpdateIntent operation if you are invoking the function in an existing intent. The fields that control the Lambda function invocation in the intent operations are `dialogCodeHook`, `initialResponseSetting`, `intentConfirmationSetting`, and `fulfillmentCodeHook`.

If you invoke the function during the elicitation of a slot, use the CreateSlot operation if you are creating a new slot, or the UpdateSlot operation to invoke the function in an existing slot. The field that controls the Lambda function invocation in the slot operations is the `slotCaptureSetting` of the `valueElicitationSetting` object.

1. To set the Lambda dialog code hook to run after every turn of the conversation, set the `enabled` field of the following DialogCodeHookSettings object in the `dialogCodeHook` field to `true`:

```
"dialogCodeHook": {
    "enabled": boolean
```

```
}
```

2. Alternatively, you can set the Lambda dialog code hook to run only at specific points in the conversations by modifying the `codeHook` and/or `elicitationCodeHook` field within the structures that correspond to the conversation stages at which you want to invoke the function. To use the Lambda dialog code hook for intent fulfillment, use the `fulfillmentCodeHook` field in the CreateIntent or UpdateIntent operation. The structures and uses of these three types of code hooks are as follows:

codeHook

The `codeHook` field defines the settings for the code hook to run at a given stage in the conversation. It is a DialogCodeHookInvocationSetting object with the following structure:

```
"codeHook": {
    "active": boolean,
    "enableCodeHookInvocation": boolean,
    "invocationLabel": string,
    "postCodeHookSpecification": PostDialogCodeHookInvocationSpecification object,
}
```

- Change the `active` field to `true` for Amazon Lex V2 to call the code hook at that point in the conversation.
- Change the `enableCodeHookInvocation` field to `true` for Amazon Lex V2 to allow the code hook to run normally. If you mark it `false`, Amazon Lex V2 acts as if the code hook returned successfully.
- The `invocationLabel` indicates the dialog step from which the code hook is invoked.
- Use the `postCodeHookSpecification` field to specify the actions and messages that occur after the code hook succeeds, fails, or times out.

elicitationCodeHook

The `elicitationCodeHook` field defines the settings for the code hook to run in the event that a slot or slots need to be re-elicited. This scenario may occur if slot elicitation fails or intent confirmation is denied. The `elicitationCodeHook` field is an ElicitationCodeHookInvocationSetting object with the following structure:

```
"elicitationCodeHook": {
```

```
    "enableCodeHookInvocation": boolean,
    "invocationLabel": string
}
```

- Change the `enableCodeHookInvocation` field to `true` for Amazon Lex V2 to allow the code hook to run normally. If you mark it `false`, Amazon Lex V2 acts as if the code hook returned successfully.
- The `invocationLabel` indicates the dialog step from which the code hook is invoked.

fulfillmentCodeHook

The `fulfillmentCodeHook` field defines the settings for the code hook to run to fulfill the intent. It maps to the following FulfillmentCodeHookSettings object:

```
"fulfillmentCodeHook": {
    "active": boolean,
    "enabled": boolean,
    "fulfillmentUpdatesSpecification": FulfillmentUpdatesSpecification object,
    "postFulfillmentStatusSpecification": PostFulfillmentStatusSpecification object
}
```

- Change the `active` field to `true` for Amazon Lex V2 to call the code hook at that point in the conversation.
- Change the `enabled` field to `true` for Amazon Lex V2 to allow the code hook to run normally. If you mark it `false`, Amazon Lex V2 acts as if the code hook returned successfully.
- Use the `fulfillmentUpdatesSpecification` field to specify the messages that appear to update the user during fulfillment of the intent and the timing associated with them.
- Use the `postFulfillmentStatusSpecification` field to specify the messages and actions that occur after the code hook succeeds, fails, or times out.

You can invoke the Lambda code hook at the following points in a conversation by setting the `active` and `enableCodeHookInvocation`/`enabled` fields to `true`:

During the initial response

To invoke the Lambda function in the initial response after the intent is recognized, use the `codeHook` structure in the `initialResponse` field of the CreateIntent or UpdateIntent operation. The `initialResponse` field maps to the following InitialResponseSetting object:

```
"initialResponse": {
    "codeHook": {
        "active": boolean,
        "enableCodeHookInvocation": boolean,
        "invocationLabel": string,
        "postCodeHookSpecification": PostDialogCodeHookInvocationSpecification object,
    },
    "initialResponse": FulfillmentUpdatesSpecification object,
    "nextStep": PostFulfillmentStatusSpecification object,
    "conditional": ConditionalSpecification object
}
```

After slot elicitation or during slot re-elicitation

To invoke the Lambda function after eliciting a slot value, use the `slotCaptureSetting` field within the `valueElicitation` field of the CreateSlot or UpdateSlot operation. The `slotCaptureSetting` field maps to the following SlotCaptureSetting object:

```
"slotCaptureSetting": {
    "captureConditional": ConditionalSpecification object,
    "captureNextStep": DialogState object,
    "captureResponse": ResponseSpecification object,
    "codeHook": {
        "active": true,
        "enableCodeHookInvocation": true,
        "invocationLabel": string,
        "postCodeHookSpecification": PostDialogCodeHookInvocationSpecification object,
    },
    "elicitationCodeHook": {
        "enableCodeHookInvocation": boolean,
        "invocationLabel": string
    },
    "failureConditional": ConditionalSpecification object,
    "failureNextStep": DialogState object,
    "failureResponse": ResponseSpecification object
}
```

- To invoke the Lambda function after slot elicitation is successful, use the `codeHook` field.
- To invoke the Lambda function after slot elicitation fails and Amazon Lex V2 attempts to retry slot elicitation, use the `elicitationCodeHook` field.

After intent confirmation or denial

To invoke the Lambda function when confirming an intent, use the `intentConfirmationSetting` field of the CreateIntent or UpdateIntent operation. The `intentConfirmation` field maps to the following IntentConfirmationSetting object:

```
"intentConfirmationSetting": {
    "active": boolean,
    "codeHook": {
        "active": boolean,
        "enableCodeHookInvocation": boolean,
        "invocationLabel": string,
        "postCodeHookSpecification": PostDialogCodeHookInvocationSpecification object,
    },
    "confirmationConditional": ConditionalSpecification object,
    "confirmationNextStep": DialogState object,
    "confirmationResponse": ResponseSpecification object,
    "declinationConditional": ConditionalSpecification object,
    "declinationNextStep": FulfillmentUpdatesSpecification object,
    "declinationResponse": PostFulfillmentStatusSpecification object,
    "elicitationCodeHook": {
        "enableCodeHookInvocation": boolean,
        "invocationLabel": string,
    },
    "failureConditional": ConditionalSpecification object,
    "failureNextStep": DialogState object,
    "failureResponse": ResponseSpecification object,
    "promptSpecification": PromptSpecification object
}
```

- To invoke the Lambda function after the user confirms the intent and its slots, use the `codeHook` field.
- To invoke the Lambda function after the user denies the intent confirmation and Amazon Lex V2 attempts to retry slot elicitation, use the `elicitationCodeHook` field.

During intent fulfillment

To invoke the Lambda function to fulfill an intent, use the `fulfillmentCodeHook` field in the CreateIntent or UpdateIntent operation. The `fulfillmentCodeHook` field maps to the following FulfillmentCodeHookSettings object:

```
{
    "active": boolean,
    "enabled": boolean,
    "fulfillmentUpdatesSpecification": FulfillmentUpdatesSpecification object,
    "postFulfillmentStatusSpecification": PostFulfillmentStatusSpecification object
}
```

3. Once you set the conversation stages at which to invoke the Lambda function, use the `BuildBotLocale` operation to rebuild the bot in order to test the function.

Debugging a Lambda function using CloudWatch Logs logs

Amazon CloudWatch Logs is a tool for tracking API calls and metrics that you can use to help debug your Lambda functions. When you test your bot in the console or with API calls, CloudWatch logs each step of the conversation. If you use a print function in your Lambda code, CloudWatch displays it as well.

To view CloudWatch logs for your Lambda function

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Under the **Logs** menu in the left side bar, select **Log groups**.
3. Select your Lambda function log group, which should be of the format `/aws/lambda/`*`function-name`*.
4. The list of **Log streams** contains a log for each session with a bot. Choose a log stream to view it.
5. In the list of **Log events**, select the right arrow next to the **Timestamp** to expand the details for that event. Anything you print from your Lambda code will appear as a log event. Use this information to debug your code.
6. After you debug your code, remember to **Deploy** the Lambda function and, if you are using the console, to reload the **Test** window before re-testing the bot's behavior.

Customizing bot interactions with users in Lex V2

Learn about the following features that you can use to customize bot interactions with your users by expanding and adjusting their default behavior:

Topics

Analyzing the sentiment of user utterances in conversations with your bot

You can use sentiment analysis to determine the sentiments expressed in a user utterance. With the sentiment information you can manage conversation flow or perform post-call analysis. For example, if the user sentiment is negative you can create a flow to hand over a conversation to a human agent.

Amazon Lex integrates with Amazon Comprehend to detect user sentiment. The response from Amazon Comprehend indicates whether the overall sentiment of the text is positive, neutral, negative, or mixed. The response contains the most likely sentiment for the user utterance and the scores for each of the sentiment categories. The score represents the likelihood that the sentiment was correctly detected.

You enable sentiment analysis for a bot using the console or by using the Amazon Lex API. You enable sentiment analysis on an alias for the bot. On the Amazon Lex console:

1. Choose an alias.
2. In **Details**, choose **Edit**.
3. Choose **Enable sentiment analysis** to sentiment analysis on or off.
4. Choose **Confirm** to save your changes.

If you are using the API, call the CreateBotAlias operation with the `detectSentiment` field set to `true`.

When sentiment analysis is enabled, the response from the RecognizeText and RecognizeUtterance operations return a field called `sentimentResponse` in the `interpretations` structure with other metadata. The `sentimentResponse` field has two fields, `sentiment` and `sentimentScore`, that contain the result of the sentiment analysis. If you are using a Lambda function, the `sentimentResponse` field is included in the event data sent to your function.

The following is an example of the `sentimentResponse` field returned as part of the `RecognizeText` or `RecognizeUtterance` response.

```
sentimentResponse {
    "sentimentScore": {
        "mixed": 0.030585512690246105,
        "positive": 0.94992071056365967,
        "neutral": 0.0141543131828308,
        "negative": 0.00893945890665054
    },
    "sentiment": "POSITIVE"
}
```

Amazon Lex calls Amazon Comprehend on your behalf to determine the sentiment in every utterance processed by the bot. By enabling sentiment analysis, you agree to the service terms and agreements for Amazon Comprehend. For more information about pricing for Amazon Comprehend, see Amazon Comprehend Pricing.

For more information about how Amazon Comprehend sentiment analysis works, see Determine the sentiment in the *Amazon Comprehend Developer Guide*.

Using confidence scores to improve conversation accuracy

There are two steps that Amazon Lex V2 uses to determine what a user says. The first, automatic speech recognition (ASR), creates a transcript of the user's audio utterance. The second, natural language understanding (NLU), determines the meaning of the user's utterance to recognize the user's intent or the value of slots.

By default, Amazon Lex V2 returns the most likely result from ASR and NLU. At times it may be difficult for Amazon Lex V2 to determine the most likely result. In that case, it returns several possible results along with a *confidence score* that indicates how likely the result is correct. A confidence score is a rating that Amazon Lex V2 provides that shows the relative confidence that it has in the result. Confidence scores range from 0.0 to 1.0.

You can use your domain knowledge with the confidence score to help determine the correct interpretation of the ASR or NLU result.

The ASR, or transcription, confidence score is a rating on how confident Amazon Lex V2 is that a particular transcription is correct. The NLU, or intent, confidence score is a rating on how confident Amazon Lex V2 is that the intent specified by the top transcription is correct. Use the confidence score that best fits your application.

Topics

Using intent confidence scores to improve intent selection with Lex V2

When a user makes an utterance, Amazon Lex V2 uses natural language understanding (NLU) to understand the user's request and return the proper intent. By default Amazon Lex V2 returns the most likely intent defined by your bot.

In some cases it may be difficult for Amazon Lex V2 to determine the most likely intent. For example, the user might make an ambiguous utterance, or there might be two intents that are similar. To help determine the proper intent, you can combine your domain knowledge with the *NLU confidence scores* in a list of interpretations. A confidence score is a rating that Amazon Lex V2 provides that shows how confident it is that an intent is the correct intent.

To determine the difference between two intents within an interpretation, you can compare their confidence scores. For example, if one intent has a confidence score of 0.95 and another has a score of 0.65, the first intent is probably correct. However, if one intent has a score of 0.75 and another has a score of 0.72, there is ambiguity between the two intents that you may be able to discriminate using domain knowledge in your application.

You can also use confidence scores to create test applications that determine if changes to an intent's utterances make a difference in the behavior of the bot. For example, you can get the confidence scores for a bot's intents using a set of utterances, then update the intents with new utterances. You can then check the confidence scores to see if there was an improvement.

The confidence scores that Amazon Lex V2 returns are comparative values. You should not rely on them as an absolute score. The values may change based on improvements to Amazon Lex V2.

Amazon Lex V2 returns the most likely intent and up to 4 alternative intents with their associated scores in the `interpretations` structure in each response. The following JSON code shows the `interpretations` structure in the response from the RecognizeText operation:

```
"interpretations": [
   {
      "intent": {
         "confirmationState": "string",
         "name": "string",
         "slots": {
            "string" : {
               "value": {
                  "interpretedValue": "string",
                  "originalValue": "string",
                  "resolvedValues": [ "string" ]
               }
            }
         },
         "state": "string"
      },
      "nluConfidence": number
   }
]
```

AMAZON.FallbackIntent

Amazon Lex V2 returns `AMAZON.FallbackIntent` as the top intent in two situations:

1. If the confidence scores of all possible intents are less than the confidence threshold. You can use the default threshold or you can set your own threshold. If you have the `AMAZON.KendraSearchIntent` configured, Amazon Lex V2 returns it as well in this situation.
2. If the interpretation confidence for `AMAZON.FallbackIntent` is higher than the interpretation confidence of all other intents.

Note that Amazon Lex V2 does not display a confidence score for `AMAZON.FallbackIntent`.

Setting and changing the confidence threshold

The confidence threshold must be a number between 0.00 and 1.00. You can set the threshold for each language in your bot in the following ways:

Using the Amazon Lex V2 console

- To set the threshold when you add a language to your bot with **Add language**, you can insert your desired value in the **Confidence score threshold** panel.
- To update the threshold, you can select **Edit** in the **Language details** panel in a language for your bot. Then insert your desired value in the **Confidence score threshold** panel.

Using API operations

- To set the threshold, set the `nluIntentConfidenceThreshold` parameter of the CreateBotLocale operation.
- To update the confidence threshold, set the `nluIntentConfidenceThreshold` parameter of the UpdateBotLocale operation.

Session Management

To change the intent that Amazon Lex V2 uses in a conversation with the user, you can use the response from your dialog code hook Lambda function, or you can use the session management APIs in your custom application.

Using a Lambda function with your Lex V2 bot

When you use a Lambda function, Amazon Lex V2 calls it with a JSON structure that contains the input to the function. The JSON structure contains a field called `currentIntent` that contains the intent that Amazon Lex V2 has identified as the most likely intent for the user's utterance. The JSON structure also includes an `alternativeIntents` field that contains up to four additional intents that may satisfy the user's intent. Each intent includes a field called `nluIntentConfidenceScore` that contains the confidence score that Amazon Lex V2 assigned to the intent.

To use an alternative intent, you specify it in the `ConfirmIntent` or the `ElicitSlot` dialog action in your Lambda function.

For more information, see Integrating an AWS Lambda function into your Amazon Lex V2 bot.

Using the Session Management API with your Lex V2 bot

To use a different intent from the current intent, use the PutSession operation. For example, if you decide that the first alternative is preferable to the intent that Amazon Lex V2 chose, you can use

the `PutSession` operation to change intents so that the next intent that the user interacts with is the one that you selected.

For more information, see Understanding Amazon Lex V2 bot sessions.

Using voice transcription confidence scores to improve conversations with your Lex V2 bot

When a user makes a voice utterance, Amazon Lex V2 uses automatic speech recognition (ASR) to transcribe the user's request before it is interpreted. By default, Amazon Lex V2 uses the most likely transcription of the audio for interpretation.

In some cases there might be more than one possible transcription of the audio. For example, a user might make an utterance with an ambiguous sound, such as "My name is John" that might be understood as "My name is Juan." In this case, you can use disambiguation techniques or combine your domain knowledge with the *transcription confidence score* to help determine which transcription in a list of transcriptions is the correct one.

Amazon Lex V2 includes the top transcription and up to two alternate transcriptions for user input in the request to your Lambda code hook function. Each transcription contains a confidence score that it is the correct transcription. Each transcription also includes any slot values inferred from the user input.

You can compare the confidence scores of two transcriptions to determine if there is ambiguity between them. For example, if one transcription has a confidence score of 0.95 and the other has a confidence score of 0.65, the first transcription is probably correct and the ambiguity between them is low. If the two transcriptions have confidence scores of 0.75 and 0.72, the ambiguity between them is high. You may be able to discriminate between them using your domain knowledge.

For example, if the inferred slot values in two transcripts with a confidence score of 0.75 and 0.72 are "John" and "Juan", you can query the users in your database for the existence of these names and eliminate one of the transcriptions. If "John" isn't a user in your database and "Juan" is, you can use the dialog code hook to change the inferred slot value for the first name to "Juan."

The confidence scores that Amazon Lex V2 returns are comparative values. Don't rely on them as an absolute score. The values may change based on improvements to Amazon Lex V2.

Audio transcription confidence scores are available only in the English (GB) (en_GB) and English (US) (en_US) languages. Confidence scores are supported only for 8 kHz audio input. Transcription

confidence scores aren't provided for audio input from the test window on the Amazon Lex V2 console because it uses 16 kHz audio input.

> **Note**
>
> Before you can use audio transcription confidence scores with an existing bot, you must first rebuild the bot. Existing versions of a bot don't support transcription confidence scores. You must create a new version of the bot to use them.

You can use confidence scores for multiple conversation design patterns:

- If the highest confidence score falls below a threshold due to a noisy environment or poor signal quality, you can prompt the user with the same question to capture better quality audio.
- If multiple transcriptions have similar confidence scores for slot values, such as "John" and "Juan," you can compare the values with a pre-existing database to eliminate inputs, or you can prompt the user to select one of the two values. For example, "say 1 for John or say 2 for Juan."
- If your business logic requires intent switching based on specific keywords in an alternative transcript with a confidence score close to the top transcript, you can change the intent using your dialog code hook Lambda function or using session management operations. For more information, see Session management.

Amazon Lex V2 sends the following JSON structure with up to three transcriptions for the user's input to your Lambda code hook function:

```
"transcriptions": [
    {
        "transcription": "string",
        "rawTranscription": "string",
        "transcriptionConfidence": "number",
        },
        "resolvedContext": {
            "intent": "string"
        },
        "resolvedSlots": {
            "string": {
                "shape": "List",
                "value": {
```

```
                        "originalValue": "string",
                        "resolvedValues": [
                            "string"
                        ]
                    },
                    "values": [
                        {
                            "shape": "Scalar",
                            "value": {
                                "originalValue": "string",
                                "resolvedValues": [
                                    "string"
                                ]
                            }
                        },
                        {
                            "shape": "Scalar",
                            "value": {
                                "originalValue": "string",
                                "resolvedValues": [
                                    "string"
                                ]
                            }
                        }
                    ]
                }
            }
        }
    ]
```

The JSON structure contains transcription text, the intent that was resolved for the utterance, and values for any slots detected in the utterance. For text user input, the transcriptions contain a single transcript with a confidence score of 1.0.

The contents of the transcripts depend on the turn of the conversation and the recognized intent.

For the first turn, intent elicitation, Amazon Lex V2 determines the top three transcriptions. For the top transcription, it returns the intent and any inferred slot values in the transcription.

On subsequent turns, slot elicitation, the results depend on the inferred intent for each of the transcriptions, as follows.

- If the inferred intent for the top transcript is the same as the previous turn and all other transcripts have the same intent, then
  - All transcripts contain inferred slot values.

- If the inferred intent for the top transcript is the different from the previous turn and all other transcripts have the previous intent, then
  - The top transcript contains the inferred slot values for the new intent.
  - Other transcripts have the previous intent and inferred slot values for the previous intent.

- If the inferred intent for the top transcript is different from the previous turn, one transcript is the same as the previous intent, and one transcript is a different intent, then
  - The top transcript contains the new inferred intent and any inferred slot values in the utterance.
  - The transcript that has the previous inferred intent contains inferred slot values for that intent.
  - The transcript with the different intent has no inferred intent name and no inferred slot values.

- If the inferred intent for the top transcript is the different from the previous turn and all other transcripts have different intents, then
  - The top transcript contains the new inferred intent and any inferred slot values in the utterance.
  - Other transcripts contain no inferred intents and no inferred slot values.

- If the inferred intent for the top two transcripts is the same and different from the previous turn, and the third transcript is a different intent, then
  - The top two transcripts contain the new inferred intent and any inferred slot values in the utterance.
  - The third transcript has no intent name and no resolved slot values.

Session management

To change the intent that Amazon Lex V2 uses in a conversation with the user, use the response from your dialog code hook Lambda function. Or you can use the session management APIs in your custom application.

Using a Lambda function with your Lex V2 bot

When you use a Lambda function, Amazon Lex V2 calls it with a JSON structure that contains the input to the function. The JSON structure contains a field called `transcriptions` that contains the possible transcriptions that Amazon Lex V2 has determined for the utterance. The `transcriptions` field contains one to three possible transcriptions, each with a confidence score.

To use the intent from an alternative transcription, you specify it in the `ConfirmIntent` or the `ElicitSlot` dialog action in your Lambda function. To use a slot value from an alternative transcription, set the value in the `intent` field in your Lambda function response. For more information, see Integrating an AWS Lambda function into your Amazon Lex V2 bot.

Example code using Lambda with Lex V2

The following code example is a Python Lambda function that uses audio transcriptions to improve the conversation experience for the user.

To use the example code, you must have:

- A bot with one language, either English (GB) (en_GB) or English (US) (en_US).
- One intent, OrderBirthStone. Make sure that the **Use a Lambda function for initialization and validation** is selected in the **Code hooks** section of the intent definition.
- The intent should have two slots, "BirthMonth" and "Name," both of type `AMAZON.AlphaNumeric`.
- An alias with the Lambda function defined. For more information, see Creating an AWS Lambda function for your Amazon Lex V2 bot.

```
import time
import os
import logging

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
```

```
# --- Helpers that build all of the responses ---

def elicit_slot(session_attributes, intent_request, slots, slot_to_elicit, message):
    return {
        'sessionState': {
            'dialogAction': {
                'type': 'ElicitSlot',
                'slotToElicit': slot_to_elicit
            },
            'intent': {
                'name': intent_request['sessionState']['intent']['name'],
                'slots': slots,
                'state': 'InProgress'
            },
            'sessionAttributes': session_attributes,
            'originatingRequestId': 'e3ab4d42-fb5f-4cc3-bb78-caaf6fc7cccd'
        },
        'sessionId': intent_request['sessionId'],
        'messages': [message],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes'
 in intent_request else None
    }


def close(intent_request, session_attributes, fulfillment_state, message):
    intent_request['sessionState']['intent']['state'] = fulfillment_state
    return {
        'sessionState': {
            'sessionAttributes': session_attributes,
            'dialogAction': {
                'type': 'Close'
            },
            'intent': intent_request['sessionState']['intent'],
            'originatingRequestId': '3ab4d42-fb5f-4cc3-bb78-caaf6fc7cccd'
        },
        'messages': [message],
        'sessionId': intent_request['sessionId'],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes'
 in intent_request else None
    }


def delegate(intent_request, session_attributes):
```

```
    return {
        'sessionState': {
            'dialogAction': {
                'type': 'Delegate'
            },
            'intent': intent_request['sessionState']['intent'],
            'sessionAttributes': session_attributes,
            'originatingRequestId': 'abc'
        },
        'sessionId': intent_request['sessionId'],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes'
 in intent_request else None
    }


def get_session_attributes(intent_request):
    sessionState = intent_request['sessionState']
    if 'sessionAttributes' in sessionState:
        return sessionState['sessionAttributes']

    return {}


def get_slots(intent_request):
    return intent_request['sessionState']['intent']['slots']


""" --- Functions that control the behavior of the bot --- """


def order_birth_stone(intent_request):
    """
    Performs dialog management and fulfillment for ordering a birth stone.
    Beyond fulfillment, the implementation for this intent demonstrates the following:
    1) Use of N best transcriptions to re prompt user when confidence for top
 transcript is below a threshold
    2) Overrides resolved slot for birth month from a known fixed list if the top
 transcript
    is not accurate.
    """

    transcriptions = intent_request['transcriptions']

    if intent_request['invocationSource'] == 'DialogCodeHook':
```

```
        # Disambiguate if there are multiple transcriptions and the top transcription
        # confidence is below a threshold (0.8 here)
        if len(transcriptions) > 1 and transcriptions[0]['transcriptionConfidence'] <
 0.8:
            if transcriptions[0]['resolvedSlots'] is not {} and 'Name' in
 transcriptions[0]['resolvedSlots'] and \
                    transcriptions[0]['resolvedSlots']['Name'] is not None:
                return prompt_for_name(intent_request)
            elif transcriptions[0]['resolvedSlots'] is not {} and 'BirthMonth' in
 transcriptions[0]['resolvedSlots'] and \
                    transcriptions[0]['resolvedSlots']['BirthMonth'] is not None:
                return validate_month(intent_request)

    return continue_conversation(intent_request)


def prompt_for_name(intent_request):
    """
    If the confidence for the name is not high enough, re prompt the user with the
 recognized names
    so it can be confirmed.
    """
    resolved_names = []
    for transcription in intent_request['transcriptions']:
        if transcription['resolvedSlots'] is not {} and 'Name' in
 transcription['resolvedSlots'] and \
                transcription['resolvedSlots']['Name'] is not None:
            resolved_names.append(transcription['resolvedSlots']['Name']['value']
['originalValue'])
    if len(resolved_names) > 1:
        session_attributes = get_session_attributes(intent_request)
        slots = get_slots(intent_request)
        return elicit_slot(session_attributes, intent_request, slots, 'Name',
                           {'contentType': 'PlainText',
                            'content': 'Sorry, did you say your name is {} ?'.format("
 or ".join(resolved_names))})
    else:
        return continue_conversation(intent_request)


def validate_month(intent_request):
    """
    Validate month from an expected list, if not valid looks for other transcriptions
 and to see if the month
```

```
    recognized there has an expected value. If there is, replace with that and if not
 continue conversation.
    """

    expected_months = ['january', 'february', 'march']
    resolved_months = []
    for transcription in intent_request['transcriptions']:
        if transcription['resolvedSlots'] is not {} and 'BirthMonth' in
 transcription['resolvedSlots'] and \
                transcription['resolvedSlots']['BirthMonth'] is not None:
            resolved_months.append(transcription['resolvedSlots']['BirthMonth']
['value']['originalValue'])

    for resolved_month in resolved_months:
        if resolved_month in expected_months:
            intent_request['sessionState']['intent']['slots']['BirthMonth']
['resolvedValues'] = [resolved_month]
            break

    return continue_conversation(intent_request)


def continue_conversation(event):
    session_attributes = get_session_attributes(event)

    if event["invocationSource"] == "DialogCodeHook":
        return delegate(event, session_attributes)


# --- Intents ---


def dispatch(intent_request):
    """
    Called when the user specifies an intent for this bot.
    """

    logger.debug('dispatch sessionId={},
 intentName={}'.format(intent_request['sessionId'],

 intent_request['sessionState']['intent']['name']))

    intent_name = intent_request['sessionState']['intent']['name']
```

```
    # Dispatch to your bot's intent handlers
    if intent_name == 'OrderBirthStone':
        return order_birth_stone(intent_request)

    raise Exception('Intent with name ' + intent_name + ' not supported')


# --- Main handler ---


def lambda_handler(event, context):
    """
    Route the incoming request based on intent.
    The JSON body of the request is provided in the event slot.

    """
    # By default, treat the user request as coming from the America/New_York time
 zone.
    os.environ['TZ'] = 'America/New_York'
    time.tzset()
    logger.debug('event={}'.format(event))

    return dispatch(event)
```

Using the session management API to choose a different intent or slot value

To use a different intent from the current intent, use the PutSession operation. For example, if you decide that the first alternative is preferable to the intent that Amazon Lex V2 chose, you can use the `PutSession` operation to change intents. That way the next intent that the user interacts with will be the one that you selected.

You can also use the `PutSession` operation to change the slot value in the `intent` structure to use a value from an alternative transcription.

For more information, see Understanding Amazon Lex V2 bot sessions.

Customizing speech transcriptions for use with your Lex V2 bot

The default behavior of your bot may sometimes result in inaccurate speech transcriptions. The following features are available to help your bot recognize words or names that are less common or easily confused.

Topics

Configuring speech recognition model preferences

Amazon Lex V2 provides different speech recognition models that you can choose from to optimize the accuracy and performance of your bot's speech recognition capabilities. You can configure speech model preferences to select the most appropriate model for your use case.

Speech recognition model types

Amazon Lex V2 supports the following speech recognition models:

Standard model

The standard speech recognition model provides reliable speech recognition performance for general use cases. This model offers consistent accuracy across a wide range of audio conditions and is suitable for most conversational AI applications.

Neural model

The neural speech recognition model provides enhanced accuracy and better handling of natural speech patterns, accents, and background noise. This model uses advanced neural network architectures to improve recognition performance, especially in challenging audio environments.

Deepgram

Deepgram provides a public speech-to-text (STT) API for users that create an account and an API key. See https://deepgram.com/ for information about their public offerings.

Configuring speech model preferences

You can configure speech model preferences when creating or updating a bot locale. The speech model preference setting determines which recognition model Amazon Lex V2 uses to process audio input for your bot.

To configure speech model preferences:

1. In the Amazon Lex V2 console, navigate to your bot and select the locale you want to configure.
2. In the bot locale settings, locate the **Speech recognition settings** section.
3. For **Speech model preference**, choose one of the following options:
   - **Standard** - Use the standard speech recognition model for reliable performance across general use cases.
   - **Neural** - Use the neural speech recognition model for enhanced accuracy and better handling of natural speech patterns.
   - **Deepgram** - Use Deepgram's Listen API for speech recognition. For setup instructions, see the section called “Setting up Deepgram speech model preference”.
4. Save your changes to apply the speech model preference to your bot locale.

> **Note**
>
> If you don't specify a speech model preference, Amazon Lex V2 uses the standard model by default.

Choosing the right speech model

Consider the following factors when choosing a speech recognition model for your bot:

- **Audio quality** - If your bot will process audio with background noise, varying audio quality, or challenging acoustic conditions, the neural model may provide better accuracy.
- **Speaker diversity** - If your bot will interact with users who have diverse accents or speech patterns, the neural model's enhanced natural language processing capabilities may improve recognition performance.
- **Performance requirements** - The standard model provides consistent performance and may be sufficient for applications with controlled audio environments and clear speech input.

You can test both models with your specific use case to determine which provides the best balance of accuracy and performance for your application.

Setting up Deepgram speech model preference

Deepgram is a third-party speech recognition service that provides advanced AI-powered speech-to-text capabilities with support for real-time and batch processing. Deepgram offers enhanced accuracy across various audio conditions, multiple languages, and specialized models for different use cases. For more information about Deepgram's offerings, see https://deepgram.com/. To use Deepgram as your speech recognition model preference, you need to complete a one-time setup process to configure your Deepgram API key and store it securely in AWS Secrets Manager.

> **Important**
>
> Deepgram is a third-party service and may not comply with certain regulatory frameworks such as GDPR, FedRAMP, or other compliance standards that AWS services adhere to. Review Deepgram's compliance documentation and your organization's requirements before using this integration.

Regional endpoint selection

Amazon Lex V2 automatically selects the appropriate Deepgram API endpoint based on your AWS region to optimize performance and data locality:

- **EU regions:** For bots deployed in AWS regions with the `eu-` prefix (such as `eu-west-1`, `eu-west-2`, and `eu-central-1`), Amazon Lex V2 uses the Deepgram EU endpoint (`api.eu.deepgram.com`).
- **All other regions:** For bots deployed in all other AWS regions, Amazon Lex V2 uses the global Deepgram endpoint (`api.deepgram.com`).

This endpoint selection is automatic and currently cannot be customized. The same Deepgram API key works with both endpoints. Amazon Lex V2 does not support the `apiTokenRegion` parameter that Amazon Connect provides for custom endpoint configuration. For more information about Amazon Connect's endpoint configuration options, see Endpoints and Regions for third-party STT providers in the Amazon Connect Administrator Guide.

Creating a Deepgram API key

Before you can use Deepgram with Amazon Lex V2, you need to obtain an API key from Deepgram.

To create a Deepgram API key:

1. Log into the Deepgram console at https://console.deepgram.com/.
2. In the left navigation pane, choose **API Keys**.
3. Choose **Create a New API Key**.
4. Follow the instructions to create the API key and copy it for use in the next section.

> **Important**
>
> Store your API key securely. You'll need it to configure AWS Secrets Manager in the next section.

Storing the API key in AWS Secrets Manager

You must store your Deepgram API key in AWS Secrets Manager for Amazon Lex V2 to access it securely. The secret must contain a single key-value pair with `apiToken` as the key and your Deepgram API key as the value.

> **Important**
>
> You must create a symmetric KMS key to use with the secret. The default AWS managed KMS key will not work with Amazon Lex V2.

To store your Deepgram API key in Secrets Manager:

1. Open the AWS Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. For **Secret type**, choose **Other type of secret**.
4. Configure the secret using one of the following methods:
   - **Key/value pairs method:** Under **Key/value pairs**, add a single key-value pair with `apiToken` as the key and your Deepgram API key as the value.

- **Plaintext method:** Under **Plaintext**, enter a JSON object with the following structure:

```
{
  "apiToken": "your-deepgram-api-key-here"
}
```

5. Choose **Next**.
6. Enter a name for your secret and choose **Next**.
7. (Optional) Configure secret rotation if needed, then choose **Next**.
8. Review your secret configuration and choose **Store**.
9. After the secret is created, navigate to your secret and copy the ARN. You'll need this ARN when configuring your bot.

Configuring resource policy for Secrets Manager

To allow Amazon Lex V2 to retrieve your Deepgram API key, you need to attach a resource policy to your secret.

The following is a sample resource policy that allows Amazon Lex V2 to retrieve the secret:

```
{
  "Version": "2012-10-17"       ,
  "Statement": [
    {
      "Sid": "LexTrust",
      "Effect": "Allow",
      "Principal": {
        "Service": "lex.amazonaws.com"
      },
      "Action": "secretsmanager:GetSecretValue",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "<YOUR_ACCOUNT_ID>"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:lex:us-east-1:<YOUR_ACCOUNT_ID>:bot-alias/*/*"
```

```
        }
      }
    }
  ]
}
```

Replace <YOUR_ACCOUNT_ID> with your actual AWS account ID and adjust the region in the ARN pattern as needed for your deployment.

Configuring your bot to use Deepgram

After setting up your Deepgram API key in Secrets Manager, you can configure your Amazon Lex V2 bot to use Deepgram for speech recognition.

To configure Deepgram for your bot:

1. In the Amazon Lex V2 console, navigate to your bot and select the locale you want to configure.
2. For **Speech model preference**, choose **Deepgram**.
3. Additional fields will appear for Deepgram configuration:
   - **Model ID** (optional) - Specify a Deepgram model ID if you want to use a specific model. For available models, see the Deepgram model documentation. If left blank, the API's default model will be used.
   - **Secret ARN** (required) - Enter the ARN of the secret you created in AWS Secrets Manager that contains your Deepgram API key.
4. Save your changes to apply the Deepgram speech model preference to your bot locale.

Your bot is now configured to use Deepgram for speech recognition. Test your bot to ensure that speech recognition is working as expected with the Deepgram integration.

Troubleshooting Deepgram integration

If you encounter issues with your Deepgram integration, check the following:

- **API key validity:** Ensure your Deepgram API key is valid and has not expired.
- **Secret configuration:** Verify that your secret in AWS Secrets Manager contains the correct key name (`apiToken`) and API key value.
- **Resource policy:** Confirm that the resource policy on your secret allows Amazon Lex V2 to access it with the correct account ID and ARN pattern.

- **KMS key:** Ensure you're using a customer-managed symmetric KMS key, not the default AWS managed key.
- **Model ID:** If you specified a model ID, verify that it's a valid Deepgram model identifier.

For additional support, consult the Amazon Lex V2 CloudWatch logs or contact AWS Support.

Improving speech recognition with a custom vocabulary

You can give Amazon Lex V2 more information about how to process audio conversations with a bot by creating a custom vocabulary in a specific language. A *custom vocabulary* is a list of specific phrases that you want Amazon Lex V2 to recognize in the audio input. These are generally proper nouns or domain-specific words that Amazon Lex V2 doesn't recognize.

For example, suppose that you have a tech support bot. You can add "backup" to a custom vocabulary to help the bot transcribe the audio correctly as "backup," even when the audio sounds like "pack up." A custom vocabulary can also help recognize rare words in the audio such as "solvency" for financial services or proper nouns such as "Cognito" or "Monitron."

Custom vocabulary basics

- A custom vocabulary works on the transcription of audio input to a bot. You must provide sample utterances to recognize an intent or slot value.
- A custom vocabulary is unique to a specific language. You must configure custom vocabularies independently for each language. Custom vocabularies are supported for the following languages: English (UK), English (US), Korean (Korea), Italian (Italy), German (Germany), French (Canada), French (France), Spanish (Spain), Spanish (US), Japanese (Japan), Mandarin (PRC), Swedish (Sweden), Gulf Arabic (United Arab Emirates), Hindi (India), Dutch (The Netherlands), Norwegian (Norway), Polish (Poland), Finnish (Finland), Portuguese (Portugal), Portuguese (Brazil), Catalan (Spain).
- Custom vocabularies are available with contact center integrations supported by Amazon Lex V2. The test window in the Amazon Lex V2 console supports custom vocabularies for all Amazon Lex V2 bots built on or after July 31, 2022. If you experience issues with custom vocabularies in the test window, rebuild the bot and try again.

Amazon Lex V2 uses custom vocabularies to elicit both intents and slots. The same custom vocabulary file is used for intents and slots. You can selectively turn off the custom vocabulary capability for a slot when you add a slot type.

Eliciting an intent – You can create a custom vocabulary for eliciting an intent. These phrases are used to transcription when your bot is determining the user's intent. For example, if you configured the phrase "backup" in your custom vocabulary, Amazon Lex V2 transcribes the user input to "can you please backup my photos?"—even when the audio sounds like "can you please pack up my photos." You can specify the degree of boosting for each phrase by configuring a weight of 0, 1, 2, or 3. You can also specify an alternate representation for the phrase in the final speech to text output by adding a `displayAs` field.

The custom vocabulary phrases used for improving transcription during intent elicitation don't affect transcriptions while eliciting slots. For more information about creating a custom vocabulary for eliciting intents, see Creating a custom vocabulary for eliciting intents and slots.

Eliciting custom slots – You can use a custom vocabulary to improve slot recognition for audio conversations. To improve your Amazon Lex V2 bot's ability to recognize slot values, create a custom slot and add the slot values to the custom slot, then choose **Use slot values as custom vocabulary**. Examples of slot values include product names, catalogs, or proper nouns. You shouldn't use common words or phrases such as "yes" and "no" in custom vocabularies.

After the slot values are added, these values are used for improving slot recognition when the bot is expecting input for the custom slot. These values aren't used for transcription when eliciting an intent. For more information, see Adding slot types.

Best practices for creating a custom vocabulary

Eliciting an intent

- Custom vocabularies work best when used to target specific words or phrases. Only add words to a custom vocabulary if they are not readily recognized by Amazon Lex V2.
- Decide how much weight to give a word based on how often the word isn't recognized in the transcription and how rare the word is in the input. Difficult to pronounce words require a higher weight.
- Use a representative test set to determine if a weight is appropriate. You can collect an audio test set by turning on audio logging in conversation logs.
- Avoid using short words like "on," "it," "to," "yes," "no" in a custom vocabulary.

Eliciting a custom slot

- Add the values to the custom slot type that you expect to be recognized. Add all the possible slot values for the custom slot type, no matter how common or rare the slot value is.
- Enable the option only when the custom slot type contains a list of catalog values or entities such as product names or mutual funds.
- Disable the option if the slot type is used to capture generic phrases such as "yes," "no," "I don't know," "maybe," or generic words such as "one," "two," "three."
- Limit the number of slot values and synonyms to 500 or less for best performance.

Enter acronyms or other words whose letters should be pronounced individually as single letters separated by a period and a space. Don't use individual letters unless they are part of a phrase, such as "J. P. Morgan" or "A. W. S." You can use upper- or lower-case letters to define an acronym.

Creating a custom vocabulary for eliciting intents and slots

You can use the Amazon Lex V2 console to create and manage a custom vocabulary, or you can use Amazon Lex V2 API operations. There are 2 ways of creating a custom vocabulary through the console:

Console

Import custom vocabulary in the console:

1. Open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home
2. From the list of bots, choose the bot which you want to add the custom vocabulary.
3. On the bot detail page, from the **Add languages** section, choose **View languages**.
4. From the list of languages, choose the language to which you want to add the custom vocabulary.

Create a new custom vocabulary directly through the console:

1. Click on **Create** in the **Custom Vocabulary** section of the language details page. This will open an editing window with no custom vocabulary present.
2. Add inputs for phrase, DisplayAs, and weight as required. You can further make inline edits to added items by updating their fields or deleting them from the list.
3. Click on **Save**. Please note: the new custom vocabulary is only saved in your bot after you click on **Save**.

4. You can continue making inline edits in this page and click **Save** when are done.
5. This page also allows you import, export, and delete a custom vocabulary file from the drop-down menu on the top right.

API

Use the `ListCustomVocabularyItems` API to view the custom vocabulary entries:

1. Use the `ListCustomVocabularyItems` operation to view the custom vocabulary entries. The request body will look like this:

```
{
   "maxResults": number,
   "nextToken": "string"
}
```

2. Please note that `maxResults` and `nextToken` are optional fields for the request body.
3. The response from the `ListCustomVocabularyItems` operation looks like this:

```
{
   "botId": "string",
   "botVersion": "string",
   "localeId": "string",
   "customVocabularyItems": [
        {
            "itemId": "string",
            "phrase": "string",
            "weight": number,
            "displayAs": "string"
        }
   ]
}
```

Use the `BatchCreateCustomVocabularyItem` API to create new custom vocabulary entries:

1. If your bot's locale does not have a custom vocabulary created yet, please follow the steps to use the StartImport to create a custom vocabulary.

2. After the custom vocabulary has been created, use the `BatchCreateCustomVocabularyItem` operation to create new custom vocabulary entries. The request body will look like this:

```
{
   "customVocabularyItemList": [
        {
            "phrase": "string",
            "weight": number,
            "displayAs": "string"
        }
   ]
}
```

3. Please note that `weight` and `displayAs` are optional fields for the request body.
4. The response from the `BatchCreateCustomVocabularyItem` will look like this:

```
{
    "botId": "string",
    "botVersion": "string",
    "localeId": "string",
    "errors": [
        {
            "itemId": "string",
            "errorMessage": "string",
            "errorCode": "string"
        }
    ],
    "resources": [
        {
            "itemId": "string",
            "phrase": "string",
            "weight": number,
            "displayAs": "string"
        }
    ]
}
```

5. As this is a batch operation, the request will not fail if one of the items fails to create. The errors list will contain information about why the operation failed for that specific entry. The resources list will contain all of the entries that were successfully created.
6. For `BatchCreateCustomVocabularyItem`, you can expect see these types of errors:

- RESOURCE_DOES_NOT_EXIST: The custom vocabulary does not exist. Follow the steps for creating a custom vocabulary before calling this operation.
- DUPLICATE_INPUT: The list of inputs contains duplicate phrases.
- RESOURCE_ALREADY_EXISTS: The given phrase for the entry already exists in your custom vocabulary.
- INTERNAL_SERVER_FAILURE: There was an error in the backend while processing your request. This may indicate a service outage or another issue.

Use the `BatchDeleteCustomVocabularyItem` API to delete existing custom vocabulary entries:

1. If your bot's locale does not have a custom vocabulary created yet, please follow the steps for Use the StartImport to create a custom vocabulary to create one.
2. After the custom vocabulary has been created, use the `BatchDeleteCustomVocabularyItem` operation to delete existing custom vocabulary entries. The request body will look like this:

```
{
   "customVocabularyItemList": [
        {
            "itemId": "string"
        }
   ]
}
```

3. The response from the `BatchDeleteCustomVocabularyItem` will look like this:

```
{
    "botId": "string",
    "botVersion": "string",
    "localeId": "string",
    "errors": [
        {
            "itemId": "string",
            "errorMessage": "string",
            "errorCode": "string"
        }
    ],
```

```
    "resources": [
        {
            "itemId": "string",
            "phrase": "string",
            "weight": number,
            "displayAs": "string"
        }
    ]
}
```

4. As this is a batch operation, the request will not fail if one of the items fails to delete. The errors list will contain information about why the operation failed for that specific entry. The resources list will contain all of the entries that were successfully deleted.
5. For `BatchDeleteCustomVocabularyItem`, you can expect see these types of errors:
   - `RESOURCE_DOES_NOT_EXIST`: The custom vocabulary entry you are trying to delete does not exist.
   - `INTERNAL_SERVER_FAILURE`: There was an error in the backend while processing your request. This may indicate a service outage or another issue.

Use the `BatchUpdateCustomVocabularyItem` API to update existing custom vocabulary entries:

1. If your bot's locale does not have a custom vocabulary created yet, please follow the steps for Use the StartImport to create a custom vocabulary to create a custom vocabulary.
2. After the custom vocabulary has been created, use the `BatchUpdateCustomVocabularyItem` operation to update existing custom vocabulary entries. The request body will look like this:

```
{
   "customVocabularyItemList": [
        {
            "itemId": "string",
            "phrase": "string",
            "weight": number,
            "displayAs": "string"
        }
   ]
}
```

3. Please note that `weight` and `displayAs` are optional fields for the request body.
4. The response from the `BatchUpdateCustomVocabularyItem` will look like this:

```
{
    "botId": "string",
    "botVersion": "string",
    "localeId": "string",
    "errors": [
        {
            "itemId": "string",
            "errorMessage": "string",
            "errorCode": "string"
        }
    ],
    "resources": [
        {
            "itemId": "string",
            "phrase": "string",
            "weight": number,
            "displayAs": "string"
        }
    ]
}
```

5. As this is a batch operation, the request will not fail if one of the items fails to delete. The errors list will contain information about why the operation failed for that specific entry. The resources list will contain all of the entries that were successfully updated.
6. For `BatchUpdateCustomVocabularyItem`, you can expect see these types of errors:

   - `RESOURCE_DOES_NOT_EXIST`: The custom vocabulary entry you are trying to update does not exist.
   - `DUPLICATE_INPUT`: The list of inputs contains duplicate itemIds.
   - `RESOURCE_ALREADY_EXISTS`: The given phrase for the entry already exists in your custom vocabulary.
   - `INTERNAL_SERVER_FAILURE`: There was an error in the backend while processing your request. This may indicate a service outage or another issue.

Creating a custom vocabulary file

A custom vocabulary file is a tab-separated list of values that contain the phrase to recognize, a weight to give the boost, and a `displayAs` field which will replace the phrase in the speech transcript. Phrases with a higher boost value are more likely to be used when they appear in the audio input.

The custom vocabulary file must be named **`CustomVocabulary.tsv`**, and must be compressed in a zip file before it can be imported. The zip file must be less than 300 MB in size. The maximum number of phrases in a custom vocabulary is 500.

- **phrase** 1–4 words that should be recognized. Separate words in the phrase with spaces. You can't have duplicate phrases in the file. The phrase field is required.
- **weight** – The degree to which the phrase recognition is boosted. The value is an integer 0, 1, 2, or 3. If you don't specify a weight, the default value is 1. Decide on the weight based on how often the word isn't recognized in the transcription and on how rare the word is in the input. The weight 0 means that no boosting will be applied and the entry will only be used for performing replacements using the `displayAs` field.
- **displayAs** – Defines how you want your phrase to look in your transcription output. This is an optional field in the custom vocabulary.

The custom vocabulary file must contain a header row with the headers "phrase," "weight," and "displayAs". The headers can be in any order, but must follow the above nomenclature.

The following example is a custom vocabulary file. The required tab character to separate the phrase, the weight, and the displayAs is represented by the text "[TAB]". If you use this example, replace the text with a tab character.

```
phrase[TAB]weight[TAB]displayAs
Newcastle[TAB]2
Hobart[TAB]2[TAB]Hobart, Australia
U. Dub[TAB]1[TAB]University of Washington, Seattle
W. S. U.[TAB]3
Issaquah
Kennewick
```

Configuring voice activity detection sensitivity

Voice Activity Detection (VAD) is a technology that determines when speech is present in an audio signal. Amazon Lex V2 uses VAD to optimize speech recognition accuracy by distinguishing between actual speech and background noise. You can configure the VAD sensitivity level to improve your bot's performance in different acoustic environments.

Understanding VAD sensitivity levels

Amazon Lex V2 provides three VAD sensitivity levels that you can configure for your bot locale:

Default

The standard sensitivity level suitable for most environments with typical background noise levels. This is the recommended setting for general use cases.

HighNoiseTolerance

Increased tolerance for moderate background noise. Use this setting when your bot operates in environments with consistent but moderate noise levels, such as busy offices or retail environments.

MaximumNoiseTolerance

Maximum tolerance for high levels of background noise. Use this setting for very noisy environments such as call centers, manufacturing floors, or outdoor locations with significant ambient noise.

> **Note**
>
> Higher noise tolerance levels may result in the system being more permissive about what it considers speech, which could potentially lead to false negatives in very quiet environments. Choose the sensitivity level that best matches your expected acoustic environment.

Configuring VAD sensitivity

You can configure VAD sensitivity when creating or updating a bot locale using the Amazon Lex V2 console, AWS CLI, or SDKs.

Using the Amazon Lex V2 console

To configure VAD sensitivity in the console

1. Open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/.
2. Choose your bot from the list.
3. In the left navigation pane, choose **Bot languages**.
4. Choose the language you want to configure, or choose **Add language** to add a new one.
5. In the **Speech detection sensitivity** section, choose one of the following options:
   - **Default** - Standard sensitivity for typical environments
   - **High noise tolerance** - For moderately noisy environments
   - **Maximum noise tolerance** - For very noisy environments
6. Choose **Save** to apply the changes.

Using the AWS CLI or SDKs

You can set the VAD sensitivity using the `speechDetectionSensitivity` parameter in the following API operations:

- `CreateBotLocale` - Set VAD sensitivity when creating a new bot locale
- `UpdateBotLocale` - Modify VAD sensitivity for an existing bot locale
- `DescribeBotLocale` - View the current VAD sensitivity setting

Example Setting VAD sensitivity with AWS CLI

```
aws lexv2-models create-bot-locale \
    --bot-id "AIDACKCEVSQ6C2EXAMPLE" \
    --bot-version "DRAFT" \
    --locale-id "en_US" \
    --nlu-intent-confidence-threshold 0.40 \
    --speech-detection-sensitivity "HighNoiseTolerance"
```

Best practices for VAD configuration

- **Test in your target environment** - Configure VAD sensitivity based on the actual acoustic conditions where your bot will be deployed.
- **Start with Default** - Begin with the Default setting and adjust based on performance testing and user feedback.
- **Monitor performance** - Use Amazon Lex V2 analytics and conversation logs to monitor speech recognition accuracy and adjust VAD sensitivity as needed.
- **Consider use case** - Higher sensitivity levels are beneficial for noisy environments but may not be necessary for controlled environments like customer service centers with headsets.

Improving recognition of slot values with runtime hints in the conversation

With *runtime hints* you can give Amazon Lex V2 a set of slot values based on context to get better recognition in audio conversations and improved slot resolutions. You can use runtime hints to provide a list of phrases at runtime that become candidates for the resolution of a slot value.

For example, if a user interacting with a flight reservation bot frequently travels to San Francisco, Jakarta, Seoul, and Moscow you can configure runtime hints with a list of these four cities when eliciting for the destination to improve recognition for frequently travelled cities.

