



Developer Guide

Deadline Cloud



Deadline Cloud: Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is Deadline Cloud?	1
Open Job Description	2
Concepts and terminology	2
Farm resources	2
Job execution resources	4
Other important concepts and terminology	7
Architecture Guidance	10
Job source	12
Interactive workflow	12
Automated workflow	12
Job submission	12
Integrated submitter with DCC	13
Custom job definition	13
Application management	13
Deadline Cloud-managed conda channel for service-managed fleets (SMF)	14
Self-managed conda channel	14
Custom application management	14
Application licensing	15
Service-managed fleets and usage-based licensing	15
Customer-managed fleets and usage-based licensing	15
Custom licensing	15
Asset access	16
Job attachments	16
Custom storage access	16
Perforce source control	17
Job monitoring and output management	18
Deadline Cloud monitor	18
Custom monitor application	18
Automated monitoring solution	18
Worker infrastructure management	18
Service-managed fleets	19
Customer-managed fleets	19
Example architectures	19
Traditional production studio	19

Studio in the Cloud	22
ECommerce Automation	23
Whitelabel/OEM/B2C Customer	26
Hybrid render farm	29
What is a Deadline Cloud workload	30
How workloads arise from production	30
The ingredients of a workload	31
Workload portability	32
Migrate from Deadline 10	35
Farm components you no longer operate	35
Map Deadline 10 concepts to Deadline Cloud	36
Route jobs to specific workers (pools and groups)	39
Features without a direct equivalent	39
Choose a starting point	40
Getting started	41
Create a farm	41
Next steps	45
Run the worker agent	45
Next steps	48
Submit jobs	48
Submit the simple_job sample	49
Submit with a parameter	52
Create a simple_file_job job	53
Next steps	56
Submit jobs with attachments	56
Configure queue for job attachments	57
Submit with job attachments	60
How job attachments are stored	62
Next steps	65
Add a service-managed fleet	66
Next steps	68
Clean up farm resources	68
Build a job	72
Job bundles	73
Job template elements	76
Task chunking	81

Parameter values elements	87
Asset references elements	89
Using files in your jobs	92
Sample project infrastructure	93
Storage profiles and path mapping	95
Job attachments	103
Submitting files with a job	103
Getting output files from a job	116
Using files in a dependent step	119
Create resource limits for jobs	122
Stopping and deleting limits	123
Create a limit	124
Associate a limit and a queue	125
Submit a job requiring limits	125
Submit a job	129
From a terminal	129
From a script	130
From within applications	131
Share job bundles	133
Upload a bundle	134
Use the command line	134
Manage queue bundles	135
Host requirements and capabilities	136
Fleet configuration provides the hardware	137
Host requirements select a fleet	137
Worker capabilities are informational	138
Auto scaling counts tasks	138
Schedule jobs	138
Scheduling configurations	138
Determine fleet compatibility	141
Fleet scaling	148
Sessions	149
Step dependencies	151
Modify jobs	153
Hooks and integration points	159
Run custom logic on the workstation	161

Run custom logic on the worker	161
Respond to events in the cloud	162
Choose an integration point	163
Additional resources	163
Submission hooks	164
Hook types	164
Pre-GUI hooks	165
Pre-submission hooks	167
Post-submission hooks	168
Configure submission hooks	168
Hook definition fields	169
Path resolution	170
Hook input	170
Environment variables	170
JSON on stdin	171
Security	171
Enabling bundle hooks	171
Enabling environment hooks	171
Confirmation prompt	172
Configuration settings summary	173
Studio deployment	173
Error handling	174
Best practices	174
Submission methods	174
Additional resources	175
Customer-managed fleets	176
Create a CMF	176
Worker host setup	181
Configure a Python environment	182
Install worker agent	183
Configure worker agent	185
Create job users and groups	186
Securing your worker host	188
Manage access	190
Grant access	191
Revoke access	191

Install software for jobs	192
Install DCC adaptors	193
Configure credentials	194
Worker host data flow	198
Endpoints and protocols	198
API operations used by workers	199
Other data transmitted	200
Private connectivity options	201
Test your worker host	201
Create an AMI	203
Prepare the instance	204
Build the AMI	206
Create fleet infrastructure	206
Auto scale your fleet	211
Fleet health check	217
Service-managed fleets	218
Connect VPC resources to your SMF	218
How VPC resource endpoints work	219
Prerequisites	219
Set up a VPC resource endpoint	220
Accessing your VPC resources	220
Authentication and security	221
Technical considerations	221
Troubleshooting	221
Job attachments	222
Choose a filesystem mode	222
Optimize transfer performance	223
Download job outputs	224
Persistent storage	224
Infrastructure as code	225
CloudFormation	225
AWS CDK	226
Terraform	226
Resources outside your templates	226
Hybrid rendering	227
Distribute jobs across your fleets	229

Choose a queue strategy	229
License applications on both fleets	230
Give both fleets access to your assets	230
Connect cloud workers to your network	231
Match applications to each fleet type	231
Control cloud costs	232
Related resources	232
Deploy and configure custom software on workers	233
Choose a deployment method	233
Use custom plugins with Deadline Cloud	234
Deliver custom plugins with plugin sync	236
Configure jobs using queue environments	240
Control the job environment	241
Provide applications for your jobs	257
Create a conda channel using S3	262
Build and test packages locally	263
Publish packages to an Amazon S3 conda channel	268
Configure production queue permissions for custom conda packages	274
Add a conda channel to a queue environment	275
Create a conda package for an application or plugin	276
Create a conda build recipe for Blender	279
Create a conda recipe for Maya	281
Create a conda recipe for the Maya adaptor	284
Create a conda recipe for MtoA plugin	286
Automate package builds with Deadline Cloud	288
Host configuration scripts	292
Troubleshooting	295
Using software licenses	299
Usage-based licensing	300
Bring your own license	300
Combining BYOL and UBL	301
Connect SMF fleets to a license server	301
Step 1: Configure the queue environment	302
Step 2: (Optional) License proxy instance setup	312
Step 3: CloudFormation template setup	313
Connect CMF fleets to a license endpoint	323