Runtime hints are only available in the English (US) and English (UK) languages. They can be used with the following slot types:

- Custom slot types
- AMAZON.City
- AMAZON.Country
- AMAZON.FirstName
- AMAZON.LastName
- AMAZON.State
- AMAZON.StreetName

Runtime hints basics

- Runtime hints are used only when eliciting a slot value from a user.
- When you use runtime hints, the values of the hints are preferred over similar values. For example, for a food ordering bot, you can set a list of menu items as runtime hints while eliciting for food items in a custom slot to prefer “fillet” over similar sounding “fella”.
- If the user input is different from the values provided in runtime hints, the original user input will be used for the slot.
- For custom slot types, values provided as runtime hints will be used for resolution of the slot even if they are not part of the custom slot during bot creation.
- Runtime hints are supported only for 8 kHz audio input. They are available with contact center integrations supported by Amazon Lex V2. Runtime hints aren't provided for audio input from the test window on the Amazon Lex V2 console because it uses 16 kHz audio input.

> **Note**
>
> Before you can use runtime hints with an existing bot, you must first rebuild the bot. Existing versions of a bot don't support runtime hints. You must create a new version of the bot to use them.

You can send runtime hints to Amazon Lex V2 using the PutSession, RecognizeText, RecognizeUtterance, or StartConversation operation. You can also add runtime hints using a Lambda function.

You can send runtime hints at the beginning of a conversation to configure the hints for each slot used in the bot, or you can send hints as part of the session state during a conversation. The `runtimeHints` attribute maps a slot to the hints for that slot.

Once you send a runtime hint to Amazon Lex V2, they persist for every turn of the conversation until the session ends. If you send a null `runtimeHints` structure, the existing hints are used. You can modify the hints by:

- Sending a new `runtimeHints` structure to the bot. The contents of the new structure replace the existing ones.
- Sending an empty `runtimeHints` structure to the bot. This clears the runtime hints for the bot.

Adding slot values in context

Add context for your bot by providing expected slot values as runtime hints when your application has information about the user's next likely utterance. Add a Lambda dialog code hook to your bot (see Integrating an AWS Lambda function into your Amazon Lex V2 bot for more information) and use the **proposedNextState** field in the AWS Lambda input event format for Lex V2 to determine the runtime hints that you should include to improve the conversation with the user.

For example, in a banking app you can generate a list of account nicknames for a specific user, and then use the list when eliciting the account that the user wants to access.

Send runtime hints at the start of the conversation when you have context to help your bot interpret user input. For example, if you have the user's phone number, you can use this information to look up the user so that you can use the `PutSession` or `StartConversation` operation to pass first and last name hints to the bot if you are eliciting for the user's name to validate their credentials.

During a conversation, you might gather information from one slot value that can help with another slot value. For example, in a car care app when you have the user's account number you can do a look up to find the cars that the customer owns and pass them along as hints to another slot.

Enter acronyms, or other words whose letters should be pronounced individually, as single letters separated by a period and a space. Don't use individual letters unless they are part of a phrase, such as "J. P. Morgan" or “A.W.S”. You can use upper- or lower-case letters to define an acronym.

Adding hints to a slot

To add runtime hints to a slot, you use the `runtimeHints` structure that is part of the `sessionState` structure. The following is an example of the `runtimeHints` structure. It provides hints for two slots, "FirstName" and "LastName" for the "MakeAppointment" intent.

```
{
    "sessionState": {
        "intent": {},
        "activeContexts": [],
        "dialogAction": {},
        "originatingRequestId": {},
        "sessionAttributes": {},
        "runtimeHints": {
            "slotHints": {
```

```
                "MakeAppointment": {
                    "FirstName": {
                        "runtimeHintValues": [
                            {
                                "phrase": "John"
                            },
                            {
                                "phrase": "Mary"
                            }
                        ]
                    },
                    "LastName": {
                        "runtimeHintValues": [
                            {
                                "phrase": "Stiles"
                            },
                            {
                                "phrase": "Major"
                            }
                        ]
                    }
                }
            }
        }
    }
}
```

You can also use a Lambda function to add runtime hints during a conversation. To add runtime hints, you add the `runtimeHints` structure to the session state of the response that your Lambda function sends to Amazon Lex V2. For more information, see AWS Lambda response format for Lex V2.

You must specify a valid `intentName` and `slotName` in the request, otherwise Amazon Lex V2 returns a runtime error.

Capturing slot values with spelling styles during the conversation

Amazon Lex V2 provides built-in slots to capture user-specific information such as first name, last name, email address or alphanumeric identifiers. For example, you can use the `AMAZON.LastName` slot to capture surnames such as "Jackson" or "Garcia." However, Amazon Lex V2 may get confused with surnames that are difficult to pronounce or that are not common in a locale, such as "Xiulan." To capture such names, you can ask the user to provide input in *spell by letter* or *spell by word* style.

Amazon Lex V2 provides three *slot elicitation styles* for you to use. When you set a slot elicitation style, it changes the way Amazon Lex V2 interprets the input from the user.

Spell by letter – With this style, you can instruct the bot to listen for spellings instead of the whole phrase. For example, to capture a last name such as "Xiulan," you can tell the user to spell out their last name one letter at a time. The bot will capture the spelling and resolve the letters to a word. For example, if the user says "x i u l a n," the bot captures the last name as "xiulan."

Spell by word – In voice conversations, especially using the telephone, there are a few letters, such as "t," "b," "p", that sound similar. When capturing alphanumeric values or spelling names results in an incorrect value, you can prompt the user to provide an identifying word along with the letter. For example, if the voice response to a request for a booking ID is "abp123," your bot might instead recognize the phrase "ab**b**123" instead. If this is an incorrect value, you can ask the user to provide the input as "a as in alpha b as in boy p as in peter one two three." The bot will resolve the input to "abp123."

When using spell by word, you can use the following formats:

For English (US), English (UK), and English (Australia):

- "as in" (a as in apple)
- "for" (a for apple)
- "like" (a like apple)

For German (de-DE):

- "wie" (p wie pizza)
- "für" (p für pizza)

For French (fr-FR):

- "comme" (p comme paris)
- "pour" (p pour paris)

For Spanish (es-419, es-US, es-ES):

- "como" (u como union)
- "de" (u de union)

Default – This is the natural style of slot capture using word pronunciation. For example, it can capture names such as "John Stiles" naturally. If a slot elicitation style isn't specified, the bot uses the default style. For the `AMAZON.AlphaNumeric` and `AMAZON.UKPostal` code slot types, the default style supports spell by letter input.

If the name "Xiulan" is spoken using a mix of letters and words , such as "x as in x-ray i u l as in lion a n" the slot elicitation style must be set to spell-by-word style. The spell-by-letter style won't recognize it.

You should create a voice interface that captures slot values with natural conversational style for a better experience. For inputs that are not correctly captured using the natural style, you can re-prompt the user and set the slot elicitation style to spell-by-letter or spell-by-word.

You can use spell-by-word and spell-by-letter styles for the following slot types in the English (US), English (UK), English (Australia), German (de-DE), French (fr-FR), and Spanish (es-419, es-US, es-ES) languages:

- AMAZON.AlphaNumeric
- AMAZON.EmailAddress
- AMAZON.FirstName
- AMAZON.LastName
- AMAZON.UKPostalCode (English (US), English (UK), and English (Australia) only)
- Custom Slot Types

Enabling spelling

You enable spell-by-letter and spell-by-word at runtime when you are eliciting slots from the user. You can set the spelling style with the PutSession, RecognizeText, RecognizeUtterance, or StartConversation operation. You can also enable spell-by-letter and spell-by-word using a Lambda function.

You set the spelling style using the `dialogAction` field of the `sessionState` field in the request of one of the aforementioned API operations or when configuring the Lambda response (see AWS Lambda response format for Lex V2 for more information). You can only set the style when the dialog action type is `ElicitSlot` and when the slot to elicit is one of the supported slot types.

The following JSON code shows the `dialogAction` field set to use the spell-by-word style:

```
"dialogAction": {
    "slotElicitationStyle": "SpellByWord",
    "slotToElicit": "BookingId",
    "type": "ElicitSlot"
}
```

The `slotElicitationStyle` field can be set to `SpellByLetter`, `SpellByWord`, or `Default`. If you don't specify a value, then the value is set to `Default`.

> **Note**
>
> You can't enable spell-by-letter or spell-by-word elicitation styles through the console.

Example code using Lambda and Lex V2

Changing the spelling style is usually performed if the first attempt to resolve a slot value that didn't work. The following code example is a Python Lambda function that uses the spell-by-word style on the second attempt to resolve a slot.

To use the example code, you must have:

- A bot with one language, English (GB) (en_GB).
- One intent, "CheckAccount" with one sample utterance, "I would like to check my account". Make sure that **Use a Lambda function for initialization and validation** is selected in the **Code hooks** section of the intent definition.
- The intent should have one slot, "PostalCode", of the `AMAZON.UKPostalCode` built-in type.
- An alias with the Lambda function defined. For more information, see Creating an AWS Lambda function for your Amazon Lex V2 bot.

```
import json
import time
import os
import logging

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
```

```
# --- Helpers that build all of the responses ---

def get_slots(intent_request):
    return intent_request['sessionState']['intent']['slots']

def get_session_attributes(intent_request):
    sessionState = intent_request['sessionState']
    if 'sessionAttributes' in sessionState:
        return sessionState['sessionAttributes']
    return {}

def get_slot(intent_request, slotName):
    slots = get_slots(intent_request)
    if slots is not None and slotName in slots and slots[slotName] is not None:
        logger.debug('resolvedValue={}'.format(slots[slotName]['value']
['resolvedValues']))
        return slots[slotName]['value']['resolvedValues']
    else:
        return None

def elicit_slot(session_attributes, intent_request, slots, slot_to_elicit,
 slot_elicitation_style, message):
    return {'sessionState': {'dialogAction': {'type': 'ElicitSlot',
                                              'slotToElicit': slot_to_elicit,
                                              'slotElicitationStyle':
 slot_elicitation_style
                                              },
                             'intent': {'name': intent_request['sessionState']
['intent']['name'],
                                        'slots': slots,
                                        'state': 'InProgress'
                                        },
                             'sessionAttributes': session_attributes,
                             'originatingRequestId': 'REQUESTID'
                             },
            'sessionId': intent_request['sessionId'],
            'messages': [ message ],
            'requestAttributes': intent_request['requestAttributes']
            if 'requestAttributes' in intent_request else None
            }

def build_validation_result(isvalid, violated_slot, slot_elicitation_style,
 message_content):
    return {'isValid': isvalid,
```

```
            'violatedSlot': violated_slot,
            'slotElicitationStyle': slot_elicitation_style,
            'message': {'contentType': 'PlainText',
            'content': message_content}
            }

def GetItemInDatabase(postal_code):
    """
    Perform database check for transcribed postal code. This is a no-op
    check that shows that postal_code can't be found in the database.
    """
    return None

def validate_postal_code(intent_request):

    postal_code = get_slot(intent_request, 'PostalCode')

    if GetItemInDatabase(postal_code) is None:
        return build_validation_result(
            False,
            'PostalCode',
            'SpellByWord',
            "Sorry, I can't find your information. " +
            "To try again, spell out your postal " +
            "code using words, like a as in apple."
        )
    return {'isValid': True}

def check_account(intent_request):
    """
    Performs dialog management and fulfillment for checking an account
    with a postal code. Besides fulfillment, the implementation for this
    intent demonstrates the following:
    1) Use of elicitSlot in slot validation and re-prompting.
    2) Use of sessionAttributes to pass information that can be used to
        guide a conversation.
    """
    slots = get_slots(intent_request)
    postal_code = get_slot(intent_request, 'PostalCode')
    session_attributes = get_session_attributes(intent_request)

    if intent_request['invocationSource'] == 'DialogCodeHook':
        # Validate the PostalCode slot. If any aren't valid,
        # re-elicit for the value.
```

```
        validation_result = validate_postal_code(intent_request)
        if not validation_result['isValid']:
            slots[validation_result['violatedSlot']] = None
            return elicit_slot(
                session_attributes,
                intent_request,
                slots,
                validation_result['violatedSlot'],
                validation_result['slotElicitationStyle'],
                validation_result['message']
            )

        return close(
            intent_request,
            session_attributes,
            'Fulfilled',
            {'contentType': 'PlainText',
             'content': 'Thanks'
             }
        )

def close(intent_request, session_attributes, fulfillment_state, message):
    intent_request['sessionState']['intent']['state'] = fulfillment_state
    return {
        'sessionState': {
            'sessionAttributes': session_attributes,
            'dialogAction': {
                'type': 'Close'
            },
            'intent': intent_request['sessionState']['intent'],
            'originatingRequestId': 'xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'
        },
        'messages': [ message ],
        'sessionId': intent_request['sessionId'],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes'
 in intent_request else None
    }

# --- Intents ---

def dispatch(intent_request):
    """
    Called when the user specifies an intent for this bot.
    """
```

```
    intent_name = intent_request['sessionState']['intent']['name']
    response = None

    # Dispatch to your bot's intent handlers
    if intent_name == 'CheckAccount':
        response = check_account(intent_request)

    return response

# --- Main handler ---

def lambda_handler(event, context):
    """
    Route the incoming request based on the intent.

    The JSON body of the request is provided in the event slot.
    """

    # By default, treat the user request as coming from
    # Eastern Standard Time.
    os.environ['TZ'] = 'America/New_York'
    time.tzset()

    logger.debug('event={}'.format(json.dumps(event)))
    response = dispatch(event)
    logger.debug("response={}".format(json.dumps(response)))

    return response
```

Using audio filler to improve bot responsiveness

Audio filler plays brief background audio, such as a light melody or soft keystrokes, during the pause between the end of a user's utterance and the start of the bot's response. This masks processing delays and keeps voice conversations feeling natural.

> **Note**
>
> At launch, audio filler is available for bot locales that support speech-to-speech interactions and have `unifiedSpeechSettings` configured. Support for additional conversation modes will roll out over the next several months.

Available audio filler types

Amazon Lex V2 provides seven built-in filler sounds, organized into two families:

- Melody - Chipper Chime
- Melody - Curious Crawl
- Melody - Rising Ripple
- Melody - Patient Ping
- Melody - Pondering Pong
- Typing - Kinetic Keys
- Typing - Quiet Qwerty

Use the **Play audio preview** button in the Amazon Lex V2 console to hear each option before you save it to a bot locale.

Timing parameters

You can tune three timing parameters to control when audio filler plays and how it transitions into the bot response:

`startDelayInMilliseconds`

Time to wait after the end of the user's utterance before starting audio filler playback. Valid range is `500` to `5000` milliseconds. Default is `1000`.

`minimumPlayDurationInMilliseconds`

Minimum time audio filler plays after it has started, even if the bot response becomes ready sooner. Valid range is `1000` to `5000` milliseconds. Default is `3000`.

`responseDeliveryDelayInMilliseconds`

Silent delay inserted between the end of audio filler playback and the start of the bot's response. Valid range is `200` to `1000` milliseconds. Default is `500`.

Configuring audio filler

You can configure audio filler when creating or updating a bot locale through the Amazon Lex V2 console, the Amazon Connect Conversational AI designer, or the AWS CLI and SDKs.

Using the console

1. Open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/.
2. Choose your bot from the list.
3. In the left navigation pane, choose **Bot languages**.
4. Choose the language you want to configure, or choose **Add language** to add a new one.
5. In the **Audio Filler** section, choose **Enable audio filler**.
6. Choose an **Audio type** from the melody or typing options. Use **Play audio preview** to hear the selected filler.
7. Adjust the timing sliders for **Start delay**, **Minimum play duration**, and **Response buffer** as needed.
8. Choose **Save** to apply the changes.

Using the Amazon Connect Conversational AI designer

1. Open the Amazon Connect admin website and navigate to the Conversational AI designer for your bot.
2. Open the language (locale) you want to configure.
3. In the **Audio Filler** section, enable audio filler and choose an audio type.
4. Adjust the **Start delay**, **Minimum play duration**, and **Response buffer** values.
5. Save your changes. The designer applies the same `audioFillerSettings` to the underlying Amazon Lex V2 bot locale.

Using the API

You can set audio filler using the `audioFillerSettings` parameter in the following API operations:

- `CreateBotLocale` - Configure audio filler for a new bot locale.
- `UpdateBotLocale` - Modify audio filler for an existing bot locale.
- `DescribeBotLocale` - View the current audio filler configuration.

Example Configure audio filler using the AWS CLI

```
aws lexv2-models update-bot-locale \
    --bot-id "bot-1234567890abcdef0" \
    --bot-version "DRAFT" \
    --locale-id "en_US" \
    --nlu-intent-confidence-threshold 0.40 \
    --audio-filler-settings '{
        "enabled": true,
        "audioType": "MELODY_CHIPPER_CHIME",
        "startDelayInMilliseconds": 1000,
        "minimumPlayDurationInMilliseconds": 3000,
        "responseDeliveryDelayInMilliseconds": 500
    }'
```

Audio filler with AI agent interim messages

Audio filler works alongside AI agent interim messages. When an AI agent sends an interim message to the caller (for example, "Let me look that up for you"), the start delay timer is measured from the end of that interim message rather than from the original invocation. This prevents audio filler from overlapping with the agent's speech and ensures the delay the caller experiences is measured from the most recent audio they heard.

Audio filler with dialog and fulfillment code hooks

Audio filler also plays during the processing gap introduced by Lambda dialog code hooks and fulfillment code hooks. The same timing parameters apply, so callers hear a consistent experience whether your bot delegates processing to an AI agent, a code hook, or both in the same turn.

Best practices for audio filler

- Match the filler to your brand voice. Use melody fillers for consumer or retail experiences, and typing fillers when users expect the bot to be actively working on a task.
- Tune the start delay to your latency profile. If most bot responses are faster than `startDelayInMilliseconds`, the filler will rarely play. Lower the delay for latency-heavy workloads and raise it for fast-responding bots.
- Keep the minimum play duration short for fast bots. A long `minimumPlayDurationInMilliseconds` on a fast bot adds perceived latency by holding the filler after the response is ready.

- Test with representative traffic. Validate the filler choice and timing in realistic conversations before rolling out to production.

Monitoring bot performance in Lex V2

Monitoring is important for maintaining the reliability, availability, and performance of your Amazon Lex V2 chatbots. This topic describes using conversation logs to monitor conversations between your users and your chatbots, using utterance statistics to determine the utterances that your bots detect and miss, and how to use Amazon CloudWatch Logs and AWS CloudTrail to monitor Amazon Lex V2. It also describes the Amazon Lex V2 runtime and channel association metrics.

Use these tools and metrics to understand what directions and actions you can take to improve your bots' performance.

Topics

Measuring business performance with Analytics

With Analytics, you can evaluate the performance of your bot with metrics that are related to success and failure rates of your bots' interactions with customers. You can also visualize patterns of conversation flows between your bot and customers. Analytics streamlines your experience by summarizing these metrics in graphs and charts. Analytics provides tools to help you filter results to identify issues and problems involving intents, slots, utterances, and conversations. You can use this data to iterate and improve upon your bot to create a better customer experience.

> **Note**
>
> For a user to access Analytics, either the AWS managed policy: AmazonLexFullAccess or a custom policy that includes analytics API permissions must be attached to their IAM role. See Managing access permissions for analytics for details on how to handle user permissions with a custom policy. If the AWS managed policy: AmazonLexReadOnly is

attached to a customer's IAM role, an error displays the missing permissions that you need to add to the user's IAM role for them to be able to access the Analytics dashboards.

To access Analytics

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the navigation pane under **Bots**, select the bot you want to view in analytics.
3. Select the section under **Analytics** that you want to view.

Topics

Key definitions

This topic provides key definitions that will help you interpret your bot analytics. These definitions are related to the performance of your bot in four contexts: **Intents**, **Slots**, **Conversations**, and **Utterances**. The following fields are relevant to many of the performance metrics:

- The `state` field of the `Intent` object.
- The `type` field of the `dialogAction` object within the SessionState object.

Intents

Amazon Lex V2 categorizes intents in the following ways:

- **Success** – The bot successfully fulfilled the intent. One of the following situations is true:

  - The intent `state` is `ReadyForFulfillment` and the `type` of `dialogAction` is `Close`.
  - The intent `state` is `Fulfilled` and the `type` of `dialogAction` is `Close`.
- **Failed** – The bot failed to fulfill the intent. The intent state. One of the following situations is true:
  - The intent `state` is `Failed` and the `type` of `dialogAction` is `Close` (for example, the user declined the confirmation prompt).
  - The bot switches to the `AMAZON.FallbackIntent` before the intent is completed.
- **Switched** – The bot recognizes a different intent and switches to that intent instead, before the original intent is categorized as a *success* or *failed*.
- **Dropped** – The customer doesn't respond before the intent is categorized as a *success* or *failed*.

Slots

Amazon Lex V2 categorizes slots in the following ways:

- **Success** – The bot filled the slot and successfully transitioned to another slot or the confirmation step.
- **Failed** – The bot wasn't able to fill the slot, even after reaching the maximum number of retries.
- **Dropped** – The customer doesn't respond or switches to another intent before the slot is categorized as a *success* or *failed*.

Conversations

When a customer makes a runtime call to Amazon Lex V2, they provide a `sessionId` and Amazon Lex V2 generates an `originatingRequestId`. If the customer doesn't respond within the Session timeout (`idleSessionTTLInSeconds`) that you set for the bot, the session expires. If a customer returns to the session by using the same `sessionId`, Amazon Lex V2 generates a new `originatingRequestId`.

For analytics, a *conversation* is a unique combination of a `sessionId` and an `originatingRequestId`. Amazon Lex V2 categorizes conversations in the following ways:

- **Success** – The final intent in the conversation is categorized as a *success*.
- **Failed** – The final intent in the conversation is *failed*. The conversation is also *failed* if Amazon Lex V2 defaults to the AMAZON.FallbackIntent.

- **Dropped** – The customer doesn't respond before the conversation is categorized as a *success* or *failed*.

Utterances

Amazon Lex V2 categorizes utterances in the following ways:

- **Detected** – Amazon Lex V2 recognizes the utterance as an attempt to invoke an intent configured for a bot.
- **Missed** – Amazon Lex V2 doesn't recognize the utterance.

Filtering results

At the top of each page, you can filter the results for your bot analytics.

You can filter by the following parameters:

- **Time** – You can filter results by a relative or absolute time range. When you select a start and end time, Amazon Lex V2 retrieves conversations that began *after* the start time and ended *before* the end time.
  - **Relative range** – Select **1d** to see results from the past day, **1w** for the past week, or **1m** for the past month.

    For more options, select **Custom** and choose a duration in the **Relative range** menu. For more control over the duration, select **Custom range**, enter a number in the **Duration** field, and choose a **Unit of time** from the dropdown menu.
  - **Absolute range** – Select **Custom** and choose the **Absolute range** menu to filter for conversations within a time range that you specify. You can choose a start and end date on the calendar or enter it in YYYY/MM/DD format.

> **Note**
>
> The analytics time range has the following restrictions:
>
> - The start date must be within the last 365 days from the current date.
> - The end date must not be more than 1 month after the start date.

- **Bot filters** – To filter by locale, alias, and version of your bot, select the dropdown menus labeled **All locales**, **All aliases**, and **All versions**.
- **Modality** – Select the gear icon and choose the **Modality** dropdown menu to choose whether to display results for **Speech** or **Text**.
- **Channel** – Select the gear icon and choose the **Channel** dropdown menu to choose the channel for which you want to display results. For more information about channel integration, see Integrating an Amazon Lex V2 bot with a messaging platform and Amazon Connect contact centers

Overview: a summary of your bot performance

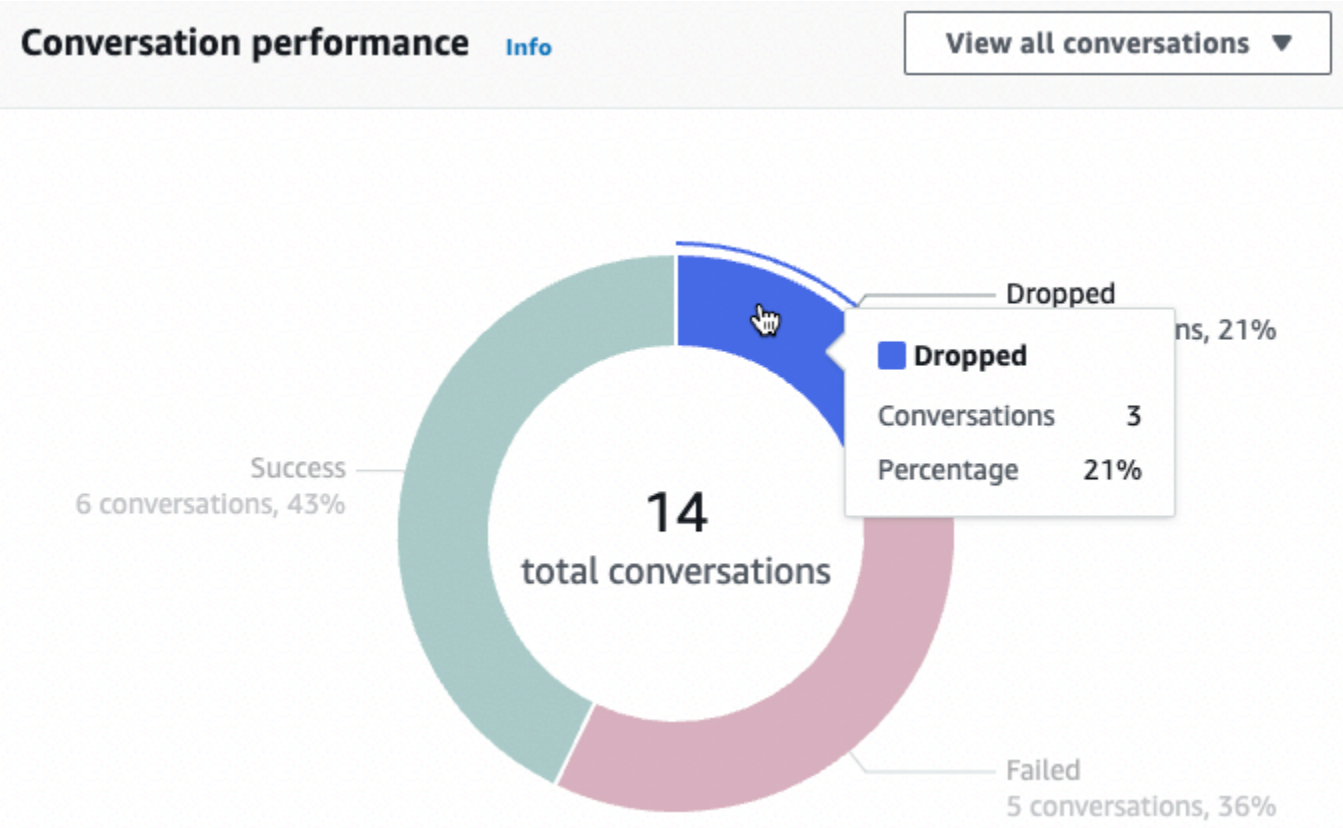
The overview page summarizes your bot's performance on conversations, utterance recognition, and intent usage. The overview consists of the following sections:

- Conversation performance
- Utterance recognition rate
- Conversation performance history
- Top 5 used intents
- Top 5 failed intents

Conversation performance

Use this chart to track the number and percentage of conversations that are categorized as a *success*, *failed*, and *dropped*. To access a list of conversations, select **View all conversations** to reveal a dropdown menu. You can choose to view a list of all user conversations with the bot or filter for conversations with a specific result (*success*, *failed*, or *dropped*). These links take you to the **Conversations** subsection of the **Conversation dashboard**. For more information, see Conversations.

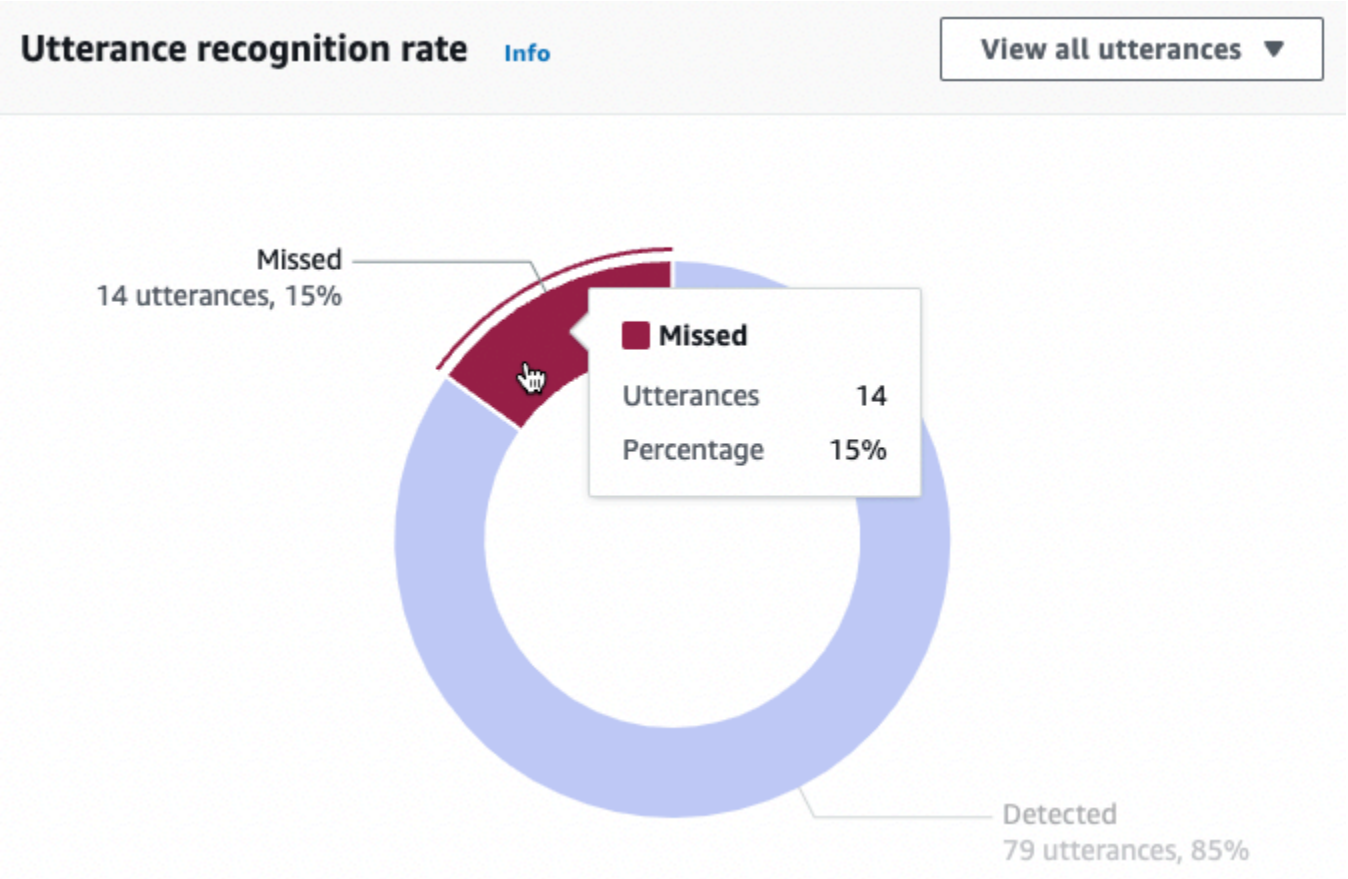
To reveal a box with the count and percentage of conversations with that result, hover over a segment of the chart, as in the following image.



Utterance recognition rate

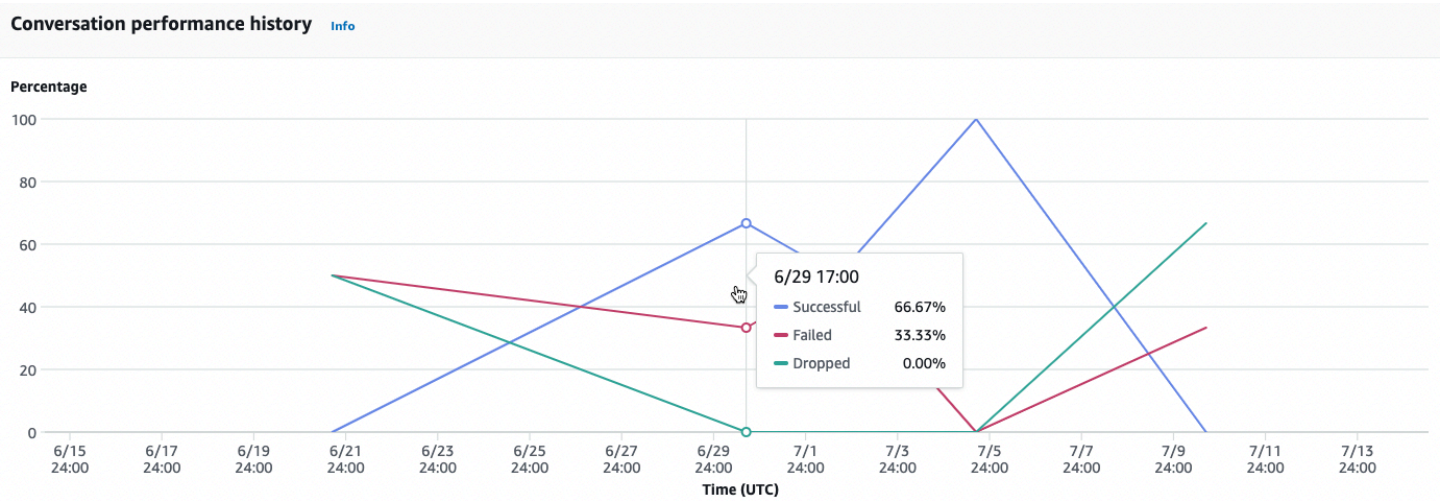
Use this chart to track the number and percentage of utterances that were **detected** and **missed** by your bot. To access a list of utterances, select **View utterances** to reveal a dropdown menu. You can choose to view a list of all user utterances or filter for utterances with a specific result (*missed* or *detected*). These links take you to the **Utterance recognition** subsection of the **Performance dashboard**. For more information, see **View Utterances** to navigate to Utterance recognition.

To reveal a box with the count and percentage of utterances, hover over a segment of the chart, as seen in the following image.



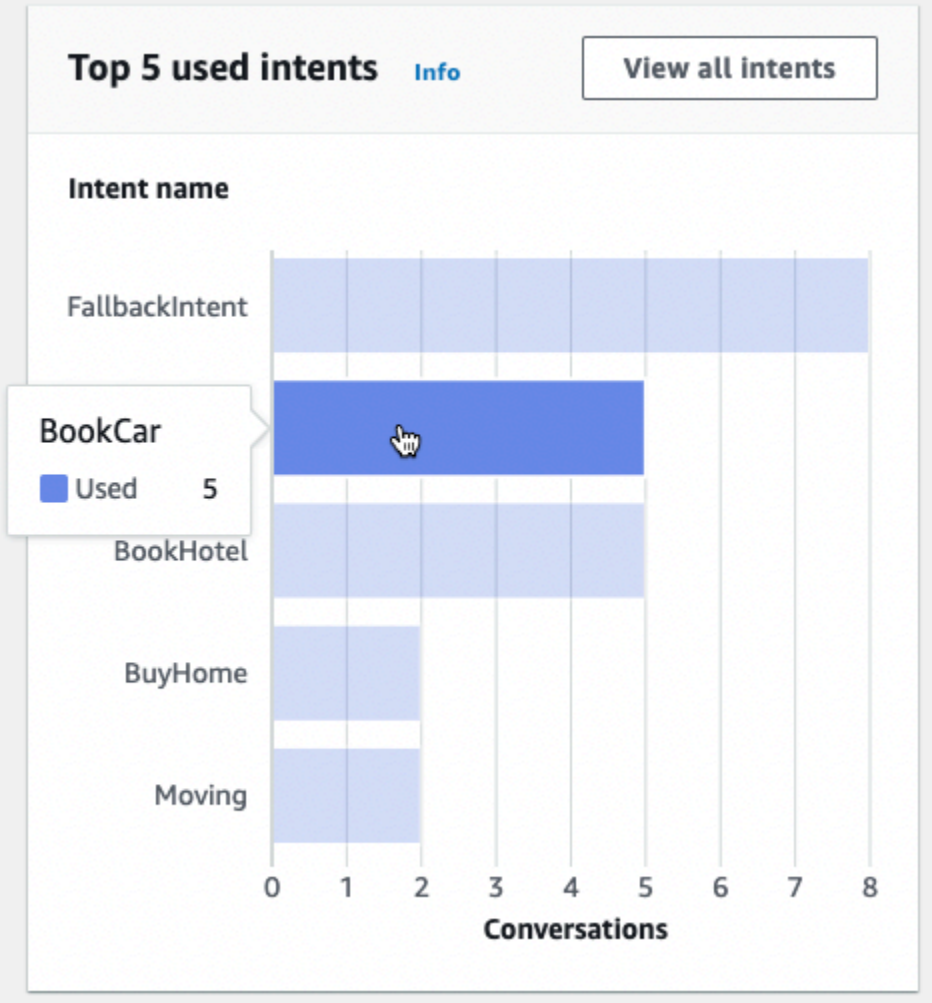
Conversation performance history

Use this graph to track the percentage of conversations categorized as a *success*, *failed*, and *dropped* over the time range that you set in the filters. To see the percentage of conversations with a specific result in an interval of time, hover over that interval, as in the following image.



Top 5 used intents

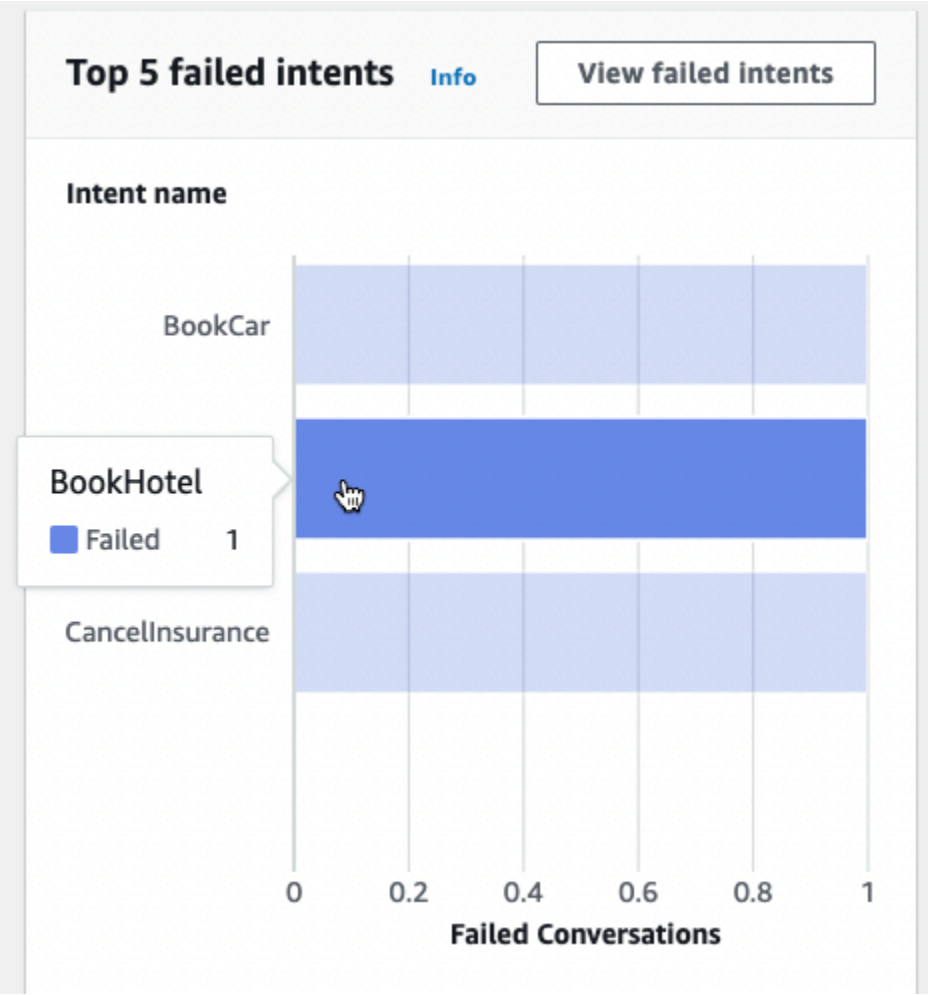
Use this chart to identify the top five intents that customers used with your bot. Hover over a bar to see the number of times that your bot recognized that intent, as in the following image.



Select **View all intents** to navigate to the **Intents performance** subsection of the **Performance dashboard**, where you can view metrics for your bot's performance in fulfilling intents. For more information, see Intent performance.

Top 5 failed intents

Use this chart to identify the top five intents that your bot failed to fulfill (see Intents for the definition of a failed intent). Hover over a bar to see the number of times that your bot failed to fulfill that intent, as in the following image.



Select **View failed intents** to navigate to the **Intents performance** subsection of the **Performance dashboard**, where you can view metrics for the intents that your bot failed to fulfill. For more information, see Intent performance.

Conversation dashboard: a summary of your bot conversations

The conversation dashboard visualizes metrics for customers' conversations (see Conversations for the definition of a *conversation*) with your bot.

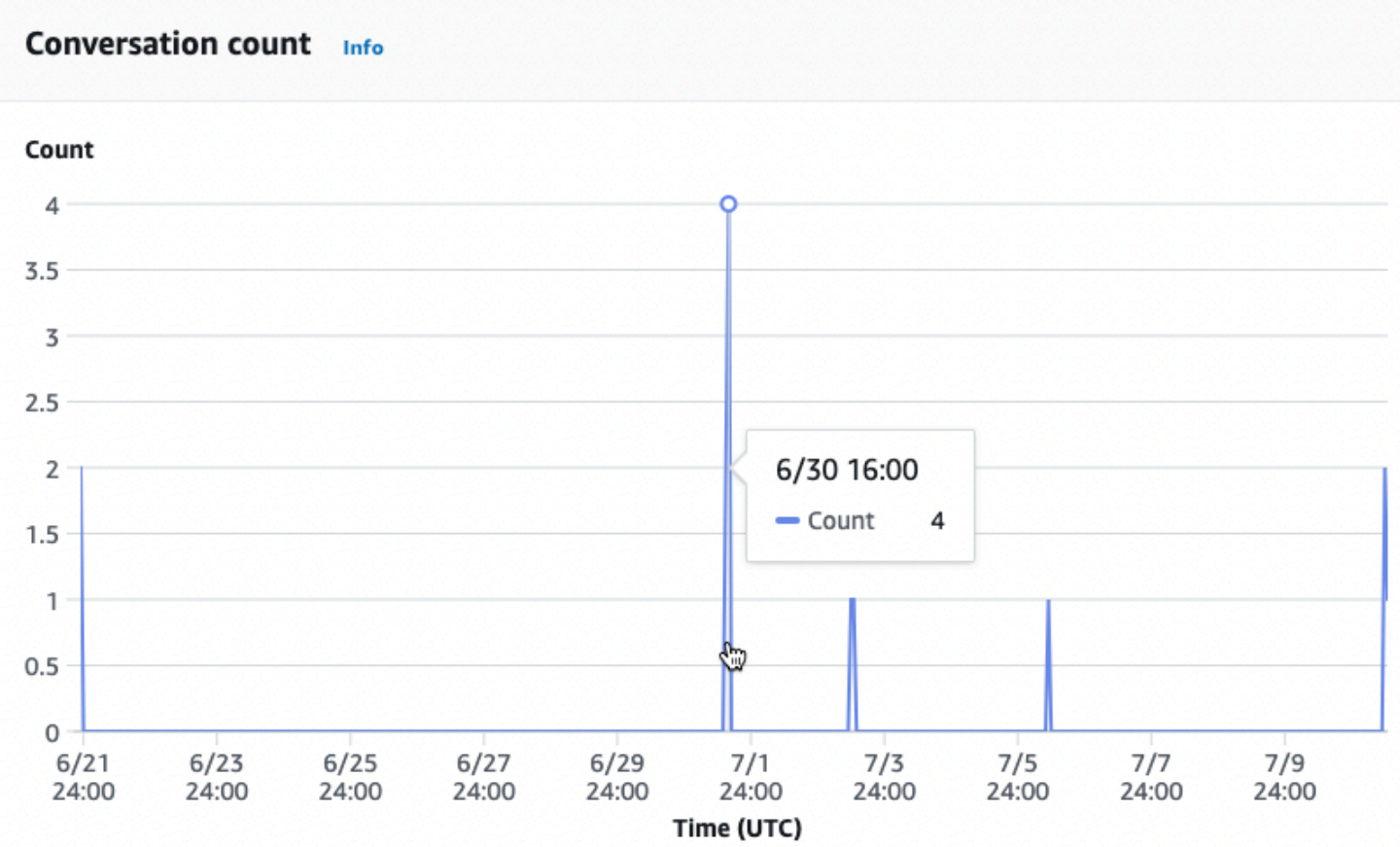
The **Summary** contains the following information about user conversations with your bot. The numbers are calculated based on the filter settings.

- **Total conversations** – The total number of conversations with the bot.
- **Average conversation duration** – The average time of user conversations with the bot in minutes and seconds. The format is mm:ss.
- **Average turns per conversation** – The average number of turns that a conversation lasts.

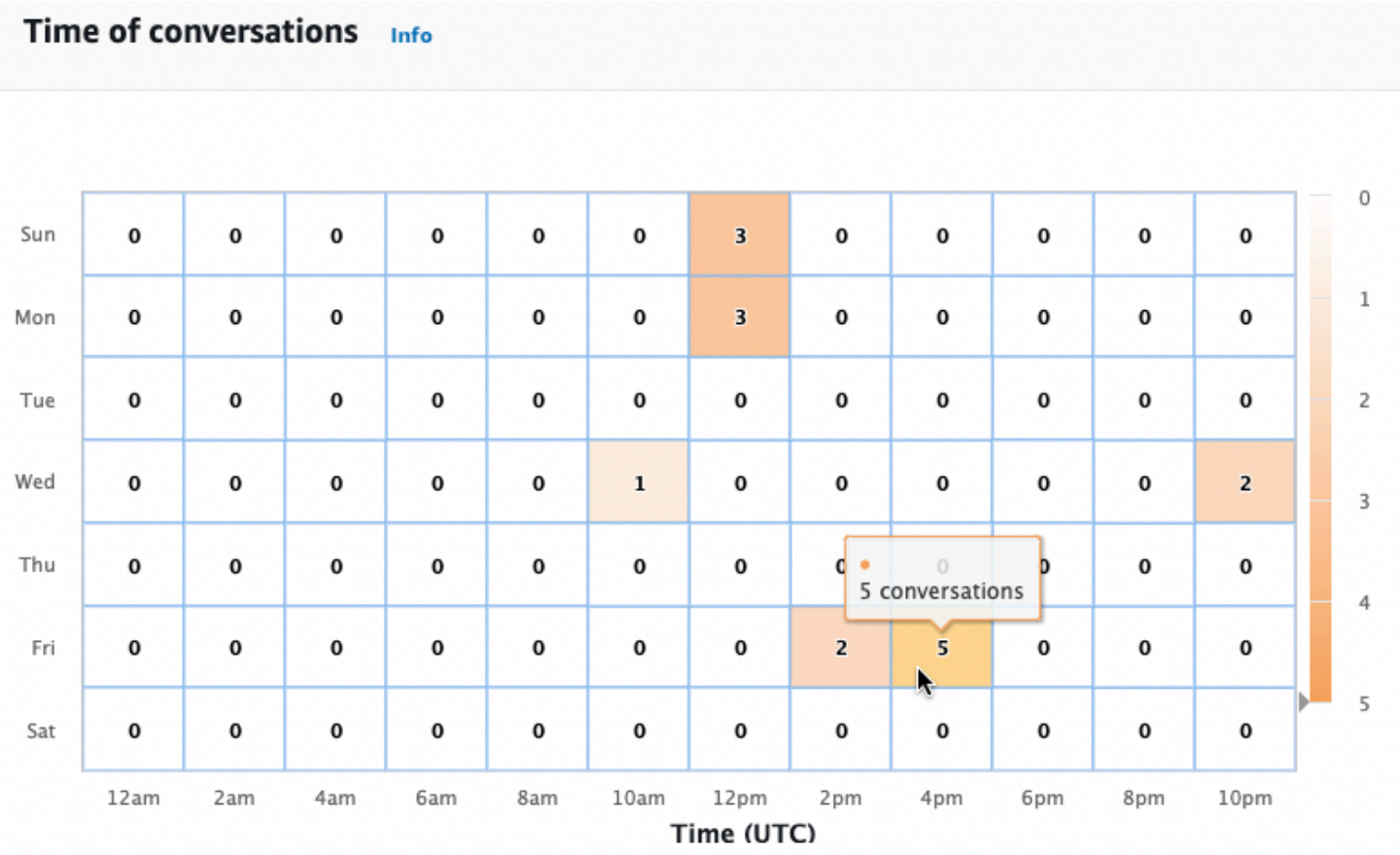
The **Conversation count** and **Message count** sections each contain a graph that shows the number of conversations and messages, respectively, over the time range that you specify in your filters. Hover over a segment of time to see the number of conversations or messages in that segment. The size of the time segment depends on the time range you specify:

- Less than 1 week – The count is displayed for each hour.
- 1 week or more – The count is displayed for each day.

See an example of the hovering behavior in the following image.



The **Time of conversations** section presents the number of conversations that took place between your bot and customers over each two hour interval on each day of the week, within the time range that you specify in the filters. More darkly shaded cells indicate times at which more conversations took place. Hover over a cell to display the number of conversations in the 2 hours beginning from that time slot. For example, the action in the following image shows the number of conversations occurring between 4:00PM and 6:00PM UTC.



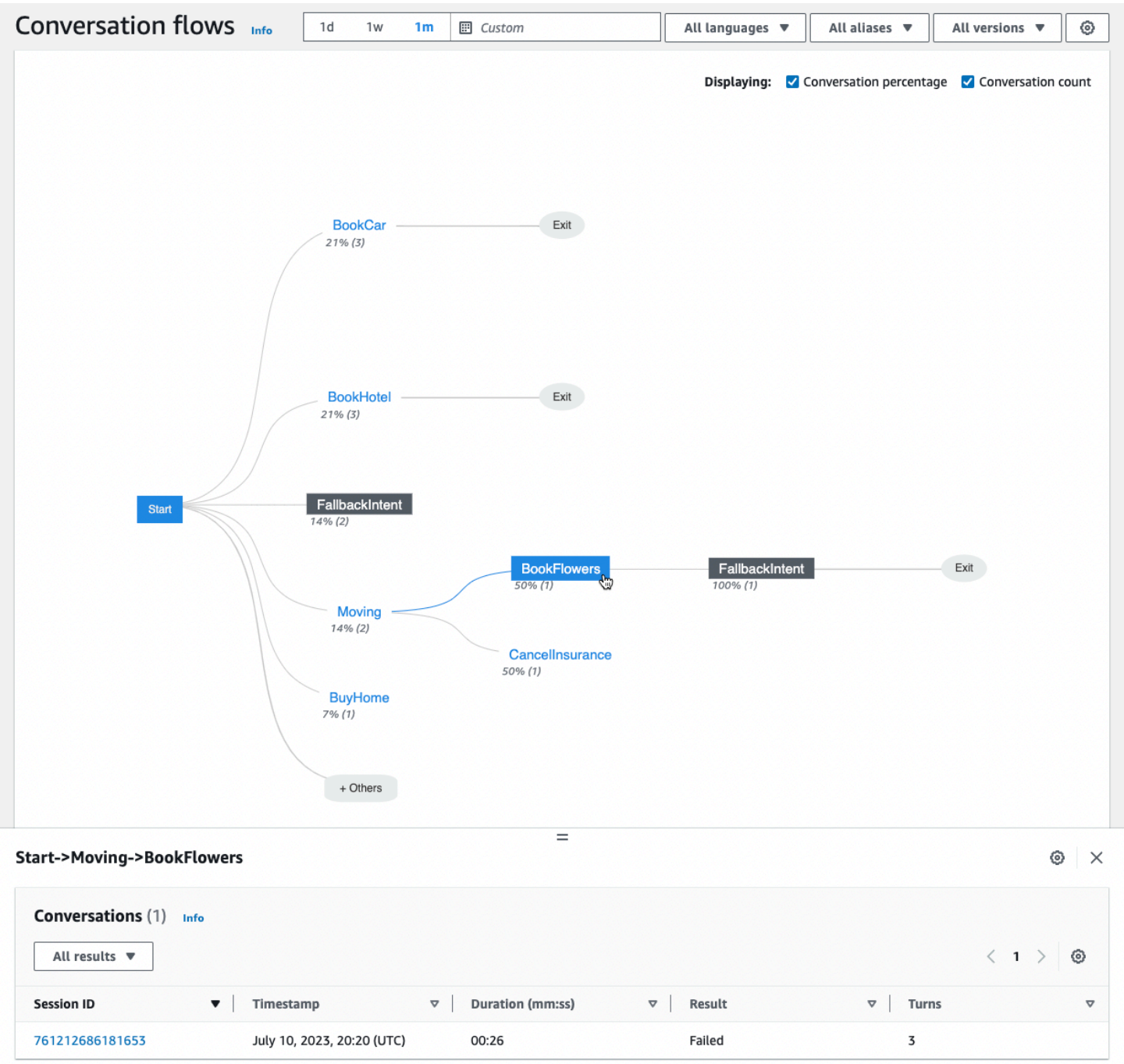
The **Conversation dashboard** contains two tools, **Conversation flows** and **Conversations**. Access a tool by selecting it under **Conversation dashboard** in the left navigation pane.

Conversation flows

Use **Conversation flows** to visualize the orders of intents that customers take in conversations with your bot. Below each intent is the percentage and count of conversations that invoked that intent at that point in the conversation. You can toggle the percentage and count off by selecting **Conversation percentage** and **Conversation count** at the top. By default, the five most common intents at that point in the conversation are shown in descending order of frequency. Select **+ Others** to display all the intents.

Choose an intent to expand to a new column of branches that shows a list of intents taken at that point in the conversation, sorted in descending frequency.

When you select a node in the conversation flow, you can expand the window below to display a list of conversations that followed that order of intents. Choose the **Session ID** corresponding to a conversation to view details about that conversation. The following image shows a conversation flow and an expanded **Conversations** window at the bottom.



Conversations

The **Conversations** tool displays a list of conversations for your bot. You can select a column to sort by that column in ascending or descending order.

To filter the conversations by result, select **All results** and choose **Success**, **Failed**, or **Dropped**.

To filter the conversations by duration

1. Select the search bar marked **Filter conversations by duration**
2. Define the filter in one of the following ways:
   - Use the predefined options.
     a. Select **Duration**.
     b. Choose between the = (equals), > (greater than), and < (less than) operators.
     c. Choose a length of time.
   - Enter an input in the format "Duration {operator} {number} sec" For example, to search for all conversations lasting longer than 30 seconds, enter **`Duration > 30 sec`**. Specify the length of time in seconds.

To view detailed information about the session, including metadata, intent usage, and a transcript, select the **Session ID** of a conversation.

> **Note**
>
> Because a conversation is a unique combination of a `sessionId` and `originatingRequestId`, the same `sessionId` might appear multiple times in the table.

The **Details** section contains the following metadata:

- **Timestamp** – Specifies the date and start time of the conversation. The time is in hh:mm:ss format.
- **Duration** – Specifies how long the conversation lasted in mm:ss format. The duration doesn't include the Session timeout duration (`idleSessionTTLInSeconds`).
- **Result** – Specifies whether the conversation was categorized as a *success*, *failed*, or *dropped*. See Conversations for more details about these results.
- **Mode** – Specifies whether the conversation was `Speech`, `Text`, or `DTMF` (touch tone keypad presses). A conversation consisting of multiple modes is `Multimode`.
- **Channel** – Specifies the channel that the conversation took place on, if applicable. See Integrating an Amazon Lex V2 bot with a messaging platform.
- **Language** – Specifies the language of the bot.

The intents that the bot elicited in the conversation are shown below **Details**. Select **Go to Intent** to go to that intent in the intent editor. Select **Snap to transcript** to automatically scroll the **Transcript** to the first instance in which the bot elicited the intent.

Select the right arrow next to the intent name to view details about the slots elicited for the intent, including the names of the slots, the value that the bot elicited for each slot, and the number of times the bot attempted to elicit each slot.

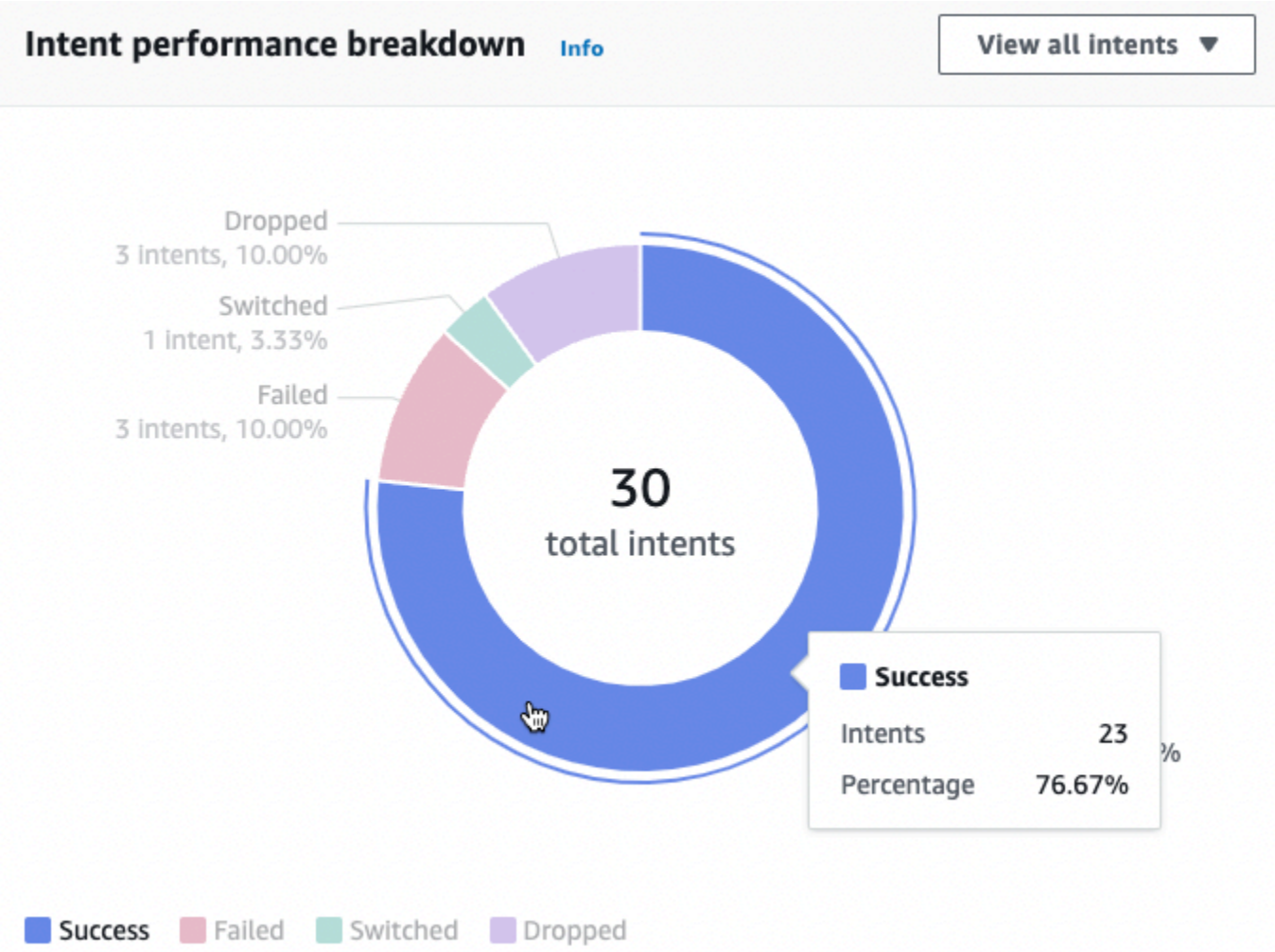
The **Transcript** lets you review the conversation utterances and your bot's behavior in eliciting intents and slots. User utterances are displayed on the left and bot utterances are displayed on the right. Use the search bar marked **Filter transcripts in this session** to find text in the transcript. Next to **Showing:** there are three pieces of information shown under each conversation turn that you can select to display or not:

- **Timestamp** – Specifies the time of the utterance.
- **Intent state** – Specifies the intent that the bot is eliciting during an utterance and the result of the intent, if applicable. The following intent states are possible:
  - Invoked intent: *`intent name`* – The bot has identified an intent that the customer is invoking.
  - Switched intent: *`intent name`* – The bot has switched to a different intent based on the utterance.
  - *`intent name`*: Success – The bot has fulfilled the intent.
- **Slot state** – Specifies the slot that the bot is eliciting during an utterance, if applicable, and the value that the customer provides.

Performance dashboard: a summary of your bot's intent and utterance metrics

In the performance dashboard, you can view details about the performance of your bot's intent fulfillment and utterance recognition.

The **Intent performance breakdown** section displays the total number of times your bot invoked an intent and breaks down the number and percentage of times that the intents were categorized as a *success*, *failed*, *dropped*, and *switched*. See Intents for an explanation of these definitions. Hover over a segment of the chart to show a box with the count and percentage of conversations with that result, as in the following image.



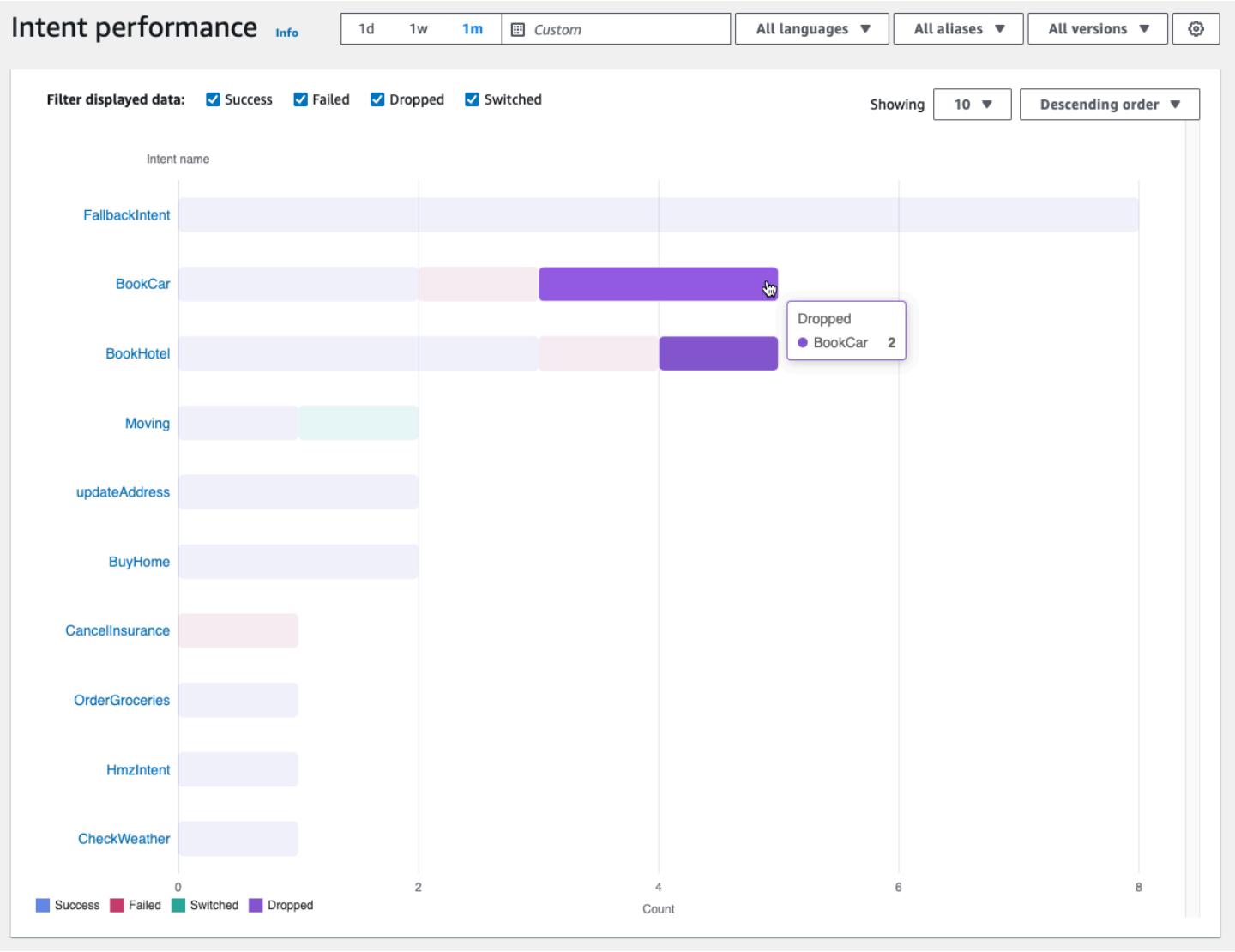
Select **View all intents** to reveal a dropdown menu, from which you can choose to view a list of intents that the bot elicited. You can also choose to view intents with a specific result (*success*, *failed*, *dropped*, or *switched*). These links take you to the **Intent performance** subsection of the **Performance dashboard**. For more information, see Intent performance.

The **Utterance recognition** section summarizes the number of utterances that were *missed* and *detected*. Select **View details** to navigate to a list of utterances for the bot. Choose the number under **Missed utterances** to see a list of missed utterances and the number under **Detected utterances** to see a list of detected utterances for the bot. For more information, see Utterance recognition.

Select **Intents performance** and **Utterance recognition** under the **Performance dashboard** in the left sidebar to view details about intents and utterances in your bot.

Intent performance

This dashboard summarizes the performance of intents used with your bot in descending order of frequency. The bar next to each intent visualizes the number of times the intent was categorized as a *success*, *failed*, *dropped*, and *switched*. See Intents for an explanation of these definitions. Hover over a segment of the bar to see the number of conversations using that intent with that result, as in the following image:



> **Note**
>
> The dashboard shows the top 1,000 results for a set of filter settings. To get more targeted results, configure granular filter settings.

At the top of the chart, you can toggle the intent statuses that you want to view with the **Success**, **Failed**, **Dropped**, and **Switched** checkboxes.

Select the dropdown menus to the right of **Showing** to adjust the number of intents to display and whether to display intents in ascending or descending order of frequency.

Select an intent name to navigate to a page that shows three charts: **Intent performance breakdown**, **Slot performance**, and **Intent switches**.

The **Intent performance breakdown** section displays the total number of times that the bot used the intent and breaks down the number and percentage of times intent fulfillment was categorized as a *success*, *failed*, *dropped*, and *switched*. See Intents for an explanation of these definitions. Hover over a segment of the chart to see the count and percentage of times that intent fulfillment yielded that result.

The **Slot performance** section displays metrics for the slots that belong to the current intent. To sort by a column, select that column once to sort in ascending order and twice to sort it in descending order. You can use the search bar to find a specific slot or use the page number buttons to navigate through the slots.

> **Note**
>
> The dashboard shows the top 1,000 results for a set of filter settings. To get more targeted results, configure granular filter settings..

The **Intent switches** section lists out the instances in which the bot switched from the current intent to another one with the following information:

- **Stage** – The stage of the conversation at which the bot switched the intent.
- **Intent switched to** – The intent that the bot switched the current intent to.
- **Session count** – The number of sessions in which the **Stage** and **Intent switched to** combination occurred.

> **Note**
>
> The dashboard shows the top 1,000 results for a set of filter settings. To get more targeted results, configure granular filter settings..

Utterance recognition

This page lists out all the utterances that were *missed* and *detected* by your bot and provides tools for you to add sample utterances to intents to help train your bot. See Utterances for an explanation of these definitions. Use the tabs at the top to switch between a list of **Missed utterances** and of **Detected utterances**.

> **Note**
>
> The dashboard shows the top 1,000 results for a set of filter settings. To get more targeted results, configure granular filter settings.

To add utterances to an intent:

1. Choose the checkbox next to the utterances that you want to add as sample utterances for an intent.
2. Select **Add to intent** and choose the intent to which you want to add the utterances in the dropdown menu under **Intent**.
3. Select **Add**.

Using APIs for analytics

This section describes the API operations that you use to retrieve analytics for a bot.

> **Note**
>
> To use the ListUtteranceMetrics and ListUtteranceAnalyticsData, your IAM role must have permissions to perform the ListAggregatedUtterances operation, which provides access to utterance-related analytics. See Viewing utterance statistics from Lex V2 conversations for details and the IAM policy to apply to the IAM role.

- The following API operations retrieve summary metrics for a bot:
  - ListSessionMetrics
  - ListIntentMetrics
  - ListIntentStageMetrics

  - ListUtteranceMetrics
- The following API operations retrieve a list of metadata for sessions and utterances:
  - ListSessionAnalyticsData
  - ListUtteranceAnalyticsData
- The ListIntentPaths operation retrieves metrics about an order of intents that customers take in conversations with a bot.

Filtering results

The Analytics API requests require you to specify the `startTime` and `endTime`. The API returns sessions, intents, intent stages, or utterances that began *after* the `startTime` and ended *before* the `endTime`.

`filters` is an optional field in the Analytics API requests. It maps to a list of AnalyticsSessionFilter, AnalyticsIntentFilter, AnalyticsIntentStageFilter, or AnalyticsUtteranceFilter objects. In each object, use the fields to create an expression to filter by. For example, if you add the following filter to the list, the bot searches for conversations that are longer than 30 seconds.

```
{
    "name": "Duration",
    "operator": "GT",
    "value": "30 sec",
}
```

Retrieving metrics for a bot

Use the `ListSessionMetrics`, `ListIntentMetrics`, `ListIntentStageMetrics`, and `ListUtteranceMetrics` operations to retrieve summary metrics for *sessions*, *intents*, *intent stages*, and *utterances*.

For these operations, fill in the following required fields:

- Provide a `startTime` and `endTime` to define a time range for which you want to retrieve results.
- Specify the metrics you want to calculate in `metrics`, a list of AnalyticsSessionMetric, AnalyticsIntentMetric, AnalyticsIntentStageMetric, or AnalyticsUtteranceMetric objects. In each object, use the `name` field to specify the metric to calculate the `statistic` field to specify

whether to calculate the `Sum`, `Average`, or `Max` number, and the `order` field to specify whether to sort the results in `Ascending` or `Descending` order.

> **Note**
>
> Both the `metrics` and `binBy` objects contain an `order` field. You can specify the sorting `order` in only one of the two objects.

The remaining fields in the request are optional. You can filter and organize the results in the following ways:

- **Filtering results** – Use the `filters` field to filter the results. See Filtering results for more details.
- **Grouping results by category** – Specify the `groupBy` field, a list containing a single AnalyticsSessionResult, AnalyticsIntentResult, AnalyticsIntentStageResult, or AnalyticsUtteranceResult object. In the object, specify the `name` field with the category by which you want to group the results.

  If you specify a `groupBy` field in the request, the `results` object in the response contains `groupByKeys`, a list of AnalyticsSessionGroupByKey, AnalyticsIntentGroupByKey, AnalyticsIntentStageGroupByKey, or AnalyticsUtteranceGroupByKey objects, each with the `name` that you specified in the request and a member of that category in the `value` field.
- **Binning results by time** – Specify the `binBy` field, a list containing a single AnalyticsBinBySpecification object. In the object, specify the `name` field with `ConversationStartTime` to bin the results by when the conversation began or `UtteranceTimestamp` to bin the results by when the utterance took place. Specify the interval of time by which you want to bin the results in the `interval` field, and whether to sort in `Ascending` or `Descending` order of time in the `order` field.

  If you specify a `binBy` field in the request, the `results` object in the response contains `binKeys`, a list of AnalyticsBinKey objects, each with the `name` that you specified in the request and the interval of time that defines that bin in the `value` field.

> **Note**
>
> Both the `metrics` and `binBy` objects contain an `order` field. You can specify the sorting `order` in only one of the two objects.

Use the following fields to handle the display of the response:

- Specify a number between 1 and 1,000 in the `maxResults` field to limit the number of results to return in a single response.
- If the number of results is greater than the number you specify in the `maxResults` field, the response contains a `nextToken`. Make the request again, but use this value in the `nextToken` field to return the next batch of results.

If you are using `ListUtteranceMetrics`, you can specify attributes to return in the `attributes` field. This field maps to a list containing a single AnalyticsUtteranceAttribute object. Specify `LastUsedIntent` in the `name` field to return the intent that Amazon Lex V2 is using at the time of the utterance.

In the response, the `results` field maps to a list of AnalyticsSessionResult, AnalyticsIntentResult, AnalyticsIntentStageResult, or AnalyticsUtteranceResult objects. Each object contains a `metrics` field which returns the value of a summary statistic for a metric that you requested, in addition to any bins or groups created from the methods you specified.

Retrieving metadata for sessions and utterances in a bot

Use the ListSessionAnalyticsData and ListUtteranceAnalyticsData operations to retrieve metadata about individual sessions and utterances.

Fill in the required `startTime` and `endTime` fields to define a time range for which you want to retrieve results.

The remaining fields in the request are optional. To filter and sort results:

- **Filtering results** – Use the `filters` field to filter the results. See Filtering results for more details.

- **Sorting results** – Sort the results with the `sortBy` field, which contains a SessionDataSortBy or UtteranceDataSortBy object. Specify the value you want to sort by in the `name` field and whether to sort in `Ascending` or `Descending` order in the `order` field.

Use the following fields to handle the display of the response:

- Specify a number between 1 and 1,000 in the `maxResults` field to limit the number of results to return in a single response.
- If the number of results is greater than the number you specify in the `maxResults` field, the response contains a `nextToken`. Make the request again, but use this value in the `nextToken` field to return the next batch of results.

In the response, the `sessions` or `utterances` field maps to a list of SessionSpecification or UtteranceSpecification objects. Each object contains metadata for a single session or utterance.

Retrieving intent path analytics data

Use the ListIntentPaths operation to retrieve metrics about an order of intents that customers take in conversation with a bot.

For this operation, fill in the following required fields:

- Provide a `startTime` and `endTime` to define a time range for which you want to retrieve results.
- Provide an `intentPath` to define an order of intents for which you want to retrieve metrics. Separate intents in the path with a forward slash. For example, populate the `intentPath` field with **`/BookCar/BookHotel`** to see details about how many times users invoked the `BookCar` and `BookHotel` intents in that order.

Use the optional `filters` field to filter the results. For more details, see Filtering results.

Viewing utterance statistics from Lex V2 conversations

You can use utterance statistics to determine the utterances that your users are sending to your bot. You can see both the utterances that Amazon Lex V2 successfully detects and the utterances that it doesn't. You can use this information to help tune your bot.

For example, if you find that your users are sending an utterance that Amazon Lex V2 is missing, you can add the utterance to an intent. The Draft version of the intent is updated with the new utterance and you can test it before deploying it to your bot.

An utterance is detected when Amazon Lex V2 recognizes the utterance as an attempt to invoke an intent configured for a bot. An utterance is missed when Amazon Lex V2 doesn't recognize the utterance and invokes the `AMAZON.FallbackIntent` instead.

Utterance statistics can be viewed using the `ListUtteranceMetrics` API and the `ListAggregatedUtterance` API.

Utterance statistics are not generated using `ListUtteranceMetrics` API under the following conditions:

- The Child Online Privacy Protection Act setting was set to **Yes** when the bot was created with the console, or the `childDirected` field was set to true when the bot was created withe the `CreateBot` operation.

The `ListUtteranceMetrics` API provides additonal features including:

- More information available, such as mapped intent for detected utterances.
- More filtering capability (including channel and mode).
- Longer retention date range (30 days).
- You can use the API even if you have opted out of data storage. The console functionality for missed and detected utterances will rely on `ListUtteranceMetrics` API.

Utterance statistics are not generated using `ListAggregatedUtterance` API under the following conditions:

- The Child Online Privacy Protection Act setting was set to **Yes** when the bot was created with the console, or the `childDirected` field was set to true when the bot was created withe the `CreateBot` operation.
- You are using slot obfuscation with one or more slots.
- You opted out of participating in improving Amazon Lex.

The `ListAggregatedUtterance` API provides features including:

- Less detailed information available (no mapped intent for utterances).
- Limited filtering capability (not including channel and mode).
- Short retention date range (15 days).

Using utterance statistics, you can see whether a specific utterance was detected or missed, alongside the last time that the utterance was used in a bot interaction.

Amazon Lex V2 stores utterances continuously while users interact with your bot. You can query the statistics using the console or the `ListAggregatedUtterances` operation. It has a data retention of 15 days and it is not available if the user has opted out of data storage. You can delete utterances using the `DeleteUtterances` operation or by opting out of data storage. All utterances are deleted if you close your AWS account. Stored utterances are encrypted with a server-managed key.

When you delete a bot version, utterance statistics are available for the version for up to 30 days with `ListUtteranceMetrics`, and 15 days using `ListAggregatedUtterances`. You can't see statistics for deleted version in the Amazon Lex V2 console. To see the statistics for deleted versions, you can use both `ListAggregatedUtterances` and `ListUtteranceMetrics`operations.

With both the `ListAggregatedUtterances` and `ListUtteranceMetrics` APIs, utterances are aggregated by the text of the utterance. For example, all instances where the customer used the phrase "I want to order a pizza" are aggregated into the same line in a response. When you use the RecognizeUtterance operation, the text used is the input transcript.

To use the `ListAggregatedUtterances` and `ListUtteranceMetrics` APIs, apply the following policy to a role.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ListAggregatedUtterancesPolicy",
            "Effect": "Allow",
            "Action": "lex:ListAggregatedUtterances",
            "Resource": "*"
```

```
        }
    ]
}
```

Managing access permissions for analytics

To provide a user access to analytics, attach a policy to an IAM role that permits the role to call the API operations for analytics. You can attach the AWS managed policy: AmazonLexFullAccess to the IAM role to provide full access to Amazon Lex API operations, or you can create a custom policy allowing only permissions to analytics and attach it to an IAM role.

To create a custom policy containing permissions for analytics

1. If you need to first create an IAM role, follow the steps at Creating a role to delegate permissions to an IAM user.
2. Follow the steps at Creating IAM policies to create a policy using the following JSON object. To enable analytics access to specific bots for the IAM role, add the ARN of each bot to the `Resource` field. Replace the *`region`*, *`account-id`*, and *`BOTID`* with the values corresponding to the bots. You can also replace the statement identifier, *`AnalyticsActions`*, with a name of your choice.
3. Attach the policy you created to the role that you want to grant analytics permissions by following the steps at Adding and removing IAM identity permissions.
4. The role should now have permissions to view analytics for the bots you specified.

Enabling conversation logs for your Lex V2 bots

Use conversation logs to store user conversations with your bot. Review these logs to identify issues with your bot's interactions with users and modify your bot's behavior with these insights. This section also describes how to obfuscate slot values to protect the privacy of users.

Topics

- Logging conversations with conversation logs in Lex V2
- Obscuring slot values in conversation logs from Lex V2
- Selective conversation log capture in Lex V2

Logging conversations with conversation logs in Lex V2

You enable *conversation logs* to store bot interactions. You can use these logs to review the performance of your bot and to troubleshoot issues with conversations. You can log text for the RecognizeText operation. You can log both text and audio for the RecognizeUtterance operation. By enabling conversation logs, you get a detailed view of conversations that users have with your bot.

For example, a session with your bot has a session ID. You can use this ID to get the transcript of the conversation including user utterances and the corresponding bot responses. You also get metadata such as intent name and slot values for an utterance.

> **Note**
>
> You can't use conversation logs with a bot subject to the Children's Online Privacy Protection Act (COPPA).

Conversation logs are configured for an alias. Each alias can have different settings for their text and audio logs. You can enable text logs, audio logs, or both for each alias. Text logs store text input, transcripts of audio input, and associated metadata in CloudWatch Logs. Audio logs store audio input in Amazon S3. You can enable encryption of text and audio logs using AWS KMS customer managed CMKs.

To configure logging, use the console or the CreateBotAlias or UpdateBotAlias operation. After enabling conversation logs for an alias, using the RecognizeText or RecognizeUtterance operation for that alias logs the text or audio utterances in the configured CloudWatch Logs log group or S3 bucket.

Topics

IAM Policies for Conversation Logs

Depending on the type of logging that you select, Amazon Lex V2 requires permission to use Amazon CloudWatch Logs and Amazon Simple Storage Service (S3) buckets to store your logs. You must create AWS Identity and Access Management roles and permissions to enable Amazon Lex V2 to access these resources.

Creating an IAM Role and Policies for Conversation Logs

To enable conversation logs, you must grant write permission for CloudWatch Logs and Amazon S3. If you enable object encryption for your S3 objects, you need to grant access permission to the AWS KMS keys used to encrypt the objects.

You can use the IAM console, the IAM API, or the AWS Command Line Interface to create the role and policies. These instructions use the AWS CLI to create the role and policies.

> **Note**
>
> The following code is formatted for Linux and MacOS. For Windows, replace the Linux line continuation character (\) with a caret (^).

To create an IAM role for conversation logs

1. Create a document in the current directory called **`LexConversationLogsAssumeRolePolicyDocument.json`**, add the following code to it, and save it. This policy document adds Amazon Lex V2 as a trusted entity to the role. This allows Amazon Lex V2 to assume the role to deliver logs to the resources configured for conversation logs.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
```

```
        "Service": "lexv2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. In the AWS CLI, run the following command to create the IAM role for conversation logs.

```
aws iam create-role \
    --role-name role-name \
    --assume-role-policy-document file://
LexConversationLogsAssumeRolePolicyDocument.json
```

Next, create and attach a policy to the role that enables Amazon Lex V2 to write to CloudWatch Logs.

To create an IAM policy for logging conversation text to CloudWatch Logs

1. Create a document in the current directory called **LexConversationLogsCloudWatchLogsPolicy.json**, add the following IAM policy to it, and save it.
2. In the AWS CLI, create the IAM policy that grants write permission to the CloudWatch Logs log group.

```
aws iam create-policy \
    --policy-name cloudwatch-policy-name \
    --policy-document file://LexConversationLogsCloudWatchLogsPolicy.json
```

3. Attach the policy to the IAM role that you created for conversation logs.

```
aws iam attach-role-policy \
    --policy-arn arn:aws:iam::account-id:policy/cloudwatch-policy-name \
    --role-name role-name
```

If you are logging audio to an S3 bucket, create a policy that enables Amazon Lex V2 to write to the bucket.

To create an IAM policy for audio logging to an S3 bucket

1. Create a document in the current directory called **LexConversationLogsS3Policy.json**, add the following policy to it, and save it.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
      {
          "Effect": "Allow",
          "Action": [
              "s3:PutObject"
          ],
          "Resource": "arn:aws:s3:::bucket-name/*"
      }
  ]
}
```

2. In the AWS CLI, create the IAM policy that grants write permission to your S3 bucket.

```
aws iam create-policy \
    --policy-name s3-policy-name \
    --policy-document file://LexConversationLogsS3Policy.json
```

3. Attach the policy to the role that you created for conversation logs.

```
aws iam attach-role-policy \
    --policy-arn arn:aws:iam::account-id:policy/s3-policy-name \
    --role-name role-name
```

Granting Permission to Pass an IAM Role

When you use the console, the AWS Command Line Interface, or an AWS SDK to specify an IAM role to use for conversation logs, the user specifying the conversation logs IAM role must have permission to pass the role to Amazon Lex V2. To allow the user to pass the role to Amazon Lex V2, you must grant `PassRole` permission to the user's IAM user, role, or group.

The following policy defines the permission to grant to the user, role, or group. You can use the `iam:AssociatedResourceArn` and `iam:PassedToService` condition keys to limit the scope of the permission. For more information, see Granting a User Permissions to Pass a Role to an AWS Service and IAM and AWS STS Condition Context Keys in the *AWS Identity and Access Management User Guide*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::111122223333:role/role-name",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "lexv2.amazonaws.com"
                },
                "StringLike": {
                    "iam:AssociatedResourceARN":
 "arn:aws:lex:region:123456789012:bot:bot-name:bot-alias"
                }
            }
        }
    ]
}
```

Configuring conversation logs for your Lex V2 bot

You enable and disable conversation logs using the console or the `conversationLogSettings` field of the `CreateBotAlias` or `UpdateBotAlias` operation. You can turn on or turn off audio logs, text logs, or both. Logging starts on new bot sessions. Changes to log settings aren't reflected for active sessions.

To store text logs, use an Amazon CloudWatch Logs log group in your AWS account. You can use any valid log group. The log group must be in the same region as the Amazon Lex V2 bot. For more information about creating a CloudWatch Logs log group, see Working with Log Groups and Log Streams in the *Amazon CloudWatch Logs User Guide*.

To store audio logs, use an Amazon S3 bucket in your AWS account. You can use any valid S3 bucket. The bucket must be in the same region as the Amazon Lex V2 bot. For more information about creating an S3 bucket, see Creating a bucket in the *Amazon Simple Storage Service Getting Started Guide*.

When you manage conversation logs using the console, the console updates your service role so that it has access to the log group and S3 bucket.

If you are not using the console, you must provide an IAM role with policies that enable Amazon Lex V2 to write to the configured log group or bucket. If you create a service-linked role using the AWS Command Line Interface, you must add a custom suffix to the role using the `custom-suffix` option as in the following example. For more information, see Creating an IAM Role and Policies for Conversation Logs.

```
aws iam create-service-linked-role \
    --aws-service-name lexv2.amazon.aws.com \
    --custom-suffix suffix
```

The IAM role that you use to enable conversation logs must have the `iam:PassRole` permission. The following policy should be attached to the role:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::111122223333:role/role"
        }
    ]
}
```

Enabling conversation logs

To turn on logs using the console

1. Open the Amazon Lex V2 console https://console.aws.amazon.com/lexv2.

2. From the list, choose a bot.
3. From the left menu, choose **Aliases**.
4. In the list of aliases, choose the alias for which you want to configure conversation logs.
5. In the **Conversation logs** section, choose **Manage conversation logs**.
6. For text logs, choose **Enable** then enter the Amazon CloudWatch Logs log group name.
7. For audio logs, choose **Enable** then enter the S3 bucket information.
8. Optional. To encrypt audio logs, choose the AWS KMS key to use for encryption.
9. Choose **Save** to start logging conversations. If necessary, Amazon Lex V2 will update your service role with permissions to access the CloudWatch Logs log group and selected S3 bucket.

Disabling conversation logs in Lex V2

To turn off logs using the console

1. Open the Amazon Lex V2 console https://console.aws.amazon.com/lexv2.
2. From the list, choose a bot.
3. From the left menu, choose **Aliases**.
4. In the list of aliases, choose the alias for which you want to configure conversation logs.
5. In the **Conversation logs** section, choose **Manage conversation logs**.
6. Disable text logging, audio logging, or both to turn off logging.
7. Choose **Save** to stop logging conversations.

Viewing text logs in Amazon CloudWatch Logs from Lex V2

Amazon Lex V2 stores text logs for your conversations in Amazon CloudWatch Logs. To view the logs, use the CloudWatch Logs console or API. For more information, see Search Log Data Using Filter Patterns and CloudWatch Logs Insights Query Syntax in the *Amazon CloudWatch Logs User Guide*.

To view logs using the Amazon Lex V2 console

1. Open the Amazon Lex V2 console https://console.aws.amazon.com/lexv2.
2. From the list, choose a bot.
3. From the left menu, choose **Analytics** and then choose **CloudWatch metrics**.

4. View metrics for your bot on the **CloudWatch metrics** page.

You can also use the CloudWatch console or API to view your log entries. To find the log entries, navigate to the log group that you configured for the alias. You can find the log stream prefix for your logs in the Amazon Lex V2 console or by using the DescribeBotAlias operation.

Log entries for a user utterance are found in multiple log streams. An utterance in the conversation has an entry in one of the log streams with the specified prefix. An entry in the log stream contains the following information:

message-version

The message schema version.

bot

Details about the bot that the customer is interacting with.

messages

The response that the bot sent back to the user.

utteranceContext

Information about processing this utterance.

- `runtimeHints`—runtime context used to transcribe and interpret the user's input. For more information, see Improving recognition of slot values with runtime hints in the conversation.
- `slotElicitationStyle`—Slot elicitation style used to interpret user input. For more information, see Capturing slot values with spelling styles during the conversation.

sessionState

The current state of the conversation between the user and the bot. For more information, see Understanding Amazon Lex V2 bot conversations.

interpretations

A list of intents that Amazon Lex V2 determined could satisfy the user's utterance. Using confidence scores to improve conversation accuracy.

interpretationSource

Indicates whether a slot is resolved by Amazon Lex V2 or Amazon Bedrock. Values: Lex | Bedrock

sessionId

The identifier of the user session that is having the conversation.

inputTranscript

A transcription of the input from the user.

- For text input, this is the text that the user typed. For DTMF input, this is the key that the user input.
- For speech input, this is the text to which Amazon Lex V2 converts the user utterance in order to invoke an intent or fill a slot.

rawInputTranscript

The raw transcript of the user input before any text processing is applied. Note: Text processing is only for en-US and en-GB locales.

transcriptions

A list of potential transcriptions of the user's input. For more information, see Using voice transcription confidence scores to improve conversations with your Lex V2 bot.

rawTranscription

Using voice transcription confidence scores. For more information, see Using voice transcription confidence scores to improve conversations with your Lex V2 bot.

missedUtterance

Indicates whether Amazon Lex V2 was able to recognize the user's utterance.

requestId

Amazon Lex V2 generated request ID for the user input.

timestamp

The timestamp of the user's input.

developerOverride

Indicates whether the conversation flow was updated using a dialog code hook. For more information on using a dialog code hook, see Integrating an AWS Lambda function into your Amazon Lex V2 bot.

inputMode

Indicates the type of input. Can be audio, DTMF, or text.

requestAttributes

The request attributes used when processing the user's input.

audioProperties

If audio conversation logs are enabled and the user input was in audio format, includes the total duration of the audio input, the duration of voice and the duration of silence in the audio. It also includes a link to the audio file.

bargeIn

Indicates whether the user input interrupted the previous bot response.

responseReason

The reason a response was generated. Can be one of:

- `UtteranceResponse` – response to user input
- `StartTimeout` – server generated response when the user didn't provide input
- `StillWaitingResponse` – server generated response when the user requests the bot wait
- `FulfillmentInitiated` – server generated response that fulfillment is about to be initiated
- `FulfillmentStartedResponse` – server generated response that fulfillment has begun
- `FulfillmentUpdateResponse` – periodic server generated response while fulfillment is in progress
- `FulfillmentCompletedResponse` – server generated response when fulfillment is complete.

operationName

The API used to interact with the bot. Can be one of `PutSession`, `RecognizeText`, `RecognizeUtterance`, or `StartConversation`.

```
{
    "message-version": "2.0",
    "bot": {
        "id": "string",
        "name": "string",
        "aliasId": "string",
        "aliasName": "string",
```

```
        "localeId": "string",
        "version": "string"
    },
    "messages": [
        {
            "contentType": "PlainText | SSML | CustomPayload | ImageResponseCard",
            "content": "string",
            "imageResponseCard": {
                "title": "string",
                "subtitle": "string",
                "imageUrl": "string",
                "buttonsList": [
                    {
                        "text": "string",
                        "value": "string"
                    }
                ]
            }
        }
    ],
    "utteranceContext": {
        "activeRuntimeHints": {
            "slotHints": {
                "string": {
                    "string": {
                        "runtimeHintValues": [
                            {
                                "phrase": "string"
                            },
                            {
                                "phrase": "string"
                            }
                        ]
                    }
                }
            }
        },
        "slotElicitationStyle": "string"
    },
    "sessionState": {
        "dialogAction": {
            "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot",
            "slotToElicit": "string"
        },
```

```
        "intent": {
            "name": "string",
            "slots": {
                "string": {
                    "value": {
                       "interpretedValue": "string",
                       "originalValue": "string",
                       "resolvedValues": [ "string" ]
                    }
                 },
                "string": {
                    "shape": "List",
                    "value": {
                        "originalValue": "string",
                        "interpretedValue": "string",
                        "resolvedValues": [ "string" ]
                    },
                    "values": [
                        {
                            "shape": "Scalar",
                            "value": {
                                "originalValue": "string",
                                "interpretedValue": "string",
                                "resolvedValues": [ "string" ]
                            }
                        },
                        {
                            "shape": "Scalar",
                            "value": {
                                "originalValue": "string",
                                "interpretedValue": "string",
                                "resolvedValues": [ "string" ]
                            }
                        }
                    ]
                }
            },
            "kendraResponse": {
                // Only present when intent is KendraSearchIntent. For details, see
                // https://docs.aws.amazon.com/kendra/latest/dg/
API_Query.html#API_Query_ResponseSyntax
                },
            "state": "InProgress | ReadyForFulfillment | Fulfilled | Failed",
            "confirmationState": "Confirmed | Denied | None"
```

```
        },
        "originatingRequestId": "string",
        "sessionAttributes": {
            "string": "string"
        },
        "runtimeHints": {
            "slotHints": {
                "string": {
                    "string": {
                        "runtimeHintValues": [
                            {
                                "phrase": "string"
                            },
                            {
                                "phrase": "string"
                            }
                        ]
                    }
                }
            }
        }
    },
   "dialogEventLogs": [
        {
   // only for conditional
     "conditionalEvaluationResult":[
      // all the branches until true

     {
     "conditionalBranchName": "string",
     "expressionString": "string",
     "evaluatedExpression": "string",
     "evaluationResult": "true | false"
     }
    ],
  "dialogCodeHookInvocationLabel": "string",
  "response": "string",
  "nextStep": {
        "dialogAction": {
            "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot",
            "slotToElicit": "string"
        },
       "intent": {
                          "name": "string",
```

```
            "slots": {
                }
        }
       }
    ]
    "interpretations": [
        {
            "interpretationSource": "Bedrock | Lex",
            "nluConfidence": "string",
            "intent": {
                "name": "string",
                "slots": {
                    "string": {
                        "value": {
                            "originalValue": "string",
                            "interpretedValue": "string",
                            "resolvedValues": [ "string" ]
                        }
                    },
                    "string": {
                        "shape": "List",
                        "value": {
                            "interpretedValue": "string",
                            "originalValue": "string",
                            "resolvedValues": [ "string" ]
                        },
                        "values": [
                            {
                                "shape": "Scalar",
                                "value": {
                                    "interpretedValue": "string",
                                    "originalValue": "string",
                                    "resolvedValues": [ "string" ]
                                }
                            },
                            {
                                "shape": "Scalar",
                                "value": {
                                    "interpretedValue": "string",
                                    "originalValue": "string",
                                    "resolvedValues": [ "string" ]
                                }

                            }
```

```
                        ]
                    }
                },
                "kendraResponse": {
                    // Only present when intent is KendraSearchIntent. For details, see

                    // https://docs.aws.amazon.com/kendra/latest/dg/
API_Query.html#API_Query_ResponseSyntax
                    },
                "state": "InProgress | ReadyForFulfillment | Fulfilled | Failed",
                "confirmationState": "Confirmed | Denied | None"
                },
            "sentimentResponse": {
                "sentiment": "string",
                "sentimentScore": {
                    "positive": "string",
                    "negative": "string",
                    "neutral": "string",
                    "mixed": "string"
                }
            }
        }
    ],
    "sessionId": "string",
    "inputTranscript": "string",
    "rawInputTranscript": "string",
    "transcriptions": [
        {
            "transcription": "string",
            "rawTranscription": "string",
            "transcriptionConfidence": "number",
            },
            "resolvedContext": {
                "intent": "string"
            },
            "resolvedSlots": {
                "string": {
                    "name": "slotName",
                    "shape": "List",
                    "value": {
                        "originalValue": "string",
                        "resolvedValues": [
                            "string"
                        ]
```

```
                    }
                }
            }
        }
    ],
    "missedUtterance": "bool",
    "requestId": "string",
    "timestamp": "string",
    "developerOverride": "bool",
    "inputMode": "DTMF | Speech | Text",
    "requestAttributes": {
        "string": "string"
    },
    "audioProperties": {
        "contentType": "string",
        "s3Path": "string",
        "duration": {
            "total": "integer",
            "voice": "integer",
            "silence": "integer"
        }
    },
    "bargeIn": "string",
    "responseReason": "string",
    "operationName": "string"
}
```

The contents of the log entry depend on the result of a transaction and the configuration of the bot and request.

- The `intent`, `slots`, and `slotToElicit` fields don't appear in an entry if the `missedUtterance` field is `true`.
- The `s3PathForAudio` field doesn't appear if audio logs are disabled or if the `inputDialogMode`field is `Text`.
- The `responseCard` field only appears when you have defined a response card for the bot.
- The `requestAttributes` map only appears if you have specified request attributes in the request.
- The `kendraResponse` field is only present when the `AMAZON.KendraSearchIntent` makes a request to search an Amazon Kendra index.

- The `developerOverride` field is true when an alternative intent was specified in the bot's Lambda function.
- The `sessionAttributes` map only appears if you have specified session attributes in the request.
- The `sentimentResponse` map only appears if you configure the bot to return sentiment values.

> **Note**
>
> The input format may change without a corresponding change in the `messageVersion`. Your code should not throw an error if new fields are present.

Accessing audio logs in Amazon S3

Amazon Lex V2 stores audio logs for your conversations in an S3 bucket.

You can use the Amazon S3 console or API to access audio logs. You can see the S3 object key prefix of the audio files in the Amazon Lex V2 console, or in the `conversationLogSettings` field in the `DescribeBotAlias` operation response.

Monitoring conversation log status with CloudWatch metrics

Use Amazon CloudWatch to monitor delivery metrics of your conversation logs. You can set alarms on metrics so that you are aware of issues with logging if they should occur.

Amazon Lex V2 provides four metrics in the `AWS/Lex` namespace for conversation logs:

- `ConversationLogsAudioDeliverySuccess`
- `ConversationLogsAudioDeliveryFailure`
- `ConversationLogsTextDeliverySuccess`
- `ConversationLogsTextDeliveryFailure`

The success metrics show that Amazon Lex V2 has successfully written your audio or text logs to their destinations.

The failure metrics show that Amazon Lex V2 couldn't deliver audio or text logs to the specified destination. Typically, this is a configuration error. When your failure metrics are above zero, check the following:

- Make sure that Amazon Lex V2 is a trusted entity for the IAM role.
- For text logging, make sure that the CloudWatch Logs log group exists. For audio logging, make sure that the S3 bucket exists.
- Make sure that the IAM role that Amazon Lex V2 uses to access the CloudWatch Logs log group or S3 bucket has write permission for the log group or bucket.
- Make sure that the S3 bucket exists in the same region as the Amazon Lex V2 bot and belongs to your account.

Obscuring slot values in conversation logs from Lex V2

Amazon Lex V2 enables you to obfuscate, or hide, the contents of slots so that the content is not visible. To protect sensitive data captured as slot values, you can enable slot obfuscation to mask those values for logging.

When you choose to obfuscate slot values, Amazon Lex V2 replaces the value of the slot with the name of the slot in conversation logs. For a slot called `full_name`, the value of the slot would be obfuscated as follows:

```
Before:
    My name is John Stiles
After:
    My name is {full_name}
```

If an utterance contains bracket characters ({}) Amazon Lex V2 escapes the bracket characters with two back slashes (\\). For example, the text `{John Stiles}` is obfuscated as follows:

```
Before:
    My name is {John Stiles}
After:
    My name is \\{{full_name}\\}
```

Slot values are obfuscated in conversation logs. The slot values are still available in the response from the `RecognizeText` and `RecognizeUtterance` operations, and the slot values are available to your validation and fulfillment Lambda functions. If you are using slot values in your prompts or responses, those slot values are not obfuscated in conversation logs.

In the first turn of a conversation, Amazon Lex V2 obfuscates slot values if it recognizes a slot and slot value in the utterance. If no slot value is recognized, Amazon Lex V2 does not obfuscate the utterance.

On the second and later turns, Amazon Lex V2 knows the slot to elicit and if the slot value should be obfuscated. If Amazon Lex V2 recognizes the slot value, the value is obfuscated. If Amazon Lex V2 does not recognize a value, the entire utterance is obfuscated. Any slot values in missed utterances won't be obfuscated.

Amazon Lex V2 also doesn't obfuscate slot values that you store in request or session attributes. If you are storing slot values that should be obfuscated as an attribute, you must encrypt or otherwise obfuscate the value.

Amazon Lex V2 doesn't obfuscate the slot value in audio. It does obfuscate the slot value in the audio transcription.

You can choose which slots to obfuscate by using the console or by using the Amazon Lex V2 API. In the console, choose **Slot obfuscation** in the settings for a slot. If you are using the API, set the `obfuscationSetting` field of the slot to `DEFAULT_OBFUSCATION` when you call the CreateSlot or UpdateSlot operation.

Selective conversation log capture in Lex V2

The selective conversation log capture allows the user to select how conversation logs are captured with text and audio data from the live conversations.

To enable and capture the output of the selective conversation log capture feature, you must activate the feature in the Amazon Lex V2 console, and enable the required session attributes in the API settings to capture the selected output from the logs.

You can select the following options for the selective conversation log capture:

- text only
- audio only
- text and audio

You can capture specific parts of the conversation, and choose if audio, text, or both are captured for the conversation log.

> **Note**
>
> Selective conversation log capture works for Amazon Lex V2 only.

Topics

Manage selective conversation log capture

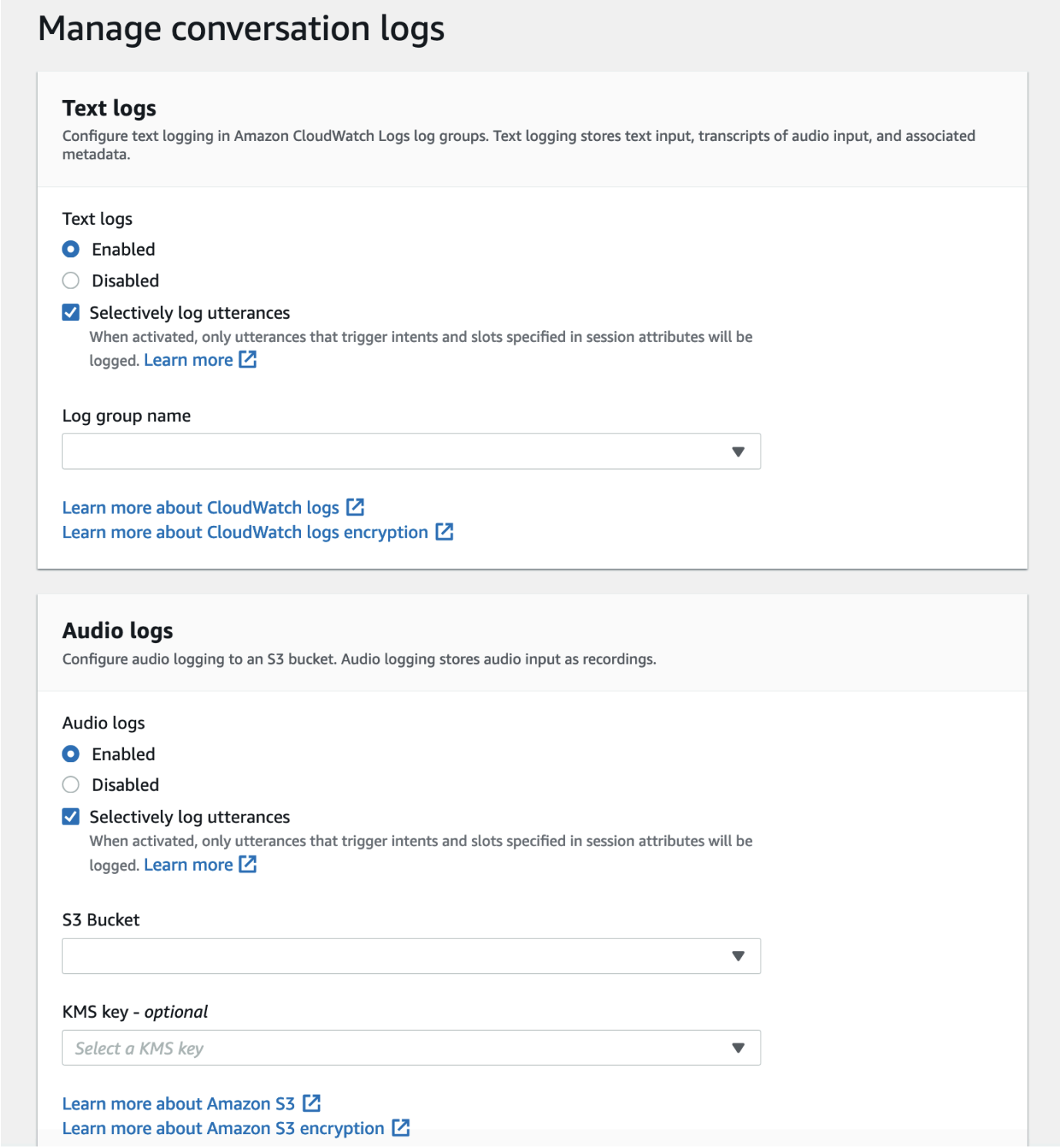
Using the Lex console, you can enable the selective conversation log capture settings and choose which slots you want to enable selective conversation log capture capture for.

Activate selective conversation log capture in the Amazon Lex V2 console:

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. Select **Bots** from the left side panels and choose the bot you want to enable the selective conversation log capture. Use an existing bot or create a new one.
3. Choose **Aliases** for your selected bot under the **Deployment** section on the left side panel.
4. Choose your bot's Alias, then select **Manage conversation logs**.
5. In the **Manage conversation logs** panel, for **Text logs**, choose whether text logs are enabled or disable by selecting the radio button. If you choose **Enabled** for text logs, then you will need to enter a **Log group name** or choose an existing log group name from the drop down menu. Select the check box for **Selectively log utterances** if you are selectively logging text files.