Step 1: Create a security group	324
Step 2: Set up the license endpoint	325
Step 3: Connect a rendering application to an endpoint	325
Step 4: Delete a license endpoint	328
Combining BYOL and UBL	328
How combined licensing works	329
Example: Using BYOL Cinema 4D licenses with UBL fallback	329
Considerations for combined licensing	330
Using AI agents	331
Monitoring	334
CloudTrail logs	335
Deadline Cloud data events in CloudTrail	336
Deadline Cloud management events in CloudTrail	338
Deadline Cloud event examples	341
Monitoring with CloudWatch	343
CloudWatch metrics	344
Recommended alarms	347
Managing events using EventBridge	348
Deadline Cloud events	348
Sending Deadline Cloud events	349
Events detail reference	350
Querying session statistics aggregated data	365
Starting an aggregation request	365
Retrieving results	366
Retrieving user metadata using userID	367
To map a user ID	367
Finding your Identity Store ID	368
Verifying the user mapping	369
Additional resources	369
Code examples	370
Tutorials	373
FLUX.2 Klein LoRA	373
vLLM LLM leaderboard	379
Gaussian Splatting pipeline	384
CARLA simulation sweep	389
LLM LoRA fine-tuning	395

vLLM batch inference	402
Text-to-image batch	406
Job bundles	410
Blender render	412
Blender wedge from CSV	413
Maya render	414
Arnold standalone render	414
Houdini Husk USD render	415
Nuke render	416
3ds Max V-Ray denoiser	417
KeyShot standalone	417
VRED render	418
After Effects render	419
V-Ray render	419
Redshift render	420
POV-Ray render	421
Tile rendering	421
Task chunking	422
FFmpeg encode video	423
Turntable with Maya/Arnold	424
Monte Carlo simulation	425
3D Gaussian Splatting pipeline	425
FLUX.2 LoRA training	427
vLLM benchmark leaderboard	427
vLLM batch inference	428
Text-to-image batch	428
Wan2.2 video generation	429
LLM LoRA fine-tuning (job bundle)	430
ESMFold protein structure prediction	430
Satellite imagery classification	431
GROMACS molecular dynamics	432
Virtual screening (AutoDock Vina)	433
Variant calling	434
Infinigen scene generation	434
MuJoCo sim-to-policy	435
CARLA autonomous driving	436

Flow Production Tracking integration	437
VFX pipeline sample	438
VTK visualization	438
FFmpeg movie from job output	439
Bash CLI job	440
GUI control showcase	440
Job environments	441
Job attachments tutorial	442
Job development progression	442
List conda packages	443
Copy S3 prefix to job attachments	443
SSH to worker	444
Custom submitters	446
Conda recipes	446
Blender conda recipes	448
Maya conda recipes	449
Houdini conda recipes	450
Cinema 4D conda recipes	450
Nuke conda recipes	451
3ds Max conda recipes	452
After Effects conda recipes	452
KeyShot conda recipe	452
VRED conda recipes	452
Unreal Engine conda recipe	453
V-Ray conda recipe	453
NeRF Studio conda recipe	454
Infinigen conda recipe	455
AutoDock Vina conda recipe	455
AYON Launcher conda recipe	456
CloudFormation templates	456
Starter farm	457
CUDA farm	457
SMF with VPC and FSx	458
Capacity manager	458
Standby scheduling	459
CMF health check	460

Budget notifications	460
Job event Slack notifications	461
AWS CDK app	461
Terraform configurations	462
Host configuration scripts	463
Docker with NVIDIA Container Toolkit	463
3ds Max host configuration	464
After Effects with Red Giant	466
Cinema 4D with Red Giant	467
Install fonts from S3	467
Passwordless sudo for job-user	468
Enable swap	468
Configure the Windows page file	469
Reboot worker	469
Queue environments	470
Console conda queue environments	471
Inline conda queue environments	471
py-rattler queue environment	472
Rez queue environment	472
Pip queue environment	473
Disconnect UBL	474
Amazon Linux 2023 worker image	474
Blender container image	475
Upload to job attachments	476
Virtual workstation setup	477
License limits hook	478
AI agent skills	479
Security	481
Security controls	482
Where to start	484
Data flow and encryption	484
Components and where they run	485
Data flows, ports, and protocols	487
Access to the monitor web application	488
Encryption summary	489
Security best practices	489

Isolate workloads	490
Job OS users	491
Job attachment buckets	492
Worker hosts	496
Workstations	497
Network restrictions	497
Verify downloaded software	500
Identity and Access Management	507
Audience	507
Authenticating with identities	508
Managing access using policies	509
How Deadline Cloud works with IAM	511
Identity-based policy examples	516
AWS managed policies	527
Service roles	531
Troubleshooting	545
Data protection	547
Encryption at rest	548
Encryption in transit	549
Key management	549
Inter-network traffic privacy	556
Opt out	556
AWS PrivateLink	558
Considerations	558
Deadline Cloud endpoints	558
Create endpoints	559
Restricted network environments	560
AWS API endpoints to allowlist	560
Web domains to allowlist	561
Environment-specific endpoints to allowlist	561
Cross-service confused deputy prevention	562
Compliance validation	563
Resilience	564
Infrastructure security	564
Configuration and vulnerability analysis	565
Document history	566

What is AWS Deadline Cloud?

AWS Deadline Cloud is a fully-managed AWS service that enables you to have a scalable processing farm up and running in minutes. It provides an administration console for managing users, farms, queues for scheduling jobs, and fleets of workers that do the processing.