   > **Note**
   >
   > Enable text and/or audio logs by selecting the **Selectively log utterances** check box in the **conversation logs settings** (text and/or audio) in build time **BotAlias** settings. You must configure the CloudWatch log group and Amazon S3 bucket to select this option.

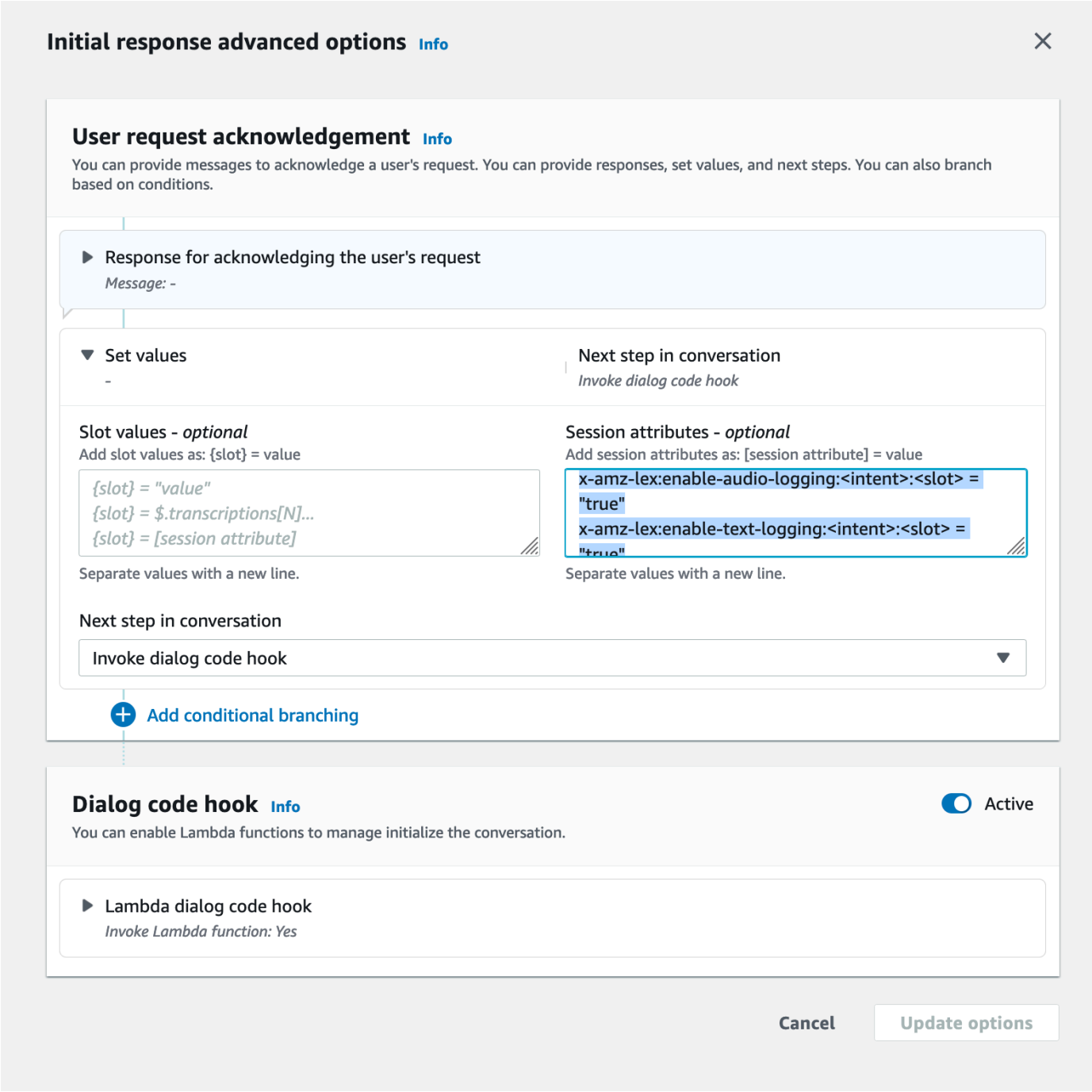
6. In the **Audio logs** section, choose whether audio logs are enabled or disable by selecting the radio button. If you choose **Enabled** for audio logs, you need to specify the Amazon S3 bucket location and (optional) the KMS key for encrypting your audio data. Select the check box for **Selectively log utterances** if you are selectively logging audio files.



7. Select **Save** in the bottom right corner of the panel to save your selective conversation log capture settings.

Activate selective conversation log capture in the Amazon Lex V2 console:

1. Go to **Intents** and select the **Intent name**, **Initial Response**, **Advanced Settings**, **Set Values**, **Session Attributes**.
2. Set the following attributes to based on the intents and slots for which you want to enable selective conversation log capture:
   - `x-amz-lex:enable-audio-logging:intent:slot = "true"`
   - `x-amz-lex:enable-text-logging:intent:slot = "true"`



Note

Set `x-amz-lex:enable-audio-logging:`*`intent`*`:`*`slot`*` = "true"` to capture utterances that contain only a specific slot in the conversation. The action to log an utterance depends on the assessment of *`intent`* :*`slot`* within the utterance, in

comparison to the session attribute expressions, and the corresponding flag value. To log an utterance, at least one expression in the session attribute must allow it, with the enable logging flag set to `true`. The value of *`intent`* and *`slot`* can be `"*"` as well. If the slot and/or intent value is `"*"`, it means that any slot and/or intent value of `"*"` will match with it. Similar to `x-amz-lex:enable-audio-logging`, a new session attribute called `x-amz-lex:enable-text-logging` will be used to control text logs.

3. Select **Update options** and build the bot to include the updated settings.

> **Note**
>
> Your IAM role must have access permission to allow you to write data to the Amazon S3 bucket and to use a KMS key to encrypt the data. Lex will update your IAM role with Lex permissions to access CloudWatch Logs log group and the selected Amazon S3 bucket.

Guidelines for using selective conversation log capture:

You can only enable selective conversation log capture for text and/or audio logs, when you have enabled text and/or audio logs in the **Conversation log settings**. By enabling selective conversation log capture for text and/or audio logs, you disable logging for all intents and slots in the conversation. To generate text and/or audio logs for particular intents and slots, you must set text or/and audio selective conversation log capture session attributes for those intents and slots to "true".

- If selective conversation log capture is enabled, and no session attributes with the prefix x-amz-lex:enable-audio-logging are present, logging will be disabled by default for all the utterances. This scenario is also true regarding x-amz-lex:enable-text-logging.
- Utterance logs will be stored exclusively for the segments of text and/or audio conversation if at least one expression in the session attribute allows it.
- The configurations for selective conversation log capture of text and/or audio, as defined in session attributes, will be effective only when selective conversation log capture for text and/or audio is enabled in Conversation Log Settings within bot alias; otherwise, session attributes will be disregarded.

- When selective conversation log capture is enabled, any slot values in SessionState, Interpretations, and Transcriptions for which logging is not enabled using session attributes will be obfuscated in the generated text log.
- The decision to produce audio and/or text logs is evaluated by matching the slot elicited by the bot with the selective conversation log capture session attributes, except for the intent elicitation turn where user can provide slot values along with intent elicitation. In an intent elicitation turn, the slots filled in current turn are matched against the selective conversation log capture session attributes.
- The slots that are considered filled are derived from the session state at the end of the turn. Therefore, any alterations made by the Dialog Codehook Lambda to the slots in session state will influence the behavior of selective conversation log capture.
- In an intent elicitation turn, if multiple slot values are given by the user, the text and/or audio log will only get generated if the text/audio session attributes allow logging for all the slots filled in this turn.
- The recommended operational approach is to set the selective conversation log capture session attribute at the beginning of the session and to refrain from modifying it during the session.
- If any slots contain sensitive data, you should always enable slot obfuscation.

Example of selective conversation log capture

Here is an example of a business use case for selective conversation log capture.

Use case:

A fintech company utilizes an Amazon Lex V2 bot to support their IVR system, which allows users to make bill payments. In order to meet compliance and auditing requirements, they must retain audio recordings of user-provided authorization consent. However, enabling general audio logs is not feasible as it would make them non-compliant, because it is not possible to obfuscate sensitive slots like CardNumber, CVV, and other information in the audio logs. Instead, they can enable selective conversation log capture for audio logs and set session attribute to only produce audio logs for utterance that has authorization consent.

BotAlias Settings:

- Text Logs Enabled : true
- Text Logs Selective Logging Enabled : false
- Audio Logs Enabled : true

- Audio Logs Selective Logging Enabled : true

Session Attributes:

```
x-amz-lex:enable-audio-logging:PayBill:AuthorizationConsent = "true"
```

Sample Conversation:

- User (Audio Input): "I want to pay my bill with bill number 35XU68."
- Bot: "What is the due amount in dollars?"
- User (Audio Input): "235."
- Bot: "What is your credit card number?"
- User (Audio Input): "9239829722200348."
- Bot: "You're paying 235 dollars using your credit card number ending with 0348. Please say 'I authorize to pay 235 dollars.'"
- User (Audio Input): "I authorize to pay 235 dollars."
- Bot: "Your bill has been paid."

Conversation Logs output:

In this situation, text logs will be produced for all turns. However, audio logs will only be recorded for the particular turn when the **AuthorizationConsent** slot within the **PayBill** intent was elicited, and no audio logs will be produced for any other turn.

Logging errors with error logs in Lex V2

You enable *error logs* to store bot interactions. You can use these error logs to review the performance of your bot and to troubleshoot errors with conversations.

Error logs are configured for an version. Each version can have different settings for their error logs. Text logs store text input in CloudWatch Logs. You can enable encryption of text logs using AWS KMS customer managed CMKs.

IAM Policies for Error Logs

Depending on the type of logging that you select, Amazon Lex V2 requires permission to use Amazon CloudWatch Logs and Amazon Simple Storage Service (S3) buckets to store your logs. You

must create AWS Identity and Access Management roles and permissions to enable Amazon Lex V2 to access these resources.

Creating an IAM Role and Policies for Error Logs

To enable conversation logs, you must grant write permission for CloudWatch Logs and Amazon S3. If you enable object encryption for your S3 objects, you need to grant access permission to the AWS KMS keys used to encrypt the objects.

You can use the IAM console, the IAM API, or the AWS Command Line Interface to create the role and policies. These instructions use the AWS CLI to create the role and policies.

To create an IAM role for error logs

The IAM role that you use to enable conversation logs must have the iam:PassRole permission. The following policy should be attached to the role:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::111122223333:role/role"
        }
    ]
}
```

Enabling Error Logs in Lex V2

To turn on error logs using the Amazon Lex V2 console:

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of **Bots**, choose the bot you want to enable for error logs.
3. From the left menu, choose **Version**.
4. In the list of **Version**, choose the Version for which you want to configure error logs.

5. In the **Version detail** section, choose **Enable**.
6. Choose **Save** to start logging conversations. If necessary, Amazon Lex V2 will update your service role with permissions to access the CloudWatch Logs log group.

Disabling Error Logs in Lex V2

To turn off error logs using the Amazon Lex V2 console:

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the list of **Bots**, choose the bot you want to enable for error logs.
3. From the left menu, choose **Version**.
4. In the list of **Version**, choose the Version for which you want to configure error logs.
5. In the **Version detail** section, choose **Disable**.
6. Choose **Save** to stop logging conversations.

Monitoring operational metrics in Lex V2

Amazon CloudWatch and AWS CloudTrail are two AWS services that integrate with Amazon Lex V2 to help you monitor user interactions with your bot. Use these services to record actions, send near real-time data, and set up notifications and automated actions when criteria are met.

Topics

Measuring operational metrics with Amazon CloudWatch

You can monitor Amazon Lex V2 using CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your web application or service is performing. You can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the Amazon CloudWatch User Guide.

The Amazon Lex V2 service reports the following metrics in the AWS/Lex namespace.

Metric	Description
AssistedSlotResolutionModelAccessDeniedErrorCount	The number of times that Amazon Lex V2 was denied access to Amazon Bedrock Valid dimensions for the RecognizeUtterance and StartConversation operations: • BotId, BotAliasId, LocaleId, Operation, InputMode, ModelType, Model • BotId, BotVersion, LocaleId, Operation, InputMode, ModelType, Model Valid dimensions for RecognizeText : • BotId, BotAliasId, LocaleId, Operation, ModelType, Model • BotId, BotVersion, LocaleId, Operation, ModelType, Model Unit: Count
AssistedSlotResolutionModelInvocationCount	The number of times that Amazon Bedrock was invoked. Valid dimensions for the RecognizeUtterance and StartConversation operations: • BotId, BotAliasId, LocaleId, Operation, InputMode, ModelType, Model • BotId, BotVersion, LocaleId, Operation, InputMode, ModelType, Model Valid dimensions for RecognizeText : • BotId, BotAliasId, LocaleId, Operation, ModelType, Model • BotId, BotVersion, LocaleId, Operation, ModelType, Model Unit: Count

Metric	Description
AssistedSlotResolutionModelSystemErrorCount	The number of times that a 5xx occurred when calling Amazon Bedrock. Valid dimensions for the RecognizeUtterance and StartConversation operations: • BotId, BotAliasId, LocaleId, Operation, InputMode, ModelType, Model • BotId, BotVersion, LocaleId, Operation, InputMode, ModelType, Model Valid dimensions for RecognizeText : • BotId, BotAliasId, LocaleId, Operation, ModelType, Model • BotId, BotVersion, LocaleId, Operation, ModelType, Model Unit: Count
AssistedSlotResolutionModelThrottlingErrorCount	The number of times that Amazon Lex was throttled by Amazon Bedrock. Valid dimensions for the RecognizeUtterance and StartConversation operations: • BotId, BotAliasId, LocaleId, Operation, InputMode, ModelType, Model • BotId, BotVersion, LocaleId, Operation, InputMode, ModelType, Model Valid dimensions for RecognizeText : • BotId, BotAliasId, LocaleId, Operation, ModelType, Model • BotId, BotVersion, LocaleId, Operation, ModelType, Model Unit: Count

Metric	Description
AssistedSlotResolutionResolvedSlotCount	The number of times that Amazon Bedrock returned a slot value. Valid dimensions for the RecognizeUtterance and StartConversation operations: • BotId, BotAliasId, LocaleId, Operation, InputMode, ModelType, Model • BotId, BotVersion, LocaleId, Operation, InputMode, ModelType, Model Valid dimensions for RecognizeText : • BotId, BotAliasId, LocaleId, Operation, ModelType, Model • BotId, BotVersion, LocaleId, Operation, ModelType, Model Unit: Count
KendraIndexAccessError	The number of times that Amazon Lex V2 could not access your Amazon Kendra index. • Operation, BotId, BotAliasId, LocaleId Unit: Count
KendraLatency	The amount of time that it takes Amazon Kendra to respond to a request from the AMAZON.KendraSearchIntent . Valid dimensions: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Milliseconds

Metric	Description
KendraSuccess	The number of times that Amazon Lex V2 couldn't access your Amazon Kendra index. Valid dimensions: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Count
KendraSystemErrors	The number of times that Amazon Lex V2 couldn't query the Amazon Kendra index. Valid dimensions: • Operation, BotId, BotAliasId, InputMode, LocaleId Unit: Count
KendraThrottledEvents	The number of times Amazon Kendra throttled requests from the AMAZON.KendraSearchIntent . Valid dimensions: • Operation, BotId, BotAliasId, InputMode. LocaleId Unit: Count

Metric	Description
RuntimeConcurrency	The number of concurrent connections in the specified time period. RuntimeConcurrency is reported as a StatisticSet . Valid dimensions for the RecognizeUtterance or StartConversation operations: • Operation, BotId, BotVersion, InputMode, LocaleId • Operation, BotId, BotAliasId, InputMode, LocaleId Valid dimensions for other operations: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Count
RuntimeInvalidLambdaResponses	The number of invalid AWS Lambda responses in the specified period. Valid dimensions: • Operation, BotId, BotAliasId, InputMode, LocaleId Unit: Count
RuntimeLambdaErrors	The number of Lambda runtime errors in the specified time period. Valid dimensions: • Operation, BotId, BotAliasId, InputMode, LocaleId Unit: Count

Metric	Description
RuntimePollyErrors	The number of invalid Amazon Polly responses in the specified time period. Valid dimensions: • Operation, BotId, BotAliasId, InputMode, LocaleId Unit: Count
RuntimeRequestCount	The number of runtime requests in the specified time period. Valid dimensions for the RecognizeUtterance and StartConversation operations: • Operation, BotId, BotVersion, InputMode, LocaleId • Operation, BotId, BotAliasId, InputMode, LocaleId Valid dimensions for other operations: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Count
RuntimeRequestLength	Total length of a conversation with an Amazon Lex V2 bot. Only applicable to the StartConversation operation. Valid dimensions: • BotAliasId, BotId, LocaleId, Operation • BotId, BotAliasId, LocaleId, Operation Unit: milliseconds

Metric	Description
RuntimeSuccessfulRequestLatency **Important** This metric is RuntimeSuccessfulRequestLatency and not RuntimeSuccessfulRequestLatency .	The latency for successful requests between the time the request was made and the response was passed back. Valid dimensions for the RecognizeUtterance and StartConversation operations: • Operation, BotId, BotVersion, InputMode, LocaleId • Operation, BotId, BotAliasId, InputMode, LocaleId Valid dimensions for other operations: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: milliseconds
RuntimeSystemErrors	The number of system errors in the specified period. The response code range for a system error is 500 to 599. Valid dimensions for the RecognizeUtterance and StartConversation operations: • Operation, BotId, BotVersion, InputMode, LocaleId • Operation, BotId, BotAliasId, InputMode, LocaleId Valid dimensions for other operations: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Count

Metric	Description
RuntimeThrottledEvents	The number of throttled events. Amazon Lex V2 throttles an event when it receives more requests than the limit of transactions per second set for your account. If the limit set for your account is frequently exceeded, you can request a limit increase. To request an increase, see AWS service limits. Valid dimensions for the RecognizeUtterance and StartConversation operations: • Operation, BotId, BotVersion, InputMode, LocaleId • Operation, BotId, BotAliasId, InputMode, LocaleId Valid dimensions for other operations: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Count
RuntimeUserErrors	The number of user errors in the specified period. The response code range for a user error is 400 to 499. Valid dimensions for the RecognizeUtterance and StartConversation operations: • Operation, BotId, BotVersion, InputMode, LocaleId • Operation, BotId, BotAliasId, InputMode, LocaleId Valid dimensions for other operations: • Operation, BotId, BotVersion, LocaleId • Operation, BotId, BotAliasId, LocaleId Unit: Count

The following dimensions are supported for the Amazon Lex V2 metrics.

Dimension	Description
`Operation`	The name of the Amazon Lex V2 operation – `RecognizeText` , `RecognizeUtterance` , `StartConversation` , `GetSession` , `PutSession` , `DeleteSession` – that generated the entry.
`BotId`	The alphanumeric unique identifier for the bot.
`BotAliasId`	The alphanumeric unique identifier for the bot alias.
`BotVersion`	The numeric version of the bot.
`InputMode`	The type of input to the bot – speech, text, or DTMF.
`LocaleId`	The identifier of the bot's locale, such as en-US or fr-CA.
`Model`	Indicates the model id of the Amazon Bedrock large language model.
`ModelType`	Indicates the type of large language model invoked from Amazon Bedrock.

Viewing events with AWS CloudTrail for Lex V2

Amazon Lex V2 is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Amazon Lex V2. CloudTrail captures API calls for Amazon Lex V2 as events. The calls captured include calls from the Amazon Lex V2 console and code calls to the Amazon Lex V2 API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Amazon Lex V2. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Amazon Lex V2, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Amazon Lex V2 information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon Lex V2, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for Amazon Lex V2, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

Amazon Lex V2 supports logging for all of the actions listed in Model Building API V2.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Understanding Amazon Lex V2 log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of

the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the CreateBotAlias action.

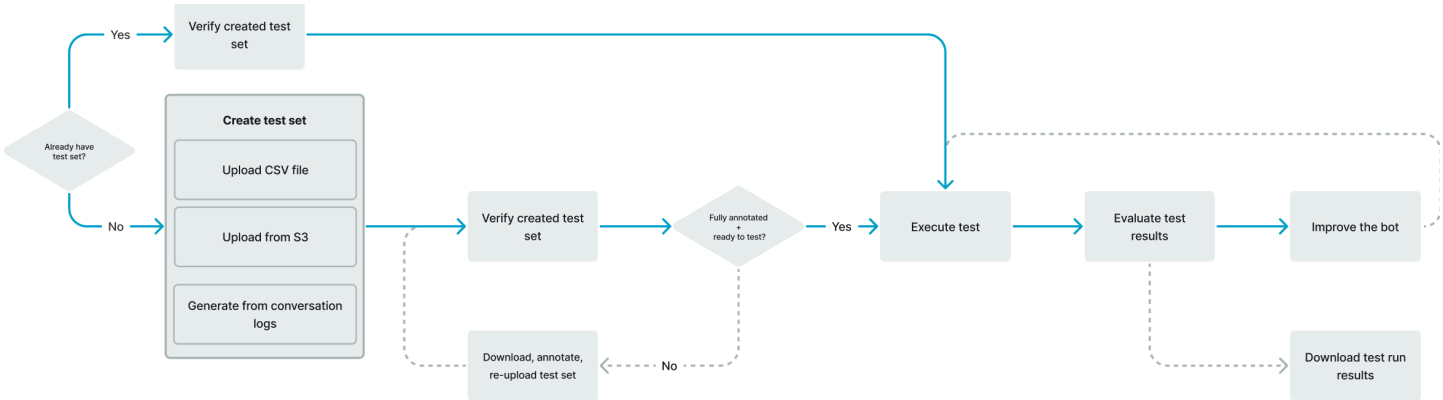
```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "ID of caller:temporary credentials",
        "arn": "arn:aws:sts::111122223333:assumed-role/role name/role ARN",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "ID of caller",
                "arn": "arn:aws:iam::111122223333:role/role name",
                "accountId": "111122223333",
                "userName": "role name"
            },
            "webIdFederationData": {},
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "creation date"
            }
        }
    },
    "eventTime": "event timestamp",
    "eventSource": "lex.amazonaws.com",
    "eventName": "CreateBotAlias",
    "awsRegion": "Region",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "user agent",
    "requestParameters": {
        "botAliasLocaleSettingsMap": {
            "en_US": {
                "enabled": true
            }
        },
        "botId": "bot ID",
        "botAliasName": "bot aliase name",
        "botVersion": "1"
    },
```

```
    "responseElements": {
        "botAliasLocaleSettingsMap": {
            "en_US": {
                "enabled": true
            }
        },
        "botAliasId": "bot alias ID",
        "botAliasName": "bot alias name",
        "botId": "bot ID",
        "botVersion": "1",
        "creationDateTime": creation timestamp
    },
    "requestID": "unique request ID",
    "eventID": "unique event ID",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```

Evaluating Lex V2 bot performance with the Test Workbench

To improve bot performance, you can evaluate the performance of your bots at scale. The results for your test evaluation are displayed in simple tables and charts.

You can use the Test Workbench to create reference test sets that use existing transcription data. You can test bots to evaluate performance before deployment, and view test result breakdowns at scale.



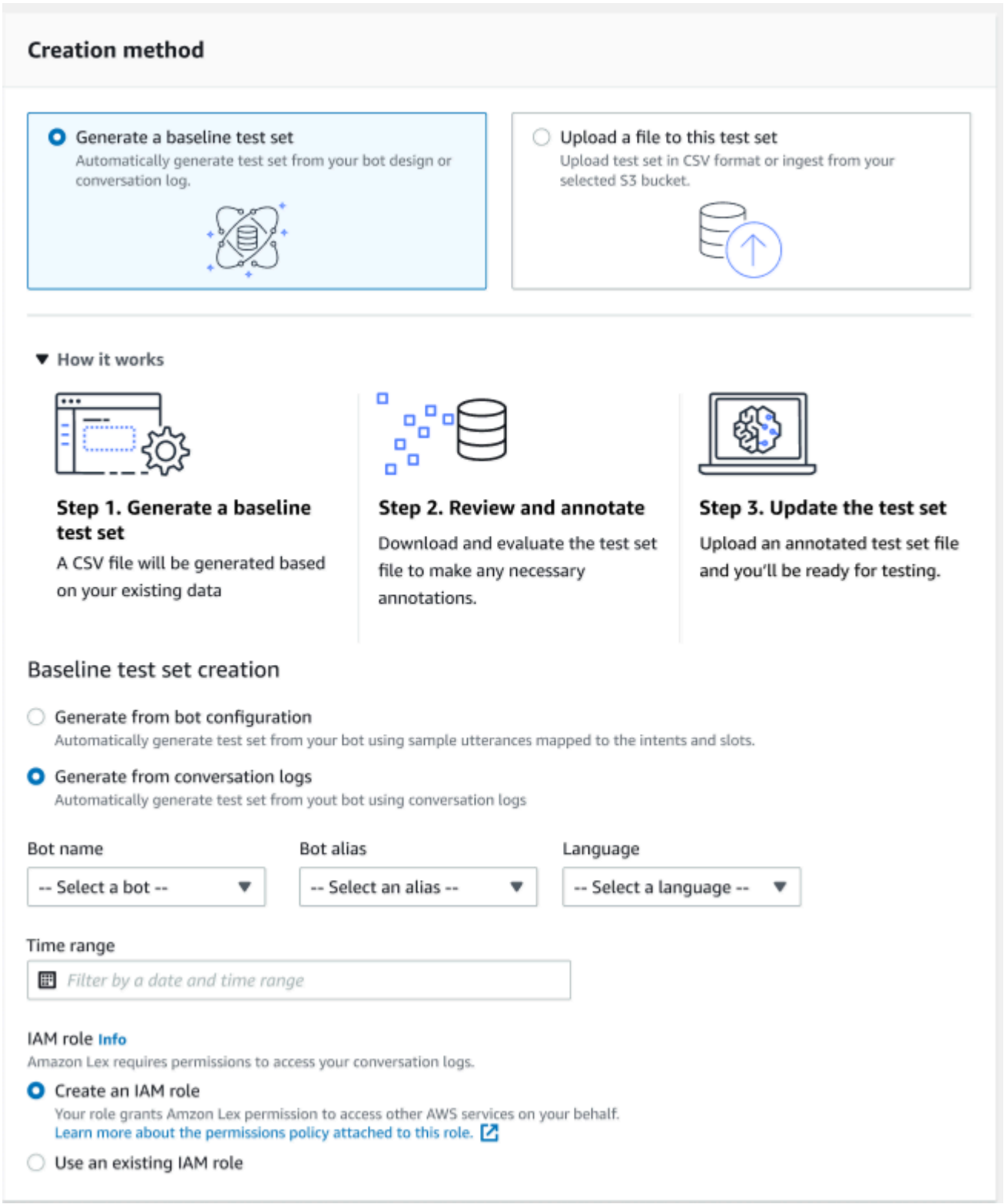
Users can use the Test Workbench to establish baseline performance for bots. This covers intent and slot performance for utterances that are in the form of single-inputs or conversations. Once

a test set is successfully loaded, you can run it against your existing pre-production or production bots. The Test Workbench helps you identify opportunities for improved slot filling and intent classification.

Topics

Generate a test set for Test Workbench

You can create a test set to evaluate the performance of your bot. Generate a test set by uploading a test set that is in a CSV file format or by generating a test set from conversation logs. The test set can contain audio or text input.



If a test set creates validation errors, remove the test set and replace it with another list of test set data, or edit the data in the CSV file by using a spreadsheet editing program.

To create a test set:

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Test workbench** from the left side panel.
3. Select **Test sets** from the options under Test workbench.
4. Select the **Create test set** button on the console.
5. In the **Details**, enter a test set name and an optional description.
6. Select **Generate a baseline test set**.
7. Select **Generate from conversation logs**.
8. Select **Bot name**, **Bot Alias**, and **Language** from the drop down menus.
9. If you are generating a baseline test from a conversation log, choose **Time range** and **IAM role**, if required. You can create a role with the basic Amazon Lex V2 permissions or use an existing role.
10. Choose a modality of **Audio** or **Text** for the test set you are creating. NOTE: The Test Workbench can import text files up to 50k, and up to 5 hours of audio.
11. Select an Amazon S3 location to store your test results, and add an optional KMS key to encrypt output transcripts.
12. Select **Create**.

To upload an existing test set in a CSV file format, or to update the test set:

1. Choose **Test workbench** from the left side panel.
2. Select **Test sets** from the options under Test workbench.
3. Select **Upload a file to this test set** on the console.
4. Choose **Upload from Amazon Amazon S3 bucket** or **Upload from your computer**. NOTE: You can upload a CSV file created from a template. Click **CSV template** to download a zip file that contains the templates.
5. Choose **Create a role with basic Amazon Lex permissions** or **Use an existing role for Role ARN**.
6. Choose a modality of **Audio** or **Text** for the test set you are creating. NOTE: The Test Workbench can import text files up to 50k, and up to 5 hours of audio.

7. Select an Amazon S3 location to store your test results, and add an optional KMS key to encrypt output transcripts.
8. Select **Create**.

If the operation is successful, the confirmation message will indicate that the test set is ready to test, and the status will display **Ready for testing**.

Tips for creating a successful test set

- You can create an IAM role for the Test Workbench in the console, or you can configure your IAM role step-by-step. For more information, see Create an IAM role for the Test Workbench.
- Before executing a test, validate the test set and the bot definition for any inconsistencies using the **Validate discrepancy** button. If the intent and slot naming conventions used in the test set are consistent with the bot, proceed to execute the test. If any anomalies are identified, revise the test set, update the test set, and choose **Validate discrepancy**. Repeat this sequence again until no inconsistencies are noted, then execute the test.
- The Test Workbench can test with different slot value formats in the **Expected Output Slot** column. For any built-in slot, you can choose the value provided in the user input (for example, Date = tomorrow), or provide its absolute resolved value (for example, Date = 2023-03-21). For more information around built-in slots and their absolute values, see Built-in slots.
- For consistency and readability in the **Expected Output Slot** columns, follow the convention of "SlotName = SlotValue" (for example, AppointmentType = cleaning) with a space before and after the equal sign.
- If the bot includes composite slots, in **Expected Output Slot** define subslots to the slot name, separated by a period (for example, “Car.Color”). No other syntax and punctuation will work.
- If the bot includes multi-value slots, in **Expected Output Slot** provide multiple slot values, separated by a comma ("FlowerType = roses, lilies"). No other syntax and punctuation will work.
- Make sure that the test set is created from valid conversation logs.
- Slot:slot value will be in the same column after the intent columns in the CSV format.
- DTMF input from a User turn is interpreted as an expected transcription and does not list an Amazon S3 location.

Creating a test case within a test set using Test Workbench

The Test Workbench results are dependent on the bot definition and its corresponding test set. You can generate a test set with the information from the bot definition to pinpoint areas that need improvement. Create a test dataset with examples that you suspect (or know) will be challenging for the bot to interpret correctly considering the current bot design and your knowledge of your customer conversations.

Review your intents based on learnings from your production bot on a regular basis. Continue to add to and adjust the bot's sample utterances and slot values. Consider improving slot resolution by using the available options, such as runtime hints. The design and development of your bot is an iterative process that is a continuous cycle.

Here are some other tips for optimizing your test set:

- Select the most common use cases with frequently used intents and slots in the test set.
- Explore different ways a customer could refer to your intents and slots. This can include user inputs in the forms of statements, questions, and commands that vary in length from minimal to extended.
- Include user inputs with a varied number of slots.
- Include commonly used synonyms or abbreviations of custom slot values supported by your bot (for example, “root canal”, “canal”, or “RC”).
- Include variations of built-in slot values (for example, “tomorrow”, “asap”, or "the next day").
- Examine the bot robustness for spoken modality by collecting user inputs that can be misinterpreted (for example, “ink”, “ankle”, or "anchor").

Creating a test set from a CSV file for Test Workbench

You can create a test set from the CSV file template provided in the Amazon Lex V2 console by entering the values directly by using a CSV spreadsheet editor. The test set is a comma-separated value (CSV) file consisting of single user utterances and multi-turn conversations recorded in the following columns:

- **Line #** – this column is an incremental counter that keeps track of the total filled rows to test.
- **Conversation #** – this column tracks the number of turns in a conversations. For single inputs, this column can be left empty, filled with "-" or "N/A". For conversations, each turn within a conversations will be assigned the same conversation number.

- **Source** – this column is set to "User" or "Agent". For single inputs, it will be always set to "User".
- **Input** – this column includes the user utterance or the bot prompts.
- **Expected Output Intent** – this column captures the intent fulfilled in the input.
- **Intent Expected Output Slot 1** – this column captures the first slot elicited in the user input. The test set should include a column called Expected Output Slot X for each slot in the user input.

Example of a test set with single inputs:

Line #	Conversation #	Source	Input	Expected Output Intent	Expected Output Slot 1	Expected Output Slot 2
1		User	book a cleaning appointment tomorrow	MakeAppointment	AppointmentType = cleaning	Date = tomorrow
2	N/A	User	book a cleaning appointment on April 15th	MakeAppointment	AppointmentType = cleaning	Date = 4/15/23
3	N/A	User	book appointment for December first	MakeAppointment	Date = December first	
4	N/A	User	book a cleaning appointment	MakeAppointment	AppointmentType = cleaning	
1		User	Can you help me	MakeAppointment		

Line #	Conversation #	Source	Input	Expected Output Intent	Expected Output Slot 1	Expected Output Slot 2
			book an appointment?			

Example of a test set with conversations

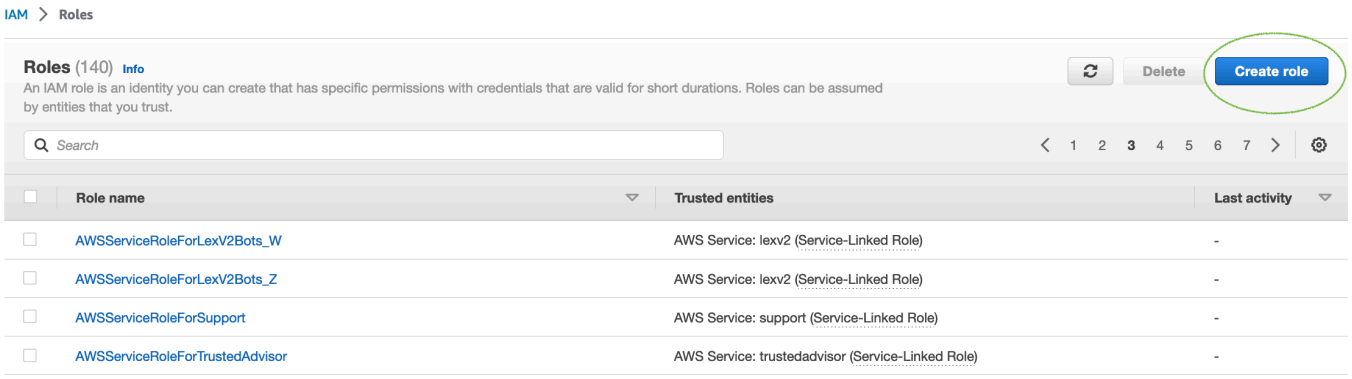
Line #	Conversation #	Source	Input	Expected Output Intent	Expected Output Slot 1	Expected Output Slot 2	Expected Output Slot 3
1	1	User	book an appointment	MakeAppointment			
2	1	Agent	What type of appointment would you like to schedule?	MakeAppointment			
3	1	User	cleaning	MakeAppointment	AppointmentType = cleaning		
4	1	Agent	When should I schedule your appointment?	MakeAppointment			

Line #	Conversation #	Source	Input	Expected Output Intent	Expected Output Slot 1	Expected Output Slot 2	Expected Output Slot 3
5	1	User	tomorrow	MakeAppointment		Date = tomorrow	
6	2	User	book a root canal appointment today	MakeAppointment	AppointmentType = root canal	Date = today	
7	2	Agent	At what time should I schedule your appointment?	MakeAppointment			
8	2	User	eleven a.m.	MakeAppointment			Time = eleven a.m.

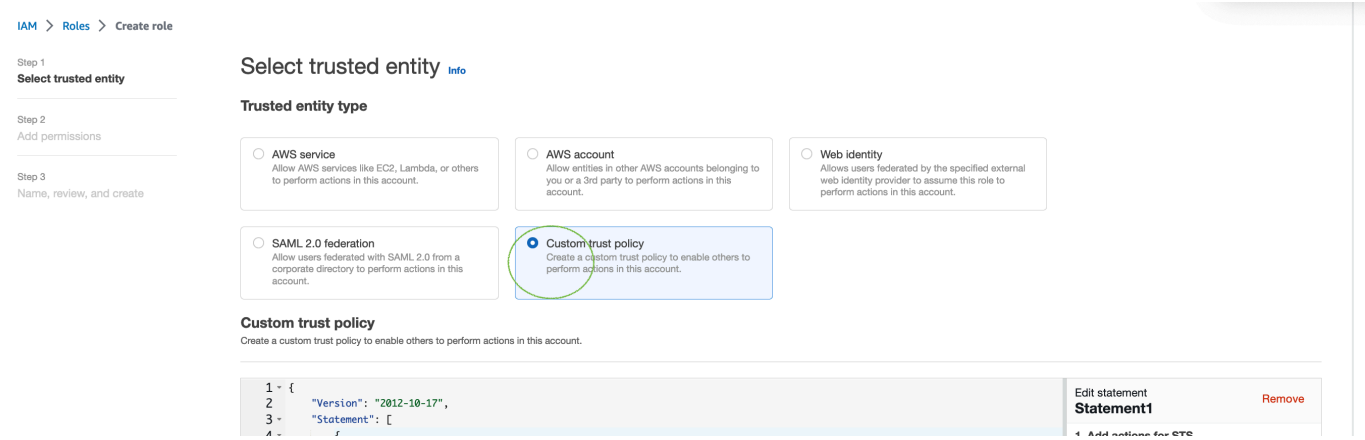
Create an IAM role for the Test Workbench

To create an IAM role for the Test Workbench

1. Follow the steps at Create an IAM user to create an IAM user which can be used to access test-workbench console.
2. Select the **Create role** button.



3. Select the option for **Custom trust policy**.



4. Enter the trust policy below and click **Next**.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "sid4",
      "Effect": "Allow",
      "Principal": {
        "Service": "lexv2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

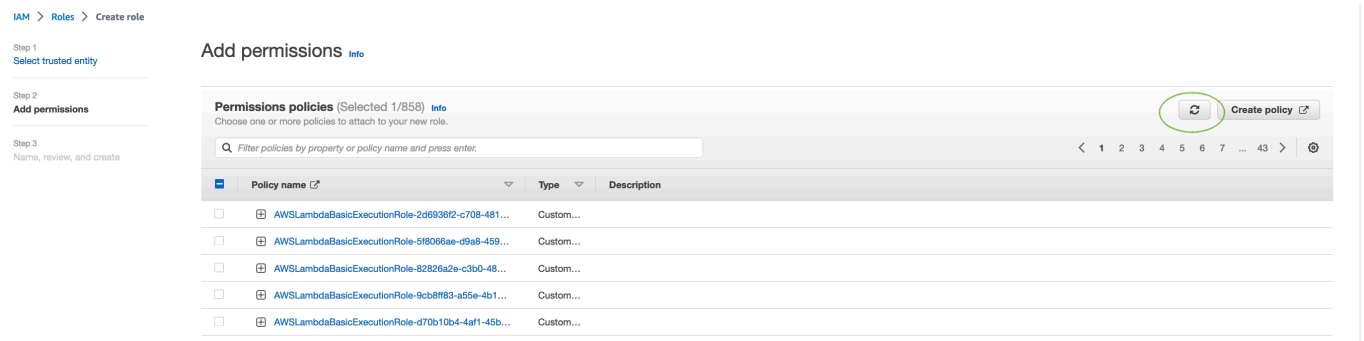
5. Select the **Create policy** button.

6. A new tab will open in your browser where you can enter the below policy and click on **Next: Tags** button.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:*"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:FilterLogEvents"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "lex:*"
            ],
            "Resource": "*"
        }
    ]
}
```

7. Enter a policy name, for example 'LexTestWorkbenchPolicy' and then click on the **Create Policy**.
8. Return to the previous tab in your browser and Refresh list of policies by clicking the **Refresh** button as shown below.



9. Search in list of policies by entering policy name that you used in the 6th step and choose the policy.
10. Select the **Next** button.
11. Enter role name and then click the **Create Role** button.
12. Choose your new IAM role when prompted in the Amazon Lex V2 console for Test Workbench.

Create an IAM role for the Test Workbench - Advanced Features

Permission setup for Test workbench IAM role

This section shows several example AWS Identity and Access Management (IAM) identity-based policies to implement least-privilege access controls for Test Workbench permissions.

1. **Policy for Test Workbench to read audio files in S3** – This policy enables Test Workbench to read audio files being used in the test sets. The below policy should be accordingly modified to update *S3BucketName* and *S3Path* to point them to an Amazon S3 location of the audio files in a test set.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "TestWorkbenchS3AudioFilesReadOnly",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion"
      ],
```

```
      "Resource": [
        "arn:aws:s3:::S3BucketName/S3Path/*"
      ]
    }
  ]
}
```

2. **Policy for Test Workbench to read and write test sets and results into an Amazon S3 bucket** – This policy enables Test Workbench to store the test set inputs and results. The below policy should be modified to update *S3BucketName* to the Amazon S3 Bucket where test-set data will be stored. Test Workbench stores these data exclusively in your Amazon S3 bucket and not in the Lex Service infrastructure. Therefore For this reason, Test Workbench requires access to your Amazon S3 bucket to function properly.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "TestSetDataUploadWithEncryptionOnly",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::S3BucketName/*/lex_testworkbench/test_set/*",
        "arn:aws:s3:::S3BucketName/*/lex_testworkbench/test_execution/*",
        "arn:aws:s3:::S3BucketName/*/lex_testworkbench/
test_set_discrepancy_report/*"
      ],
      "Condition": {
        "StringEquals": {
          "s3:x-amz-server-side-encryption": "aws:kms"
        }
      }
    },
    {
      "Sid": "TestSetDataGetObject",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
```

```
        "s3:GetObjectVersion"
      ],
      "Resource": [
        "arn:aws:s3:::S3BucketName/*/lex_testworkbench/test_set/*",
        "arn:aws:s3:::S3BucketName/*/lex_testworkbench/test_execution/*",
        "arn:aws:s3:::S3BucketName/*/lex_testworkbench/
test_set_discrepancy_report/*"
      ]
    },
    {
      "Sid": "TestSetListS3Objects",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::S3BucketName"
      ]
    }
  ]
}
```

3. **Policy for Test Workbench to read CloudWatch Logs** – This policy enables Test Workbench to generate test-sets from Lex Conversation Text Logs stored in Amazon CloudWatch Logs. The below policy should be modified to update *Region*, *AwsAccountId*, *LogGroupName*.
4. **Policy for Test Workbench to call Lex Runtime** – This policy enables Test Workbench to execute a test set against Lex bots. The below policy should be modified to update *Region*, *AwsAccountId*, *BotId*. Since Test Workbench can test any bot in your Lex environment, you can replace the resource with "arn:aws:lex:*Region*:*AwsAccountId*:bot-alias/*" to allow Test Workbench access to all Amazon Lex V2 bots in an account.
5. **(Optional) Policy for Test Workbench to encrypt and decrypt test set data** – If Test Workbench is configured to store test-set inputs and results in Amazon S3 buckets using a customer managed KMS key, Test Workbench will need both encryption and decryption permission to the KMS key. The below policy should be modified to update *Region*, *AwsAccountId*, and *KmsKeyId* where *KmsKeyId* is the ID of the customer managed KMS key.
6. **(Optional) Policy for Test Workbench to decrypt audio files** – If Audio files are stored in the S3 bucket using customer managed KMS key, Test Workbench will need decryption permission to the KMS keys. The below policy should be modified to update *Region*, *AwsAccountId*, and *KmsKeyId* where *KmsKeyId* is the ID of the customer managed KMS key used to encrypt the audio files.

Manage test sets

You can download, update, and delete test sets from the test set window. Or you can use the list of available test sets to edit or manually annotate your test set file. Then, upload it again to retry validation, due to errors or other input issues.

To download the test set file from test set record:

1. Select the name of the test set from the list of test sets.
2. In the test set record window, select the **Download** button on the right side of the screen in the **Test Inputs** section.
3. if there are any validation error details at the top of the window regarding the test set, select the **Download** button. The file will be saved to your Downloads folder. You can fix the validation errors in the test set from the error messages in the test set CSV file. Find the error identified in the validation step, fix the line or remove it, and upload the file to retry the validation step.
4. if you successfully download the test set, a green banner messages will appear.

To download a test set from the list of test sets:

1. From the list of test sets, select the radio button next to the test set item you want to download.
2. From the Action menu at the top right, choose **Download**.
3. A green banner message will indicate if you successfully have downloaded the test set. The file will be saved to your Downloads folder.

Test set columns supported by Test Workbench

Below is the complete list of test set columns supported by Test Workbench and the instruction on how to use them with Amazon Lex V2.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
Line Number	Text and Audio	Number	No	This is a user column which is ignored by Amazon Lex V2. It is intended to help a test set author to sort and filter the test-set rows. "Line #" can be used as an alternative column name.
Conversation Number	Text and Audio	Number	No	This column allows you to put rows in a conversat ion together. "Conversation #" can be used as an alternative column name.
Source	Text and Audio	Enum ("User", "Agent")	No	The value in this column indicates if the row is for a user or an agent. "Conversation Participant" can be used as

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				an alternative column name.
Input	Text	String	No	This column is used to add the transcrip t for text test set. Text input should be used in User rows. The Agent prompt should be used in Agent rows.
Expected Transcription	Audio	String	No	This column is used to add the transcript for the audio test set. Expected transcription of the audio file should be used in User rows with audio input. DTMF input can be used in User rows with DTMF input. The Agent prompt should be used in Agent rows.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
S3 Audio Location	Audio	String	No	This column is used to add the audio file location and is applicable only to audio test sets. The S3 path should be used in the User rows with the audio input. This field should be left empty in User rows with DTMF input. This field should be left empty in Agent rows.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
Input Context Tag	Text and Audio	String	Yes	This column is used to provide name of an input context which will be used in input to Lex while executing the row in the test set. This refers to input context in Setting intent context for your Lex V2 bot. Note that Test Workbench only supports name of context. It does not support the parameters in a context. Multiple columns named such as 'Input Context Tag 1', 'Input Context Tag 2', and so on, may be used.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
Request Attribute	Text and Audio	String	Yes	This column is used to provide a request attribute which will be used in input to Lex while executing the row in the test set. Value in a column should be provided in format `<request-attribute-name> = <request-attribute-value>`. Spaces can be added around '=' for readability. For example: request-attribute-foo = this is a dummy response request-attribute-foo = 'this is a "dummy response" ' request-attribute-foo = "this is a 'dummy

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				response' ". Multiple columns named such as 'Request Attribute 1', 'Request Attribute 2', and so on, may be used.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
Session Attribute	Text and Audio	String	Yes	This column is used to provide a session attribute which will be used in input to Lex while executing the row in the test set. • Value in a column should be provided in format `<session-attribute-name> = <session-attribute-value>`. Spaces can be added around '=' for readability. • Examples: • session-attribute-foo = this is a dummy response • session-attribute-

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				foo = 'this is a "dummy response"' • session-attribute-foo = "this is a 'dummy response'" • Multiple columns named such as: 'Session Attribute 1', 'Session Attribute 2', and so on, may be used.

Column Name	Test set type	Value Type	Multiple Columns Support	Description

RunTime Hint	Text and Audio	String	Yes	This column is used to provide a Runtime Hint for a slot within an intent which will be used in input to Lex while executing the row in the test set. Below are examples:
				• Value in a column should be provided in format `<intent-name>.<slot -name> = < slot-value>`. Spaces may be added around '=' for readability.
				• Examples:
				  • IntentNameFoo.slotNameFoo = a dummy value
				  • IntentNameFoo.slot

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				NameFoo = 'a "dummy value"' • IntentNameFoo.slotNameFoo = "a 'dummy value'" • Test workbench does not support composite slots for runtime hints. • Multiple columns named such as 'RunTime Hint 1', 'RunTime Hint 2', and so on, may be used.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
Barge In	Audio	Boolean	No	This column is used specify if Test Workbench should barge-in when sending audio file to Lex Runtime for the row in the test set. • Only applicable for audio test set for the streaming API. • This column is ignored when executing a test set in non-streaming API mode.
Expected Output Intent	Text and Audio	String	No	This column is used specify name of an intent expected in output from Lex for the row in the test set.

Column Name	Test set type	Value Type	Multiple Columns Support	Description

Expected Output Slot	Text and Audio	String	Yes	This column is used to provide a slot value expected in output from Lex while executing the row in the test set.
				• Value in a column should be provided in format `<slot-name> = < slot-value>`. Spaces may be added around '=' for readability.
				• Examples for slots that are neither multi-value slots nor composite slots:
				  • slotNameFoo = a dummy value
				  • slotNameFoo = 'a

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				"dummy value"' • slotNameFoo = "a 'dummy value'" • Examples for multi-value slots: • slotNameFoo = value1, value2 • slotNameFoo = value1, "Foo's item" • slotNameFoo = value1, 'value2' • Examples for composite slots where slot name is ‘Car’ and sub slot name is ‘Make’: • Car.Make = Toyota • Car.Make = "Toyota" • Car.Make = 'Toyota'

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				• Multiple columns named such as: 'Expected Output Slot 1', 'Expected Output Slot 2', etc., may be used.

Column Name	Test set type	Value Type	Multiple Columns Support	Description
Expected Output Context Tag	Text and Audio	String	Yes	This column is used to specify name of an output context expected in output from Lex for the row in the test set. • This refers to output context in Setting intent context for your Lex V2 bot. • Note that Test Workbench only supports name of context and does not yet support parameter s inside the context. • Multiple columns named such as: 'Expected Output Context Tag

Column Name	Test set type	Value Type	Multiple Columns Support	Description
				1', 'Expected Output Context Tag 2', etc., may be used.

View test validation errors in test workbench

You can correct test sets that report validation errors. These validation errors are generated when a test set is not ready to be tested. The Test Workbench can show you which required columns in the test set input CSV file did not have a value in the expected format.

To view test validation errors:

1. From the list of test sets, select the name of the test set that reports a Status of **Validation Error** that you want to view. The names of the test sets are active links that take you to details regarding the test set.
2. The test set record displays validation error details at the top of the screen. Choose **View Details** to see the report on Validation Errors.
3. From the error report window, review the Line # and Error Type to see where the error occurs. For a lengthy list of errors, you can choose to **Download** the error report.
4. Compare the errors listed in your test set input CSV file to your original test file to correct any issues and upload the test set again.

The following table lists the input CSV validation error messages with scenarios.

Scenario	Error message	Notes
Test Set File Size Exceeds	Test Set file size is larger than 200 MB. Provide smaller file and try your request again.	

Scenario	Error message	Notes
Test set exceeds max records	Input file had records more than supported maximum number of 200,000.	
Upload Empty Test set	Imported test set is empty. Provide non-empty test set and try your request again.	
Empty column header name	Column Headers Row: found empty column name in column number 5.	
Unrecognized column header name	Column Headers Row: could not recognize column name 'dummy' in column number 2.	
Duplicate column header name	Column Headers Row: found multiple columns 'S3 audio link' and 'S3 audio link' that are same or equivalent. Remove or rename one of those columns.	
Multi value column name exceeded the limit	Column Headers Row: count of columns for 'Expected Output Slot' exceeded maximum supported count: 6. Remove some columns for 'Expected Output Slot' and try again.	Maximum Number of columns supported for multi value column is 6.

Scenario	Error message	Notes
Text or Audio related column header not present	Could not find columns for text or audio conversations. For text conversations, use {'Text input'} columns. For audio conversations, use {'S3 audio link', 'Expected transcription'} columns.	Audio Mandatory Columns: {'S3 audio link', 'Expected transcription'}Text Mandatory Columns: {'Text input'}
Both Text and Audio related column header exist	Found columns for both text and audio conversations. You can either use {'Text input'} columns for text conversations, or {'S3 audio link', 'Expected transcription'} columns for audio conversat ions.	Audio Mandatory Columns: {'S3 audio link', 'Expected transcription'}Text Mandatory Columns: {'Text input'}
Mandatory column is missing	Could not find mandatory columns ["Expected Output Intent"].	Mandatory Columns:{"Line #", "Source", "Expected Output Intent"}
Found a data in column with no header	Found data in column number 8 for row number 6, but corresponding column did not have a column header.	
Data not found for mandatory columns	Row=12: no values found for mandatory columns: {"Source", "Expected Output Intent"}	

Scenario	Error message	Notes
Duplicate conversation id found	conversation number '19' was seen for previous conversat ion at row number 39." Make sure that same conversat ion number has not been provided for two conversat ions, you can do this by ensuring that all rows for a conversation number are grouped together.	
Invalid conversation id provided	Found invalid value 'test_con versation' in 'Conversation #' column. Value for this column must be either numeric or N/ A (i.e. Not Applicable) for a user row.	
Non numeric value provided for line number	Found non-numeric value 'test_line' in 'Line #' column. Its value must be numeric.	
Conversation id not found in agent row	No value found for 'Conversa tion #' column. It must be provided for an agent row.	
Non numeric conversation id found in agent row	Found non-numeric value 'test_conversation' in 'Conversation #' column. Its value must be numeric for an agent row.	
Invalid S3 location	Invalid value 'bucket/folder' was provided. Valid format is S3://<bucketName>/ <keyName>.	

Scenario	Error message	Notes
Invalid S3 bucket name	Invalid s3 bucket name 'test_bucket' was provided. Check the bucket name.	
S3 audio location is folder	Provided audio location 'S3://bucket/folder' is invalid. It points to an S3 folder.	
Invalid intent name	Invalid characters were present in intent 'intent@name'. Check the intent name.	Regex check: ^([0-9a-zA-Z][_-]?)+$
Invalid slot name	Invalid characters were present in slot 'Slot@Name'. Check the slot name.	Regex: ^([0-9a-zA-Z][_-]?)+$It should not start or end with dot(.)
Slot value provided for parent slot	Slot values were provided for subslot 'Address.City' as well as parent slot 'Address'. Values should be only provided for the subslot.	Parent slot in CST should not have slot value
Invalid character in context name	Invalid characters were present in context name 'context@1'. Check the context name.	Regex: ^([A-Za-z]_?)+$
Invalid slot spelling style	Invalid value 'test' was provided. Make sure that they are all upper case. Valid values are ["Default", "SpellByLetter", "SpellByWord"].	Supported values["Default", "SpellByLetter", "SpellByWord"

Scenario	Error message	Notes
Participant or source has to be either agent or User	Invalid value 'bot' was provided. Valid values are ["Agent", "User"].	Supported Enums: "Agent", "User"
Line Number should not be decimal	Invalid value '10.1' was provided. It should be a valid number without any fractions.	
Conversation Number should not be decimal	Invalid value '10.1' was provided. It should be a valid number without any fractions.	
Line number should be with in range	Invalid value '92233720 368547758071' was provided. It should be greater than or equal to 1 and less than or equal to 922337203 6854775807.	
Barge-in column only accepts boolean value	Invalid value 'test' was provided. It should be a valid boolean value such as 'true' or 'false'. Alternatively 'yes' and 'no' can be used.	Possible Values:"True", "true", "T", "Yes", "yEs", "Y", "1", "1.0", "False", "false", "F", "No", "no", "N", "0", "0.0"
Expected slot, Session Attribute, Request Attribute should be separated by equal to (=)	Value 'slotName:slotValu e' does not have '='. Such value should be provided as a key-value pair in format '<key>=<value>'.	For example: slotName = slotType
Expected slot, Session Attribute, Request Attribute should be have key value pair	'=slotValue' does not have a key before '='. Such value should be provided as a key-value pair in format '<key>=<value>'.	For example: slotName = slotType

Scenario	Error message	Notes
Invalid quote at end	Found incorrect quoting in 'Foo's item'“. It starts with quote character `"` but does not end with same quote character.	For example: `*"Foo's item", KFC*`
Invalid quote at middle	Found incorrect quoting in `"Foo's" Burger, etc.`. It contains quote character `"` inside its content. Values containing single quotes should be wrapped within double quotes and vice-versa.	Correct For example: `*"Foo's item", KFC*`
Required quotes	`key = Foo's item` contains single-quotes or double-quotes but has not been wrapped inside quotes. Values containing single quotes should be wrapped inside double quotes and vice-versa.	
Duplicate Key repeated in column	Key `key1` was repeated in two columns: `Session Attribute 3` and `Session Attribute 1`.	
Invalid format in Runtime hint	Invalid key `BookFlight.Car."` provided for Runtime Hints. For Runtime Hints, key should be in format <intentName>.<slotName>.	If '.' must be present in middle of the key, intent name and slot name cannot be extracted from such key. examples of such incorrect formatting: "BookFlight", ".BookFlight.Car", "BookFlight.Car."

Scenario	Error message	Notes
Invalid Intent name in runtime hint key	Found invalid intent `intent@name` for Runtime Hints. Check intent name.	Regex check: ^([0-9a-zA-Z][_-]?)+$
Invalid Slot name in runtime hint key	Found invalid slot name in `Slot@Name` for Runtime Hints. Check slot name.	Regex: ^([0-9a-zA-Z][_-]?)+$It should not start or end with dot(.)

Delete a test set in Test Workbench

You can easily delete a test set from your list of test sets.

To delete a test set:

1. Go to the list of **Test Sets** from the left side menu to see the list of test sets.
2. From the list of test sets, select the test set you want to delete.
3. Go to the **Actions** drop down menu in the top right, and choose **Delete**.
4. A message confirms that the test set is deleted.

Edit test set details

You can edit a Test Set name and details in the list of test sets. The name or details can be added or updated later. However, you will have to update your test set before running the test with your bot or transcription data.

To edit test set details:

1. Go to the list of test set from the left side menu to see the list of test sets.
2. From the list of test sets, select the check box for the test set you want to edit.
3. Go to the **Actions** drop down menu in the top right, and choose **Edit Details**.
4. A message confirms that the test set is successfully edited.

Update test set

You can update, correct, modify, or delete items from the test set to optimize your baseline results, or to correct other errors that may have occurred in the test set

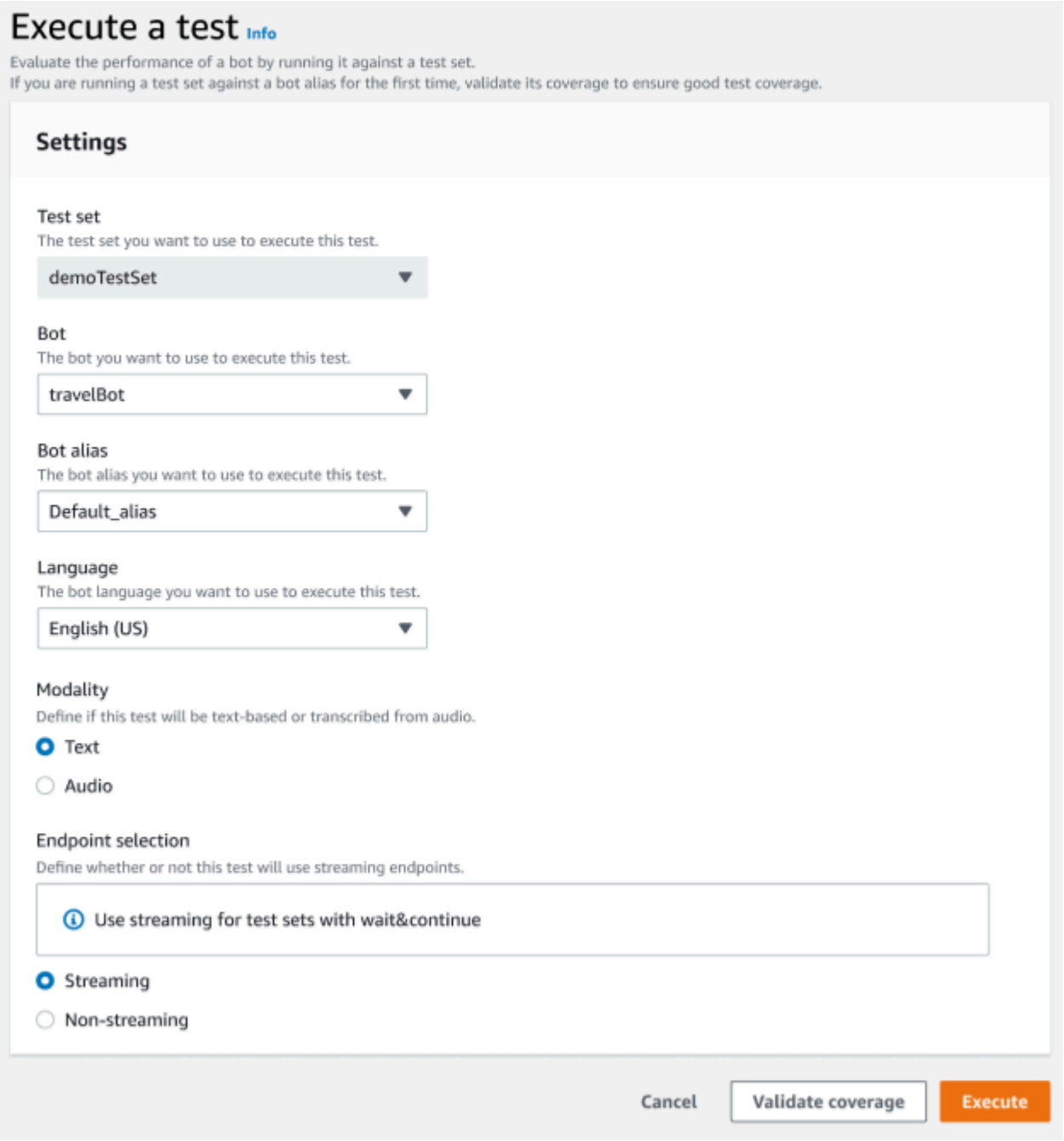
You can download a test set and fix the validation errors before uploading the corrected test set. See View test validation errors.

To update a test set:

1. From the test set record, choose the **Update Test Set** button in the top right.
2. Choose a file to upload from your Amazon S3 account or upload a CSV test file from your computer. NOTE: Updating a test set will overwrite the existing data.
3. Select the **Update** button.
4. A message confirms that the test set is successfully updated. NOTE: This operation can take a few minutes, depending on the complexity and size of the test set.
5. A message confirms that the test set is successfully updated and the **Status** displays **Ready for Testing**.

Execute a test

To execute a test set, you must choose the appropriate bot to run the test against the test set. You can choose a bot from your AWS account from the drop down menu under Test Set. This operation will test your selected bot against your validated test data to report performance metrics against the baseline data from the test set.



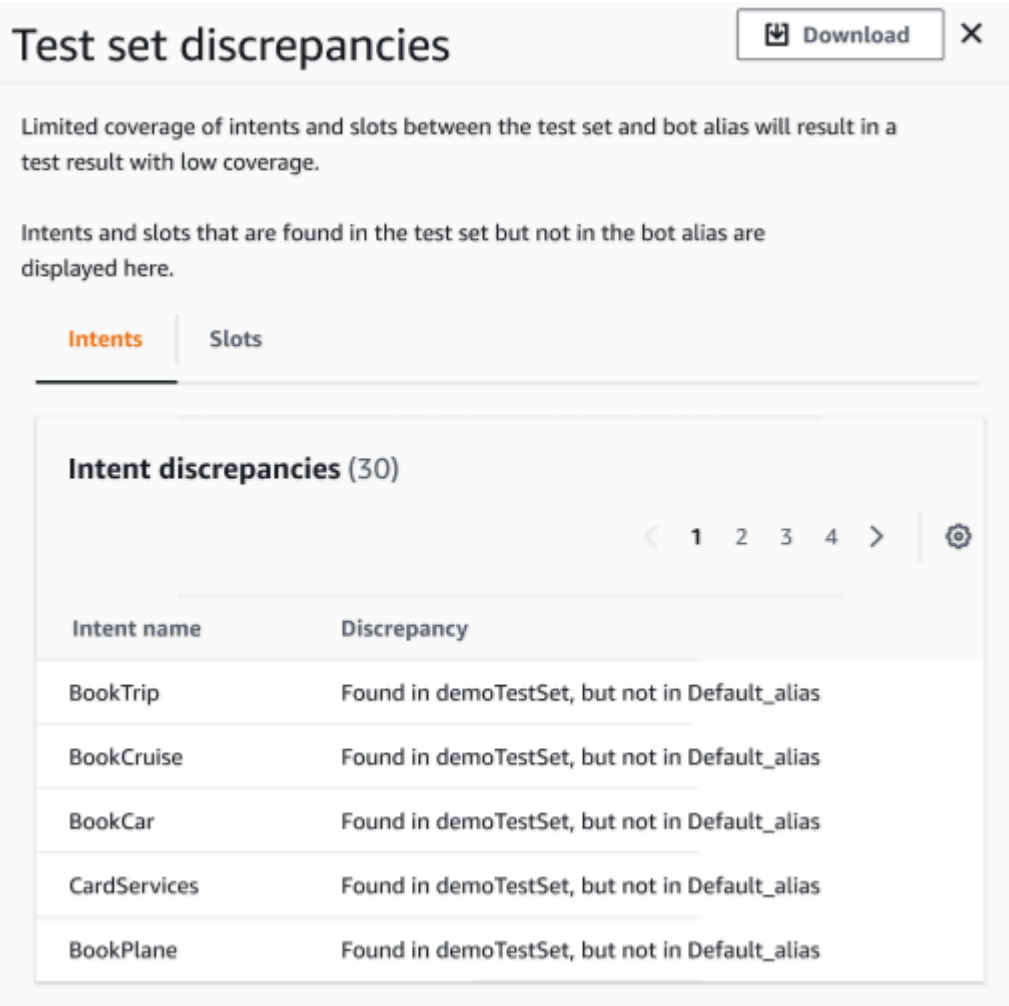
To execute a test in the Test Workbench

1. In the test set record page, choose **Execute Test**.
2. Select the test set you want to use in the test.
3. Select the name of the bot to use in the test from the **Bot** drop down menu.
4. Choose a bot alias, if applicable, from the **Bot alias** drop down menu.
5. From the **Languages** selection, choose a version of English.

6. Select **Text** or **Audio** for the Modality type.
7. Choose your Amazon S3 location. (audio only)
8. Select your **Endpoint selection** for your bot. (streaming only)
9. Select the **Validate coverage** button to confirm your test in ready to run. If there are any errors present in the validation step, review the previous parameters and make corrections.
10. Select **Execute** to run the test.
11. A message confirms that the test is successfully executed.

Test set coverage in Test Workbench

Limited coverage of intents and slots between the test set and the bot can result in expected performance measures. We recommend that you review the test set coverage ahead of running the test.



To review validation coverage

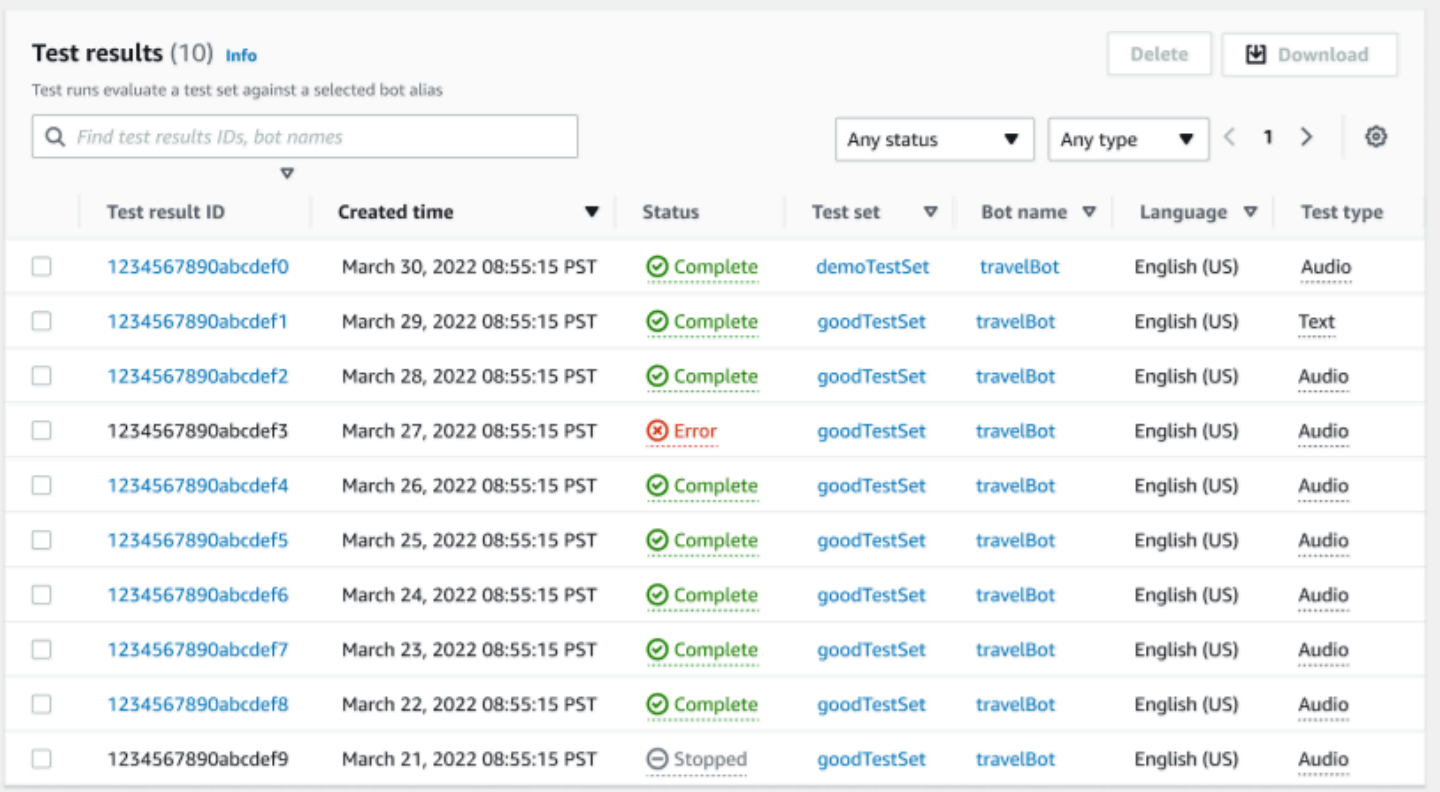
1. In the test set records, choose the **Validate coverage** button.
2. The message indicates it is validating coverage between the test set and the bot selected.
3. Once the operation is completed, the message indicates **Coverage validation successful**.
4. Choose the **View Details** button at the bottom of the window.
5. View the test set discrepancies for intents and slots by choosing the tab for each. You can download this data into a CSV format by choosing the **Download** button.
6. Review the validation results for your test set data, bot intents, and slots. Identify issues and make changes in your bot test set architecture to improve results. Upload the edited test set and bot to run the test once you have made changes to the CSV file. NOTE: Validation coverage runs against the test set and not against the bot. Intents in the bot but not present in the test set will not be covered.

View test results

Interpret test results from the Test Workbench to determine where the conversation between your bot and the customer might be failing, or requiring the customer to make multiple attempts to fulfill the intent.

By locating these issues in your test results, you can optimize your bot's performance by improving intent performance using different training data or utterances that are more consistent with the real time bot transcription values.

You can get a detailed view of intents and slots that had performance discrepancies. Once you have identified intents or slots that have discrepancies, you can further drill down and review the utterances and conversation flow.



To review test results:

1. Go to the list of test sets from the left side menu to select the **Test results** option under Test workbench. NOTE: Test results indicate a **Status** of complete if they were successful.
2. Select the **Test Result ID** for the test results you want to review.

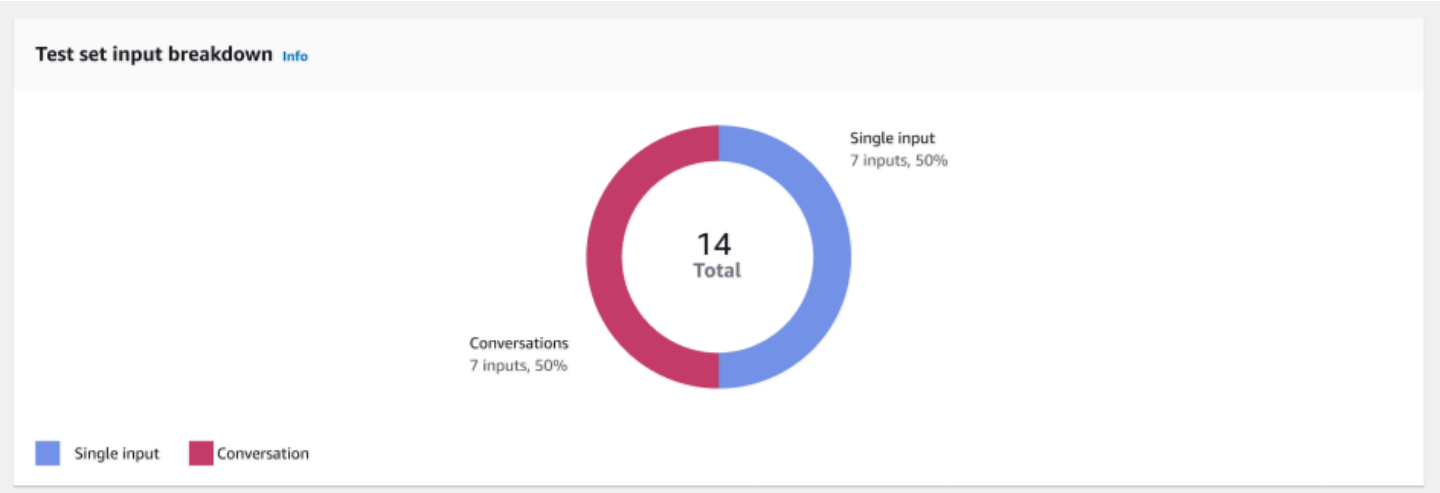
Test results details in Test Workbench

The test results show the test set details, intents used, and the slots used. It also provides the overall test set input breakdown includes the overall results, conversation results, intent, and slot results.

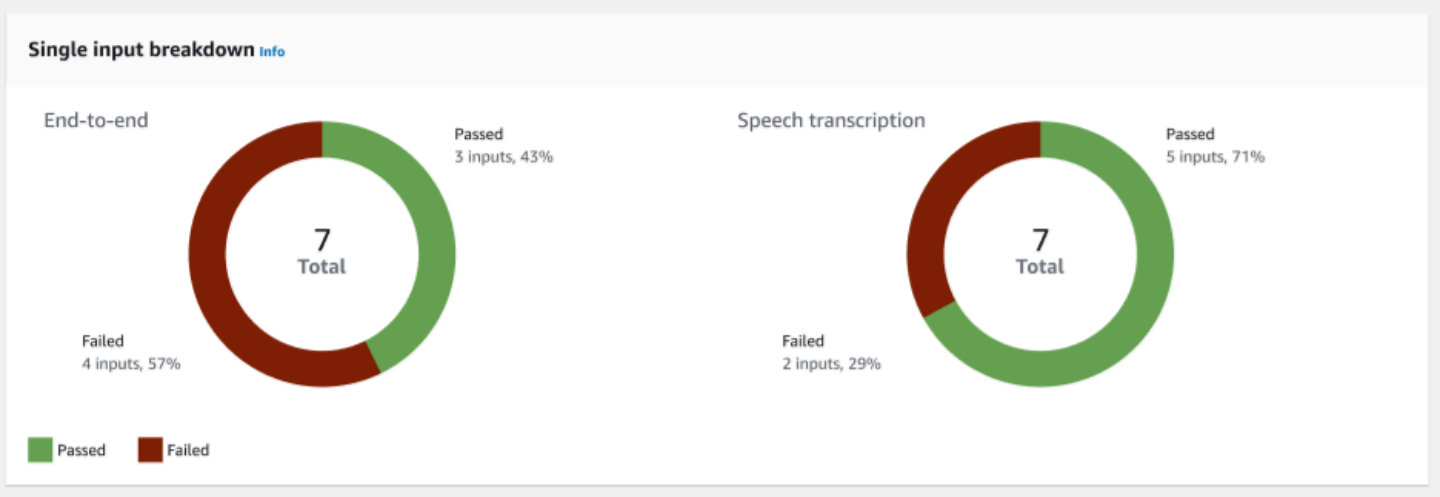
Test results comprise all testing related information such as:

- Test details metadata
- Overall results
- Conversation results
- Intent and slot results
- Detailed results

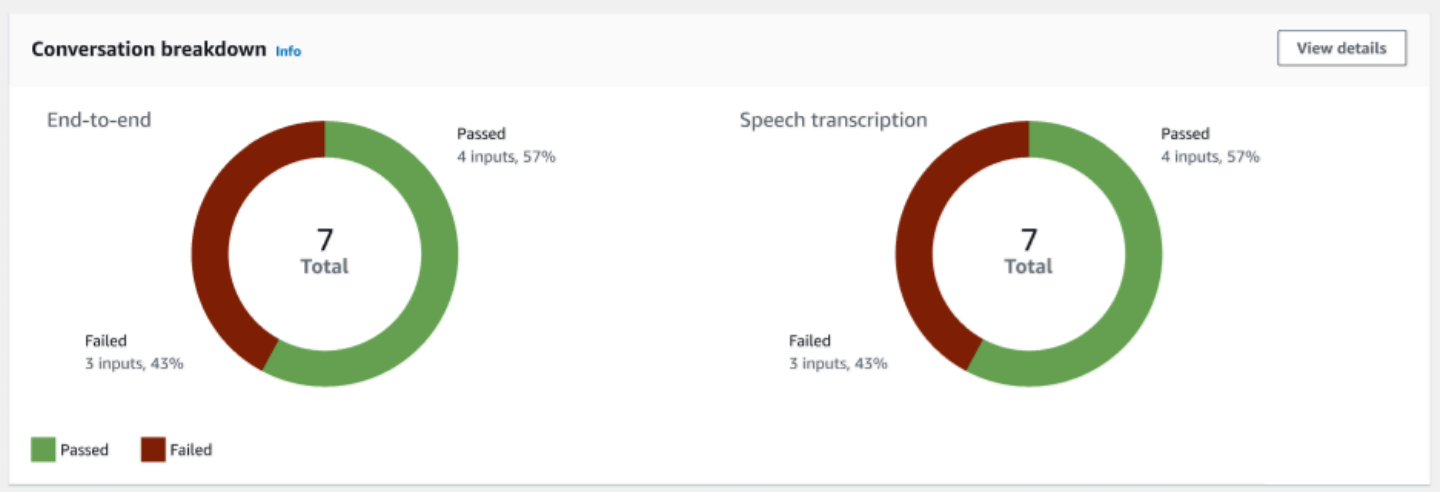
Overall results tab:



Test set input breakdown – This chart shows the breakdown of number of conversations and single input utterances in the test set.

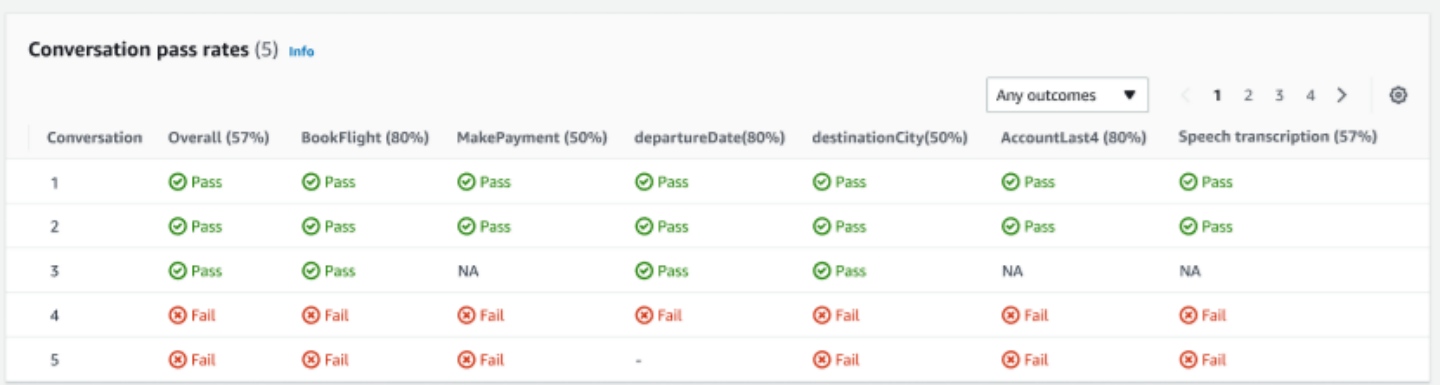


Single input breakdown – Displays two charts that included end-to-end conversations and speech transcriptions. The number of passed and failed inputs are indicated on each chart. Note: Speech transcription chart will be visible only for the audio test set.

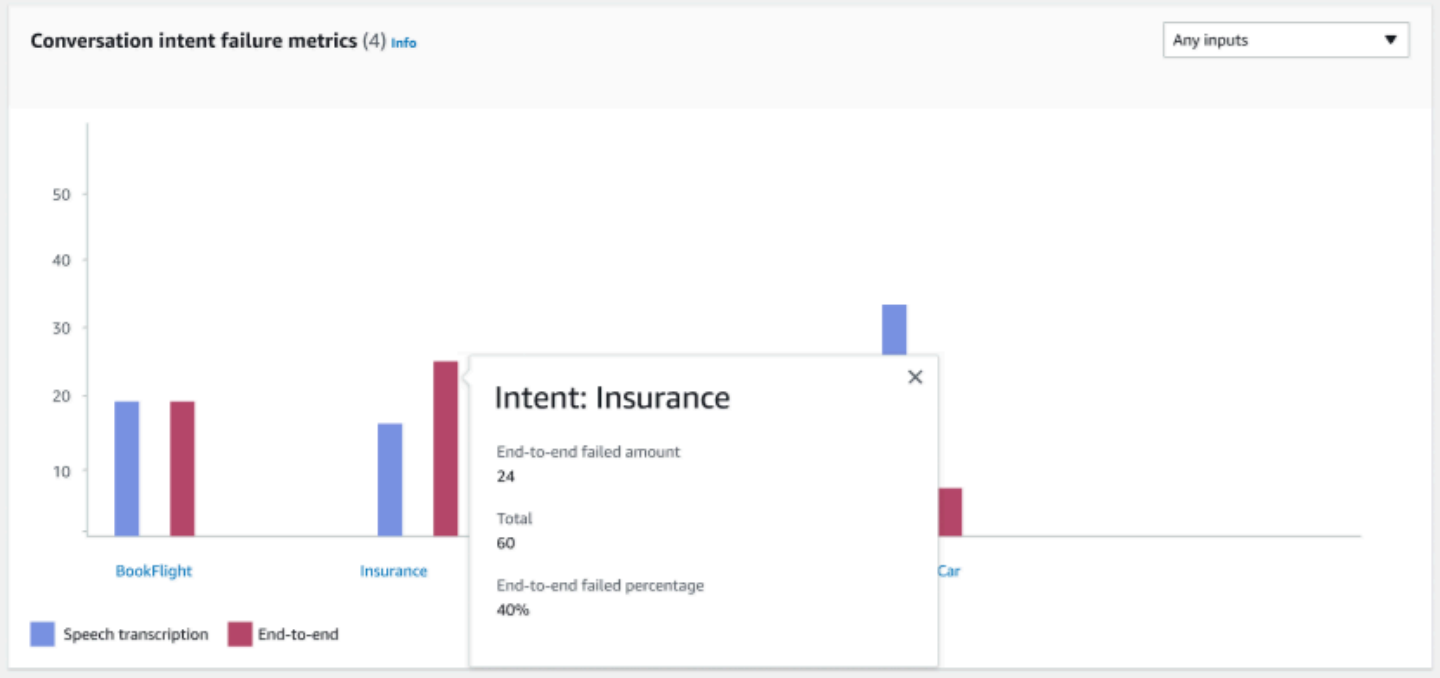


Conversation breakdown – Displays two charts that included end-to-end conversations and speech transcriptions. The number of passed and failed inputs are indicated on each chart. Note: Speech transcription chart will be visible only for the audio test set.

Conversation results tab:



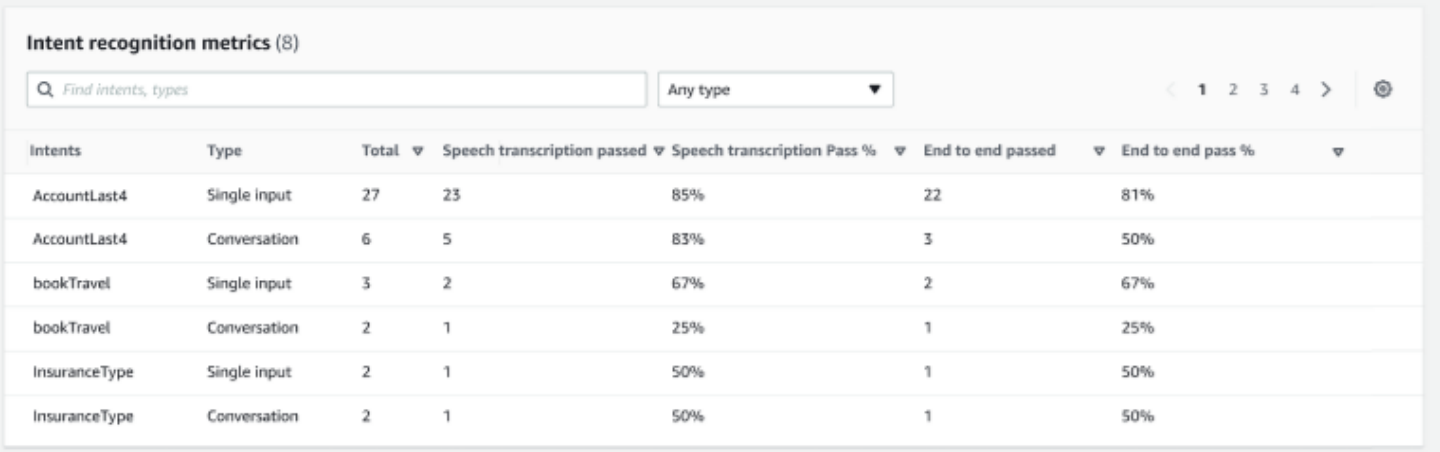
Conversation pass rates – The conversation pass rates table is used to see which intents and slots are used in each conversation in the test set. You can visualize where the conversation has failed by reviewing which intent or slot failed, along with the pass percentage of each intent and slot.



Conversation intent failure metrics – This metric shows the top 5 worst performing intents in the test set. This panel shows a chart of what percent or number of intents were successful or failed based on the bot's conversation logs or transcription. A successful intent does not mean that the entire conversation was successful. These metrics only apply to the value of the intents, regardless of which intent came before or after.

Conversation slot failure metrics – This metric shows the top 5 worst performing slots in the test set. Indicated the success rate for each slot in the intent. Bar graph shows both speech transcription and end-to-end conversations for each slot in the intent.

Intent and slot results tab:

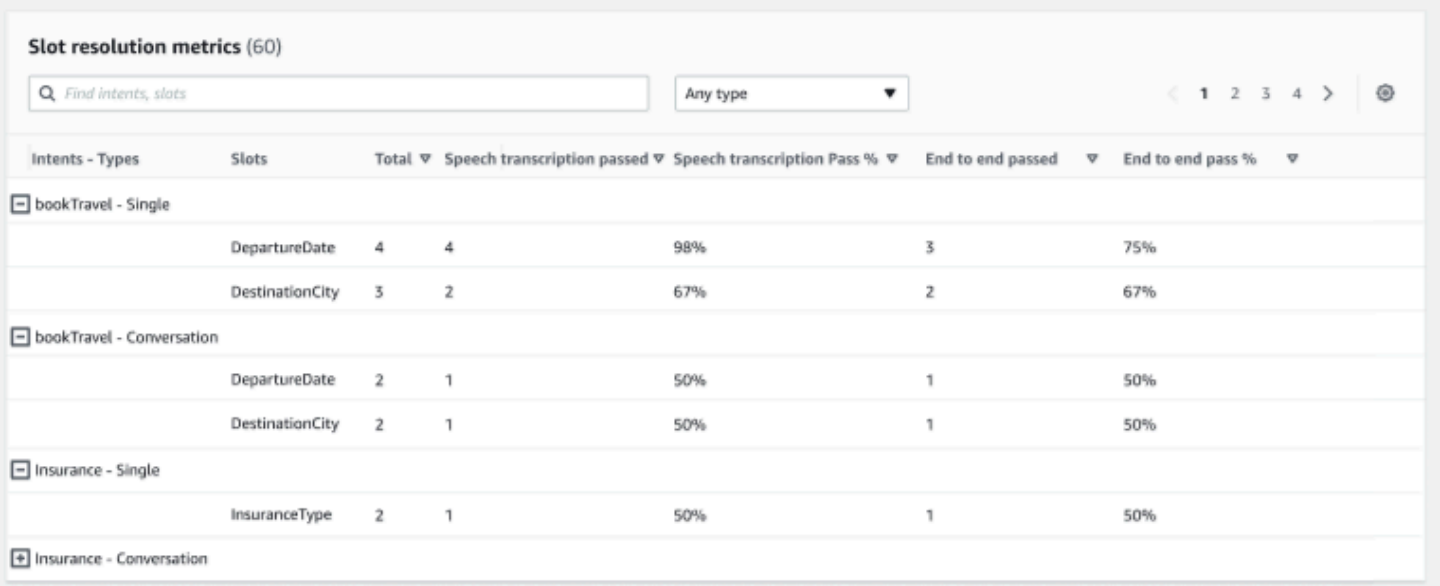


Intent recognition metrics (8)

Find intents, types | Any type | 1 2 3 4

Intents	Type	Total	Speech transcription passed	Speech transcription Pass %	End to end passed	End to end pass %
AccountLast4	Single input	27	23	85%	22	81%
AccountLast4	Conversation	6	5	83%	3	50%
bookTravel	Single input	3	2	67%	2	67%
bookTravel	Conversation	2	1	25%	1	25%
InsuranceType	Single input	2	1	50%	1	50%
InsuranceType	Conversation	2	1	50%	1	50%

Intent recognition metrics – Shows a table of how many intents were recognized successfully. Displays the pass rate of speech transcription and end-to-end conversations.



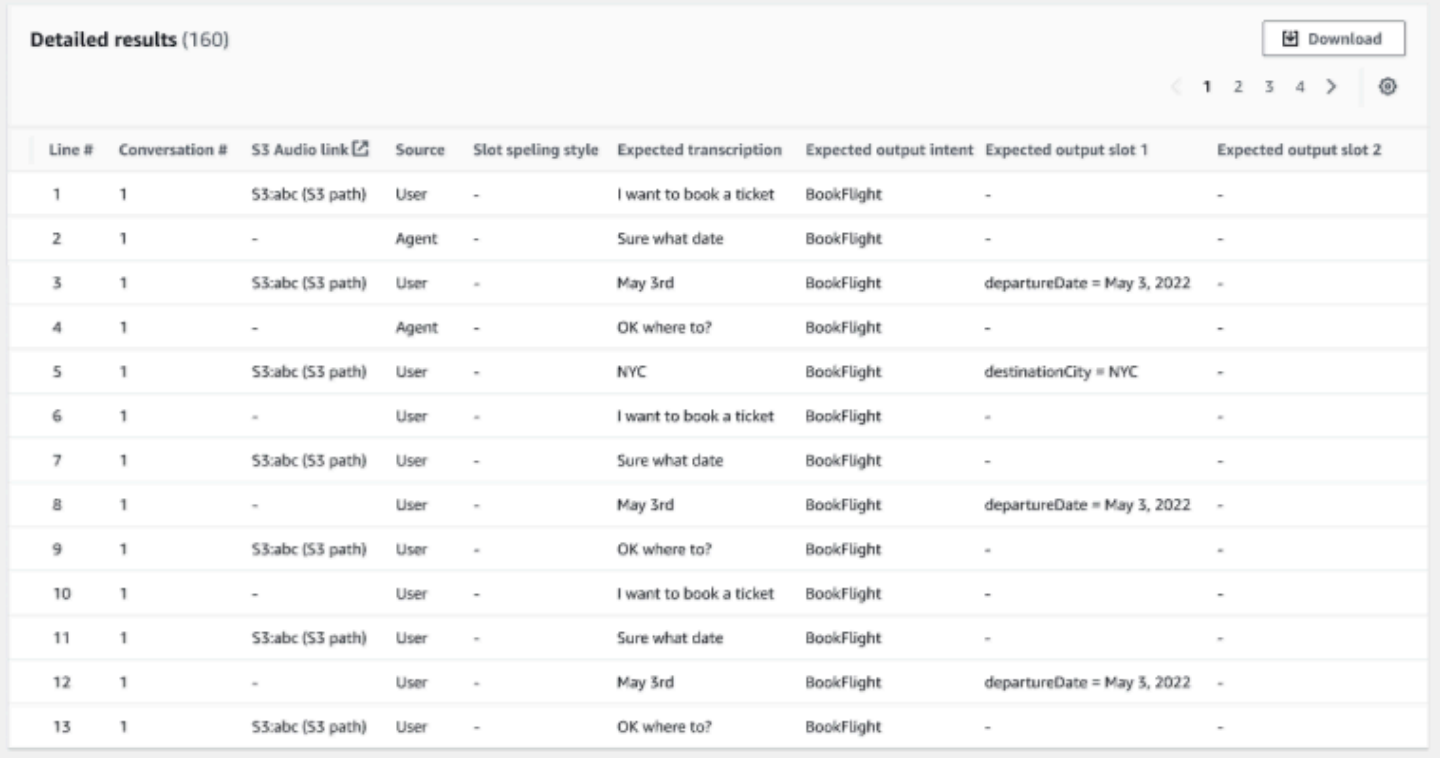
Slot resolution metrics (60)

Find intents, slots | Any type | 1 2 3 4

Intents - Types	Slots	Total	Speech transcription passed	Speech transcription Pass %	End to end passed	End to end pass %
bookTravel - Single						
	DepartureDate	4	4	98%	3	75%
	DestinationCity	3	2	67%	2	67%
bookTravel - Conversation						
	DepartureDate	2	1	50%	1	50%
	DestinationCity	2	1	50%	1	50%
Insurance - Single						
	InsuranceType	2	1	50%	1	50%
Insurance - Conversation						

Slot resolution metrics – Shows the intents and slots separately, and the success and failure rate of each slot for each intent used in the conversation or single input. Displays the pass rate of speech transcription and end-to-end conversations.

Detailed results tab:



Detailed results (160)

Download

1 2 3 4

Line #	Conversation #	S3 Audio link	Source	Slot speling style	Expected transcription	Expected output intent	Expected output slot 1	Expected output slot 2
1	1	S3:abc (S3 path)	User	-	I want to book a ticket	BookFlight	-	-
2	1	-	Agent	-	Sure what date	BookFlight	-	-
3	1	S3:abc (S3 path)	User	-	May 3rd	BookFlight	departureDate = May 3, 2022	-
4	1	-	Agent	-	OK where to?	BookFlight	-	-
5	1	S3:abc (S3 path)	User	-	NYC	BookFlight	destinationCity = NYC	-
6	1	-	User	-	I want to book a ticket	BookFlight	-	-
7	1	S3:abc (S3 path)	User	-	Sure what date	BookFlight	-	-
8	1	-	User	-	May 3rd	BookFlight	departureDate = May 3, 2022	-
9	1	S3:abc (S3 path)	User	-	OK where to?	BookFlight	-	-
10	1	-	User	-	I want to book a ticket	BookFlight	-	-
11	1	S3:abc (S3 path)	User	-	Sure what date	BookFlight	-	-
12	1	-	User	-	May 3rd	BookFlight	departureDate = May 3, 2022	-
13	1	S3:abc (S3 path)	User	-	OK where to?	BookFlight	-	-

Detailed results – Shows a detailed table on the conversation log with User and Agent utterances and the expected output and expected transcription for each slot. You can download this report by selecting the **Download** button.

The following table lists the result failure error messages with scenarios.

Scenario	Error message	Action
Intent Mismatch	Expected BookFlight intent but it was BookHotel intent.	Skip other turns in the conversation
Slot Elicitation mismatch	Expected departureDate slot to be elicited but it was cabinType.	Skip other turns in the conversation
Slot value mismatch	Mismatch between expected and actual slot value.	Continue with other turns in the conversations
Back-to-back agent prompt is missing	Expected bot to return an agent prompt in this turn but it was not received.	Skip other turns in the conversation

Scenario	Error message	Action
Transcription Mismatch	Expected transcription didn't match actual transcription.	Continue with other turns in the conversations
Optional slot not elicited	Expected to elicit cabinType slot in next turn, however current intent fulfilled before that.	Skip other turns in the conversation
Slot not recognized	Expected departureDate slot was not recognized in this turn.	Skip other turns in the conversation
Extra back-to-back agent prompt	Expected a user turn but it was agent prompt	Skip other turns in the conversation

Streaming conversations to an Amazon Lex V2 bot

You can use the Amazon Lex V2 streaming API to start a bidirectional stream between an Amazon Lex V2 bot and your application. Starting a stream enables the bot to manage the conversation between the bot and the user. The bot responds to user input without you writing code to handle responses from the user. The bot can:

- Handle interruptions from the user while it's playing a prompt. For more information, see Enabling your Amazon Lex V2 bot to be interrupted by the user.
- Wait for the user to provide input. For example, the bot can wait for the user to gather credit card information. For more information, see Enabling the Amazon Lex V2 bot to wait for the user to provide more information during a pause.
- Take both dual-tone multiple-frequency (DTMF) and audio input in the same stream.
- Handle pauses in user input better than if you were managing the conversation from your application.

Not only does the Amazon Lex V2 bot respond to data sent from your application, but it also sends information about the state of the conversation to your application. You can use this information to change how your application responds to customers.

The Amazon Lex V2 bot also monitors the connection between the bot and your application. It can determine if the connection has timed out.

To use the API to start a stream to an Amazon Lex V2 bot, see Starting a conversation stream to an Amazon Lex V2 bot.

When you start streaming to an Amazon Lex V2 bot from your application, you can configure the bot to accept audio input or text input from the user. You can also choose whether the user receives audio or text in response to their input.

If you've configured the Amazon Lex V2 bot to accept audio input from the user, it can't take text input. If you've configured the bot to accept text input, the user can only use written text to communicate with it.

When an Amazon Lex V2 bot takes a streaming audio input, the bot determines when a user begins speaking and when they stop speaking. It handles any pauses or any interruptions from the user. It can also take DTMF (dual-tone multi-frequency) input and speech input in the same stream. This

helps the user interact with the bot more naturally. You can present users with welcome messages and prompts. You can also enable users to interrupt those messages and prompts.

When you start a bidirectional stream, Amazon Lex V2 uses the HTTP/2 protocol. Your application and the bot exchange data in a single stream as a series of *events*. An event can be one of the following:

- Text, audio, or DTMF input from the user.
- Signals from the application to the Amazon Lex V2 bot. These include an indication that audio playback of a message has been completed, or that the user has disconnected from the session.

For more information about events, see Starting a conversation stream to an Amazon Lex V2 bot. For information about how to encode events, see Event stream encoding.

Topics

Starting a conversation stream to an Amazon Lex V2 bot

You use the StartConversation operation to start a stream between the user and the Amazon Lex V2 bot in your application. The POST request from the application establishes a connection between your application and the Amazon Lex V2 bot. This enables your application and the bot to start exchanging information with each other through events.

> **Note**
>
> When using StartConversation, if the utterance audio duration exceeds the configured value for `max-length-ms`, Amazon Lex V2 cuts off the audio at the specified duration.

The `StartConversation` operation is supported only in the following SDKs:

- AWS SDK for C++
- AWS SDK for Java V2
- AWS SDK for JavaScript v3
- AWS SDK for Ruby V3

The first event your application must send to the Amazon Lex V2 bot is a ConfigurationEvent. This event includes information such as the response type format. The following are the parameters that you can use in a configuration event:

- **responseContentType** – Determines whether the bot responds to user input with text or speech.
- **sessionState** – Information relating to the streaming session with the bot such as predetermined intent or dialog state.
- **welcomeMessages** – Specifies the welcome messages that play for the user at the beginning of their conversation with a bot. These messages play before the user provides any input. To activate a welcome message, you must also specify values for the `sessionState` and `dialogAction` parameters.
- **disablePlayback** – Determines whether the bot should wait for a cue from the client before it starts listening for caller input. By default, playback is activated, so the value of this field is `false`.
- **requestAttributes** – Provides additional information for the request.

For information about how to specify values for the preceding parameters, see the ConfigurationEvent data type of the StartConversation operation.

Each stream between a bot and your application can only have one configuration event. After your application has sent a configuration event, the bot can take additional communication from your application.

If you've specified that your user is using audio to communicate with the Amazon Lex V2 bot, your application can send the following events to the bot during that conversation:

- AudioInputEvent – Contains an audio chunk that has maximum size of 320 bytes. Your application must use multiple audio input events to send a message from the server to the bot. Every audio input event in the stream must have the same audio format.
- DTMFInputEvent – Sends a DTMF input to the bot. Each DTMF key press corresponds to a single event.

- PlaybackCompletionEvent – Informs the server that a response from the user's input has been played back to them. You must use a playback completion event if you're sending an audio response to the user. If `disablePlayback` of your configuration event is `true`, you can't use this feature.
- DisconnectionEvent – Informs the bot that the user has disconnected from the conversation.

If you've specified that the user is using text to communicate with the bot, your application can send the following events to the bot during that conversation:

- TextInputEvent – Text that is sent from your application to the bot. You can have up to 512 characters in a text input event.
- PlaybackCompletionEvent – Informs the server that a response from the user's input has been played back to them. You must use this event if you're playing audio back to the user. If `disablePlayback` of your configuration event is `true`, you can't use this feature.
- DisconnectionEvent – Informs the bot that the user has disconnected from the conversation.

You must encode every event that you send to an Amazon Lex V2 bot in the correct format. For more information, see Event stream encoding.

Every event has an event ID. To help troubleshoot any issues that might occur in the stream, assign a unique event ID to each input event. You can then troubleshoot any processing failures with the bot.

Amazon Lex V2 also uses timestamps for each event. You can use these timestamps in addition to the event ID to help troubleshoot any network transmission issues.

During the conversation between the user and the Amazon Lex V2 bot, the bot can send the following outbound events in response to the user:

- IntentResultEvent – Contains the intent that Amazon Lex V2 determined from the user utterance. Each internal result event includes:
  - **inputMode** – The type of user utterance. Valid values are `Speech`, `DTMF`, or `Text`.
  - **interpretations** – Interpretations that Amazon Lex V2 determines from the user utterance.
  - **requestAttributes** – If you haven't modified the request attributes by using a lambda function, these are the same attributes that were passed at the start of the conversation.
  - **sessionId** – Session identifier used for the conversation.

- **sessionState** – The state of the user's session with Amazon Lex V2.

- TranscriptEvent – If the user provides an input to your application, this event contains the transcript of the user's utterance to the bot. Your application does not receive a `TranscriptEvent` if there's no user input.

  The value of the transcript event sent to your application depends on whether you've specified audio (speech and DMTF) or text as a conversation mode:

  - Transcript of speech input – If the user is speaking with the bot, the transcript event is the transcription of the user's audio. It's a transcript of all the speech from the time the user begins speaking to the time they end speaking.
  - Transcript of DTMF input – If the user is typing on a keypad, the transcript event contains all the digits the user pressed in their input.
  - Transcript of text input – If the user is providing text input, the transcript event contains all of the text in the user's input.

- TextResponseEvent – Contains the bot response in text format. A text response is returned by default. If you've configured Amazon Lex V2 to return an audio response, this text is used to generate an audio response. Each text response event contains an array of message objects that the bot returns to the user.

- AudioResponseEvent – Contains the audio response synthesized from the text generated in the `TextResponseEvent`. To receive audio response events, you must configure Amazon Lex V2 to provide an audio response. All audio response events have the same audio format. Each event contains audio chunks of no more than 100 bytes. Amazon Lex V2 sends an empty audio chunk with the `bytes` field set to `null` to indicate that the end of the audio response event to your application.

- PlaybackInterruptionEvent – When a user interrupts a response that the bot has sent to your application, Amazon Lex V2 triggers this event to stop the playback of the response.

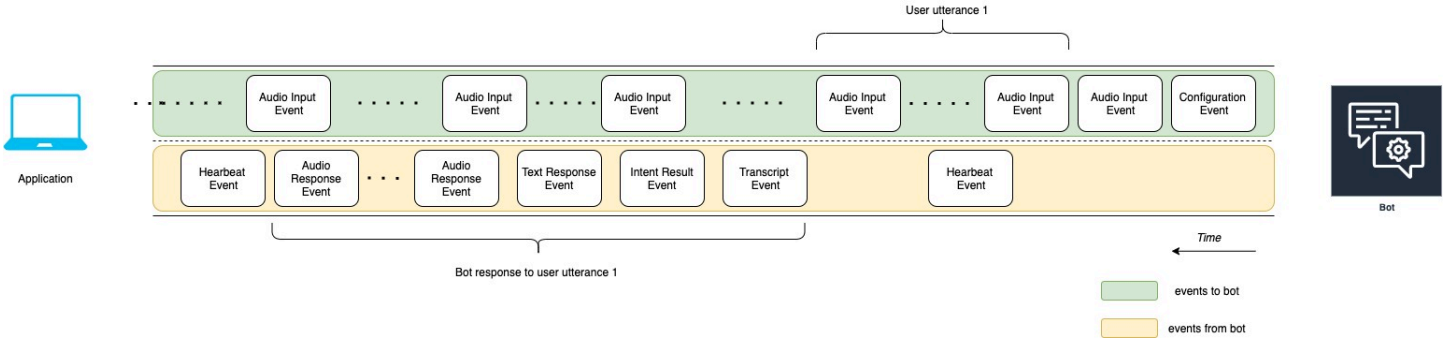
- HeartbeatEvent – Amazon Lex V2 sends this event back periodically to keep the connection between your application and the bot from timing out.

Time sequence of events for an audio conversation when using an Amazon Lex V2 bot

The following diagrams show a streaming audio conversation between a user and an Amazon Lex V2 bot. The application continuously streams audio to the bot, and the bot looks for user input

from the audio. In this example, both the user and the bot are using speech to communicate. Each diagram corresponds to a user utterance and the response of the bot to that utterance.

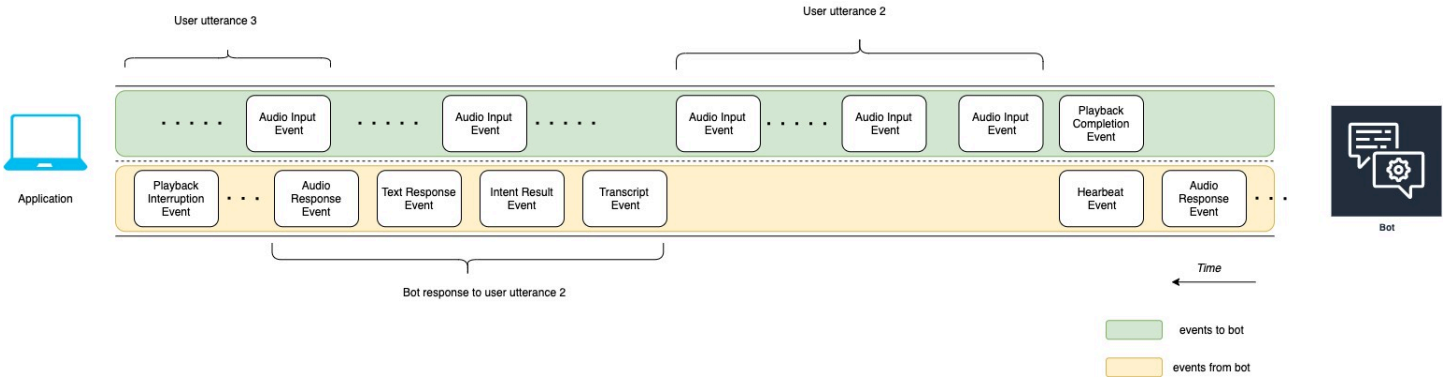
The following diagram shows the beginning of a conversation between the application and the bot. The stream begins at time zero (t0).



The following list describes the events of the preceding diagram.

- t0: The application sends a configuration event to the bot to start the stream.
- t1: The application streams audio data. This data is broken into a series of input events from the application.
- t2: For *user utterance 1*, the bot detects an audio input event when the user begins speaking.
- t2: While the user is speaking, the bot sends a heartbeat event to maintain the connection. It sends these events intermittently to make sure the connection doesn't time out.
- t3: The bot detects the end of the user's utterance.
- t4: The bot sends back a transcript event that contains a transcript of the user's speech to the application. This is the beginning of *Bot response to user utterance 1*.
- t5: The bot sends an intent result event to indicate the action that the user wants to perform.
- t6: The bot begins providing its response as text in a text response event.
- t7: The bot sends a series of audio response events to the application to play for the user.
- t8: The bot sends another heartbeat event to intermittently maintain the connection.

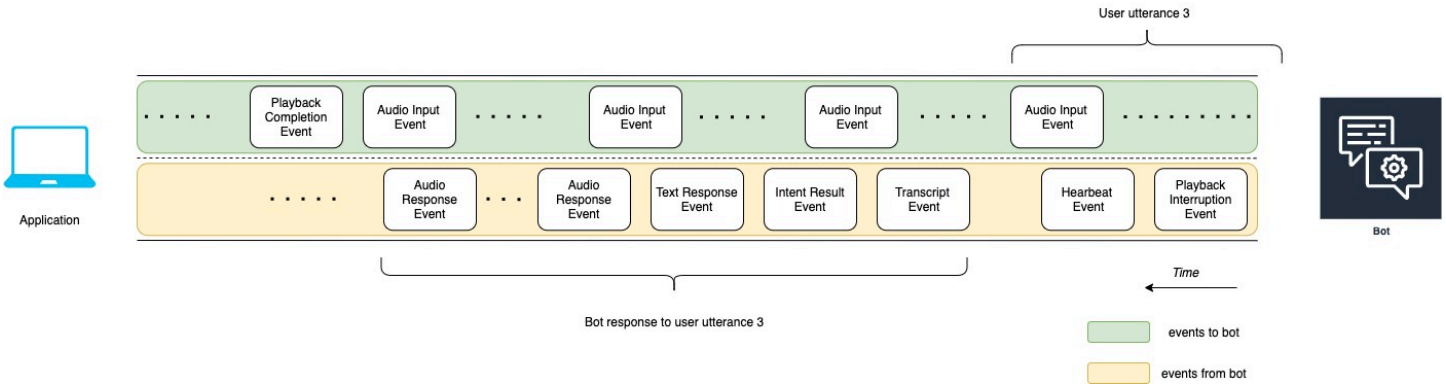
The following diagram is a continuation of the previous diagram. It shows the application sending a playback completion event to the bot to indicate that it has stopped playing the audio response for the user. The application plays back *Bot response to user utterance 1* to the user. The user responds to *Bot response to user utterance 1* with *User utterance 2*.



The following list describes the events of the preceding diagram:

- t10: The application sends a playback completion event to indicate that it has finished playing the bot's message to the user.
- t11: The application sends the user response back to the bot as *User utterance 2*.
- t12: For *Bot response to user utterance 2*, the bot waits for the user to stop speaking and then begins to provide an audio response.
- t13: While the bot sends *Bot response to user utterance 2* to the application, the bot detects the start of *User utterance 3*. The bot stops *Bot response to user utterance 2* and sends a playback interruption event.
- t14: The bot sends a playback interruption event to the application to signal that the user has interrupted the prompt.

The following diagram shows the *Bot response to user utterance 3*, and that the conversation continues after the bot responds to the user utterance.



Using the API to start a streaming conversation

When you start a stream to an Amazon Lex V2 bot, you accomplish the following tasks:

1. Create an initial connection to the server.
2. Configure the security credentials and bot details. Bot details include whether the bot takes DTMF and audio input, or text input.
3. Send events to the server. These events are text data or audio data from the user.
4. Process events sent from the server. In this step, you determine whether the bot output is presented to the user as text or speech.

The following code examples initialize a streaming conversation with an Amazon Lex V2 bot and your local machine. You can modify the code to meet your needs.

The following code is an example request using the AWS SDK for Java to start the connection to a bot and configure the bot details and credentials.