This developer guide is for pipeline, tools, and applications developers with use cases including the following:

- Pipeline developers and technical directors can integrate Deadline Cloud APIs and features into their custom production pipelines.
- Independent software vendors can integrate Deadline Cloud into their applications, enabling users to submit Deadline Cloud jobs from their workstations or web interfaces.
- Data scientists and engineers can distribute compute-intensive workloads such as scientific simulations, financial modeling, machine learning training, and data processing across scalable fleets of workers.
- Web and cloud-based service developers can integrate Deadline Cloud processing into their platforms, enabling customers to submit and track jobs on demand.

This developer guide covers job bundles, submitters, customer-managed fleets, and integrations with the Deadline Cloud API, AWS Command Line Interface (AWS CLI), and SDKs. If you submit and monitor jobs from a digital content creation (DCC) application, see the [AWS Deadline Cloud User Guide](#). The User Guide also covers setting up farms, queues, users, and budgets in the console. Each topic is documented once, in the guide for its audience. The other guide links to it.

We provide tools that enable you to work directly with any step of your pipeline:

- A command-line interface that you can use directly or from scripts.
- The AWS SDK for 11 popular programming languages.
- A REST-based web interface that you can call from your applications.

You can also use other AWS services in your custom applications. For example, you can use:

- **AWS CloudFormation** to automate creating and removing farms, queues, and fleets.
- **Amazon CloudWatch** to gather metrics for jobs.

- **Amazon Simple Storage Service** to store and manage digital assets and job output.
- **AWS IAM Identity Center** to manage users and groups for your farms.

Open Job Description

Deadline Cloud uses the Open Job Description (OpenJD) specification to specify the details of a job. OpenJD was developed to define jobs that are portable between solutions. You use it to define a job that is a set of commands that run on worker hosts. For more information, see the [OpenJD specification](#) on the GitHub website.

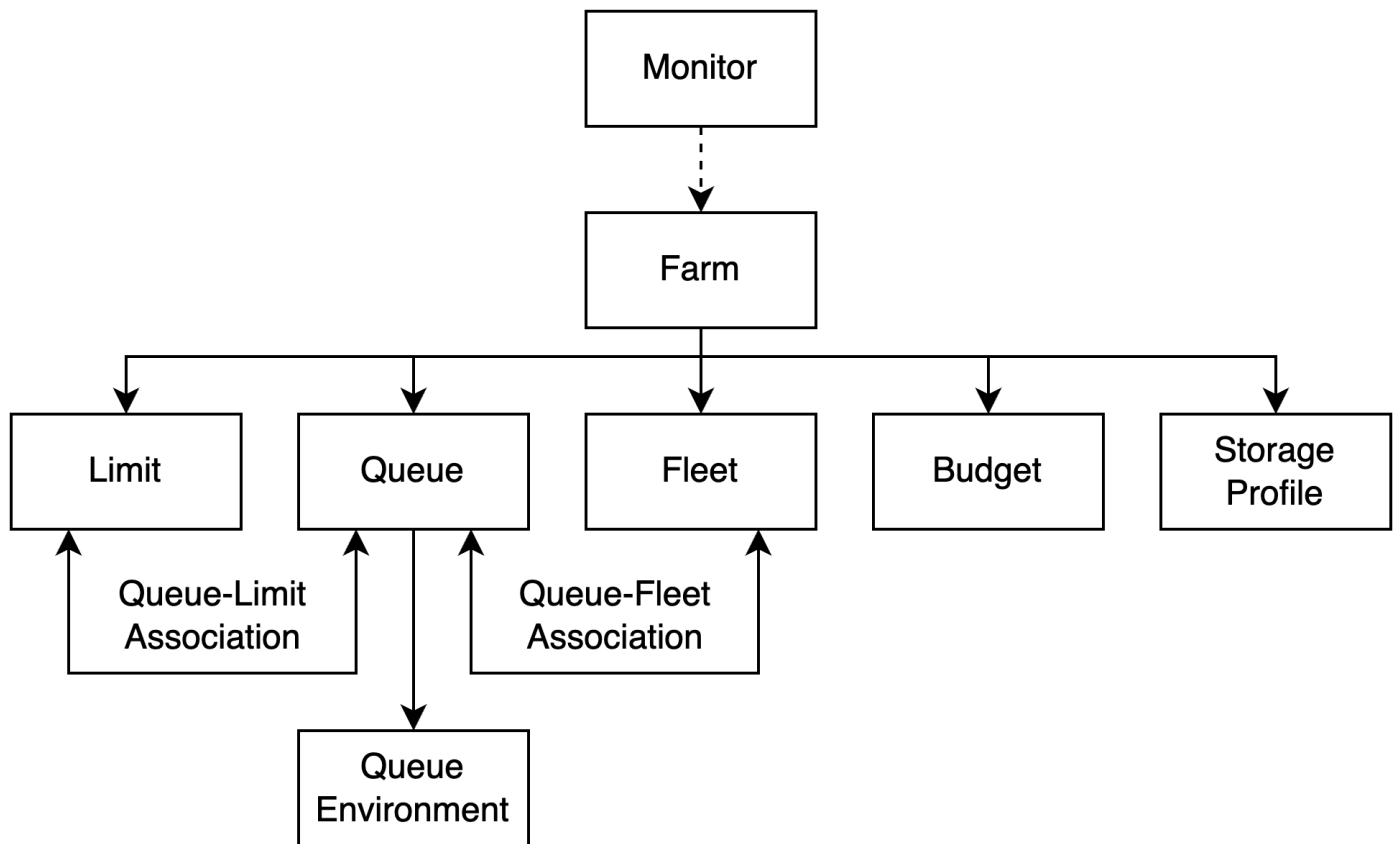
You can create an OpenJD job template using a submitter that Deadline Cloud provides, or you can use any tool that you want to create the template. After creating the template, you send it to Deadline Cloud. If you use a submitter, it takes care of sending the template. If you created the template another way, you call a Deadline Cloud command-line action, or you can use one of the AWS SDKs to send the job. Either way, Deadline Cloud adds the job to the specified queue and schedules the work.