```
package com.lex.streaming.sample;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lexruntimev2.LexRuntimeV2AsyncClient;
import software.amazon.awssdk.services.lexruntimev2.model.ConversationMode;
import software.amazon.awssdk.services.lexruntimev2.model.StartConversationRequest;

import java.net.URISyntaxException;
import java.util.UUID;
import java.util.concurrent.CompletableFuture;

/**
 * The following code creates a connection with the Amazon Lex bot and configures the
 bot details and credentials.
 * Prerequisite: To use this example, you must be familiar with the Reactive streams
 programming model.
 * For more information, see
 * https://github.com/reactive-streams/reactive-streams-jvm.
 * This example uses AWS SDK for Java for Amazon Lex V2.
 * <p>
 * The following sample application interacts with an Amazon Lex bot with the streaming
 API. It uses the Audio
 * conversation mode to return audio responses to the user's input.
 * <p>
```

```
 * The code in this example accomplishes the following:
 * <p>
 * 1. Configure details about the conversation between the user and the Amazon Lex bot.
 These details include the conversation mode and the specific bot the user is speaking
 with.
 * 2. Create an events publisher that passes the audio events to the Amazon Lex bot
 after you establish the connection. The code we provide in this example tells your
 computer to pick up the audio from
 * your microphone and send that audio data to Amazon Lex.
 * 3. Create a response handler that handles the audio responses from the Amazon Lex
 bot and plays back the audio to you.
 */
public class LexBidirectionalStreamingExample {

    public static void main(String[] args) throws URISyntaxException,
 InterruptedException {
        String botId = "";
        String botAliasId = "";
        String localeId = "";
        String accessKey = "";
        String secretKey = "";
        String sessionId = UUID.randomUUID().toString();
        Region region = Region.region_name; // Choose an AWS Region where the Amazon
 Lex Streaming API is available.

        AwsCredentialsProvider awsCredentialsProvider = StaticCredentialsProvider
                .create(AwsBasicCredentials.create(accessKey, secretKey));

        // Create a new SDK client. You need to use an asynchronous client.
        System.out.println("step 1: creating a new Lex SDK client");
        LexRuntimeV2AsyncClient lexRuntimeServiceClient =
 LexRuntimeV2AsyncClient.builder()
                .region(region)
                .credentialsProvider(awsCredentialsProvider)
                .build();


        // Configure the bot, alias and locale that you'll use to have a conversation.
        System.out.println("step 2: configuring bot details");
        StartConversationRequest.Builder startConversationRequestBuilder =
 StartConversationRequest.builder()
                .botId(botId)
                .botAliasId(botAliasId)
                .localeId(localeId);
```

```
        // Configure the conversation mode of the bot. By default, the
        // conversation mode is audio.
        System.out.println("step 3: choosing conversation mode");
        startConversationRequestBuilder =
startConversationRequestBuilder.conversationMode(ConversationMode.AUDIO);

        // Assign a unique identifier for the conversation.
        System.out.println("step 4: choosing a unique conversation identifier");
        startConversationRequestBuilder =
startConversationRequestBuilder.sessionId(sessionId);

        // Start the initial request.
        StartConversationRequest startConversationRequest =
startConversationRequestBuilder.build();

        // Create a stream of audio data to the Amazon Lex bot. The stream will start
after the connection is established with the bot.
        EventsPublisher eventsPublisher = new EventsPublisher();

        // Create a class to handle responses from bot. After the server processes the
user data you've streamed, the server responds
        // on another stream.
        BotResponseHandler botResponseHandler = new
BotResponseHandler(eventsPublisher);

        // Start a connection and pass in the publisher that streams the audio and
process the responses from the bot.
        System.out.println("step 5: starting the conversation ...");
        CompletableFuture<Void> conversation =
lexRuntimeServiceClient.startConversation(
                startConversationRequest,
                eventsPublisher,
                botResponseHandler);

        // Wait until the conversation finishes. The conversation finishes if the
dialog state reaches the "Closed" state.
        // The client stops the connection. If an exception occurs during the
conversation, the
        // client sends a disconnection event.
        conversation.whenComplete((result, exception) -> {
            if (exception != null) {
                eventsPublisher.disconnect();
            }
```

```
        });

        // The conversation finishes when the dialog state is closed and last prompt
 has been played.
        while (!botResponseHandler.isConversationComplete()) {
            Thread.sleep(100);
        }

        // Randomly sleep for 100 milliseconds to prevent JVM from exiting.
        // You won't need this in your production code because your JVM is
        // likely to always run.
        // When the conversation finishes, the following code block stops publishing
 more data and informs the Amazon Lex bot that there is no more data to send.
        if (botResponseHandler.isConversationComplete()) {
            System.out.println("conversation is complete.");
            eventsPublisher.stop();
        }
    }
}
```

The following code is an example request using the AWS SDK for Java to send events to the bot. The code in this example uses the microphone on your computer to send audio events.

```
package com.lex.streaming.sample;

import org.reactivestreams.Publisher;
import org.reactivestreams.Subscriber;
import
 software.amazon.awssdk.services.lexruntimev2.model.StartConversationRequestEventStream;

/**
 * You use the Events publisher to send events to the Amazon Lex bot. When you
 establish a connection, the bot uses the
 * subscribe() method and enables the events publisher starts sending events to
 * your computer. The bot uses the "request" method of the subscription to make more
 requests. For more information on the request method, see https://github.com/reactive-
streams/reactive-streams-jvm.
 */
public class EventsPublisher implements Publisher<StartConversationRequestEventStream>
 {
```

```
    private AudioEventsSubscription audioEventsSubscription;

    @Override
    public void subscribe(Subscriber<? super StartConversationRequestEventStream>
 subscriber) {
        if (audioEventsSubscription == null) {

            audioEventsSubscription = new AudioEventsSubscription(subscriber);
            subscriber.onSubscribe(audioEventsSubscription);

        } else {
            throw new IllegalStateException("received unexpected subscription
 request");
        }
    }

    public void disconnect() {
        if (audioEventsSubscription != null) {
            audioEventsSubscription.disconnect();
        }
    }

    public void stop() {
        if (audioEventsSubscription != null) {
            audioEventsSubscription.stop();
        }
    }

    public void playbackFinished() {
        if (audioEventsSubscription != null) {
            audioEventsSubscription.playbackFinished();
        }
    }
}
```

The following code is an example request using the AWS SDK for Java to handle responses from the bot. The code in this example configures Amazon Lex V2 to play an audio response back to you.

```
package com.lex.streaming.sample;
```

```
import javazoom.jl.decoder.JavaLayerException;
import javazoom.jl.player.advanced.AdvancedPlayer;
import javazoom.jl.player.advanced.PlaybackEvent;
import javazoom.jl.player.advanced.PlaybackListener;
import software.amazon.awssdk.core.async.SdkPublisher;
import software.amazon.awssdk.services.lexruntimev2.model.AudioResponseEvent;
import software.amazon.awssdk.services.lexruntimev2.model.DialogActionType;
import software.amazon.awssdk.services.lexruntimev2.model.IntentResultEvent;
import software.amazon.awssdk.services.lexruntimev2.model.PlaybackInterruptionEvent;
import software.amazon.awssdk.services.lexruntimev2.model.StartConversationResponse;
import
 software.amazon.awssdk.services.lexruntimev2.model.StartConversationResponseEventStream;
import
 software.amazon.awssdk.services.lexruntimev2.model.StartConversationResponseHandler;
import software.amazon.awssdk.services.lexruntimev2.model.TextResponseEvent;
import software.amazon.awssdk.services.lexruntimev2.model.TranscriptEvent;

import java.io.IOException;
import java.io.UncheckedIOException;
import java.util.concurrent.CompletableFuture;

/**
 * The following class is responsible for processing events sent from the Amazon Lex
 bot. The bot sends multiple audio events,
 * so the following code concatenates those audio events and uses a publicly available
 Java audio player to play out the message to
 * the user.
 */
public class BotResponseHandler implements StartConversationResponseHandler {

    private final EventsPublisher eventsPublisher;

    private boolean lastBotResponsePlayedBack;
    private boolean isDialogStateClosed;
    private AudioResponse audioResponse;


    public BotResponseHandler(EventsPublisher eventsPublisher) {
        this.eventsPublisher = eventsPublisher;
        this.lastBotResponsePlayedBack = false;// At the start, we have not played back
 last response from bot.
        this.isDialogStateClosed = false; // At the start, the dialog state is open.
    }
```

```
    @Override
    public void responseReceived(StartConversationResponse startConversationResponse) {
        System.out.println("successfully established the connection with server.
request id:" + startConversationResponse.responseMetadata().requestId()); // would
have 2XX, request id.
    }

    @Override
    public void onEventStream(SdkPublisher<StartConversationResponseEventStream>
sdkPublisher) {

        sdkPublisher.subscribe(event -> {
            if (event instanceof PlaybackInterruptionEvent) {
                handle((PlaybackInterruptionEvent) event);
            } else if (event instanceof TranscriptEvent) {
                handle((TranscriptEvent) event);
            } else if (event instanceof IntentResultEvent) {
                handle((IntentResultEvent) event);
            } else if (event instanceof TextResponseEvent) {
                handle((TextResponseEvent) event);
            } else if (event instanceof AudioResponseEvent) {
                handle((AudioResponseEvent) event);
            }
        });
    }

    @Override
    public void exceptionOccurred(Throwable throwable) {
        System.err.println("got an exception:" + throwable);
    }

    @Override
    public void complete() {
        System.out.println("on complete");
    }

    private void handle(PlaybackInterruptionEvent event) {
        System.out.println("Got a PlaybackInterruptionEvent: " + event);
    }

    private void handle(TranscriptEvent event) {
        System.out.println("Got a TranscriptEvent: " + event);
    }
```

```
   private void handle(IntentResultEvent event) {
       System.out.println("Got an IntentResultEvent: " + event);
       isDialogStateClosed =
DialogActionType.CLOSE.equals(event.sessionState().dialogAction().type());
   }

   private void handle(TextResponseEvent event) {
       System.out.println("Got an TextResponseEvent: " + event);
       event.messages().forEach(message -> {
           System.out.println("Message content type:" + message.contentType());
           System.out.println("Message content:" + message.content());
       });
   }

   private void handle(AudioResponseEvent event) {//Synthesize speech
       // System.out.println("Got a AudioResponseEvent: " + event);
       if (audioResponse == null) {
           audioResponse = new AudioResponse();
           //Start an audio player in a different thread.
           CompletableFuture.runAsync(() -> {
               try {
                   AdvancedPlayer audioPlayer = new AdvancedPlayer(audioResponse);

                   audioPlayer.setPlayBackListener(new PlaybackListener() {
                       @Override
                       public void playbackFinished(PlaybackEvent evt) {
                           super.playbackFinished(evt);

                           // Inform the Amazon Lex bot that the playback has
finished.
                           eventsPublisher.playbackFinished();
                           if (isDialogStateClosed) {
                               lastBotResponsePlayedBack = true;
                           }
                       }
                   });
                   audioPlayer.play();
               } catch (JavaLayerException e) {
                   throw new RuntimeException("got an exception when using audio
player", e);
               }
           });
```

```
        }

        if (event.audioChunk() != null) {
            audioResponse.write(event.audioChunk().asByteArray());
        } else {
            // The audio audio prompt has ended when the audio response has no
            // audio bytes.
            try {
                audioResponse.close();
                audioResponse = null;  // Prepare for the next audio prompt.
            } catch (IOException e) {
                throw new UncheckedIOException("got an exception when closing the audio
 response", e);
            }
        }
    }

    // The conversation with the Amazon Lex bot is complete when the bot marks the
 Dialog as DialogActionType.CLOSE
    // and any prompt playback is finished. For more information, see
    // https://docs.aws.amazon.com/lexv2/latest/dg/API_runtime_DialogAction.html.
    public boolean isConversationComplete() {
        return isDialogStateClosed && lastBotResponsePlayedBack;
    }

}
```

To configure a bot to respond to input events with audio, you must first subscribe to audio events from Amazon Lex V2 and then configure the bot to provide an audio response to the input events from the user.

The following code is an AWS SDK for Java example for subscribing to audio events from Amazon Lex V2.

```
package com.lex.streaming.sample;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.services.lexruntimev2.model.AudioInputEvent;
```

```
import software.amazon.awssdk.services.lexruntimev2.model.ConfigurationEvent;
import software.amazon.awssdk.services.lexruntimev2.model.DisconnectionEvent;
import software.amazon.awssdk.services.lexruntimev2.model.PlaybackCompletionEvent;
import
 software.amazon.awssdk.services.lexruntimev2.model.StartConversationRequestEventStream;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.DataLine;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.TargetDataLine;
import java.io.IOException;
import java.io.UncheckedIOException;
import java.nio.ByteBuffer;
import java.util.Arrays;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicLong;

public class AudioEventsSubscription implements Subscription {
    private static final AudioFormat MIC_FORMAT = new AudioFormat(8000, 16, 1, true,
 false);
    private static final String AUDIO_CONTENT_TYPE = "audio/lpcm; sample-rate=8000;
 sample-size-bits=16; channel-count=1; is-big-endian=false";
    //private static final String RESPONSE_TYPE = "audio/pcm; sample-rate=8000";
    private static final String RESPONSE_TYPE = "audio/mpeg";
    private static final int BYTES_IN_AUDIO_CHUNK = 320;
    private static final AtomicLong eventIdGenerator = new AtomicLong(0);

    private final AudioInputStream audioInputStream;
    private final Subscriber<? super StartConversationRequestEventStream> subscriber;
    private final EventWriter eventWriter;
    private CompletableFuture eventWriterFuture;


    public AudioEventsSubscription(Subscriber<? super
 StartConversationRequestEventStream> subscriber) {
        this.audioInputStream = getMicStream();
        this.subscriber = subscriber;
        this.eventWriter = new EventWriter(subscriber, audioInputStream);
        configureConversation();
    }
```

```
    private AudioInputStream getMicStream() {
        try {
            DataLine.Info dataLineInfo = new DataLine.Info(TargetDataLine.class,
MIC_FORMAT);
            TargetDataLine targetDataLine = (TargetDataLine)
AudioSystem.getLine(dataLineInfo);

            targetDataLine.open(MIC_FORMAT);
            targetDataLine.start();

            return new AudioInputStream(targetDataLine);
        } catch (LineUnavailableException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void request(long demand) {
        // If a thread to write events has not been started, start it.
        if (eventWriterFuture == null) {
            eventWriterFuture = CompletableFuture.runAsync(eventWriter);
        }
        eventWriter.addDemand(demand);
    }

    @Override
    public void cancel() {
        subscriber.onError(new RuntimeException("stream was cancelled"));
        try {
            audioInputStream.close();
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }

    public void configureConversation() {
        String eventId = "ConfigurationEvent-" +
String.valueOf(eventIdGenerator.incrementAndGet());

        ConfigurationEvent configurationEvent = StartConversationRequestEventStream
                .configurationEventBuilder()
                .eventId(eventId)
                .clientTimestampMillis(System.currentTimeMillis())
```

```
                .responseContentType(RESPONSE_TYPE)
                .build();

        System.out.println("writing config event");
        eventWriter.writeConfigurationEvent(configurationEvent);
    }

    public void disconnect() {

        String eventId = "DisconnectionEvent-" +
String.valueOf(eventIdGenerator.incrementAndGet());

        DisconnectionEvent disconnectionEvent = StartConversationRequestEventStream
                .disconnectionEventBuilder()
                .eventId(eventId)
                .clientTimestampMillis(System.currentTimeMillis())
                .build();

        eventWriter.writeDisconnectEvent(disconnectionEvent);

        try {
            audioInputStream.close();
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }

    }
    //Notify the subscriber that we've finished.
    public void stop() {
        subscriber.onComplete();
    }

    public void playbackFinished() {
        String eventId = "PlaybackCompletion-" +
String.valueOf(eventIdGenerator.incrementAndGet());

        PlaybackCompletionEvent playbackCompletionEvent =
StartConversationRequestEventStream
                .playbackCompletionEventBuilder()
                .eventId(eventId)
                .clientTimestampMillis(System.currentTimeMillis())
                .build();

        eventWriter.writePlaybackFinishedEvent(playbackCompletionEvent);
```

```
    }

    private static class EventWriter implements Runnable {
        private final BlockingQueue<StartConversationRequestEventStream> eventQueue;
        private final AudioInputStream audioInputStream;
        private final AtomicLong demand;
        private final Subscriber subscriber;

        private boolean conversationConfigured;

        public EventWriter(Subscriber subscriber, AudioInputStream audioInputStream) {
            this.eventQueue = new LinkedBlockingQueue<>();

            this.demand = new AtomicLong(0);
            this.subscriber = subscriber;
            this.audioInputStream = audioInputStream;
        }

        public void writeConfigurationEvent(ConfigurationEvent configurationEvent) {
            eventQueue.add(configurationEvent);
        }

        public void writeDisconnectEvent(DisconnectionEvent disconnectionEvent) {
            eventQueue.add(disconnectionEvent);
        }

        public void writePlaybackFinishedEvent(PlaybackCompletionEvent
playbackCompletionEvent) {
            eventQueue.add(playbackCompletionEvent);
        }

        void addDemand(long l) {
            this.demand.addAndGet(l);
        }

        @Override
        public void run() {
            try {

                while (true) {
                    long currentDemand = demand.get();

                    if (currentDemand > 0) {
                        // Try to read from queue of events.
```

```
                        // If nothing is in queue at this point, read the audio events
directly from audio stream.
                        for (long i = 0; i < currentDemand; i++) {

                            if (eventQueue.peek() != null) {
                                subscriber.onNext(eventQueue.take());
                                demand.decrementAndGet();
                            } else {
                                writeAudioEvent();
                            }
                        }
                    }
                }
            } catch (InterruptedException e) {
                throw new RuntimeException("interrupted when reading data to be sent to
server");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        private void writeAudioEvent() {
            byte[] bytes = new byte[BYTES_IN_AUDIO_CHUNK];

            int numBytesRead = 0;
            try {
                numBytesRead = audioInputStream.read(bytes);
                if (numBytesRead != -1) {
                    byte[] byteArrayCopy = Arrays.copyOf(bytes, numBytesRead);

                    String eventId = "AudioEvent-" +
String.valueOf(eventIdGenerator.incrementAndGet());

                    AudioInputEvent audioInputEvent =
StartConversationRequestEventStream
                            .audioInputEventBuilder()

.audioChunk(SdkBytes.fromByteBuffer(ByteBuffer.wrap(byteArrayCopy)))
                            .contentType(AUDIO_CONTENT_TYPE)
                            .clientTimestampMillis(System.currentTimeMillis())
                            .eventId(eventId).build();

                    //System.out.println("sending audio event:" + audioInputEvent);
                    subscriber.onNext(audioInputEvent);
```

```
                    demand.decrementAndGet();
                    //System.out.println("sent audio event:" + audioInputEvent);
                } else {
                    subscriber.onComplete();
                    System.out.println("audio stream has ended");
                }

            } catch (IOException e) {
                System.out.println("got an exception when reading from audio stream");
                System.err.println(e);
                subscriber.onError(e);
            }
        }
    }
}
```

The following AWS SDK for Java example configures the Amazon Lex V2 bot to provide an audio response to the input events.

```
package com.lex.streaming.sample;

import java.io.IOException;
import java.io.InputStream;
import java.io.UncheckedIOException;
import java.util.Optional;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class AudioResponse extends InputStream{

    // Used to convert byte, which is signed in Java, to positive integer (unsigned)
    private static final int UNSIGNED_BYTE_MASK = 0xFF;
    private static final long POLL_INTERVAL_MS = 10;

    private final LinkedBlockingQueue<Integer> byteQueue = new LinkedBlockingQueue<>();

    private volatile boolean closed;

    @Override
    public int read() throws IOException {
```

```
        try {
            Optional<Integer> maybeInt;
            while (true) {
                maybeInt = Optional.ofNullable(this.byteQueue.poll(POLL_INTERVAL_MS,
 TimeUnit.MILLISECONDS));

                // If we get an integer from the queue, return it.
                if (maybeInt.isPresent()) {
                    return maybeInt.get();
                }

                // If the stream is closed and there is nothing queued up, return -1.
                if (this.closed) {
                    return -1;
                }
            }
        } catch (InterruptedException e) {
            throw new IOException(e);
        }
    }

    /**
     * Writes data into the stream to be offered on future read() calls.
     */
    public void write(byte[] byteArray) {
        // Don't write into the stream if it is already closed.
        if (this.closed) {
            throw new UncheckedIOException(new IOException("Stream already closed when
 attempting to write into it."));
        }

        for (byte b : byteArray) {
            this.byteQueue.add(b & UNSIGNED_BYTE_MASK);
        }
    }

    @Override
    public void close() throws IOException {
        this.closed = true;
        super.close();
    }
}
```

Event stream encoding

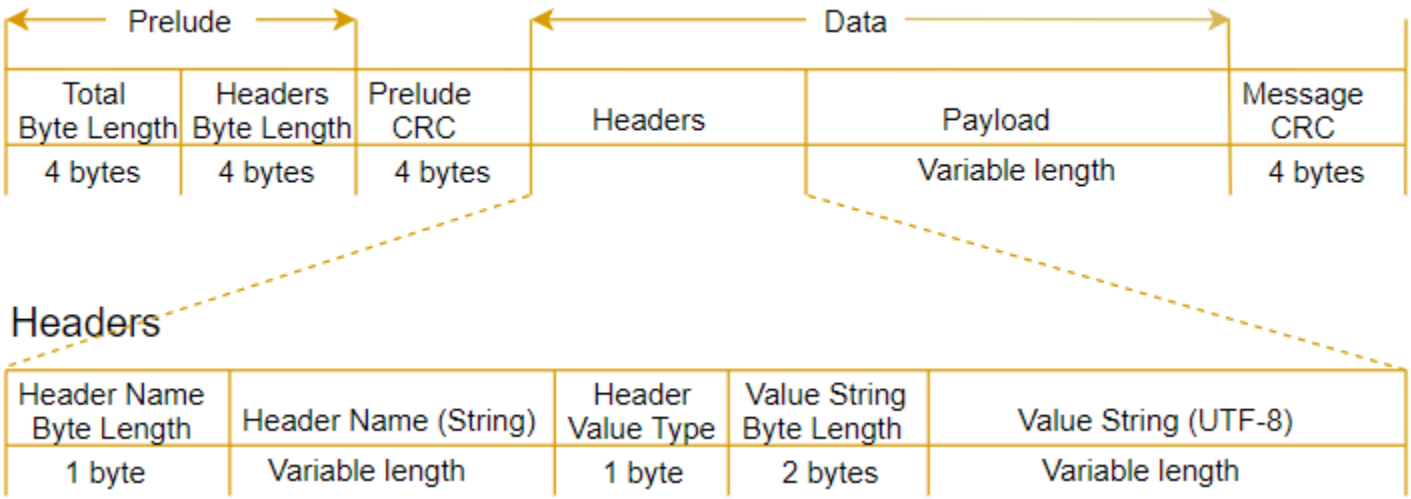
Event stream encoding provides bidirectional communication using messages between a client and a server. Data frames sent to the Amazon Lex V2 streaming service are encoded in this format. The response from Amazon Lex V2 also uses this encoding.

Each message consists of two sections: the prelude and the data. The prelude section contains the total byte length of the message and the combined byte length of all of the headers. The data section contains the headers and a payload.

Each section ends with a 4-byte big-endian integer CRC checksum. The message CRC checksum includes the prelude section and the data section. Amazon Lex V2 uses CRC32 (often referred to as GZIP CRC32) to calculate both CRCs. For more information about CRC32, see *GZIP file format specification version 4.3*.

Total message overhead, including the prelude and both checksums, is 16 bytes.

The following diagram shows the components that make up a message and a header. There are multiple headers per message.



Each message contains the following components:

- **Prelude:** Always a fixed size of 8 bytes, two fields of 4 bytes each.
  - *First 4 bytes:* The total byte-length. This is the big-endian integer byte-length of the entire message, including the 4-byte length field itself.

  - *Second 4 bytes:* The headers byte-length. This is the big-endian integer byte-length of the headers portion of the message, excluding the headers length field itself.
- **Prelude CRC:** The 4-byte CRC checksum for the prelude portion of the message, excluding the CRC itself. The prelude has a separate CRC from the message CRC to ensure that Amazon Lex V2 can detect corrupted byte-length information immediately without causing errors such as buffer overruns.
- **Headers:** Metadata annotating the message, such as the message type, content type, and so on. Messages have multiple headers. Headers are key-value pairs where the key is a UTF-8 string. Headers can appear in any order in the headers portion of the message and any given header can appear only once. For the required header types, see the following sections.
- **Payload:** The audio or text content being sent to Amazon Lex.
- **Message CRC:** The 4-byte CRC checksum from the start of the message to the start of the checksum. That includes everything in the message except the CRC itself.

Each header contains the following components. There are multiple headers per frame.

- **Header name byte-length:** The byte-length of the header name.
- **Header name:** The name of the header indicating the header type. For valid values, see the following frame descriptions.
- **Header value type:** An enumeration indicating the header value type.
- **Value string byte length:** The byte-length of the header value string.
- **Header value:** The value of the header string. Valid values for this field depend on the type of header. For valid values, see the following frame descriptions.

Enabling your Amazon Lex V2 bot to be interrupted by the user

When you start a bidirectional audio stream between an Amazon Lex V2 bot and your application, you can configure the bot to listen for user input while it is sending back a prompt. This input can be set as an interrupt. With this, the user can interrupt the prompt before the bot has finished playing it back. You can use this configuration for situations where the user might already know the answer to a question, such as when they're being prompted to provide a CVV code.

A bot knows when the user interrupts a prompt when it detects user input before your application can send back a `PlaybackCompletion` event. When the user interrupts a bot, the bot sends a `PlaybackInterruptionEvent`.

By default, the user can interrupt any prompt that the bot is streaming to your application. You can change this setting in the Amazon Lex V2 console.

You can change how a user can respond to a prompt by editing a slot. A slot is part of an intent, and it is the means by which the user provides you the information you want. Each slot has a prompt for the user to provide you with that information. To learn more about slots, see Amazon Lex V2 core concepts.

To change whether the user can interrupt a prompt (console)

1. Sign in to AWS Management Console and open the Amazon Lex V2 console at Amazon Lex V2 console.
2. Under **Bots**, select a bot.
3. Under **Language**, select the language of the bot.
4. Choose **View intents**.
5. Choose the intent.
6. For **Slots**, choose a slot.
7. Under **Advanced options**, choose **Slot prompts**.
8. Choose **More prompt options**.
9. Select or deselect **Users can interrupt the prompt when it is being read**.

You can test this functionality by creating a bot with two slots and specifying that users can't interrupt a prompt for one slot. If you interrupt an interruptible prompt, the bot sends a playback interruption event. If you interrupt an uninterruptible, the prompt continues to play.

Enabling the Amazon Lex V2 bot to wait for the user to provide more information during a pause

When you start a bidirectional stream from an Amazon Lex V2 bot to your application, you can configure the bot to wait for the user to provide additional information. There are circumstances when a user might not be ready to respond to a prompt. For example, a user might not be ready to provide their credit card information because their wallet is in another room.

By using the *Wait and continue* behavior of the Amazon Lex V2 bot, users can say phrases such as "hold on a second" to make the bot wait for them to find the information and provide it. When you

enable this behavior, the bot sends periodic reminders to the user to provide the information. It does not send back transcript events because there are no user utterances for it to transcribe.

The Amazon Lex V2 bot automatically manages a streaming conversation. You don't have to write any additional code to enable this functionality. When a bot is prompted to wait by the user, the `state` of the `Intent` is `Waiting` and the `type` of the `DialogAction` is `ElicitSlot`. You can use this information to help customize your application for your needs. For example, you can configure your application to play music when the user is looking for their credit card.

You enable the wait and continue behavior for an individual slot. To learn more about slots, see Amazon Lex V2 core concepts.

To enable wait and continue

1. Sign in to AWS Management Console and open the Amazon Lex V2 console at Amazon Lex V2 console.
2. Under **Bots**, select a bot.
3. Under **Language**, select the language of the bot.
4. Choose **View intents**.
5. Choose the intent.
6. Under **Slots**, choose a slot.
7. Under **Advanced options**, choose **Wait and continue**.
8. Under **Wait and continue** specify the following fields:

    - **Response when user wants the bot to wait** – This is how the bot responds when the user asks it to wait for the additional information.
    - **Response if the user needs the bot to continue waiting** – This is the response the bot sends to remind the user that it's still waiting for the information. You can change how frequently the bot reminds the user.
    - **Response when the user wants to continue** – This is the bot's response when the user has the requested information.

For every bot response, you can give multiple variations of the response, and one is presented to the user at random. You can also choose whether these responses can be interrupted by the user.

To test the wait and continue functionality, configure your bot to wait for user input and start a stream to an Amazon Lex V2 bot. For information on streaming to a bot, see Using the API to start a streaming conversation.

You may need to turn off the wait and continue responses. Use the **Active** toggle to set whether or not the wait and continue responses are used.

> **Note**
>
> Wait and continue is available in the following locales: ca-ES, de-AT, de-DE, en-AU, en-GB, en-IN, en-US, en-ZA, es-419, es-ES, es-US, fr-CA, fr-FR, it-IT, ja-JP, ko-KR, pt-BR, pt-PT, zh-CN, zh-HK.

Configuring fulfillment progress updates for your Lex V2 bot

When the fulfillment Lambda function for an intent is called, the bot doesn't send a response until the function completes. If the Lambda function takes more than a few seconds to complete, the user may think that the bot is unresponsive. To address this, you can configure your bot to send updates to the user while the fulfillment Lambda function is running so that the user knows that the bot is still working on their request.

When you add fulfillment updates to an intent, the bot responds at the start of fulfillment and periodically while fulfillment is in progress. When you configure the start response, you can specify a delay before the bot sends the response. With this, you can support cases where the fulfillment doesn't finish relatively quickly. When you configure an update response, you specify the frequency that you want the updates sent. You also configure a timeout to limit the time that the fulfillment function has to run.

You can also add post-fulfillment responses to a bot. This enables the bot to send a different response depending on whether fulfillment succeeds, fails, or times out.

Fulfillment updates are used only when interacting with a bot using the StartConversation operation. You can use the post-fulfillment update when interacting with the bot using the StartConversation, RecognizeText, and RecognizeUtterance operations

Fulfillment updates

Fulfillment updates are sent while your Lambda function is fulfilling an intent. When you turn on fulfillment updates, you provide a start response that is sent at the beginning of fulfillment and an update response that is sent periodically while fulfillment is in progress.

When you specify an update response, you also specify a timeout that determines how long the fulfillment function can run. You can specify a timeout length of up to 15 minutes (900 seconds).

If you turn off fulfillment updates by setting `active` to false in the console or using the CreateIntent or UpdateIntent operation, the timeout specified for the fulfillment updates isn't used and the default timeout of 30 seconds is used instead.

If the fulfillment function times out, Amazon Lex V2 does one of three things:

- Post-fulfillment response is configured and active – returns the timeout response.
- Post-fulfillment response is configured and not active – returns an exception.
- Post-fulfillment response isn't configured – returns an exception.

Start response

Amazon Lex V2 returns the start response when the Lambda fulfillment function is called during a streaming conversation. It typically tells the user that fulfilling the intent takes some time and that they should wait. The start response isn't returned when you use the `RecognizeText` or `RecognizeUtterance` operations.

You can specify up to five response messages. Amazon Lex V2 chooses one of the messages to play to the user.

You can configure a delay between when the Lambda function is called and when the start response is returned. The start response isn't returned if the Lambda function completes its work before the delay is complete.

You can use the `active` toggle in the console or the FulfillmentUpdatesSpecification structure to turn the start response on and off. When `active` is false, the start response isn't played.

Update response

Amazon Lex returns the update response periodically during a streaming conversation while the Lambda fulfillment function is running. The update response isn't played when you use

the `RecognizeText` or `RecognizeUtterance` operations. You can configure how often the update response plays. For example, you can play an update response every 30 seconds while the fulfillment function runs to let the user know that the process is running and that they should continue to wait.

You can specify up to five update messages. Amazon Lex V2 chooses a message to play to the user. Using multiple messages keeps the updates from being repetitive.

If the user provides input via voice, DTMF, or text while the fulfillment Lambda function is running, Amazon Lex V2 returns the update response to the user.

If the Lambda function completes its work before the first update period ends, the update response isn't returned.

You can use the `active` toggle in the console or the FulfillmentUpdatesSpecification structure to turn the update response on and off. When `active` is false, the update response isn't returned.

Post-fulfillment response

Amazon Lex V2 returns a post-fulfillment response when the fulfillment function ends. A post-fulfillment response can be used when fulfilling any intent, not just when streaming conversations. The post-fulfillment response lets the user know that the function is complete and the result.

You can use the `active` toggle in the console or the PostFulfillmentStatusSpecification structure to turn the post-fulfillment response on and off. When `active` is false, the response is not played.

There are three types of post-fulfillment responses:

- **Success** – returned when the fulfillment Lambda function completes its work successfully. If post-fulfillment responses aren't active. Amazon Lex V2 takes the next configured action.
- **Timeout** – returned if the Lambda function doesn't complete its work before the configured timeout period elapses. If post-fulfillment responses aren't active, Amazon Lex V2 returns an exception.
- **Failure** – returned when the Lambda function returns the status `Failed` in the response or when Amazon Lex V2 encounters an error while fulfilling the intent. If post-fulfillment responses aren't active, Amazon Lex V2 returns an exception.

You can specify up to five messages for each type. Amazon Lex V2 chooses one of the messages to play to the user.

Unlike fulfillment start and fulfillment update responses, post-fulfillment responses are played back for both streaming and non-streaming conversations.

You also have the option to override these messages by configuring the Lambda function to return a post-fulfillment message.

> **Note**
>
> If the intent has a closing response, it is returned after the post-fulfillment response.

Post-fulfillment example for Lex V2

To better understand the post-fulfillment response, let's take, as an example, a *BookTrip* bot, created to help plan a trip, with a *BookFlight* intent, configured with a fulfillment Lambda function that reserves the customer's flight with an airline. Once the slots for *BookFlight* have been elicited, Amazon Lex V2 invokes the fulfillment Lambda function. During this fulfillment process one of the following three results can happen:

- **Success** – The flight is successfully booked.
- **Timeout** – The booking process takes longer than the configured fulfillment Lambda execution time (for example, if the airline cannot be contacted within the allotted time).
- **Failure** – The booking fails for another reason.

You can leverage the post-fulfillment response to provide a more meaningful response to your customers in each of these situations. Examples for each situation are as follows:

- **Success response** – "We were able to successfully book your ticket and have sent you a confirmation email. Please feel free to reach out to us using the contact information provided in that email if you have any questions."
- **Timeout response** – "Due to heavy traffic on our systems, booking your ticket is taking longer than expected. We have your request in our queue and have sent you an email with the reference number corresponding to this request. Once we book the ticket, we will send you a confirmation of the reservation. Please feel free to reach out to us using the contact information provided in that email if you have any questions."

> **Note**
>
> If you do not configure a timeout message, Amazon Lex V2 throws a 4XX error corresponding to the use case.

- **Failure response** – "Unfortunately, we were unable to book your ticket. We have sent an email with details regarding the issue we encountered while booking your reservation."

Configuring timeouts for capturing user input with a Lex V2 bot

The Amazon Lex V2 streaming API enables a bot to automatically detect utterances in user input. When you create an intent or a slot, you can configure aspects of an utterance, such as maximum duration of an utterance, timeout while waiting for user input, or the end character for DTMF input. You can customize a bot's behavior for your use case. For example, you can limit the number of digits for a credit card number to 16.

You can also configure timeouts through session attributes when starting a conversation with a bot, and overwrite them in your Lambda function if necessary.

The configuration keys for an attribute use the following syntax:

```
x-amz-lex:<InputType>:<BehaviorName>:<IntentName>:<SlotName>
```

`InputType` can be **`audio`**, **`dtmf`**, or **`text`**.

You can configure default settings for all intents or slots in a bot by specifying **`*`** as the intent or slot name. Any intent- or slot-specific settings take precedence over default settings.

Amazon Lex V2 provides predefined session attributes for managing the way the StartConversation operations works with text, voice, or DTMF input to your bot. All predefined attributes are in the `x-amz-lex` namespace.

You can configure default settings for all intents, slots or subslots in a bot by specifying `*` as the intent or slot name. Any intent or slot-specific settings take precedence over default settings. Use these patterns for all the timeouts below.

For a composite slot's subslot you can separate by `..` For example:

```
<slotName>.<subSlotName>
```

```
x-amz-lex:allow-interrupt:<intentName>:<slotName>.<subSlotName>
```

Expression	Scenario
Intent:Slot.SubSlot	Applicable to only sub slot named 'SubSlot' inside composite slot named 'Slot'
Intent:Slot.*	Applicable to any sub slot inside composite slot named 'Slot'
Intent:*.SubSlot	Applicable to only sub slot named 'SubSlot' inside any composite slot
Intent:*.*	Applicable to any sub slot inside any composite slot

How interrupt behavior works in a Lex V2 bot

You can set up the interrupt behavior for the bot. The attribute is defined by Amazon Lex V2.

Allow interrupt

```
x-amz-lex:allow-interrupt:<intentName>:<slotName>
```

Defines whether user can interrupt the prompt played by Amazon Lex V2 bot. You can selectively turn it off.

Default: True

Set the timeouts for voice input

You can set time-out values for voice interaction with your bot using session attributes. The attributes are defined by Amazon Lex V2. These attributes enable you to specify how long Amazon Lex V2 waits for a customer to finish speaking before collecting input speech.

All of these attributes are in the `x-amz-lex:audio` namespace.

Maximum utterance length

```
x-amz-lex:audio:max-length-ms:<intentName>:<slotName>
```

Defines how long Amazon Lex V2 waits before speech input is truncated and the speech is returned to your application. You can increase the length of the input when you expect long responses, or if you want to give customers more time to provide information.

Default: 55,000 milliseconds (55 seconds). The maximum value is 55,000 milliseconds (55 seconds)

If you set the `max-length-ms` attribute to more than 55,000 milliseconds, the value will default to 55,000 milliseconds.

Voice timeout

```
x-amz-lex:audio:start-timeout-ms:<intentName>:<slotName>
```

How long a bot waits before assuming that the customer isn't going to speak. You can increase the time in situations where the customer may need more time to find or recall information before speaking. For example, you might want to give customers time to get out their credit card so they can enter the number.

Default: 4,000 milliseconds (4 seconds)

Silence timeout

```
x-amz-lex:audio:end-timeout-ms:<intentName>:<slotName>
```

How long a bot waits after the customer stops speaking to assume the utterance is finished. You can increase the time in situations where periods of silence are expected while providing input.

Default: 600 milliseconds (0.6 seconds)

Allow audio input

```
x-amz-lex:allow-audio-input:<intentName>:<slotName>
```

You can enable this attribute so that the bot accepts user input only via audio modality. The bot will not accept audio input if this flag is set to false. The value is set to true by default.

Default: True

Timeouts for text input

Use the following session attribute to specify how your bot behaves with the text conversation mode.

This attribute is in the `x-amz-lex:text` namespace.

Start timeout threshold

```
x-amz-lex:text:start-timeout-ms:<intentName>:<slotName>
```

How long the bot waits before re-prompting a customer for text input. You can increase the time in situations where you'd like to allow the customer more time to find or recall information before providing text input. For example, you might want to give customers more time to find details on their order. Alternatively, you may reduce the threshold to prompt customers earlier.

Default: 30,000 milliseconds (30 seconds)

Set configuration for DTMF input

Use the following session attributes to specify how your Amazon Lex V2 bot responds to DTMF input when using an audio conversation.

All of these attributes are in the `x-amz-lex:dtmf` namespace.

Deletion character

```
x-amz-lex:dtmf:deletion-character:<intentName>:<slotName>
```

The DTMF character that clears the accumulated DTMF digits and immediately ends the input.

Default: *

End character

```
x-amz-lex:dtmf:end-character:<intentName>:<slotName>
```

The DTMF character that immediately ends input. If the user does not press this character, the input ends after the end timeout.

Default: #

End timeout

```
x-amz-lex:dtmf:end-timeout-ms:<intentName>:<slotName>
```

How long the bot should wait from the last DTMF character input before assuming that the input has concluded.

Default: 5000 milliseconds (5 seconds)

Maximum number of DTMF digits per utterance

```
x-amz-lex:dtmf:max-length:<intentName>:<slotName>
```

The maximum number of DTMF digits allowed in an utterance. For example, you could set this value to 16 to limit the number of characters that can be input for a credit card number. This value can't be increased.

Default: 1024 characters

Allow DTMF input

You can set the type of input that the bot can accept using session attributes. The attributes are defined by Amazon Lex V2.

```
x-amz-lex:allow-dtmf-input:<intentName>:<slotName>
```

You can enable this attribute so that the bot accepts user input via DTMF modality. The bot will not accept DTMF input if this flag is set to false. The value is set to true by default.

Default: True

Importing and exporting bots in Lex V2

You can export a bot definition, a bot locale or a custom vocabulary and then import it back to create a new resource or to overwrite an existing resource in an AWS account. For example, you can export a bot from a test account and then create a copy of the bot in your production account. You can also copy a bot from one AWS Region to another Region.

You can change the resources of the exported resource before importing it. For example, you can export a bot and then edit the JSON file for a slot to add or remove slot value elicitation utterances from a specific slot. After you finish editing the definition, you can import the modified file.

Topics

Exporting bots from Lex V2

You export a bot, bot locale, or custom vocabulary using the console or the `CreatExport` operation. You specify the resource to export, and you can provide an optional password to help protect the .zip file when you start an export. After you download the .zip file, you must use the password to access the file before you can use it. For more information, see Using a password when importing or exporting.

Exporting is an asynchronous operation. Once you have started the export, you can use the console or the `DescribeExport` operation to monitor the progress of the export. Once the export is complete, the console or the `DescribeExport` operation shows a status of `COMPLETED`, and the console downloads the export .zip file to your browser. If you use the `DescribeExport` operation, Amazon Lex V2 provides a pre-signed Amazon S3 URL where you can download the results of the export. The download URL is available for only five minutes, but you can get a new URL by calling the `DescribeExport` operation again.

You can see the history of exports for a resource with the console or with the `ListExports` operation. The results show the exports along with their current status. An export is available in the history for seven days.

When you export the `Draft` version of a bot or a bot locale, it is possible for the definition in the JSON file to be in an inconsistent state because the `Draft` version of a bot or bot locale can be changed while an export is in progress. If the `Draft` version is changed while it is being exported, the changes may not be included in the export file.

When you export a bot locale, Amazon Lex exports all of the information that defines the locale, including the locale, custom vocabulary, intents, slot types, and slots.

When you export a bot, Amazon Lex exports all of the locales defined for the bot, including the intents, slot types, and slots. The following items are not exported with a bot:

- Bot aliases
- Role ARN associated with a bot
- Tags associated with bots and bot aliases
- Lambda code hooks associated with a bot alias

The role ARN and tags are entered as request parameters when you import a bot. You need to create bot aliases and assign Lambda code hooks after importing, if necessary.

You can remove an export and the associated .zip file using the console or the `DeleteExport` operation.

For an example of exporting a bot using the console, see Exporting a Lex V2 bot (console).

IAM permissions required to export bots in Lex V2

To export bots, bot locales, and custom vocabularies, the user running the export must have the following IAM permissions.

API	• Required IAM actions	Resource
CreateExport	• CreateExport	Bot
UpdateExport	• UpdateExport	Bot
DescribeExport	• DescribeExport • DescribeBot • DescribeCustomVocabulary	Bot

API	• Required IAM actions	Resource
	• DescribeLocale • DescribeIntent • DescribeSlot • DescribeSlotType • ListLocale • ListIntent • ListSlot • ListSlotType	
DescribeExport for custom vocabularies	• DescribeExport • DescribeCustomVocabulary	bot
DeleteExport	• DeleteExport	Bot
ListExports	• ListExports	*

For an example IAM policy, see Allow a user to export bots and bot locales.

Exporting a Lex V2 bot (console)

You can export a bot from the bot list, from the list of versions, or from the version details page. When you choose a version, Amazon Lex V2 exports that version. The following instructions assume that you start exporting the bot from the list of bots, but when you start with a version the steps are the same.

To export a bot using the console

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. From the list of bots, choose the bot to export.
3. From **Action**, choose **Export**.
4. Choose the bot version, platform, and export format.
5. (Optional) Enter a password for the .zip file. Providing a password helps protect the output archive.

6. Choose **Export**.

After you start the export, you return to the list of bots. To monitor the progress of the export, use the **Import/export history** list. When the status of the export is **Complete**, the console automatically downloads the .zip file to your computer.

To download the export again, on the import/export list, choose the export, and then choose **Download**. You can provide a password for the downloaded .zip file.

To export a bot language

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. From the list of bots, choose the bot whose language you want to export.
3. From **Add languages**, choose **View languages**.
4. From the **All languages** list, choose the language to export.
5. From **Action**, choose **Export**.
6. Choose the bot version, platform, and format.
7. (Optional) Enter a password for the .zip file. Providing a password helps protect the output archive.
8. Choose **Export**.

After you start the export, you return to the list of languages. To monitor the progress of the export, use the **Import/export history** list. When the status of the export is **Complete**, the console automatically downloads the .zip file to your computer.

To download the export again, on the import/export list, choose the export, and then choose **Download**. You can provide a password for the downloaded .zip file.

Importing bots in Lex V2

To use the console to import a previously exported bot, bot locale or custom vocabulary, you provide the file location on your local computer and the optional password to unlock the file. For an example, see Importing a Lex V2 bot (console).

When you use the API, importing a resource is a three step process:

1. Create an upload URL using the `CreateUploadUrl` operation. You don't need to create an upload URL when you are using the console.
2. Upload the .zip file that contains the resource definition.
3. Start the import with the `StartImport` operation.

The upload URL is a pre-signed Amazon S3 URL with write permission. The URL is available for five minutes after it is generated. If you password protect the .zip file, you must provide the password when you start the import. For more information, see Using a password when importing or exporting.

An import is an asynchronous process. You can monitor the progress of an import using the console or the `DescribeImport` operation.

When you import a bot or bot locale, there may be conflicts between resource names in the import file and existing resource names in Amazon Lex V2. Amazon Lex V2 can handle the conflict in three ways:

- **Fail on conflict** – The import stops and no resources are imported from the import .zip file.
- **Overwrite** – Amazon Lex V2 imports all of the resources from the import .zip file and replaces any existing resource with the definition from the import file.
- **Append** – Amazon Lex V2 imports all of the resources from the import .zip file and adds to any existing resource with the definition from the import file. This is available only for the bot locale.

You can see a list of the imports to a resource using the console or the `ListImports` operation. Imports remain in the list for seven days. You can use the console or the `DescribeImport` operation to see details about a specific import.

You can also remove an import and the associated .zip file using the console or the `DeleteImport` operation.

For an example of importing a bot using the console, see Importing a Lex V2 bot (console).

IAM permissions required to import

To import bots, bot locales, and custom vocabularies, the user running the import must have the following IAM permissions.

API	Required IAM actions	Resource
CreateUploadUrl	• CreateUploadUrl	*
StartImport for bot and bot locale	• StartImport • iam:PassRole • CreateBot • CreateCustomVocabulary • CreateLocale • CreateIntent • CreateSlot • CreateSlotType • UpdateBot • UpdateCustomVocabulary • UpdateLocale • UpdateIntent • UpdateSlot • UpdateSlotType • DeleteBot • DeleteCustomVocabulary • DeleteLocale • DeleteIntent • DeleteSlot • DeleteSlotType	1. To import a new bot: bot, bot alias. 2. To overwrite an existing bot: bot. 3. To import a new locale: bot.
StartImport for custom vocabularies	• StartImport • CreateCustomVocabulary • DeleteCustomVocabulary • UpdateCustomVocabulary	bot
DescribeImport	• DescribeImport	Bot
DeleteImport	• DeleteImport	Bot

API	Required IAM actions	Resource
ListImports	• ListImports	*

For an example IAM policy, see Allow a user to import bots and bot locales.

Importing a Lex V2 bot (console)

To import a bot using the console

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. From **Action**, choose **Import**.
3. In **Input file**, give the bot a name and then choose the .zip file that contains the JSON files that define the bot.
4. If the .zip file is password protected, enter the password for the .zip file. Password protecting the archive is optional, but it helps protect the contents.
5. Create or enter the IAM role that defines permissions for your bot.
6. Indicate whether your bot is subject to the Children's Online Privacy Protection Act (COPPA).
7. Provide an idle timeout setting for your bot. If you don't provide a value, the value from the zip file is used. If the .zip file does not contain a timeout setting, Amazon Lex V2 uses the default of 300 seconds (five minutes).
8. (Optional) Add tags for your bot.
9. Choose whether to warn about overwriting existing bots with the same name. If you enable warnings, if the bot you are importing would overwrite an existing bot, you receive a warning and the bot is not imported. If you disable warnings, the imported bot replaces the existing bot with the same name.
10. Choose **Import**.

After you start the import, you return to the list of bots. To monitor the progress of the import, use the **Import/export history** list. When the status of the import is **Complete**, you can choose the bot from the list of bots to modify or build the bot.

To import a bot language

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. From the list of bots, choose the bot to which you want to import a language.
3. From **Add languages**, choose **View languages**.
4. From **Action**, choose **import**.
5. In **Input file**, choose the file that contains the language to import. If you protected the .zip file, provide the password in **Password**.
6. In **Language**, choose the language to import as. The language doesn't have to match the language in the import file. You can copy the intents from one language to another.
7. In **Voice**, choose the Amazon Polly voice to use for voice interaction, or choose **None** for a text-only bot.
8. In **Confidence score threshold**, enter the threshold where Amazon Lex V2 inserts the `AMAZON.FallbackIntent`, the `AMAZON.KendraSearchIntent`, or both when returning alternative intents.
9. Choose whether to warn about overwriting an existing language. If you enable warnings, if the language you are importing would overwrite an existing language, you receive a warning and the language is not imported. If you disable warnings, the imported language replaces the existing language.
10. Choose **Import** to start importing the language.

After you start the import, you return to the list of languages. To monitor the progress of the import, use the **Import/export history** list. When the status of the import is **Complete**, you can choose the language from the list of bots to modify or build the bot.

Using a password when importing or exporting

Amazon Lex V2 can password protect your export archives or read your protected import archives using standard .zip file compression. You should always password protect your import and export archives.

Amazon Lex V2 sends your export archive to an S3 bucket, and it is available to you with a pre-signed S3 URL. The URL is only available for five minutes. The archive is available to anyone with

access to the download URL. To help protect the data in the archive, provide a password when you export the resource. If you need to get the archive after the URL expires, you can use the console or the `DescribeExport` operation to get a new URL.

If you lose the password for an export archive, you can create a new password for an existing file by choosing **Download** from the import/export history table or by using the `UpdateExport` operation. If you choose **Download** in the history table for an export and you don't provide a password, Amazon Lex V2 downloads an unprotected zip file.

JSON format for importing and exporting bots in Lex V2

You import and export bots, bot locales, or custom vocabularies from Amazon Lex V2 using a .zip file that contains JSON structures that describe the parts of the resource. When you export a resource, Amazon Lex V2 creates the .zip file and makes it available to you using an Amazon S3 pre-signed URL. When you import a resource, you must create a .zip file that contains the JSON structures and upload it to an S3 pre-signed URL.

Amazon Lex creates the following directory structure in the .zip file when you export a bot. When you export a bot locale, only the structure under the locale is exported. When you export a custom vocabulary, only the structure under the custom vocabulary is exported.

```
BotName_BotVersion_ExportID_LexJson.zip
            -or-
BotName_BotVersion_LocaleId_ExportId_LEX_JSON.zip
        --> manifest.json
        --> BotName
        ----> Bot.json
        ----> BotLocales
        ------> Locale_A
        --------> BotLocale.json
        --------> Intents
        ----------> Intent_A
        ------------> Intent.json
        ------------> Slots
        --------------> Slot_A
        ----------------> Slot.json
        --------------> Slot_B
        ----------------> Slot.json
        ----------> Intent_B
                        ...
```

```
        --------> SlotTypes
        ----------> SlotType_A
        ------------> SlotType.json
        ----------> SlotType_B
                        ...
        --------> CustomVocabulary
        ------------> CustomVocabulary.json

        ------> Locale_B
                        ...
```

Manifest file structure

The manifest file contains metadata for the export file.

```
{
    "metadata": {
        "schemaVersion": "1.0",
        "fileFormat": "LexJson",
        "resourceType": "Bot | BotLocale | CustomVocabulary"
    }
}
```

Bot file structure

The bot file contains the configuration information for the bot.

```
{
    "name": "BotName",
    "identifier": "identifier",
    "version": "number",
    "description": "description",
    "dataPrivacy": {
        "childDirected": true | false
    },
    "idleSessionTTLInSeconds": seconds
}
```

Bot locale file structure

The bot locale file contains a description of the locale or language of a bot. When you export a bot, there can be more than one bot locale file in the .zip file. When you export a bot locale, there is only one locale in the zip file.

```
{
    "name": "locale name",
    "identifier": "locale ID",
    "version": "number",
    "description": "description",
    "voiceSettings": {
        "voiceId": "voice",
        "engine": "standard | neural
    },
    "nluConfidenceThreshold": number
}
```

Intent file structure

The intent file contains the configuration information for an intent. There is one intent file in the .zip file for each intent in a specific locale.

The following is an example of a JSON structure for the BookCar intent in the sample BookTrip bot. For a complete example of the JSON structure for an intent, see the CreateIntent operation.

```
{
    "name": "BookCar",
    "identifier": "891RWHHICO",
    "description": "Intent to book a car.",
    "parentIntentSignature": null,
    "sampleUtterances": [
        {
            "utterance": "Book a car"
        },
        {
            "utterance": "Reserve a car"
        },
        {
            "utterance": "Make a car reservation"
        }
    ],
```

```
    "intentConfirmationSetting": {
        "confirmationPrompt": {
            "messageGroupList": [
                {
                    "message": {
                        "plainTextMessage": {
                            "value": "OK, I have you down for a {CarType} hire in
{PickUpCity} from {PickUpDate} to {ReturnDate}.  Should I book the reservation?"
                        },
                        "ssmlMessage": null,
                        "customPayload": null,
                        "imageResponseCard": null
                    },
                    "variations": null
                }
            ],
            "maxRetries": 2
        },
        "declinationResponse": {
            "messageGroupList": [
                {
                    "message": {
                        "plainTextMessage": {
                            "value": "OK, I have cancelled your reservation in
progress."
                        },
                        "ssmlMessage": null,
                        "customPayload": null,
                        "imageResponseCard": null
                    },
                    "variations": null
                }
            ]
        }
    },
    "intentClosingSetting": null,
    "inputContexts": null,
    "outputContexts": null,
    "kendraConfiguration": null,
    "dialogCodeHook": null,
    "fulfillmentCodeHook": null,
    "slotPriorities": [
        {
            "slotName": "DriverAge",
```

```
            "priority": 4
        },
        {
            "slotName": "PickUpDate",
            "priority": 2
        },
        {
            "slotName": "ReturnDate",
            "priority": 3
        },
        {
            "slotName": "PickUpCity",
            "priority": 1
        },
        {
            "slotName": "CarType",
            "priority": 5
        }
    ]
}
```

Slot file structure

The slot file contains the configuration information for a slot in an intent. There is one slot file in the .zip file for each slot defined for an intent in a specific locale.

The following example is the JSON structure of a slot that enables the customer to choose the type of car they wish to rent in the BookCar intent in the BookTrip example bot. For a complete example of the JSON structure for an slot, see the CreateSlot operation.

```
{
    "name": "CarType",
    "identifier": "KDHJWNGZGC",
    "description": "Type of car being reserved.",
    "multipleValuesSetting": {
        "allowMutlipleValues": false
    },
    "slotTypeName": "CarTypeValues",
    "obfuscationSetting": null,
    "slotConstraint": "Required",
    "defaultValueSpec": null,
    "slotValueElicitationSetting": {
        "promptSpecification": {
```

```
            "messageGroupList": [
                {
                    "message": {
                        "plainTextMessage": {
                            "value": "What type of car would you like to rent?  Our
 most popular options are economy, midsize, and luxury"
                        },
                        "ssmlMessage": null,
                        "customPayload": null,
                        "imageResponseCard": null
                    },
                    "variations": null
                }
            ],
            "maxRetries": 2
        },
        "sampleValueElicitingUtterances": null,
        "waitAndContinueSpecification": null,
    }
}
```

The following example shows the JSON structure of a composite slot.

```
{
  "name": "CarType",
  "identifier": "KDHJWNGZGC",
  "description": "Type of car being reserved.",
  "multipleValuesSetting": {
      "allowMutlipleValues": false
  },
  "slotTypeName": "CarTypeValues",
  "obfuscationSetting": null,
  "slotConstraint": "Required",
  "defaultValueSpec": null,
  "slotValueElicitationSetting": {
      "promptSpecification": {
          "messageGroupList": [
              {
                  "message": {
                      "plainTextMessage": {
                          "value": "What type of car would you like to rent?  Our most
 popular options are economy, midsize, and luxury"
                      },
```

```
                    "ssmlMessage": null,
                    "customPayload": null,
                    "imageResponseCard": null
                },
                "variations": null
            }
        ],
        "maxRetries": 2
    },
    "sampleValueElicitingUtterances": null,
    "waitAndContinueSpecification": null,
},
"subSlotSetting": {
  "slotSpecifications": {
    "firstname": {
      "valueElicitationSetting": {
        "promptSpecification": {
          "allowInterrupt": false,
          "messageGroupsList": [
            {
              "message": {
                "imageResponseCard": null,
                "ssmlMessage": null,
                "customPayload": null,
                "plainTextMessage": {
                  "value": "please provide firstname"
                }
              },
              "variations": null
            }
          ],
          "maxRetries": 2,
          "messageSelectionStrategy": "Random"
        },
        "defaultValueSpecification": null,
        "sampleUtterances": [
          {
            "utterance": "my name is {firstName}"
          }
        ],
        "waitAndContinueSpecification": null
      },
      "slotTypeId": "AMAZON.FirstName"
    },
```

```
      "eyeColor": {
        "valueElicitationSetting": {
          "promptSpecification": {
            "allowInterrupt": false,
            "messageGroupsList": [
              {
                "message": {
                  "imageResponseCard": null,
                  "ssmlMessage": null,
                  "customPayload": null,
                  "plainTextMessage": {
                    "value": "please provide eye color"
                  }
                },
                "variations": null
              }
            ],
            "maxRetries": 2,
            "messageSelectionStrategy": "Random"
          },
          "defaultValueSpecification": null,
          "sampleUtterances": [
            {
              "utterance": "eye color is {eyeColor}"
            },
            {
              "utterance": "I have eyeColor eyes"
            }
          ],
          "waitAndContinueSpecification": null
        },
        "slotTypeId": "7FEVCB2PQE"
      }
    },
    "expression": "(firstname OR eyeColor)"
  }
}
```

Slot type file structure

The slot type file contains the configuration information for a custom slot type used in a language or locale. There is one slot type file in the .zip file for each custom slot type in a specific locale.

The following is the JSON structure for the slot type that lists the types of cars available in the BookTrip example bot. For a complete example of the JSON structure for a slot type, see the CreateSlotType operation.

```
{
    "name": "CarTypeValues",
    "identifier": "T1YUHGD9ZR",
    "description": "Enumeration representing possible types of cars available for
 hire",
    "slotTypeValues": [{
        "synonyms": null,
        "sampleValue": {
            "value": "economy"
        }
    }, {
        "synonyms": null,
        "sampleValue": {
            "value": "standard"
        }
    }, {
        "synonyms": null,
        "sampleValue": {
            "value": "midsize"
        }
    }, {
        "synonyms": null,
        "sampleValue": {
            "value": "full size"
        }
    }, {
        "synonyms": null,
        "sampleValue": {
            "value": "luxury"
        }
    }, {
        "synonyms": null,
        "sampleValue": {
            "value": "minivan"
        }
    }],
    "parentSlotTypeSignature": null,
    "valueSelectionSetting": {
        "resolutionStrategy": "TOP_RESOLUTION",
```

```
        "advancedRecognitionSetting": {
            "audioRecognitionStrategy": "UseSlotValuesAsCustomVocabulary"
        },
        "regexFilter": null
    }
}
```

The following example shows the JSON structure for a composite slot type.

```
{
  "name": "CarCompositeType",
  "identifier": "TPA3CC9V",
  "description": null,
  "slotTypeValues": null,
  "parentSlotTypeSignature": null,
  "valueSelectionSetting": {
    "regexFilter": null,
    "resolutionStrategy": "CONCATENATION"
  },
  "compositeSlotTypeSetting": {
    "subSlots": [
      {
        "name": "model",
        "slotTypeId": "MODELTYPEID" # custom slot type Id for model
      },
      {
        "name": "city",
        "slotTypeId": "AMAZON.City"
      },
      {
        "name": "country",
        "slotTypeId": "AMAZON.Country"
      },
      {
        "name": "make",
        "slotTypeId": "MAKETYPEID" # custom slot type Id for make
      }
    ]
  }
}
```

The following is a slot type that uses a custom grammar to understand the customer's utterances. For more information, see Grammar slot type.

```
{
  "name": "custom_grammar",
  "identifier": "7KEAQIQKPX",
  "description": "Slot type using a custom grammar",
  "slotTypeValues": null,
  "parentSlotTypeSignature": null,
  "valueSelectionSetting": null,
  "externalSourceSetting": {
    "grammarSlotTypeSetting": {
      "source": {
        "kmsKeyArn": "arn:aws:kms:Region:123456789012:alias/customer-grxml-key",
        "s3BucketName": "grxml-test",
        "s3ObjectKey": "grxml_files/grammar.grxml"
      }
    }
  }
}
```

Custom vocabulary file structure

The custom vocabulary file contains the entries in a custom vocabulary for single language or locale. There is one custom vocabulary file in the .zip file for each locale that has a custom vocabulary.

The following is a custom vocabulary file for a bot that takes restaurant orders. There is one file per locale in the bot.

```
{
    "customVocabularyItems": [
        {
            "weight": 3,
            "phrase": "wafers"
        },
        {
            "weight": null,
            "phrase": "extra large"
        },
        {
            "weight": null,
            "phrase": "cremini mushroom soup"
        },
        {
```

```
            "weight": null,
            "phrase": "ramen"
        },
        {
            "weight": null,
            "phrase": "orzo"
        }
    ]
}
```

Tagging resources in Lex V2

To help you manage your Amazon Lex V2 bots and bot aliases, you can assign metadata to each resource as *tags*. A tag is a label that you assign to an AWS resource. Each tag consists of a key and a value.

Tags enable you to categorize your AWS resources in different ways, for example, by purpose, owner, or application. Tags help you to:

- Identify and organize your AWS resources. Many AWS resources support tagging, so you can assign the same tag to resources in different services to indicate that the resources are the same. For example, you can tag a bot and the Lambda functions that it uses with the same tag.
- Allocate costs. You activate tags on the AWS Billing and Cost Management dashboard. AWS uses the tags to categorize your costs and deliver a monthly cost allocation report to you. For Amazon Lex V2, you can allocate costs for each alias using tags specific to the alias. For more information, see Use cost allocation tags in the *AWS Billing and Cost Management User Guide*.
- Control access to your resources. You can use tags with Amazon Lex V2 to create policies to control access to Amazon Lex V2 resources These policies can be attached to an IAM role or user to enable tag-based access control.

You can work with tags using the AWS Management Console, the AWS Command Line Interface, or the Amazon Lex V2 API.

Tagging your resources with the console or API

If you are using the Amazon Lex V2 console, you can tag resources when you create them, or you can add the tags later. You can also use the console to update or remove existing tags.

If you are using the AWS CLI or Amazon Lex V2 API, you use the following operations to manage tags for your resource:

- CreateBot and CreateBotAlias – apply tags when you create a bot or a bot alias.
- ListTagsForResource – view the tags associated with a resource.
- TagResource – add and modify tags on an existing resource.
- UntagResource – remove tags from a resource.

The following resources in Amazon Lex V2 support tagging:

- Bots – use an Amazon Resource Name (ARN) like the following:
  - `arn:aws:lex:${Region}:${account}:bot/${bot-id}`
- Bot aliases – use an ARN like the following:
  - `arn:aws:lex:${Region}:${account}:bot-alias/${bot-id}/${bot-alias-id}`

The `bot-id` and `bot-alias-id` values are capitalized alphanumeric strings 10 characters long.

Tag restrictions when using Lex V2

The following basic restrictions apply to tags on Amazon Lex V2 resources:

- Maximum number of keys – 50 using the console, 200 using the API
- Maximum key length – 128 characters
- Maximum value length – 256 characters
- Valid characters for key and value – a-z, A-Z, 0-9, space, and the following characters: _.:/=+- and @
- Keys and values are case-sensitive
- Don't use `aws:` as a prefix for keys, it's reserved for AWS use

Tagging resources (console)

You can use the console to manage tags on a bot or bot alias. You can add tags when you create the resource, or you can add, modify, or remove tags from existing resources.

To add a tag when you create a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose **Create bot**.
3. In the **Advanced settings** section of **Configure bot settings**, choose **Add new tag**. You can add tags to the bot and to the `TestBotAlias` alias.
4. Choose **Next** to continue creating your bot.

To add a tag when you create a bot alias

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot that you want to add the bot alias to.
3. From the left menu, choose **Aliases** and then choose **Create alias**.
4. In **General info**, choose **Add new tag** from **Tags.**
5. Choose **Create**.

To add, remove, or modify a tag on an existing bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot that you want to modify.
3. From the left menu, choose **Settings**, and then choose **Edit**.
4. In **Tags**, make your changes.
5. Choose **Save** to save your changes to the bot.

To add, remove, or modify a tag on an existing alias

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose the bot that you want to modify.
3. From the left menu, choose **Aliases** and then from the list of aliases, choose the alias to modify.
4. From **Alias details**, in **Tags**, choose **Modify tags**.
5. In **Manage tags**, make your changes.
6. Choose **Save** to save your changes to the alias.

Security in Amazon Lex V2

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Lex V2, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon Lex V2. The following topics show you how to configure Amazon Lex V2 to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Amazon Lex V2 resources.

Topics

Data protection in Amazon Lex V2

Amazon Lex V2 conforms to the AWS shared responsibility model, which includes regulations and guidelines for data protection. AWS is responsible for protecting the global infrastructure that runs all the AWS services. AWS maintains control over data hosted on this infrastructure, including the security configuration controls for handling customer content and personal data. AWS customers and APN partners, acting either as data controllers or data processors, are responsible for any personal data that they put in the AWS Cloud.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM), so that each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources.

- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This includes when you work with Amazon Lex V2 or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Amazon Lex V2 or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

For more information about data protection, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

Encryption at rest

Amazon Lex V2 encrypts user utterances and other information that it stores.

Topics

- Sample utterances
- Session attributes
- Request attributes

Sample utterances

When you develop a bot, you can provide sample utterances for each intent and slot. You can also provide custom values and synonyms for slots. This information is encrypted at rest, and it is used only to build the bot and create the customer experience.

Session attributes

Session attributes contain application-specific information that is passed between Amazon Lex V2 and client applications. Amazon Lex V2 passes session attributes to all AWS Lambda functions configured for a bot. If a Lambda function adds or updates session attributes, Amazon Lex V2 passes the new information back to the client application.

Session attributes persist in an encrypted store for the duration of the session. You can configure the session to remain active for a minimum of 1 minute and up to 24 hours after the last user utterance. The default session duration is 5 minutes.

Request attributes

Request attributes contain request-specific information and apply only to the current request. A client application uses request attributes to send information to Amazon Lex V2 at runtime.

You use request attributes to pass information that doesn't need to persist for the entire session. Because request attributes don't persist across requests, they aren't stored.

Encryption in transit

Amazon Lex V2 uses the HTTPS protocol to communicate with your client application. It uses HTTPS and AWS signatures to communicate with other services, such as Amazon Polly and AWS Lambda on your application's behalf.

Identity and access management for Amazon Lex V2

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon Lex V2 resources. IAM is an AWS service that you can use with no additional charge.

Topics

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting Amazon Lex V2 identity and access)
- **Service administrator** - determine user access and submit permission requests (see How Amazon Lex V2 works with IAM)
- **IAM administrator** - write policies to manage access (see Identity-based policy examples for Amazon Lex V2)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-

based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Lex V2 works with IAM

Before you use IAM to manage access to Amazon Lex V2, learn what IAM features are available to use with Amazon Lex V2.

IAM features you can use with Amazon Lex V2

IAM feature	Amazon Lex V2 support
Identity-based policies	*Yes*
Resource-based policies	*Yes*
Policy actions	*Yes*
Policy resources	*Yes*
Policy condition keys	*No*
ACLs	*No*
ABAC (tags in policies)	*Yes*
Temporary credentials	*No*
Principal permissions	*Yes*
Service roles	*Yes*
Service-linked roles	*Partial*

To get a high-level view of how Amazon Lex V2 and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Amazon Lex V2

Supports identity-based policies: *Yes*

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Amazon Lex V2

To view examples of Amazon Lex V2 identity-based policies, see Identity-based policy examples for Amazon Lex V2.

Resource-based policies within Amazon Lex V2

Supports resource-based policies: *Yes*

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM role trust policies and Amazon S3 bucket policies. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include users, roles, federated users, or AWS services.

You can't use cross-account or cross-region policies with Amazon Lex V2. If you create a policy for a resource with a cross-account or cross-region ARN, Amazon Lex V2 returns an error.

The Amazon Lex V2 service supports resource-based policies called a *bot policy* and a *bot alias* policy, which are attached to a bot or a bot alias. These policies define which principals can perform actions on the bot or bot alias.

Actions can only be used on specific resources. For example, the `UpdateBot` action can only be used on bot resources, the `UpdateBotAlias` action can only be used on bot alias resources. If you specify an action in a policy that can't be used on the resource specified in the policy, Amazon Lex V2 returns an error. For the list of actions and the resources that they can be used with, see the following table.

Action	Supports resource-based policy	Resource
BuildBotLocale	Supported	BotId
CreateBot	No	
CreateBotAlias	No	
CreateBotChannel [permission only]	Supported	BotId

Action	Supports resource-based policy	Resource
CreateBotLocale	Supported	BotId
CreateBotVersion	Supported	BotId
CreateExport	Supported	BotId
CreateIntent	Supported	BotId
CreateResourcePolicy	Supported	BotId, BotAliasId
CreateSlot	Supported	BotId
CreateSlotType	Supported	BotId
CreateUploadUrl	No	
DeleteBot	Supported	BotId, BotAliasId
DeleteBotAlias	Supported	BotAliasId
DeleteBotChannel [permission only]	Supported	BotId
DeleteBotLocale	Supported	BotId
DeleteBotVersion	Supported	BotId
DeleteExport	Supported	BotId
DeleteImport	Supported	BotId
DeleteIntent	Supported	BotId
DeleteResourcePolicy	Supported	BotId, BotAliasId
DeleteSession	Supported	BotAliasId
DeleteSlot	Supported	BotId

Action	Supports resource-based policy	Resource
DeleteSlotType	Supported	BotId
DescribeBot	Supported	BotId
DescribeBotAlias	Supported	BotAliasId
DescribeBotChannel [permission only]	Supported	BotId
DescribeBotLocale	Supported	BotId
DescribeBotVersion	Supported	BotId
DescribeExport	Supported	BotId
DescribeImport	Supported	BotId
DescribeIntent	Supported	BotId
DescribeResourcePolicy	Supported	BotId, BotAliasId
DescribeSlot	Supported	BotId
DescribeSlotType	Supported	BotId
GetSession	Supported	BotAliasId
ListBotAliases	Supported	BotId
ListBotChannels [permission only]	Supported	BotId
ListBotLocales	Supported	BotId
ListBots	No	
ListBotVersions	Supported	BotId
ListBuiltInIntents	No	

Action	Supports resource-based policy	Resource
ListBuiltIntSlotTypes	No	
ListExports	No	
ListImports	No	
ListIntents	Supported	BotId
ListSlots	Supported	BotId
ListSlotTypes	Supported	BotId
PutSession	Supported	BotAliasId
RecognizeText	Supported	BotAliasId
RecognizeUtterance	Supported	BotAliasId
StartConversation	Supported	BotAliasId
StartImport	Supported	BotId, BotAliasId
TagResource	No	
UpdateBot	Supported	BotId
UpdateBotAlias	Supported	BotAliasId
UpdateBotLocale	Supported	BotId
UpdateBotVersion	Supported	BotId
UpdateExport	Supported	BotId
UpdateIntent	Supported	BotId
UpdateResourcePolicy	Supported	BotId, BotAliasId
UpdateSlot	Supported	BotId

Action	Supports resource-based policy	Resource
UpdateSlotType	Supported	BotId
UntagResource	No	

To learn how to attach a resource-based policy to a bot or bot alias, see Resource-based policy examples for Amazon Lex V2.

Resource-based policy examples within Amazon Lex V2

To view examples of Amazon Lex V2 resource-based policies, see Resource-based policy examples for Amazon Lex V2.

Policy actions for Amazon Lex V2

Supports policy actions: *Yes*

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Amazon Lex V2 actions, see Actions defined by Amazon Lex V2 in the *Service Authorization Reference*.

Policy actions in Amazon Lex V2 use the following prefix before the action:

```
lex
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "lex:action1",
      "lex:action2"
         ]
```

To view examples of Amazon Lex V2 identity-based policies, see Identity-based policy examples for Amazon Lex V2.

Policy resources for Amazon Lex V2

Supports policy resources: *Yes*

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Amazon Lex V2 resource types and their ARNs, see Resources defined by Amazon Lex V2 in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by Amazon Lex V2.

To view examples of Amazon Lex V2 identity-based policies, see Identity-based policy examples for Amazon Lex V2.

Policy condition keys for Amazon Lex V2

Supports service-specific policy condition keys: *No*

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Amazon Lex V2 condition keys, see Condition keys for Amazon Lex V2 in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by Amazon Lex V2.

To view examples of Amazon Lex V2 identity-based policies, see Identity-based policy examples for Amazon Lex V2.

Access control lists (ACLs) in Amazon Lex V2

Supports ACLs: *No*

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with Amazon Lex V2

Supports ABAC (tags in policies): *Yes*

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using Temporary credentials with Amazon Lex V2

Supports temporary credentials: *No*

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Cross-service principal permissions for Amazon Lex V2

Supports forward access sessions (FAS): *Yes*

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for Amazon Lex V2

Supports service roles: *Yes*

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Amazon Lex V2 functionality. Edit service roles only when Amazon Lex V2 provides guidance to do so.

Service-linked roles for Amazon Lex V2

Supports service-linked roles: *Partial*

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon Lex V2

By default, users and roles don't have permission to create or modify Amazon Lex V2 resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Amazon Lex V2, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for Amazon Lex V2 in the *Service Authorization Reference*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Amazon Lex V2 resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Amazon Lex V2 console

To access the Amazon Lex V2 console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Amazon Lex V2 resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum

required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Amazon Lex V2 console, users need to have Console access. For more information about creating a user with Console access, see Creating an IAM user in your AWS account in the *IAM User Guide*.

Allow users to add functions to a bot

This example shows a policy that allows IAM users to add Amazon Comprehend, sentiment analysis and Amazon Kendra query permissions to an Amazon Lex V2 bot.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Id1",
            "Effect": "Allow",
            "Action": "iam:PutRolePolicy",
            "Resource": "arn:aws:iam::*:role/aws-service-role/
lexv2.amazonaws.com/AWSServiceRoleForLexV2Bots*"
        },
        {
            "Sid": "Id2",
            "Effect": "Allow",
            "Action": "iam:GetRolePolicy",
            "Resource": "arn:aws:iam::*:role/aws-service-role/
lexv2.amazonaws.com/AWSServiceRoleForLexV2Bots*"
        }
    ]
}
```

Allow users to add channels to a bot

This example is a policy that allows IAM users to add a messaging channel to a bot. A user must have this policy in place before they can deploy a bot on a messaging platform.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Id1",
            "Effect": "Allow",
            "Action": "iam:PutRolePolicy",
            "Resource": "arn:aws:iam::*:role/aws-service-role/
channels.lexv2.amazonaws.com/AWSServiceRoleForLexV2Channels*"
        },
        {
            "Sid": "Id2",
            "Effect": "Allow",
            "Action": "iam:GetRolePolicy",
            "Resource": "arn:aws:iam::*:role/aws-service-role/
channels.lexv2.amazonaws.com/AWSServiceRoleForLexV2Channels*"
        }
    ]
}
```

Allow users to create and update bots

This example shows an example policy that allows IAM users to create and update any bot. The policy includes permissions to complete this action on the console or using the AWS CLI or AWS API.

Allow users to use the Automated Chatbot Designer

This example shows an example policy that allows IAM users to run the Automated Chatbot Designer.

Allow users to use a AWS KMS key to encrypt and decrypt files

This example shows an example policy that allows IAM users to use a AWS KMS customer managed key to encrypt and decrypt data.

Allow users to delete bots

This example shows an example policy that allows IAM users to delete any bot. The policy includes permissions to complete this action on the console or using the AWS CLI or AWS API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex:DeleteBot",
                "lex:DeleteBotLocale",
                "lex:DeleteBotAlias",
                "lex:DeleteIntent",
                "lex:DeleteSlot",
                "lex:DeleteSlottype"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:lex:us-east-1:123412341234:bot/*",
                "arn:aws:lex:us-east-1:123412341234:bot-alias/*"
            ]
        }
    ]
}
```

Allow users to have a conversation with a bot

This example shows an example policy that allows IAM users have a conversation with any bot. The policy includes permissions to complete this action on the console or using the AWS CLI or AWS API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex:StartConversation",
                "lex:RecognizeText",
                "lex:RecognizeUtterance",
                "lex:GetSession",
                "lex:PutSession",
                "lex:DeleteSession"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:lex:us-east-1:123412341234:bot-alias/*"
        }
    ]
}
```

Allow a specific user to manage resource-based policies

The following example grants permission for a specific user to manage the resource-based policies. It allows console and API access to the policies associated with bots and bot aliases.

Allow a user to export bots and bot locales

The following IAM permission policy enables a user to create, update, and get an export for a bot or bot locale.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex:CreateExport",
                "lex:UpdateExport",
                "lex:DescribeExport",
```

```
                "lex:DescribeBot",
                "lex:DescribeBotLocale",
                "lex:ListBotLocales",
                "lex:DescribeIntent",
                "lex:ListIntents",
                "lex:DescribeSlotType",
                "lex:ListSlotTypes",
                "lex:DescribeSlot",
                "lex:ListSlots",
                "lex:DescribeCustomVocabulary"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot/*"
            ]
        }
    ]
}
```

Allow a user to export a custom vocabulary

The following IAM permission policy allows a user to export a custom vocabulary from a bot locale.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex:CreateExport",
                "lex:UpdateExport",
                "lex:DescribeExport",
                "lex:DescribeCustomVocabulary"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot/*"
            ]
        }
    ]
```

```
}
```

Allow a user to import bots and bot locales

The following IAM permission policy allows a user to import a bot or bot locale and to check the status of an import.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex:CreateUploadUrl",
                "lex:StartImport",
                "lex:DescribeImport",
                "lex:CreateBot",
                "lex:UpdateBot",
                "lex:DeleteBot",
                "lex:CreateBotLocale",
                "lex:UpdateBotLocale",
                "lex:DeleteBotLocale",
                "lex:CreateIntent",
                "lex:UpdateIntent",
                "lex:DeleteIntent",
                "lex:CreateSlotType",
                "lex:UpdateSlotType",
                "lex:DeleteSlotType",
                "lex:CreateSlot",
                "lex:UpdateSlot",
                "lex:DeleteSlot",
                "lex:CreateCustomVocabulary",
                "lex:UpdateCustomVocabulary",
                "lex:DeleteCustomVocabulary",
                "iam:PassRole"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot/*",
                "arn:aws:lex:us-east-1:123456789012:bot-alias/*"
```

```
            ]
        }
    ]
}
```

Allow a user to import a custom vocabulary

The following IAM permission policy allows a user to import a custom vocabulary to a bot locale.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex:CreateUploadUrl",
                "lex:StartImport",
                "lex:DescribeImport",
                "lex:CreateCustomVocabulary",
                "lex:UpdateCustomVocabulary",
                "lex:DeleteCustomVocabulary"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot/*"
            ]
        }
    ]
}
```

Allow a user to migrate a bot from Amazon Lex to Amazon Lex V2

The following IAM permission policy allows a user to start migrating a bot from Amazon Lex to Amazon Lex V2.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Allow a user to draw conversation flow with visual conversation builder in Amazon Lex V2

The following IAM permission policy allows a user to draw the conversation flow with visual conversation builder in Amazon Lex V2.

Allow users to create and view bot replicas, but not to delete them

You can attach the following permissions to an IAM role to allow it to only create and view bot replicas. By omitting `lex:DeleteBotReplica`, you prevent the role from deleting bot replicas. For more information, see Permissions to replicate bots and manage bot replicas in Lex V2.

Resource-based policy examples for Amazon Lex V2

A *resource-based policy* is attached to a resource, such as a bot or a bot alias. With a resource-based policy you can specify who has access to the resource and the actions that they can perform on it. For example, you can add resource-based policies that enable a user to modify a specific bot, or to allow a user to use runtime operations on a specific bot alias.

When you use a resource-based policy you can allow other AWS services to access resources in your account. For example, you can allow Connect Customer to access an Amazon Lex V2 bot.

To learn how to create a bot or bot alias, see Working with Amazon Lex V2 bots.

Topics

- Use the console to specify a resource-based policy
- Use the API to specify a resource-based policy
- Allow an IAM role to update a bot and list bot aliases
- Allow a user to have a conversation with a bot
- Allow an AWS service to use a specific Amazon Lex V2 bot

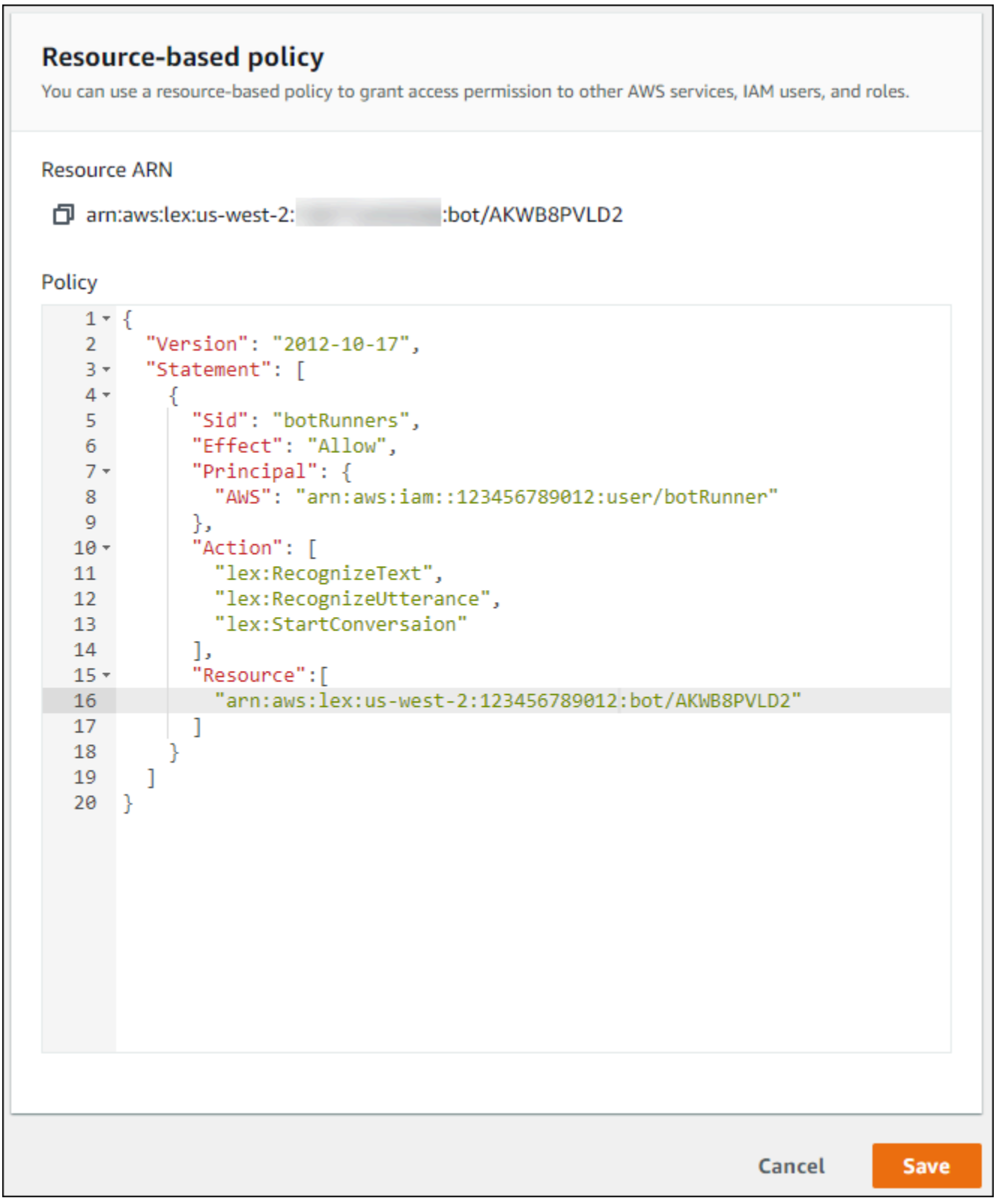
Use the console to specify a resource-based policy

You can use the Amazon Lex V2 console to manage the resource-based policies for your bots and bot aliases. You enter the JSON structure of a policy and the console associates it with the resource.

If there is a policy already associated with a resource, you can use the console to view and modify the policy.

When you save a policy with the policy editor, the console checks the syntax of the policy. If the policy contains errors, such as a non-existent user or an action that is not supported by the resource, it returns an error and doesn't save the policy.

The following shows the resource-based policy editor for a bot in the console. The policy editor for a bot alias is similar.


Resource-based policy
You can use a resource-based policy to grant access permission to other AWS services, IAM users, and roles.
Resource ARN
arn:aws:lex:us-west-2: :bot/AKWB8PVLD2
Policy
{
"Version": "2012-10-17",
"Statement": [
{
"Sid": "botRunners",
"Effect": "Allow",
"Principal": {
"AWS": "arn:aws:iam::123456789012:user/botRunner"
},
"Action": [
"lex:RecognizeText",
"lex:RecognizeUtterance",
"lex:StartConversaion"
],
"Resource":[
"arn:aws:lex:us-west-2:123456789012:bot/AKWB8PVLD2"
]
}
]
}
Cancel
Save

To open the policy editor for a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the **Bots** list, choose the bot whose policy you want to edit.
3. In the **Resource-based policy** section, choose **Edit**.

To open the policy editor for a bot alias

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the **Bots** list, choose the bot that contains the alias that you want to edit.
3. From the left menu, choose **Aliases**, then choose the alias to edit.
4. In the **Resource-based policy** section, choose **Edit**.

Use the API to specify a resource-based policy

You can use API operations to manage the resource-based policies for your bots and bot aliases. There are operations to create, update and delete policies.

CreateResourcePolicy

Adds a new resource policy with the specified policy statements to a bot or bot alias.

CreateResourcePolicyStatement

Adds a new resource policy statement to a bot or bot alias.

DeleteResourcePolicy

Removes a resource policy from a bot or bot alias.

DeleteResourcePolicyStatement

Removes a resource policy statement from a bot or bot alias.

DescribeResourcePolicy

Gets a resource policy and the policy revision.

UpdateResourcePolicy

Replaces the existing resource policy for a bot or bot alias with a new one.

Examples

Java

The following example shows how to use the resource-based policy operations to manage a resource-based policy.

```
        /*
         * Create a new policy for the specified bot alias
         * that allows a role to invoke lex:UpdateBotAlias on it.
         * The created policy will have revision id 1.
         */

        CreateResourcePolicyRequest createPolicyRequest =
                CreateResourcePolicyRequest.builder()
                        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
                        .policy("{\"Version\": \"2012-10-17\",\"Statement
\": [{\"Sid\": \"BotAliasEditor\",\"Effect\": \"Allow\",\"Principal\":
 {\"AWS\": \"arn:aws:iam::123456789012:role/BotAliasEditor\"},\"Action\":
 [\"lex:UpdateBotAlias\"],\"Resource\":[\"arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID\"]]}")

        lexmodelsv2Client.createResourcePolicy(createPolicyRequest);

        /*
         * Overwrite the policy for the specified bot alias with a new policy.
         * Since no expectedRevisionId is provided, this request overwrites the
 current revision.
         * After this update, the revision id for the policy is 2.
         */
        UpdateResourcePolicyRequest updatePolicyRequest =
        UpdateResourcePolicyRequest.builder()
                        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
                        .policy("{\"Version\": \"2012-10-17\",\"Statement
\": [{\"Sid\": \"BotAliasEditor\",\"Effect\": \"Deny\",\"Principal\":
 {\"AWS\": \"arn:aws:iam::123456789012:role/BotAliasEditor\"},\"Action\":
 [\"lex:UpdateBotAlias\"],\"Resource\":[\"arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID\"]]}")

        lexmodelsv2Client.updateResourcePolicy(updatePolicyRequest);

        /*
```

```
         * Creates a statement in an existing policy for the specified bot alias
         * that allows a role to invoke lex:RecognizeText on it.
         * This request expects to update revision 2 of the policy. The request will
 fail
         * if the current revision of the policy is no longer revision 2.
         * After this request, the revision id for this policy will be 3.
         */

        CreateResourcePolicyStatementRequest createStatementRequest =
                CreateResourcePolicyStatementRequest.builder()
                        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
                        .effect("Allow")

 .principal(Principal.builder().arn("arn:aws:iam::123456789012:role/
BotRunner").build())
                        .action("lex:RecognizeText")
                        .statementId("BotRunnerStatement")
                        .expectedRevisionId(2)
                        .build();

        lexmodelsv2Client.createResourcePolicyStatement(createStatementRequest);

        /*
         * Deletes a statement from an existing policy for the specified bot alias
 by statementId.
         * Since no expectedRevisionId is supplied, the request will remove the
 statement from
         * the current revision of the policy for the bot alias.
         * After this request, the revision id for this policy will be 4.
         */
        DeleteResourcePolicyRequest deleteStatementRequest =
                DeleteResourcePolicyRequest.builder()
                        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
                        .statementId("BotRunnerStatement")
                        .build();

        lexmodelsv2Client.deleteResourcePolicy(deleteStatementRequest);

        /*
         * Describe the current policy for the specified bot alias
         * It always returns the current revision.
         */
```

```
        DescribeResourcePolicyRequest describePolicyRequest =
                DescribeResourcePolicyRequest.builder()
                        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
                        .build();

        lexmodelsv2Client.describeResourcePolicy(describePolicyRequest);


        /*
         * Delete the current policy for the specified bot alias
         * This request expects to delete revision 3 of the policy. Since the
 revision id for
         * this policy is already at 4, this request will fail.
         */
        DeleteResourcePolicyRequest deletePolicyRequest =
                DeleteResourcePolicyRequest.builder()
                        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
                        .expectedRevisionId(3);
                        .build();

        lexmodelsv2Client.deleteResourcePolicy(deletePolicyRequest);
```

Allow an IAM role to update a bot and list bot aliases

The following example grants permissions for a specific IAM role to call Amazon Lex V2 model building API operations to modify an existing bot. The user can list aliases for a bot and update the bot, but can't delete the bot or bot aliases.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "botBuilders",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/BotBuilder"
            },
```

```
            "Action": [
                "lex:ListBotAliases",
                "lex:UpdateBot"
            ],
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot/MYBOT"
            ]
        }
    ]
}
```

Allow a user to have a conversation with a bot

The following example grants permission for a specific user to call Amazon Lex V2 runtime API operations on a single alias of a bot.

The user is specifically denied permission to update or delete the bot alias.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "botRunners",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:user/botRunner"
            },
            "Action": [
                "lex:RecognizeText",
                "lex:RecognizeUtterance",
                "lex:StartConversation",
                "lex:DeleteSession",
                "lex:GetSession",
                "lex:PutSession"
            ],
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot-alias/MYBOT/MYBOTALIAS"
            ]
        },
        {
```

```
            "Sid": "botRunners",
            "Effect": "Deny",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:user/botRunner"
            },
            "Action": [
                "lex:UpdateBotAlias",
                "lex:DeleteBotAlias"
            ],
            "Resource": [
                "arn:aws:lex:us-east-1:123456789012:bot-alias/MYBOT/MYBOTALIAS"
            ]
        }
    ]
}
```

Allow an AWS service to use a specific Amazon Lex V2 bot

The following example grants permission for AWS Lambda and Connect Customer to call Amazon Lex V2 runtime API operations.

The condition block is required for service principals, and must use the global context keys `AWS:SourceAccount` and `AWS:SourceArn`.

The `AWS:SourceAccount` is the account ID that is calling the Amazon Lex V2 bot.

The `AWS:SourceArn` is the resource ARN of the Connect Customer service instance or Lambda function that the call to the Amazon Lex V2 bot alias originates from.

AWS managed policies for Amazon Lex V2

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: AmazonLexReadOnly

You can attach the `AmazonLexReadOnly` policy to your IAM identities.

This policy grants read-only permissions that allow users to view all actions in the Amazon Lex V2 and Amazon Lex model building service.

Permissions details

This policy includes the following permissions:

- `lex` – Read-only access to Amazon Lex V2 and Amazon Lex resources in the model building service.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AmazonLexReadOnlyStatement1",
            "Effect": "Allow",
            "Action": [
                "lex:GetBot",
                "lex:GetBotAlias",
                "lex:GetBotAliases",
                "lex:GetBots",
                "lex:GetBotChannelAssociation",
                "lex:GetBotChannelAssociations",
                "lex:GetBotVersions",
                "lex:GetBuiltinIntent",
```

```
                "lex:GetBuiltinIntents",
                "lex:GetBuiltinSlotTypes",
                "lex:GetIntent",
                "lex:GetIntents",
                "lex:GetIntentVersions",
                "lex:GetSlotType",
                "lex:GetSlotTypes",
                "lex:GetSlotTypeVersions",
                "lex:GetUtterancesView",
                "lex:DescribeBot",
                "lex:DescribeBotAlias",
                "lex:DescribeBotChannel",
                "lex:DescribeBotLocale",
                "lex:DescribeBotRecommendation",
                "lex:DescribeBotReplica",
                "lex:DescribeBotVersion",
                "lex:DescribeExport",
                "lex:DescribeImport",
                "lex:DescribeIntent",
                "lex:DescribeResourcePolicy",
                "lex:DescribeSlot",
                "lex:DescribeSlotType",
                "lex:ListBots",
                "lex:ListBotLocales",
                "lex:ListBotAliases",
                "lex:ListBotAliasReplicas",
                "lex:ListBotChannels",
                "lex:ListBotRecommendations",
                "lex:ListBotReplicas",
                "lex:ListBotVersions",
                "lex:ListBotVersionReplicas",
                "lex:ListBuiltInIntents",
                "lex:ListBuiltInSlotTypes",
                "lex:ListExports",
                "lex:ListImports",
                "lex:ListIntents",
                "lex:ListRecommendedIntents",
                "lex:ListSlots",
                "lex:ListSlotTypes",
                "lex:ListTagsForResource",
                "lex:SearchAssociatedTranscripts",
                "lex:ListCustomVocabularyItems"
            ],
            "Resource": "*"
```

```
        }
    ]
}
```

AWS managed policy: AmazonLexRunBotsOnly

You can attach the `AmazonLexRunBotsOnly` policy to your IAM identities.

This policy grants read-only permissions that allow access to run Amazon Lex V2 and Amazon Lex conversational bots. .

Permissions details

This policy includes the following permissions:

- `lex` – Read-only access to all actions in the Amazon Lex V2 and Amazon Lex runtime.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lex:PostContent",
                "lex:PostText",
                "lex:PutSession",
                "lex:GetSession",
                "lex:DeleteSession",
                "lex:RecognizeText",
                "lex:RecognizeUtterance",
                "lex:StartConversation"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AmazonLexFullAccess

You can attach the `AmazonLexFullAccess` policy to your IAM identities.

This policy grants administrative permissions that allow the user permission to create, read, update, and delete Amazon Lex V2 and Amazon Lex resources; and to run Amazon Lex V2 and Amazon Lex conversational bots.