Concepts and terminology for Deadline Cloud

To help you get started with AWS Deadline Cloud, this topic explains some of its key concepts and terminology.

Farm resources

This diagram shows how Deadline Cloud farm resources work together.



Farm

A farm contains all other resources related to submitting and running jobs. Farms are independent from each other making them useful for separating production environments.

Queue

A queue holds jobs for scheduling on associated fleets. Users can submit jobs to a queue and manage their priority and status inside the queue. A queue must be associated with a fleet with a queue-fleet association for its jobs to be run, and queues can be associated with multiple fleets.

Fleet

A fleet contains compute capacity for running jobs. Fleets can be service-managed or customer-managed. Service-managed fleets run in Deadline Cloud and include built-in functionality like autoscaling, licensing, and software access. Customer-managed fleets run on your own compute resources like Amazon EC2 instances or on-premises servers.

Budget

A budget sets spending thresholds for your job activity and allows you to take actions when thresholds are reached, such as stopping job scheduling.

Queue environment

A queue environment defines scripts that run on each worker to set up or tear down the workload environment. They are useful for setting environment variables, installing software, and configuring asset storage.

Storage profile

A storage profile is a configuration for a group of hosts and workstations, that tells where data is located on the file system. Deadline Cloud uses storage profiles to map paths when running jobs on differently configured hosts, such as a job submitted from Windows and running on Linux.

Limit

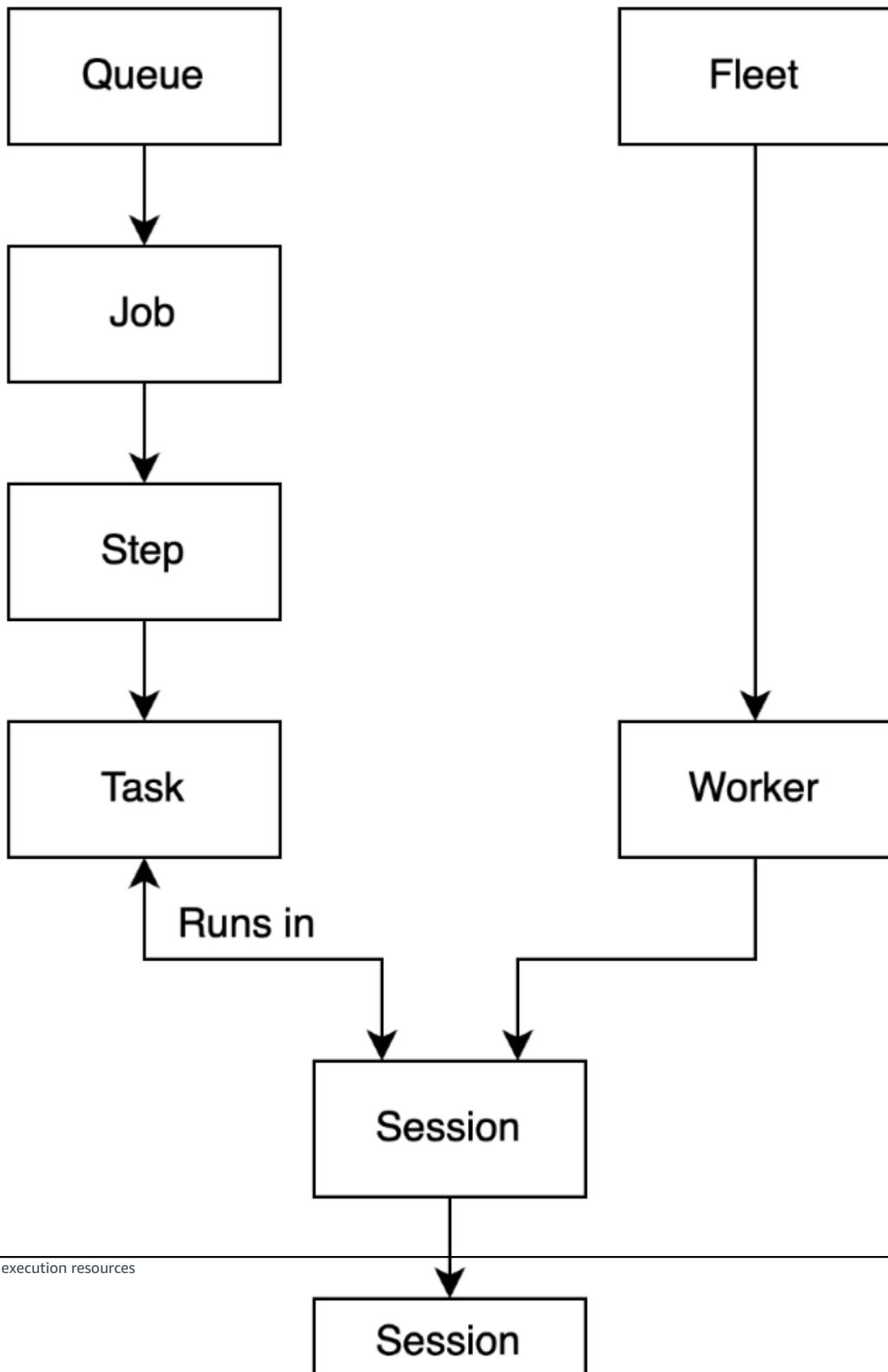
A limit allows you to track usage of shared resources such as floating licenses and control how they are allocated between jobs. Limits are associated with queues with queue-limit associations.

Monitor

The monitor configures the URL for the Deadline Cloud monitor web application, allowing end users to monitor and manage jobs. It can be accessed in a browser or through the Deadline Cloud monitor desktop application.

Job execution resources

This diagram shows how Deadline Cloud job resources work together.



Job

A job is a set of work that a user submits to Deadline Cloud to be scheduled and run on available workers. A job may render a 3D scene or run a simulation. Jobs are created from reusable job templates, which define the runtime environment and processes, and job-specific parameters. Jobs contain steps and tasks that define the work to be performed, and they can be configured with priorities, maximum worker counts, and retry settings.

Job priority

Job priority is the approximate order that Deadline Cloud processes a job in a queue. You can set the job priority between 0 and 100, jobs with a higher number priority are generally processed first. Jobs with the same priority are processed in the order received.

Job properties

Job properties are settings that you define when submitting a render job. Some examples include frame range, output path, job attachments, renderable camera, and more. The properties vary based on the DCC that the render is submitted from.

Step

A step is part of a job that provides a template for running many tasks that are identical except for the task parameter values. Steps can have dependencies on other steps, allowing you to create complex workflows with sequential or parallel execution paths. In rendering jobs, a step often defines the command for rendering a frame and uses the frame number as the task parameter.

Task

A task is the smallest unit of work in Deadline Cloud. Tasks are part of steps and are executed by workers, representing individual operations that need to be performed as part of a job. Tasks can be configured with specific parameters and are assigned to workers based on their capabilities and availability. In rendering jobs, a task often renders a single frame.

Worker

Workers are part of a fleet and execute tasks from jobs. Workers can be configured with specific capabilities such as GPU accelerators, CPU architecture, and operating system. In service-managed fleets, workers are created automatically as the fleet scales out and in.

Instance

Fleets use instances for CPU resources. An instance is an Amazon EC2 performance instance. Deadline Cloud uses On-Demand and Spot instances.

On-Demand instance

On-Demand instances are priced by the second, have no long-term commitment, and will not be interrupted.

Spot instance

Spot instances are unreserved capacity that you can use at a discounted price, but may be interrupted by On-Demand requests.

Wait and Save

The Wait and Save feature provides delayed job scheduling for lower cost and can be interrupted by On-Demand and Spot requests. Wait and Save is only available within Deadline Cloud service-managed fleets.

Wait and Save is for managing the execution of visual computing workloads in AWS Deadline Cloud. See [AWS service terms](#) for details.

Session

A session represents a worker's sequence of work on a job. During a single session, a worker may be assigned multiple tasks which it runs one after another. Sessions often have setup actions which configure environments and load assets before running the task actions.

Session action

A session action represents specific operations performed during a session such as setting up the environment, running a task, and syncing assets.

Other important concepts and terminology

Usage explorer

Usage explorer is a feature of Deadline Cloud monitor. It provides an approximate estimate of your costs and usage.

Budget manager

Budget manager is part of the Deadline Cloud monitor. Use the budget manager to create and manage budgets. You can also use it to limit activities to stay within budget.

Deadline Cloud client library

The open-source client library includes a command line interface and library for managing Deadline Cloud. Functionality includes submitting job bundles based on the Open Job Description specification to Deadline Cloud, downloading job attachment outputs, and monitoring your farm using the command line interface (CLI).

Digital content creation application (DCC)

Digital content creation applications (DCCs) are third-party products where you create digital content. Deadline Cloud has built-in integrations with many DCCs such as Autodesk Maya, Blender, and Maxon Cinema 4D allowing you to submit jobs from within the DCC and render on service-managed fleets with pre-configured software and licensing.

Job attachments

Job attachments are a Deadline Cloud feature that you upload and download assets as part of a job such as textures, 3D models, and lighting rigs. Job attachments are stored in Amazon S3 and avoid the need for shared network storage.

Job template

A job template defines the runtime environment and all processes that run as part of a Deadline Cloud job.

Deadline Cloud submitter

A Deadline Cloud submitter is a plugin for a DCC that allows users to easily submit jobs from within the DCC.

License endpoint

A license endpoint makes Deadline Cloud's usage-based licensing for third-party products available inside your VPC. This model is pay as you go, and you are charged for the number of hours and minutes that you use. License endpoints are not connected to farms and can be used independently.

Tags

A tag is a label that you can assign to an AWS resource. Each tag consists of a key and an optional value that you define. With tags, you can categorize your AWS resources in different ways, such as by purpose, owner, or environment.

Usage-based licensing (UBL)

Usage-based licensing (UBL) is an on-demand licensing model that is available for select third-party products. This model is pay as you go, and you are charged for the number of hours and minutes that you use.

Deadline Cloud Architecture Guidance

This topic provides guidance and best practices for designing and building reliable, secure, efficient, and cost-effective render farms for your workloads using Deadline Cloud. Using this guidance can help you build stable and efficient workloads, allowing you to focus on innovation, reduce costs, and improve your customer's experience.

This content is intended for chief technology officers (CTOs), architects, developers, and operations team members.

An end-to-end rendering workflow requires solutions at multiple layers of the process such as job generation, asset access, and job monitoring. Deadline Cloud offers multiple solutions for each layer of the rendering process. By selecting from Deadline Cloud's options in each layer, you can design a workflow that matches your use case.

For each layer, decide which approach is best for your use case. These scenarios are not strict definitions and are not the only way to use Deadline Cloud. Instead they form a high-level set of concepts to help you understand how Deadline Cloud might be able to fit into your business or workflow. You can separate Deadline Cloud workloads into the following layers: Job Source, Job Submission, Application Management, Application Licensing, Asset Access, Output Management, and Worker Infrastructure Management.

In general, you can mix-and-match any scenario in one layer with any other scenario in another layer, except specific combinations which are specified below.