Permissions details

This policy includes the following permissions:

- `lex` – Allows principals read and write access to all actions in the Amazon Lex V2 and Amazon Lex model building and runtime services.
- `cloudwatch` – Allows principals to view Amazon CloudWatch metrics and alarms.
- `iam` – Allows principals to create and delete service-linked roles, pass roles, and attach and detach policies to a role. The permissions are restricted to "lex.amazonaws.com" for Amazon Lex operations and to "lexv2.amazonaws.com" for Amazon Lex V2 operations.
- `kendra` – Allows principals to list Amazon Kendra indexes.
- `kms` – Allows principals to describe AWS KMS keys and aliases.
- `lambda` – Allows principals to list AWS Lambda functions and manage permissions attached to any Lambda function.
- `polly` – Allows principals to describe Amazon Polly voices and synthesize speech.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AmazonLexFullAccessStatement1",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:DescribeAlarms",
                "cloudwatch:DescribeAlarmsForMetric",
                "kms:DescribeKey",
                "kms:ListAliases",
```

```
                "lambda:GetPolicy",
                "lambda:ListFunctions",
                "lambda:ListAliases",
                "lambda:ListVersionsByFunction",
                "lex:*",
                "polly:DescribeVoices",
                "polly:SynthesizeSpeech",
                "kendra:ListIndices",
                "iam:ListRoles",
                "s3:ListAllMyBuckets",
                "logs:DescribeLogGroups",
                "s3:GetBucketLocation"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "AmazonLexFullAccessStatement2",
            "Effect": "Allow",
            "Action": [
                "bedrock:ListFoundationModels"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": "arn:aws:bedrock:*::foundation-model/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:*:*:function:AmazonLex*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "lex.amazonaws.com"
                }
            }
```

```
        },
        {
            "Sid": "AmazonLexFullAccessStatement3",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:GetRolePolicy"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots",
                "arn:aws:iam::*:role/aws-service-role/channels.lex.amazonaws.com/
AWSServiceRoleForLexChannels",
                "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexV2Bots*",
                "arn:aws:iam::*:role/aws-service-role/
channels.lexv2.amazonaws.com/AWSServiceRoleForLexV2Channels*",
                "arn:aws:iam::*:role/aws-service-role/
replication.lexv2.amazonaws.com/AWSServiceRoleForLexV2Replication*"
            ]
        },
        {
            "Sid": "AmazonLexFullAccessStatement4",
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "lex.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement5",
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
```

```
                "arn:aws:iam::*:role/aws-service-role/channels.lex.amazonaws.com/
AWSServiceRoleForLexChannels"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "channels.lex.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement6",
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexV2Bots*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "lexv2.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement7",
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/
channels.lexv2.amazonaws.com/AWSServiceRoleForLexV2Channels*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "channels.lexv2.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement8",
            "Effect": "Allow",
```

```
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/
replication.lexv2.amazonaws.com/AWSServiceRoleForLexV2Replication*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "replication.lexv2.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement9",
            "Effect": "Allow",
            "Action": [
                "iam:DeleteServiceLinkedRole",
                "iam:GetServiceLinkedRoleDeletionStatus"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots",
                "arn:aws:iam::*:role/aws-service-role/channels.lex.amazonaws.com/
AWSServiceRoleForLexChannels",
                "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexV2Bots*",
                "arn:aws:iam::*:role/aws-service-role/
channels.lexv2.amazonaws.com/AWSServiceRoleForLexV2Channels*",
                "arn:aws:iam::*:role/aws-service-role/
replication.lexv2.amazonaws.com/AWSServiceRoleForLexV2Replication*"
            ]
        },
        {
            "Sid": "AmazonLexFullAccessStatement10",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots"
            ],
            "Condition": {
```

```
                "StringEquals": {
                    "iam:PassedToService": [
                        "lex.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement11",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexV2Bots*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "lexv2.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Sid": "AmazonLexFullAccessStatement12",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/
channels.lexv2.amazonaws.com/AWSServiceRoleForLexV2Channels*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "channels.lexv2.amazonaws.com"
                    ]
                }
            }
        },
        {
```

```
            "Sid": "AmazonLexFullAccessStatement13",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/
replication.lexv2.amazonaws.com/AWSServiceRoleForLexV2Replication*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "lexv2.amazonaws.com"
                    ]
                }
            }
        }
    ]
}
```

AWS managed policy: AmazonLexReplicationPolicy

You can't attach `AmazonLexReplicationPolicy` to your IAM entities. This policy is attached to a service-linked role that allows Amazon Lex V2 to perform actions on your behalf. For more information, see Using service-linked roles for Amazon Lex V2.

This policy grants administrative permissions that allows Amazon Lex V2 to replicate AWS resources across Regions on your behalf. You can attach this policy to permit a role to easily replicate resources, including bots, locales, versions, aliases, intents, slot types, slots, and custom vocabularies.

Permissions details

This policy includes the following permissions.

- `lex` – Allows principals to replicate resources in other Regions.
- `iam` – Allows principals to pass roles from IAM. This is required so that Amazon Lex V2 has permissions to replicate resources in other Regions.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
 {
  "Sid": "ReplicationPolicyStatement1",
  "Effect": "Allow",
  "Action": [
   "lex:BuildBotLocale",
   "lex:ListBotLocales",
   "lex:CreateBotAlias",
   "lex:UpdateBotAlias",
   "lex:DeleteBotAlias",
   "lex:DescribeBotAlias",
   "lex:CreateBotVersion",
   "lex:DeleteBotVersion",
   "lex:DescribeBotVersion",
   "lex:CreateExport",
   "lex:DescribeBot",
   "lex:UpdateExport",
   "lex:DescribeExport",
   "lex:DescribeBotLocale",
   "lex:DescribeIntent",
   "lex:ListIntents",
   "lex:DescribeSlotType",
   "lex:ListSlotTypes",
   "lex:DescribeSlot",
   "lex:ListSlots",
   "lex:DescribeCustomVocabulary",
   "lex:StartImport",
   "lex:DescribeImport",
   "lex:CreateBot",
   "lex:UpdateBot",
   "lex:DeleteBot",
   "lex:CreateBotLocale",
   "lex:UpdateBotLocale",
   "lex:DeleteBotLocale",
   "lex:CreateIntent",
   "lex:UpdateIntent",
   "lex:DeleteIntent",
   "lex:CreateSlotType",
```

```
    "lex:UpdateSlotType",
    "lex:DeleteSlotType",
    "lex:CreateSlot",
    "lex:UpdateSlot",
    "lex:DeleteSlot",
    "lex:CreateCustomVocabulary",
    "lex:UpdateCustomVocabulary",
    "lex:DeleteCustomVocabulary",
    "lex:DeleteBotChannel",
    "lex:ListTagsForResource",
    "lex:TagResource",
    "lex:UntagResource",
    "lex:CreateResourcePolicy",
    "lex:DeleteResourcePolicy",
    "lex:DescribeResourcePolicy",
    "lex:UpdateResourcePolicy"
   ],
   "Resource": [
    "arn:aws:lex:*:*:bot/*",
    "arn:aws:lex:*:*:bot-alias/*"
   ]
  },
  {
   "Sid": "ReplicationPolicyStatement2",
   "Effect": "Allow",
   "Action": [
    "lex:CreateUploadUrl",
    "lex:ListBots"
   ],
   "Resource": "*"
  },
  {
   "Sid": "ReplicationPolicyStatement3",
   "Effect": "Allow",
   "Action": [
    "iam:PassRole"
   ],
   "Resource": "*",
   "Condition": {
    "StringEquals": {
     "iam:PassedToService": "lexv2.amazonaws.com"
    }
   }
  }
```

```
    ]
}
```

AWS managed policy: AmazonLexV2BedrockAgentPolicy

Amazon Lex V2 policy for Amazon Bedrock agents

Response

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "LexV2TrustPolicy",
      "Principal": {
        "Service": "lexv2.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "{accountId}"
        }
      }
    }
  ]
}
```

AWS managed policy: AmazonLexV2BedrockKnowledgeBasePolicy

Amazon Lex V2 policy for Amazon Bedrock knowledge bases

Response

JSON

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "LexV2TrustPolicy",
      "Principal": {
        "Service": "lexv2.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
     "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "{accountId}"
        }
      }
    }
  ]
}
```

AWS managed policy: AmazonLexV2BedrockAgentPolicyInternal

Amazon Lex V2 internal policy for Amazon Bedrock agents

Response

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Sid": "LexV2InternalTrustPolicy",
      "Principal": {
        "Service": "lexv2.aws.internal"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "{accountId}"
        }
      }
```

```
    }
  ]
}
```

AWS managed policy: AmazonLexV2BedrockKnowledgeBasePolicyInternal

Amazon Lex V2 internal policy for Amazon Bedrock knowledge bases

Response

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "LexV2InternalTrustPolicy",
      "Effect": "Allow",
      "Principal": {
        "Service": "lexv2.aws.internal"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "{accountId}"
        }
      }
    }
  ]
}
```

Amazon Lex V2 updates to AWS managed policies

View details about updates to AWS managed policies for Amazon Lex V2 since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Amazon Lex V2 Document history for Amazon Lex V2 page.

Change	Description	Date
AmazonLexReplicationPolicy – Updated policy	Amazon Lex V2 updated the policy to allow replication of tags and ResourceBasedPolicy.	June 24, 2025
AmazonLexV2BedrockKnowledgeBasePolicyInternal – New policy	Amazon Lex V2 added a new policy to allow replication of Amazon Bedrock knowledge base resources.	August 30, 2024
AmazonLexV2BedrockAgentPolicyInternal – New policy	Amazon Lex V2 added a new policy to allow replication of Amazon Bedrock agent resources.	August 30, 2024
AmazonLexV2BedrockKnowledgeBasePolicy – New policy	Amazon Lex V2 added a new policy to allow replication of Amazon Bedrock knowledge base resources.	August 30, 2024
AmazonLexV2BedrockAgentPolicy – New policy	Amazon Lex V2 added a new policy to allow replication of Amazon Bedrock agent resources.	August 30, 2024
AmazonLexReadOnly – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access replicas of bot resources.	May 10, 2024
AmazonLexFullAccess – Update to an existing policy	Amazon Lex V2 added new permissions to allow replication of bot resources to other regions.	April 16, 2024

Change	Description	Date
AmazonLexFullAccess – Update to an existing policy	Amazon Lex V2 added new permissions to allow replication of bot resources to other regions.	January 31, 2024
AmazonLexReplicationPolicy – New policy	Amazon Lex V2 added a new policy to allow replication of bot resources to other regions.	January 31, 2024
AmazonLexReadOnly – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to list custom vocabulary items.	November 29, 2022
AmazonLexFullAccess – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 model building service operations.	August 18, 2021
AmazonLexReadOnly – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 Automated Chatbot Designer operations.	December 1, 2021
AmazonLexFullAccess – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 model building service operations.	August 18, 2021

Change	Description	Date
AmazonLexReadOnly – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 model building service operations.	August 18, 2021
AmazonLexRunBotsOnly – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 runtime service operations.	August 18, 2021
Amazon Lex V2 started tracking changes	Amazon Lex V2 started tracking changes for its AWS managed policies.	August 18, 2021

Using service-linked roles for Amazon Lex V2

Amazon Lex V2 uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Amazon Lex V2. Service-linked roles are predefined by Amazon Lex V2 and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up Amazon Lex V2 easier because you don't have to manually add the necessary permissions. Amazon Lex V2 defines the permissions of its service-linked roles, and unless defined otherwise, only Amazon Lex V2 can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

For information about other services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

You can delete a service-linked role only after first deleting related resources. This protects your Amazon Lex V2 resources because you can't inadvertently remove permissions to access the resources.

Topics

Creating a service-linked role for Amazon Lex V2

You don't need to manually create a service-linked role, because Amazon Lex V2 creates the service-linked role for you when you carry out the relevant action (see Service-linked role permissions for Amazon Lex V2 for more information) in the AWS Management Console, AWS CLI, or AWS API.

If you delete this service-linked role, and then need to create one again, you can use the same process to create a new role in your account.

Editing a service-linked role for Amazon Lex V2

Amazon Lex V2 doesn't allow you to edit service-linked roles. After you create a service-linked role, you can't change the name of the role because various entities might reference the role. However, you can edit the description of a role using IAM. For more information, see Editing a Service-Linked Role in the *IAM User Guide*.

Deleting a service-linked role for Amazon Lex V2

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don’t have an unused entity that is not actively monitored or maintained. However, you must clean up the resources for your service-linked role before you can manually delete it.

> **Note**
>
> If the Amazon Lex V2 service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To see the steps for deleting resources for specific service-linked roles in Amazon Lex V2, refer to the section specific to the role in Service-linked role permissions for Amazon Lex V2.

To manually delete a service-linked role using IAM

After deleting resources related to a service-linked role, use the IAM console, the AWS CLI, or the AWS API to delete the role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Service-linked role permissions for Amazon Lex V2

Amazon Lex V2 uses service-linked roles with the following prefixes.

Topics

- AWSServiceRoleForLexV2Bots_
- AWSServiceRoleForLexV2Channels_
- AWSServiceRoleForLexV2Replication

AWSServiceRoleForLexV2Bots_

The AWSServiceRoleForLexV2Bots_ role gives permissions to connect your bot to other required services. This role includes a trust policy to allow the lexv2.amazonaws.com service to assume the role and includes permissions to carry out the following actions.

- Use Amazon Polly to synthesize speech on all Amazon Lex V2 resources that the action supports.
- If a bot is configured to use Amazon Comprehend sentiment analysis, detect the sentiment on all Amazon Lex V2 resources that the action supports.
- If a bot is configured to store audio logs in an S3 bucket, put objects in a specified bucket.
- If a bot is configured to store audio and text logs, create a log stream in and put logs into a specified log group.

- If a bot is configured to use a AWS KMS key to encrypt data, generate a specific data key.
- If a bot is configured to use the `KendraSearchIntent` intent, query access to a specified Amazon Kendra index.

To create the role

Amazon Lex V2 creates a new AWSServiceRoleForLexV2Bots_ role with a random suffix in your account each time that you create a bot. Amazon Lex V2 modifies the role when you add additional capabilities to a bot. For example, if you add Amazon Comprehend sentiment analysis to a bot, Amazon Lex V2 adds permission for the `lex:DetectSentiment` action to the service role.

To delete the role

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the left navigation pane, select **Bots** and choose the bot whose service-linked role you want to delete.
3. Select any version of the bot.
4. The **IAM permissions runtime role** is in the **Version details**.
5. Return to the **Bots** page and choose the radio button next to the bot to delete.
6. Select **Action** and then choose **Delete**.
7. Follow the steps at Deleting a service-linked role to delete the IAM role.

AWSServiceRoleForLexV2Channels_

The AWSServiceRoleForLexV2Channels_ role gives permission to list bots in an account and to call conversation APIs for a bot. This role includes a trust policy to allow the channels.lexv2.amazonaws.com service to assume the role. If a bot is configured to use a channel to communicate with a messaging service, the AWSServiceRoleForLexV2Channels_ role permissions policy allows Amazon Lex V2 to complete the following actions.

- List permissions on all bots in an account.
- Recognize text, get session and put session permissions on a specified bot alias.

To create the role

When you create a channel integration to deploy a bot on a messaging platform, Amazon Lex V2 creates a new service-linked role in your account for each channel with a random suffix.

To delete the role

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. From the left navigation pane, select **Bots**.
3. Choose a bot.
4. From the left navigation pane, choose **Channel integrations** under **Deployments**.
5. Select a channel whose service-linked role you want to delete.
6. The **IAM permissions runtime role** is in the **General configuration**
7. Choose **Delete**, then choose **Delete** again to delete the channel.
8. Follow the steps at Deleting a service-linked role to delete the IAM role.

AWSServiceRoleForLexV2Replication

The AWSServiceRoleForLexV2Replication role gives permission to replicate bots in a second region. This role includes a trust policy to allow the replication.lexv2.amazonaws.com service to assume the role and also includes the AmazonLexReplicationPolicy AWS managed policy, which allows permissions for the following actions.

- Pass bot IAM roles to the replica bot to reduplicate the appropriate permissions for the replica bot.
- Create and manage bots and bot resources (versions, aliases, intents, slots, custom vocabularies, and so on) in other Regions.

To create the role

When you enable Global Resiliency for a bot, Amazon Lex V2 creates the AWSServiceRoleForLexV2Replication service-linked role in your account. Ensure that you have the correct permissions to grant the Amazon Lex V2 service permissions to create the service-linked role.

To delete Amazon Lex V2 resources used by AWSServiceRoleForLexV2Replication so that you can delete the role

1. Sign in to the AWS Management Console and open the Amazon Lex console at https://console.aws.amazon.com/lex/.
2. Choose a bot for which Global Resiliency is enabled.
3. Select **Global Resiliency** under **Deployment**.
4. Select **Disable Global Resiliency**.
5. Repeat the process for all bots that have Global Resiliency enabled.
6. Follow the steps at Deleting a service-linked role to delete the IAM role.

Supported regions for Amazon Lex V2 service-linked roles

Amazon Lex V2 supports using service-linked roles in all of the regions where the service is available. For more information, see AWS Regions and Endpoints.

Using service roles for Amazon Lex V2

Amazon Lex V2 uses an IAM service role to access other AWS services on your behalf at runtime. The service role provides the permissions needed to build and run your bot. A service role is a standard customer-managed IAM role in your account. It respects the service control policies (SCPs) that you configure in AWS Organizations and gives you full control over the permissions that Amazon Lex V2 uses.

AmazonLexServiceRole-

The AmazonLexServiceRole- role gives permissions to connect your bot to other required services. This role includes a trust policy to allow the lexv2.amazonaws.com service to assume the role and includes permissions to carry out the following actions.

- If a bot is configured to use text-to-speech, use Amazon Polly to synthesize speech on all Amazon Lex V2 resources that the action supports.
- If a bot is configured to use Amazon Comprehend sentiment analysis, detect the sentiment on all Amazon Lex V2 resources that the action supports.
- If a bot is configured to store audio logs in an S3 bucket, put objects in a specified bucket.
- If a bot is configured to store audio and text logs, create a log stream in and put logs into a specified log group.

- If a bot is configured to use a AWS KMS key to encrypt data, generate a specific data key.
- If a bot is configured to use Amazon Bedrock generative AI features, invoke the specified Amazon Bedrock models, inference profiles, and knowledge bases.

Creating the role

Amazon Lex V2 creates a new AmazonLexServiceRole- role with a random suffix in your account when you create a bot and choose to create a new role. Amazon Lex V2 modifies the role when you add additional capabilities to a bot. For example, if you add Amazon Comprehend sentiment analysis to a bot, Amazon Lex V2 adds permission for the `comprehend:DetectSentiment` action to the service role.

You can also choose an existing IAM role that you have created and configured yourself, as long as it trusts the Amazon Lex V2 service principal (`lexv2.amazonaws.com`).

Editing or deleting the role

Because a service role is a standard customer-managed IAM role, you manage it the same way as any other IAM role. Use the IAM console, the AWS CLI, or the IAM API. You can review the role's trust policy and attached permissions policies at any time in the IAM console.

> **Important**
>
> Before you delete a service role, make sure that no bots are still using it. Deleting a role that a bot depends on causes the bot to lose the permissions it needs to run.

Supported Regions for Amazon Lex V2 service roles

Amazon Lex V2 supports using service roles in all of the Regions where the service is available. For more information, see AWS Regions and Endpoints.

Migrating to service roles for Amazon Lex V2

Amazon Lex V2 is transitioning from AWS-managed *service-linked roles* (SLRs) to customer-managed service roles for bot runtime permissions. This topic explains why Amazon Lex V2 is making this change, how to migrate your existing bots, and the timeline for the transition.

Why Amazon Lex V2 is moving to service roles

A service-linked role is an IAM role that is linked directly to Amazon Lex V2 and is managed by AWS. Service-linked roles do not honor the service control policies (SCPs) that you configure in AWS Organizations.

A service role is an IAM role that you own and manage in your own account. Because a service role is a standard customer-managed role, it respects the service control policies (SCPs) that you configure in your organization. This gives you full control over the permissions that Amazon Lex V2 uses on your behalf. Moving to service roles aligns Amazon Lex V2 with IAM best practices and improves your security and compliance posture.

Migrate an existing bot to a service role

You can migrate an existing bot that uses a service-linked role to a customer-managed service role directly from the Amazon Lex V2 console.

> **Note**
>
> Amazon Lex V2 plans to make single-bot migration using the **Migrate to Service Role** button available before September 1, 2026. After the button is available, complete the following steps to migrate a bot.

1. Open the Amazon Lex V2 console at https://console.aws.amazon.com/lexv2/home.
2. In the left navigation pane, choose **Bots**, and then choose the bot that you want to migrate.
3. On the bot's **Draft version** page, if the bot is using a service-linked role, a banner appears indicating that the bot is using a service-linked role. Choose **Migrate to Service Role**.
4. Amazon Lex V2 creates a new customer-managed service role, attaches the permissions that your bot requires, and updates the bot to use the new role.
5. When the migration completes, the service-linked role banner and the **Migrate to Service Role** button no longer appear for the bot, confirming that the bot now uses a service role.

> **Note**
>
> The **Migrate to Service Role** button appears only for a bot that is using a service-linked role. Migration does not modify or delete your original service-linked role. If the migration

does not complete, your bot continues to use its existing service-linked role and remains fully functional.

Migration timeline

The transition from service-linked roles to service roles follows the timeline described in the following list.

- **New bots** – New bots created in the console now default to a service role. We recommend creating new bots with a service role.
- **Migrating a single bot** – The one-click **Migrate to Service Role** button appears only for bots that use a service-linked role. With one click, you can migrate a bot to a service role. Amazon Lex V2 plans to make this available before September 1, 2026.
- **Migrating multiple bots** – A wizard lists your bots so that you can select multiple service-linked role bots and migrate them to service roles at once. Amazon Lex V2 plans to make this available by September 15, 2026.
- **End of support for service-linked roles** – Support for service-linked roles ends by the end of 2026.

Troubleshooting Amazon Lex V2 identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Amazon Lex V2 and IAM.

Topics

- I am not authorized to perform an action in Amazon Lex V2
- I am not authorized to perform iam:PassRole
- I'm an administrator and want to allow others to access Amazon Lex V2
- Grant programmatic access to a user
- I want to allow people outside of my AWS account to access my Amazon Lex V2 resources

I am not authorized to perform an action in Amazon Lex V2

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your sign-in credentials.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but does not have the fictional `lex:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lex:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`my-example-widget`* resource using the `lex:`*`GetWidget`* action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Amazon Lex V2.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Amazon Lex V2. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I'm an administrator and want to allow others to access Amazon Lex V2

To allow others to access Amazon Lex V2, you must grant permission to the people or applications that need access. If you are using AWS IAM Identity Center to manage people and applications, you assign permission sets to users or groups to define their level of access. Permission sets automatically create and assign IAM policies to IAM roles that are associated with the person or application. For more information, see Permission sets in the *AWS IAM Identity Center User Guide*.

If you are not using IAM Identity Center, you must create IAM entities (users or roles) for the people or applications that need access. You must then attach a policy to the entity that grants them the correct permissions in Amazon Lex V2. After the permissions are granted, provide the credentials to the user or application developer. They will use those credentials to access AWS. To learn more about creating IAM users, groups, policies, and permissions, see IAM Identities and Policies and permissions in IAM in the *IAM User Guide*.

Grant programmatic access to a user

For information about how to get your access key ID and secret access key, see Understanding and getting your AWS credentials in the AWS General Reference.

I want to allow people outside of my AWS account to access my Amazon Lex V2 resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Amazon Lex V2 supports these features, see How Amazon Lex V2 works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.

- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Logging and monitoring in Amazon Lex V2

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Lex V2 and your other AWS solutions. AWS provides the following monitoring tools to watch Amazon Lex V2, report when something is wrong, and take automatic actions when appropriate:

- *Amazon CloudWatch* monitors your AWS resources and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the Amazon CloudWatch User Guide.
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Compliance validation for Amazon Lex V2

Third-party auditors assess the security and compliance of Amazon Lex V2 as part of multiple AWS compliance programs. Amazon Lex V2 is a HIPAA eligible service. It is PCI, SOC, and ISO compliant.

Our new AWS sign-up experience is not designed for regulated workloads. If you're using our new AWS sign-up experience, but you want to use AWS for regulated workloads, you can sign up for AWS (advanced) or activate advanced features for your AWS environment.

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience in Amazon Lex V2

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Amazon Lex V2 offers several features to help support your data resiliency and backup needs.

> **Note**
>
> For more information on Global Resiliency in Amazon Lex V2, which lets you create a replicated bot in a second region in pre-determined pairs, see Global Resiliency.

Infrastructure security in Amazon Lex V2

As a managed service, Amazon Lex V2 is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Amazon Lex V2 through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Amazon Lex V2 and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Lex V2 by creating an *interface VPC endpoint*. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to privately access Amazon Lex V2 APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Amazon Lex V2 APIs. Traffic between your VPC and Amazon Lex V2 does not leave the Amazon network.

Each interface endpoint is represented by one or more Elastic Network Interfaces in your subnets.

For more information, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*.

Considerations for Amazon Lex V2 VPC endpoints

Before you set up an interface VPC endpoint for Amazon Lex V2, ensure that you review Interface endpoint properties and limitations in the *Amazon VPC User Guide*.

Amazon Lex V2 supports making calls to all of its API actions from your VPC.

Creating an interface VPC endpoint for Amazon Lex V2

You can create a VPC endpoint for the Amazon Lex V2 service using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Creating an interface endpoint in the *Amazon VPC User Guide*.

Create a VPC endpoint for Amazon Lex V2 using the following service name:

- com.amazonaws.*region*.models-v2-lex
- com.amazonaws.*region*.runtime-v2-lex

If you enable private DNS for the endpoint, you can make API requests to Amazon Lex V2 using its default DNS name for the Region, for example, `runtime-v2-lex.us-east-1.amazonaws.com`.

For more information, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

Creating a VPC endpoint policy for Amazon Lex V2

You can attach an endpoint policy to your VPC endpoint that controls access to Amazon Lex V2. The policy specifies the following information:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for Amazon Lex V2 actions

The following is an example of an endpoint policy for Amazon Lex V2. When attached to an endpoint, this policy grants access to the listed Amazon Lex V2 actions for all principals on all resources.

```
{
   "Statement":[
      {
         "Principal":"*",
         "Effect":"Allow",
         "Action":[
            "lex:RecognizeText",
            "lex:RecognizeUtterance",
            "lex:StartConversation",
            "lex:DeleteSession",
            "lex:GetSession",
            "lex:DeleteSession"
         ],
         "Resource":"*"
      }
   ]
}
```

Code examples for Amazon Lex using AWS SDKs

The following code examples show how to use Amazon Lex with an AWS software development kit (SDK).

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

For a complete list of AWS SDK developer guides and code examples, see Using Amazon Lex V2 with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Scenarios for Amazon Lex using AWS SDKs
  - Create an Amazon Lex chatbot to engage your website visitors

Scenarios for Amazon Lex using AWS SDKs

The following code examples show you how to implement common scenarios in Amazon Lex with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Amazon Lex or combined with other AWS services. Each scenario includes a link to the complete source code, where you can find instructions on how to set up and run the code.

Scenarios target an intermediate level of experience to help you understand service actions in context.

Examples

- Create an Amazon Lex chatbot to engage your website visitors

Create an Amazon Lex chatbot to engage your website visitors

The following code examples show how to create a chatbot to engage your website visitors.

Java

SDK for Java 2.x

Shows how to use the Amazon Lex API to create a Chatbot within a web application to engage your web site visitors.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Amazon Comprehend
- Amazon Lex
- Amazon Translate

JavaScript

SDK for JavaScript (v3)

Shows how to use the Amazon Lex API to create a Chatbot within a web application to engage your web site visitors.

For complete source code and instructions on how to set up and run, see the full example Building an Amazon Lex chatbot in the AWS SDK for JavaScript developer guide.

Services used in this example

- Amazon Comprehend
- Amazon Lex
- Amazon Translate

For a complete list of AWS SDK developer guides and code examples, see Using Amazon Lex V2 with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Guidelines and best practices

Refer to the following guidelines and best practices to optimize your bot's behavior and interactions with customers.

Signing requests

All Amazon Lex V2 model-building and runtime requests in the API Reference use signature V4 for authenticating requests. For more information about authenticating requests, see Signature Version 4 signing process in the AWS General Reference.

Protecting confidential information

The runtime API operations RecognizeText and RecognizeUtterance take a session ID as a required parameter. Developers can set this to any value that meets the constraints described in the API. We recommend that you not use this parameter to send any confidential information, such as user logins, emails, or social security numbers. This ID is primarily used to uniquely identify a conversation with a bot.

Capturing slot values from user utterances

Amazon Lex V2 uses the enumeration values that you provide in a slot type definition to train its machine learning models. Suppose that you define an intent called `GetPredictionIntent` with the following sample utterance:

```
"Tell me the prediction for {sign}"
```

where {sign} is a slot with the custom type `ZodiacSign` that has 12 enumeration values: `Aries` through `Pisces`. Now suppose the user says "Tell me the prediction for earth":

- Amazon Lex V2 infers that "earth" is a ZodiacSign value if you do one of the following actions:
  - Set the `valueSelectionStrategy` field to `ORIGINAL_VALUE` using the CreateSlotType operation
  - Select **Expand values** in the console
- Amazon Lex V2 does not recognize the value "earth" if you limit recognition to the values that you defined for the slot type by doing one of the following actions:
  - Set the `valueSelectionStrategy` field to `TOP_RESOLUTION` using the `CreateSlotType` operation

- Select **Restrict to slot values and synonyms** in the console

When you define synonyms for slot values, they are recognized to be the same as a slot value. However, the slot value is returned instead of the synonym.

Because Amazon Lex V2 passes this value to your client application or to the Lambda function, you should check that the slot values are valid values before using them in your fulfillment activity.

When Amazon Lex V2 calls a Lambda function or returns the result of a speech interaction with your client, the case of the slot values is not guaranteed. In text interactions, the case of the slot values matches the text entered or the slot value, depending on the value of the `valueResolutionStrategy` field.

Acronyms in slot values

When defining slot values that contain acronyms, use the following patterns:

- Capital letters separated by periods (D.V.D.)
- Capital letters separated by spaces (D V D)

Built-in slots for date and time

The AMAZON.Date and AMAZON.Time built-in slot types capture dates and times (both absolute and relative). Relative dates and times are resolved at the time and date that Amazon Lex V2 receives the request and in the region where it processes the request.

For the `AMAZON.Time` built-in slot type, if the user doesn't specify that a time is before or after noon, the time is ambiguous. In that case, Amazon Lex V2 will prompt the user again. We recommend prompts that elicit an absolute time. For example, use a prompt such as "When do you want your pizza delivered? You can say 6 PM or 6 in the evening."

Avoiding ambiguity in training data for your bot

Providing confusable training data in your bot reduces the ability of Amazon Lex V2 to understand user input. Suppose you have two intents (`OrderPizza` and `OrderDrink`) in your bot, and you include "I want to order" as a sample utterance. When you build your bot, Amazon Lex V2 is unable to map this utterance to a specific intent. As a result, when a user inputs this utterance at runtime, Amazon Lex V2 can't pick an intent with a high degree of confidence.

If you have two intents with the same sample utterance, use input contexts to help Amazon Lex V2 distinguish between the two intents at runtime. For more information, see Setting intent context.

Using the TSTALIASID alias

- The TSTALIASID alias of your bot points to the Draft version and should only be used for manual testing. Amazon Lex limits the number of runtime requests that you can make to the TSTALIASID alias of the bot.
- When you update the Draft version of the bot, Amazon Lex shuts down any in-progress conversations for any client application using the TSTALIASID alias of the bot. Generally, you should not use the TSTALIASID alias of a bot in production because the Draft version can be updated. You should publish a version and an alias and use them instead.
- When you update an alias, Amazon Lex takes a few minutes to pick up the changes. When you modify the Draft version of the bot, the change is picked up by the TSTALIASID alias immediately.

Quotas

Service quotas, also referred to as limits, are the maximum number of service resources allowed for your AWS account. For more information, see AWS service quotas in the *AWS general reference*.

Some service quotas can be adjusted or increased. Refer to the **Adjustable** column in the following tables to see whether a quota can be adjusted and to the **Self-service** column to see whether you can request a quota adjustment through the Service quotas console. Contact Support to increase a quota that is adjustable, but not through self-service. It can take a few days to increase a service quota. If you're increasing your quota as part of a larger project, be sure to add this time to your plan.

> **Note**
>
> Character limits are calculated as the number of Unicode code units. In most cases, one Unicode character is equivalent to one Unicode code unit. Some special characters might be greater than one unit and counts might differ for different encodings. For more information on calculating string length, see this documentation.

Build-time quotas

The following maximum quotas are enforced when you are creating a bot.

Description	Default	Adjustable	Self-service
Bots per AWS account	100	Yes	Yes
Bot channel associations per AWS account	5,000	No	N/A
Parallel locale builds per AWS account	5	Yes	No
Bots per bot network	5	No	N/A
Bot networks per bot	25	No	N/A

Description	Default	Adjustable	Self-service
Versions per bot	100	No	N/A
Intents per locale in each bot	• 1,000 in en-AU, en-GB, and en-US • 250 in all other locales	Yes	No
Slots per locale in each bot	• 4,000 in en-AU, en-GB, and en-US • 2,000 in all other locales	No	N/A
Custom slot types per bot locale	• 250 in en-AU, en-GB, and en-US • 100 in all other locales	No	N/A
Custom slot type values and synonyms per locale in each bot	50,000	No	N/A
Total characters in sample utterances per locale in each bot	• 2,000,000 in en-AU, en-GB, and en-US • 200,000 in all other locales	No	N/A
Channel associations per bot alias	10	No	N/A
Slots per intent	100	No	N/A
Sample utterances per intent	1,500	Yes	Yes

Description	Default	Adjustable	Self-service
Characters per sample utterance	500	No	N/A
Text response length	4,000	No	N/A
Sample utterances per slot	10	Yes	Yes
Characters per sample slot utterance	500	No	N/A
Prompts per slot	30	No	N/A
Values and synonyms per custom slot type	10,000	No	N/A
Characters per custom slot type value	500	No	N/A
Characters in a channel association name	100	No	N/A
Number of concurrent Automated Chatbot Designer analysis jobs across all bots in your account per Region	10	No	N/A
Size of custom grammar slot type XML file	100 KB	No	N/A

Runtime quotas

The following maximum quotas are enforced at runtime.

Description	Default	Adjustable	Self-service
Input text size for RecognizeText and RecognizeUtterance	1024 characters	No	N/A
Speech input length for `RecognizeUtterance` operation	55 seconds	Yes	No
Size of `RecognizeUtterance` headers	16 KB	No	N/A
Size of combined request and session headers for `RecognizeUtterance`	12 KB	No	N/A
Maximum number of concurrent text-mode conversations for `RecognizeText`, `RecognizeUtterance`, or `StartConversation` for the TestBotAlias	2	No	N/A
Maximum number of concurrent text-mode conversat	50	Yes	No

Description	Default	Adjustable	Self-service
ions for `RecognizeText` , `RecognizeUtterance` , or `StartConversation` for other aliases			
Maximum number of concurrent voice-mode conversations for `RecognizeUtterance` for the TestBotAlias	2	No	N/A
Maximum number of concurrent voice-mode conversations for `RecognizeUtterance` for other aliases	125	Yes	No
Maximum number of concurrent voice-mode conversations for `StartConversation` for the TestBotAlias	2	No	N/A
Maximum number of concurrent voice-mode conversations for `StartConversation` for other aliases	400	Yes	No

Description	Default	Adjustable	Self-service
Maximum number of concurrent session management operations (`PutSession`, `GetSession`, or `DeleteSession`) when using the TestBotAlias	2	No	N/A
Maximum number of concurrent session management operations (`PutSession`, `GetSession`, or `DeleteSession`) when using other aliases	50	Yes	No
Maximum input size to a Lambda function	12 KB	No	N/A
Maximum output size of a Lambda function	50 KB	No	N/A
Maximum size of session attributes in Lambda function output (after base-64 encoding)	12 KB	No	N/A
Maximum timeout of a Lambda function	30 seconds	Yes	No

Description	Default	Adjustable	Self-service
Maximum duration for a single conversation	15 minutes	No	No
Maximum audio length	55 seconds	No	No

Amazon Lex V1 to V2 migration guide

The Amazon Lex V2 console and APIs make it easier to build and manage bots. Use this guide to learn about the improvements in the Amazon Lex V2 API as you migrate bots.

You migrate a bot using the Amazon Lex console or API. For more information see Migrating a bot in the *Amazon Lex developer guide*.

Amazon Lex V2 overview

Multiple languages can be added to a bot so you can manage them as a single resource. A simplified information architecture lets you efficiently manage your bot versions. Capabilities such as a 'conversation flow', partial saving of bot configuration and bulk upload of utterances give you more flexibility.

Multiple languages in a bot

You can add multiple languages with the Amazon Lex V2 API. You add, modify, and build each language independently. Resources such as slot types are scoped at the language level. You can quickly move between different languages to compare and refine the conversations. You can use one dashboard in the console to review utterances for all languages for faster analysis and iterations. A bot operator can manage permissions and logging operations for all languages with one bot configuration. You must provide a language as a runtime parameter to converse with a Amazon Lex V2 bot. For more information, see Languages and locales supported by Amazon Lex V2.

Simplified information architecture

The Amazon Lex V2 API follows a simplified information architecture (IA) with intent and slot types scoped to a language. You version at the bot level so that resources such as intents and slot types aren't versioned individually. By default, a bot is created with a *Draft* version that is mutable and used for testing changes. You can create numbered snapshots from the draft version. You choose the languages to include in a version. All resources within the bot (languages, intents, slot types) are archived as part of creating a bot version. For more information, see Versions.

Improved builder productivity

You have additional builder productivity tools and capabilities that give you more flexibility and control of your bot design process.

Save partial configuration

The Amazon Lex V2 API enables you to save partial changes during development. For example, you can save a slot that references a deleted slot type. This flexibility enables you to save your work and return to it later. You can resolve these changes before building the bot. In Amazon Lex V2 the partial save can be applied to slots, versions, and aliases.

Renaming resources

With Amazon Lex V2 you can rename a resource after it's created. Use a resource name to associate user-friendly metadata with each resource. The Amazon Lex V2 API assigns every resource a unique 10-character resource ID. All resources have a resource name. You can rename the following resources:

- Bot
- Intent
- Slot type
- Slot
- Alias

You can use resource IDs to read and modify your resources. If you are using the AWS Command Line Interface or the Amazon Lex V2 API to work with Amazon Lex V2, resource IDs are required for certain commands.

Simplified management of Lambda functions

In the Amazon Lex V2 API you define one Lambda function per language instead of a function for each intent. The Lambda function is configured in the alias for the language and is used for both the dialog and fulfillment code hook. You can still choose to enable or disable the dialog and fulfillment code hooks independently for each intent. For more information, see Integrating an AWS Lambda function into your Amazon Lex V2 bot.

Granular settings

The Amazon Lex V2 API moves the voice and intent classification confidence score threshold from the bot to the language scope. The sentiment analysis flag moves from bot scope to alias scope. Session time out and privacy settings at the bot scope, and conversation logs at the alias scope, remain unchanged.

Default fallback intent

The Amazon Lex V2 API adds a default fallback intent when you create a language. Use it to configure error handling for your bot instead of specific error-handling prompts.

Optimized session variable update

With the Amazon Lex V2 API you can update session state directly with the RecognizeText and RecognizeUtterance operations without any dependency on session APIs.

Creating Amazon Lex V2 resources with AWS CloudFormation

Amazon Lex V2 is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want (such as Amazon Lex V2 chatbots), and CloudFormation provisions and configures those resources for you.

When you use CloudFormation, you can reuse your template to set up your Amazon Lex V2 resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

Amazon Lex V2 and CloudFormation templates

To provision and configure resources for Amazon Lex V2 and related services, you must understand CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use CloudFormation Designer to help you get started with CloudFormation templates. For more information, see What is CloudFormation Designer? in the *AWS CloudFormation User Guide*.

Amazon Lex V2 supports creating the following resources in CloudFormation:

- AWS::Lex::Bot
- AWS::Lex::BotAlias
- AWS::Lex::BotVersion
- AWS::Lex::ResourcePolicy

For more information, including examples of JSON and YAML templates for these resources, see the Amazon Lex V2 resource type reference in the *AWS CloudFormation User Guide*.

Learn more about CloudFormation

To learn more about CloudFormation, see the following resources:

- AWS CloudFormation

- AWS CloudFormation user guide
- CloudFormation API reference
- AWS CloudFormation Command line interface user guide

Document history for Amazon Lex V2

- **Latest documentation update:** October 27th, 2025

The following table describes important changes in each release of Amazon Lex V2. For notification about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Date
Feature update	Wait and continue is now available in additional locales: ca-ES, de-AT, de-DE, es-419, es-ES, es-US, fr-CA, fr-FR, it-IT, ja-JP, ko-KR, pt-BR, pt-PT, zh-CN, zh-HK. For more information, see Enabling the Amazon Lex V2 bot to wait for the user to provide more information during a pause.	October 27, 2025
Feature update	AMAZON.Currency and AMAZON.Confirmation built-in slot types now support all locales. For more information, see Built-in slot types.	August 18, 2025
Documentation update	Updated documentation for Assisted NLU feature with improved examples and clarifications. For more information, see Assisted NLU.	August 4, 2025
New feature	You can use the AMAZON.BedrockAgentIntent built-in intent to connect your Amazon Lex V2 bot with	July 3, 2025

	Amazon Bedrock Agents and Knowledge Bases. For more information, see AMAZON.BedrockAgentIntent built-in intent.	
Update to AWS managed policy	Amazon Lex V2 updated the policy to allow replication of tags and ResourceBasedPolicy when using global resiliency see AmazonLexReplicationPolicy.	June 24, 2025
New feature	You can now use Assisted NLU to improve intent classification and slot resolution in Amazon Lex V2. This feature uses Large Language Models (LLMs) to enhance accuracy while staying within your bot's configured intents and slots. For more information, see Assisted NLU.	June 11, 2025
Update to feature (2025-06-03)	You can use 17 new languages with your custom vocabularies in Amazon Lex V2. For more information, see Custom vocabularies.	June 3, 2025
New feature (2024-12-01)	You can use the QinConnect built-in intent to connect your Amazon Lex V2 bot with Amazon Connect. For more information, see QinConnect built-in intent.	December 1, 2024

New feature (2024-10-01)	You can use AMAZON.Currency built-in slot for US and Spanish locales in Amazon Lex V2. For more information, see AMAZON.Currency.	October 1, 2024
Update to feature (2024-07-26)	The QnA built-in intent for generative AI now supports Bedrock Knowledge Base and Guardrails in Amazon Lex V2. For more information, see AMAZON.QnAIntent.	July 26, 2024
Update to feature (2024-07-10)	The QnA built-in intent for generative AI now supports Anthropic Claude 3 Haiku and Anthropic Claude 3 Sonnet models in Amazon Lex V2. For more information, see Optimize bot creation and performance with generative AI.	July 10, 2024
Update to AWS managed policy (2024-05-10)	Amazon Lex V2 added new permissions to the AmazonLexReadOnly managed policy to allow read access to bot resources that have been replicated in other regions.	May 10, 2024
Update to AWS managed policy (2024-04-15)	Amazon Lex V2 added new permissions to the AmazonLexFullAccess managed policy to allow updating of replicated bot resources to other regions.	April 15, 2024

Region expansion	Amazon Lex V2 is now available in AWS GovCloud (US-West) (us-gov-west-1).	March 22, 2024
Update to AWS managed policy (2024-03-07)	Amazon Lex V2 added new permissions to the AmazonLexReplicationPolicy managed policy to allow updating of replicated bot resources to other regions.	March 7, 2024
New function	You can use fn.LENGTH() function to determine the value length of a string value in Amazon Lex V2. For more information, see Conditional branching - Functions.	March 4, 2024
Update to feature (2024-02-28)	The QnA built-in slot for generative AI capabilities in Amazon Lex V2 is now GA. For more information, see Optimize bot creation and performance with generative AI.	February 28, 2024
Update to AWS managed policy (2024-02-08)	Amazon Lex V2 added new permissions to the AmazonLexFullAccess managed policy to allow replication of bot resources to other regions.	February 8, 2024

New managed policy	Amazon Lex V2 added a managed policy providing permissions to replicate bot resources in other regions. For more information, see AmazonLexReplicationPolicy.	February 8, 2024
New feature (2024-02-08)	You can use Global resilienc y to replicate your bot in a second AWS region in Amazon Lex V2. For more information, see Global resiliency.	February 8, 2024
New feature (2023-11-29)	You can now take advantage of generative AI capabilities in Amazon Lex V2. For more information, see Optimize bot creation and performance with generative AI.	November 29, 2023
New feature (2023-11-08)	Amazon Lex V2 can now use selective logging to capture text and/or audio at the intent or slot level. For more information, see Selective logging.	November 8, 2023
New feature (2023-08-17)	Amazon Lex V2 can now use a built-in slot to determine Yes/No/Maybe/Don't know responses using `AMAZON.Co nfirmation` . For more information, see Built-in Slot Types.	August 17, 2023

New feature (2023-07-18)	You can view the performance metrics of intents and slots, in addition to other conversational metrics using the analytics dashboard. For more information, see Analytics.	July 18, 2023
New feature (2023-06-06)	You can improve the accuracy and fulfillment success of your bot with the Test Workbench. For more information, see Test Workbench.	June 6, 2023
New feature (2023-02-23)	You can now create bots from a template for some popular business verticals. For more information, see Bot templates.	February 23, 2023
New feature (2023-02-09)	You can now combine multiple bots into a network of bots to create an integrated customer experience. For more information, see Network of bots.	February 9, 2023

New feature (2022-12-06)	Amazon Lex V2 now supports the Gulf Arabic (United Arab Emirates), Cantonese (Hong Kong), Finnish (Finland) , Norwegian (Norway), Polish (Poland), and Swedish (Sweden) locales. For more information, see Languages and locales supported by Amazon Lex V2.	December 6, 2022
Updated to AWS managed policy	Amazon Lex V2 added new permissions to the AmazonLexReadOnly managed policy to allow the display of custom vocabulary items.	November 29, 2022
New feature (2022-11-07)	Amazon Lex V2 can display an alternative representation for a phrase or word by using the console or APIs to customize the speech to text output. For more information, see Creating a custom vocabulary for speech recognition.	November 7, 2022
New feature (2022-10-28)	Amazon Lex V2 can add a weight attribute to an item element that will represent the degree to which the phrase is boosted during speech recognition. For more information, see Grammar Weights.	October 28, 2022

New feature (2022-10-19) (2022-10-19) - 2	Amazon Lex V2 can display an alternative representation for a phrase or word in the final speech to text output. For more information, see Creating a custom vocabulary for speech recognition.	October 19, 2022
New feature (2022-10-19) (2022-10-19)	Amazon Lex V2 can be used to capture free form input from the end user made up of words or characters using `AMAZON.FreeFormInput` . For more information, see Built-in Slot Types.	October 19, 2022
New feature (2022-10-14)	Amazon Lex V2 now supports the Hindi (India) and Dutch (The Netherlands) locales. For more information, see Languages and locales supported by Amazon Lex V2.	October 14, 2022
New feature (2022-09-22)	There was an update to the way Amazon Lex V2 manages user input. Now you can select whether Amazon Lex V2 accepts text, audio or DTMF input at any point in the conversation flow. For more information, see Configurable Attributes.	September 22, 2022

New feature (2022-09-14)	There was an update to the way Amazon Lex V2 manages conversation flows. Visual conversation builder is a drag and drop conversation builder to easily design and visualize conversation paths. For more information, see Visual Conversation Builder.	September 14, 2022
New feature (2022-09-09)	There was an update to the way Amazon Lex V2 builds complex slots. You can now create complex subslots within slots to manage intents in complex conversat ion design. For more informati on, see Building composite slots.	September 9, 2022
New feature (2022-08-17)	There was an update to the way Amazon Lex V2 manages the flow of conversational paths with your users. You can now create complex conversat ional paths by ordering the next step in the conversation. For more information, see Creating conversation paths.	August 17, 2022

New feature (2022-07-05)	There was an update to the way Amazon Lex V2 manages the flow of conversations with your users. You can now create complex conversat ions by ordering the prompts. For more information, see Configuring prompts.	July 5, 2022
New feature (2022-05-03)	There was an update to the way Amazon Lex V2 manages the flow of conversations with your users. You can now create complex conversations using conditions. For more information, see Understan ding the new conversation flows.	May 3, 2022
New feature (2022-03-22)	Added example industry grammars for the built-in grammar slot type. For more information, see Industry grammars.	March 22, 2022
New feature (2022-03-18)	Added documentation about integrating Amazon Lex V2 with the Amazon Chime SDK. For more information, see Amazon Chime SDK.	March 18, 2022

New feature (2022-01-27)	Amazon Lex V2 now provides confidence scores for voice transcriptions. Use the score to help determine the correct response from the user. For more information, see Using voice transcription confidence scores.	January 27, 2022
New feature (2022-01-13)	You can now add contextual and dynamic hints to slots to improve the accuracy of your bot. For more information, see Using hints to improve accuracy.	January 13, 2022
New feature (2022-01-12)	Amazon Lex V2 adds support for custom vocabularies to improve speech recognition for audio input. For more information, see Creating a custom vocabulary to improve speech recognition.	January 12, 2022
New feature (2022-01-07)	Amazon Lex V2 now supports AWS PrivateLink. For more information, see VPC endpoints (AWS PrivateLink) .	January 7, 2022
New feature (2022-01-03)	Amazon Lex V2 now supports the Catalan (Spain) locale. For more information, see Languages and locales supported by Amazon Lex V2.	January 3, 2022

New feature (2021-12-20) (2021-12-20) - 2	AWS CloudFormation now supports Amazon Lex V2. For more information, see AWS CloudFormation resources.	December 20, 2021
New feature (2021-12-20) (2021-12-20)	You can now create slot types using your own custom grammar. For more informati on, see Using a custom grammar slot type.	December 20, 2021
New feature (2021-12-16)	Amazon Lex V2 now supports the Portuguese (Brazil), Portuguese (Portugal), and Mandarin (PRC) locales. For more information, see Languages and locales supported by Amazon Lex V2.	December 16, 2021
New feature (2021-12-01)	Amazon Lex V2 now provides a preview of the Automated Chatbot Designer to help you get started creating a chatbot from contact center transcrip ts. For more information, see Using the Automated Chatbot Designer (Preview).	December 1, 2021
New feature (2021-11-19) (2021-11-19) - 2	You can now use Amazon Polly neural text to speech (NTTS) voices for audio conversations with your users. For more information, see Voices in Amazon Polly.	November 19, 2021

New feature (2021-11-19) (2021-11-19)	You can now use spell-by-letter and spell-by-word styles for entering slot values that Amazon Lex V2 has difficulty understanding. For more information, see Using spelling styles to capture slot values.	November 19, 2021
New feature (2021-11-09)	Amazon Lex V2 now supports the English (South Africa) locale. For more information, see Languages and locales supported by Amazon Lex V2.	November 9, 2021
New feature (2021-11-05)	Amazon Lex V2 now supports the German (Austria) locale. For more information, see Languages and locales supported by Amazon Lex V2.	November 5, 2021
New feature (2021-10-07)	You can now provide users with update messages that play at the start of a fulfillment function and periodically while the function runs. You can also create messages that inform the user of status of fulfillment when the function is complete. For more information, see Configuring fulfillment progress updates.	October 7, 2021

Region expansion (2021-09-22)	Amazon Lex V2 is now available in Africa (Cape Town) (af-south-1) and Asia Pacific (Seoul) (ap-north east-2).	September 22, 2021
New feature (2021-09-22)	You can now view statistic s for the utterances that your users send to your bot. For more information, see Viewing utterance statistics.	September 22, 2021
New feature (2021-09-09)	Amazon Lex V2 now supports the Korean (Korea) locale. For more information, see Languages and locales supported by Amazon Lex V2.	September 9, 2021
New feature (2021-07-27)	Amazon Lex V2 now provides a built-in slot type for UK postal codes. For more information, see AMAZON.UK PostalCode.	July 27, 2021
New feature (2021-07-15)	Amazon Lex V2 now supports the English (Indian) locale. For more information, see Languages and locales supported by Amazon Lex V2.	July 15, 2021
New feature (2021-07-13)	Amazon Lex V2 now provides a tool to migrate a bot from Amazon Lex V1 to the Amazon Lex V2 API. For more information, see Migrating a bot in the *Amazon Lex Developer Guide*.	July 13, 2021

New feature (2021-06-26)	You can now configure your Amazon Lex V2 bot to re-elicit slots that have already been filled by setting slot values to null and configuring the conversation flow to loop back. This feature enables more dynamic conversations based on customer responses. For more information, see Re-eliciting slots.	June 26, 2021
New documentation	Amazon Lex V2 now provides comprehensive guidelines and best practices for bot development to help you create more effective conversational experiences. For more information, see Guidelines for Amazon Lex V2 bot development.	June 26, 2021
New feature (2021-06-15)	Amazon Lex V2 now enables you to accept multiple values for a single slot in the English (US) language. For more information, see Using multiple values in a slot.	June 15, 2021
New feature (2021-06-11)	You can now build larger bots for English languages. For more information, see Quotas.	June 11, 2021

New feature (2021-05-20)	Use Amazon Lex V2 resource-based policies to manage access to your bots and bot aliases. For more information, see Resource-based policies within Amazon Lex V2.	May 20, 2021
New feature (2021-05-18)	Amazon Lex V2 now enables you to import and export bots and bot locales. You can use this feature to copy bots and bot locales between accounts and AWS Regions. For more information, see Importing and exporting.	May 18, 2021
Region expansion (2021-05-17)	Amazon Lex V2 is now available in Canada (Central) (ca-central-1).	May 17, 2021
New feature (2021-04-01)	Amazon Lex V2 now supports the Japanese (Japan) locale. For more information, see Languages and locales supported by Amazon Lex V2.	April 1, 2021
New feature (2021-03-12)	Amazon Lex V2 now supports three new built-in slot types: `AMAZON.City` , `AMAZON.Country` , and `AMAZON.State` .	March 12, 2021
New guide	This is the first release of the *Amazon Lex V2 user guide*.	January 21, 2021

API reference

The API Reference is now a separate document.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.