Job Source



Interactive Workflow



Automated Workflow

Job Submission



Integrated Submitter



Custom Job Definition

Application Management



Conda Application Management



Custom Application Management

Application Licensing



Deadline Cloud UBL



Custom Licensing

Asset Access



Job Attachments



Custom Storage

Job Monitoring



Deadline Cloud monitor



Custom Monitor Application



Automated Monitoring Solution

Worker Infrastructure



Job source

The job source is the access point where new jobs will enter the system to be rendered by Deadline Cloud. At a high-level, there are two primary sources of jobs: human interactivity and automated computer systems.

Interactive workflow

In this scenario, an artist or other creative role is the primary generator of work to be processed in the Deadline Cloud farm. Usually the output from these jobs are a primary artifact for the larger project or team. They perform their work using software such as an industry standard digital content creation (DCC) tool. They are manually submitting jobs to the Deadline Cloud farm and are viewing the outputs afterwards to review. The workstation itself is not managed by AWS.

In most cases, these artists use Deadline Cloud integrated submitters and the Deadline Cloud monitor in the Workload Application and Monitoring layers.

Automated workflow

In this scenario, a programmatic system owned by the customer is the primary generator of jobs in the Deadline Cloud farm. The jobs could be asset generation in a retail pipeline, like a turntable video generated from a 3D model or scan. Another example is automated compositing of broadcast graphics and player cards for sports. The theme of this scenario is an individual isn't manually submitting each job to Deadline Cloud, but instead the job is generated as part of a larger system.

With automated jobs, it is less common for Deadline Cloud integrated submitters and the Deadline Cloud monitor to be used. Often the job definitions will be custom application development written by you and job outputs will automatically flow into a Digital Asset Management (DAM) system or Media Asset Management (MAM) system for approval and distribution.

Job submission

Jobs are submitted to Deadline Cloud using [OpenJobDescription](#) templates on the GitHub website. OpenJobDescription is a flexible open specification for defining batch processing jobs that are portable between different scheduling system deployments. The Job definition file describes the parameters of the job, the steps of the job, how a step is parameterized based upon the job inputs, as well as the actual script that will run on a Worker to perform the processing. The idea of

Workload Submission is how these job definitions are created, who creates them and how they are submitted.

Integrated submitter with DCC

A Deadline Cloud integrated submitter is a piece of software that ties together Deadline Cloud with an industry standard DCC or software package. The integrated submitter determines how to transform the data and configuration for a render, composite or other workload into a job template, something that can be understood by Deadline Cloud. Many integrated submitters are created and maintained by the Deadline Cloud team or the creator of the software package, but if one does not already exist for the desired application then you can create and maintain your own submitter. There is a finite set of DCCs that are supported by the Deadline Cloud team.

Interactive workflows usually involve integrated submitters, but not always. For templated, automated workflows, a common workflow is for an artist to setup a template job in their DCC and perform a one-time export of the **job bundle**. This job bundle defines how to run that particular kind of job on Deadline Cloud in a parameterized manner. This job bundle can be integrated into the Automated Workflow scenario for automation purposes.

Custom job definition

For custom applications and workflows, it is possible to fully control how these job definitions are created and submitted to Deadline Cloud. For example, an e-commerce site might ask sellers to upload 3D models of the object they are selling. After this upload, the e-commerce platform could dynamically generate a job definition to submit to Deadline Cloud to automatically generate a turntable animation on a common background using common lighting to match the other 3D objects available on the site. During development of the ecommerce platform, a software developer would create a job definition, embed it into the ecommerce platform with parameters eventually provided by the sellers, and code the platform to submit this job during the platforms product upload workflow.

Deadline Cloud provides a number of sample job definitions in the [samples repository](#) on the GitHub website.

Application management

After a job is submitted to Deadline Cloud and assigned to a worker, the script from the job definition is executed on the worker. In most cases, this script will invoke an application to perform

the actual processing, such as a renderer, composite, encode, filtering or any other of a number of compute-intensive tasks. Application management is the concept of ensuring the necessary version of the required software is available to the workers.

You can manage applications using any package management system you like, but Deadline Cloud provides a number of tools to easily enable the use of conda packages. Conda is an open-source, cross-platform, language agnostic package manager and environment management system. For more information, see [conda](#) on the Anaconda website.

Deadline Cloud-managed conda channel for service-managed fleets (SMF)

When using service-managed fleets, a Deadline Cloud-managed conda channel is automatically setup and configured for use by your jobs. The Deadline Cloud service provides a number of partner DCC applications and renders in this conda channel. For more information see [Create a queue environment](#) in the Deadline Cloud user guide. These packages are automatically kept up to date by the Deadline Cloud service and require no maintenance from you. This conda channel is only available when using service-managed fleets and are not available when using customer-managed fleets.

Self-managed conda channel

If you are not able to use the Deadline Cloud-managed conda channel, you must determine how to install, patch and otherwise manage applications on your Deadline Cloud fleet. One option is to create a conda channel that you setup and maintain. This approach most closely interoperates with the Deadline Cloud-managed conda channel. For example, you can use a DCC from the Deadline Cloud-managed conda channel but bring your own package that contains a specific DCC plugin. For more information about this process, see [Create a conda channel using S3](#).

Custom application management

For application management, the requirement from Deadline Cloud is that the application is available in the PATH when the job script is executed on the worker.

If you already build and maintain Rez packages, you can use a queue environment to install the applications from Rez repositories. For an example queue environment, see the [deadline-cloud-samples repository](#) on the GitHub website.

If you already manage applications on a customer-managed fleets with long-lived workers or in system images, then no queue environment is required for application management. Ensure the application appears on the job user's path and submit the job.

Application licensing

Many workloads commonly ran on Deadline Cloud require software licensing from the software vendor. These applications are often licensed by seat, CPU, or host. It is your responsibility to ensure that your usage of 3rd party software on Deadline Cloud abides by the 3rd party licensing agreement. If you are using open-source software, custom software, or otherwise license-free software, then configuring this layer is not required. Deadline Cloud only supports render licensing and does not support workstation licensing.

Service-managed fleets and usage-based licensing

When using Deadline Cloud service-managed fleets, usage-based licensing (UBL) is automatically configured for supported software. Jobs run on service-managed fleets automatically have environment variables set for supported applications to direct them to use the Deadline Cloud license servers. When using Deadline Cloud UBL, you are only charged for the number of hours you use the licensed application.

Customer-managed fleets and usage-based licensing

Deadline Cloud usage-based licensing (UBL) is also available when not using service-managed fleets. In this scenario, you will setup Deadline Cloud license endpoints which provide IP addresses in your selected VPC subnets that provide access to Deadline Cloud license servers. After you configure the appropriate software-specific environment variables on your workers and configure network connectivity from the workers to those license endpoint IP addresses, the workers are able to check-out and check-in licenses for supported software. You are charged per hour for licenses the same as when using UBL in service-managed fleets.

Custom licensing

You might use an application that is not supported by Deadline Cloud UBL or you might have preexisting licenses that are still valid. In this scenario, you are responsible for configuring the network path from your workers (customer- or service-managed) to the license servers. For more information about custom licensing, see [Connect service-managed fleets to a custom license server](#).

Asset access

After a job is submitted to a worker and the application is configured, the worker must be configured to access the asset data required for the job. The assets could be 3D data, texture data, animation data, video frames or any other sort of data used in your job.

Start with thinking about where your data is currently stored. The data might be on the workstation hard drive, a user collaboration tool, source control, a shared filesystem on-premises or in the cloud, Amazon S3 or any number of other locations.

Next, consider what is necessary for a worker to access this data. Is this data only made available on your corporate network? What identity or credentials is required to access the data? Is the data source scaled to support the job with the number of workers you expect to process the job?

Job attachments

The easiest to start with mechanism for asset access is Deadline Cloud job attachments. When a job is submitted using job attachments, the data required by the job is uploaded to an Amazon S3 bucket along with a manifest file specifying which files the job requires. With job attachments, no complicated networking or shared storage setup is required. Files are only uploaded once, so subsequent uploads complete more quickly. After a worker has finished processing a job, the output data is uploaded to Amazon S3 so it can be downloaded by the artist or another client. Job attachments scale for fleets of any size and is simple and fast to onboard to and use.

Job attachments is not the best tool for all situations. If your data is already on AWS, then job attachments add an additional copy of your data, including associated transfer time and storage costs. Job attachments require that the job can fully specify the data it requires at submission time, so that the data can be uploaded.

To use job attachments, your Deadline Cloud queue must have an associated job attachments bucket and the queue role must be used to provide access to that bucket. By default, Deadline Cloud integrated submitters all support job attachments. If you are not using a Deadline Cloud integrated submitter, job attachments can be used with your custom software by integrating the [Deadline Cloud python library](#) on the GitHub website.

Custom storage access

If you do not use job attachments, you are responsible for ensuring workers have access to the data required for jobs. Deadline Cloud provides a number of tools to support this and to keep

jobs portable. You might want to use a custom storage solution when you already have shared network storage for artists and workers, you prefer to use an external service like LucidLink, or other reasons.

Use [storage profiles](#) to model file systems on your workstation and worker hosts. Each storage profile describes the operating system and file system layout of one of your system configurations. Using storage profiles, when an artist using a windows workstation submits a job that is processed by a Linux worker, Deadline Cloud ensures path mapping occurs so the worker can access the data storage you have configured.

When using Deadline Cloud service-managed fleets, [host configuration scripts](#) and [VPC resource endpoints](#) enable workers to directly mount and access shared storage or other services available in your VPC.

Perforce source control

Many studios keep their project files in Perforce. When your artists already submit their work to a Perforce server as changelists, workers can sync a Perforce client workspace directly from that server as an alternative to job attachments. Your jobs then render the same file revisions your artists submitted. The Deadline Cloud integration for Unreal Engine includes Perforce sync support. For more information, see [Creating a Perforce render job](#) in the *Deadline Cloud User Guide*.

Two practices keep Perforce sync fast at scale:

Sync deltas onto persistent worker storage

Persist the client workspace between jobs so that each sync downloads only the file revisions that changed since the last job that used that workspace, instead of the full project. On service-managed fleets, enable [persistent storage](#) so volumes carrying the synced workspace are reused across workers.

Cache Perforce data close to your fleet

Sync speed depends on the network path between your workers and your Perforce server. That path can have low throughput or high latency, for example when the commit server is on-premises or in a distant AWS Region. A Perforce edge server in the same Region as your fleet acts as a cache for depot data. Deploy it using the Perforce commit-edge topology and Perforce-native replication. Each asset revision then crosses the slower link once into the edge server, and workers sync from the edge over the regional network. Workers on service-managed fleets can reach the edge server through [VPC resource endpoints](#). An edge server comes with operational responsibilities, including Perforce licensing, monitoring of replication

lag, and backups. A job that targets a specific changelist can start only after the edge server has replicated that changelist.

Job monitoring and output management

After jobs submitted to Deadline Cloud are successfully completed, a person or process will download the job output to use in the business workflow outside of Deadline Cloud. After job failure, job logs and monitoring information help diagnose issues.

Deadline Cloud monitor

The Deadline Cloud monitor application is available on the web and for desktop. This solution is best suited for studios using interactive workflows for many DCCs using job attachments for storage. The monitor only supports you when using IAM Identity Center. IAM Identity Center is a Workforce Identity product, not a consumer identity (B2C) solution so it is not appropriate for many B2C scenarios.

Custom monitor application

If you wish to customize the monitoring experience of your users, you are building a B2C product, or building a highly specialized system using Deadline Cloud you opt to create a custom monitoring application. You can use the [AWS Deadline Cloud API](#) to create this custom application, combining the context of your overall workflow with Deadline Cloud concepts. For example, your B2C product might have its own project concept which users setup and your application can nest Deadline Cloud jobs in the same interface.

Automated monitoring solution

In some scenarios, no dedicated monitoring application is needed for Deadline Cloud. This scenario is common in automated workflows where Deadline Cloud is used to automatically render assets in a pipeline, such as broadcast graphics for sports or news. In this scenario, the Deadline Cloud API and EventBridge events are used to integrate with an external Media Asset Management system for approvals and moving data to the next step in the process.

Worker infrastructure management

Deadline Cloud fleets are a grouping of servers (workers) that are able to process jobs submitted to a Deadline Cloud queue and are the core infrastructure of any Deadline Cloud farm.

Service-managed fleets

In a service-managed fleet, Deadline Cloud takes responsibility for the worker hosts, operating system, networking, patching, auto scaling, and other factors of running a render farm. You specify the minimum and maximum number of workers you want, along with the system specifications required for your application and Deadline Cloud does the rest. Service-managed fleets are the only fleet option that can use Deadline Cloud-managed conda channels to easily manage industry DCC applications. Additionally, Deadline Cloud UBL is automatically configured with service-managed fleets. Wait and Save fleets for lower cost, delay-tolerant workloads is only available using service-managed fleets.

Customer-managed fleets

You use customer-managed fleets when you need more control over the worker hosts and their environment. Customer-managed fleets are best suited when using Deadline Cloud on-premises. To learn more, see [Create and use Deadline Cloud customer-managed fleets](#).

Example architectures

Traditional production studio

The traditional production studio requires significant compute, storage, and networking infrastructure that can span multiple physical locations to service rendering workloads. Each individual software package and vendor has unique hardware, software, networking, and licensing requirements that must be met while resolving versioning, compatibility, and resource conflicts.

It is common to have separate infrastructure requirements for artist workstations, render nodes, network storage, license servers, job queuing systems, monitoring tools, and asset management. Studios typically need to maintain multiple versions of DCC tools, renderers, plugins, and custom tools while managing complex licensing arrangements across their render farm. Your studio infrastructure becomes more complicated when you consider development, quality assurance, and production environments.

A typical Deadline Cloud deployment using service-managed options solves or reduces many of these challenges through:

- Interactive workflow job submission through integrated DCC submitters

- Application management via Deadline Cloud-managed conda channels
- Usage-based licensing automatically configured for supported software
- Asset management through job attachments
- Monitoring via the Deadline Cloud monitor application
- Infrastructure management through service-managed fleets

With this approach, artists can submit jobs directly from their familiar DCC tools to a scalable cloud render farm without managing complex infrastructure. The service automatically handles software deployment, licensing, data transfer, and infrastructure scaling. Artists can monitor their jobs through a web interface or desktop application, and outputs are automatically stored in Amazon S3 for easy access.

With this configuration, studios can create development and production environments in minutes, only pay for the compute and licensing they use, and focus on creative work rather than infrastructure management. The service-managed approach provides the fastest path to adopting cloud rendering while maintaining familiar workflows for artists.



Studio in the Cloud

Modern visual effects and animation studios are increasingly moving their entire pipeline to the cloud, including artist workstations. This approach eliminates the need for on-premises infrastructure, enables global collaboration, and scales for both interactive work and rendering. However, it also introduces new challenges in managing cloud resources, ensuring low-latency access to data, and integrating cloud-based workstations with render farms.

A typical cloud-native studio requires a unified approach to managing cloud workstations, shared storage, rendering infrastructure, and software deployment across all these components. Traditional approaches often resulted in complex, manually-managed systems that struggled to balance performance, cost, and flexibility.

A Deadline Cloud deployment for a cloud-native studio can be implemented using:

- Interactive workflow job submission through integrated DCC submitters on cloud workstations
- Application management via Deadline Cloud-managed conda channels render nodes
- Usage-based licensing automatically configured for supported software
- Custom storage access using FSx for Windows File Server for shared project data
- Monitoring via the Deadline Cloud monitor application
- Infrastructure management using service-managed fleets

This approach allows artists to work on cloud-based workstations with direct access to high-performance shared storage and submit jobs to the Deadline Cloud farm. The studio can manage software deployment across both workstations and render nodes using the same conda channels, ensuring consistency and reducing maintenance overhead.

Key benefits of this configuration include:

- Global collaboration with artists able to access workstations from anywhere
- Consistent software environments across workstations and render nodes
- High-performance shared storage accessible to both workstations and render nodes
- Flexible scaling of both interactive and batch compute resources
- Centralized management of all studio infrastructure in the cloud

Storage configuration in this scenario typically involves:

- FSx for Windows File Server for project data, accessible by both cloud workstations and Deadline Cloud workers
- Storage profiles in Deadline Cloud to manage path mapping between workstations and render nodes
- Direct mounting of FSx shares on Deadline Cloud workers using VPC resource endpoints and host configuration scripts

This cloud-native approach allows studios to eliminate on-premises infrastructure, enabling rapid scaling for projects of any size while maintaining familiar artist workflows. It provides the flexibility to use a mix of service-managed and customer-managed resources, optimizing for both ease of management and specific performance requirements.

By using cloud workstations alongside Deadline Cloud, studios can achieve a fully integrated, globally accessible production pipeline that scales from small teams to large productions.

ECommerce Automation

The modern ecommerce platform requires automated asset generation at scale to provide rich product visualization across millions of items. Traditional approaches would require significant infrastructure investment to process large volumes of 3D models into standardized product media, often resulting in either under-provisioned systems that create processing backlogs or over-provisioned systems with idle capacity.

A typical automated ecommerce workflow needs to handle product upload processing, 3D model validation, render farm management, output processing, and integration with product information systems. Managing these workflows traditionally requires coordinating multiple rendering applications, compute resources, and data processing pipelines while ensuring consistent quality and maintaining cost efficiency at scale.

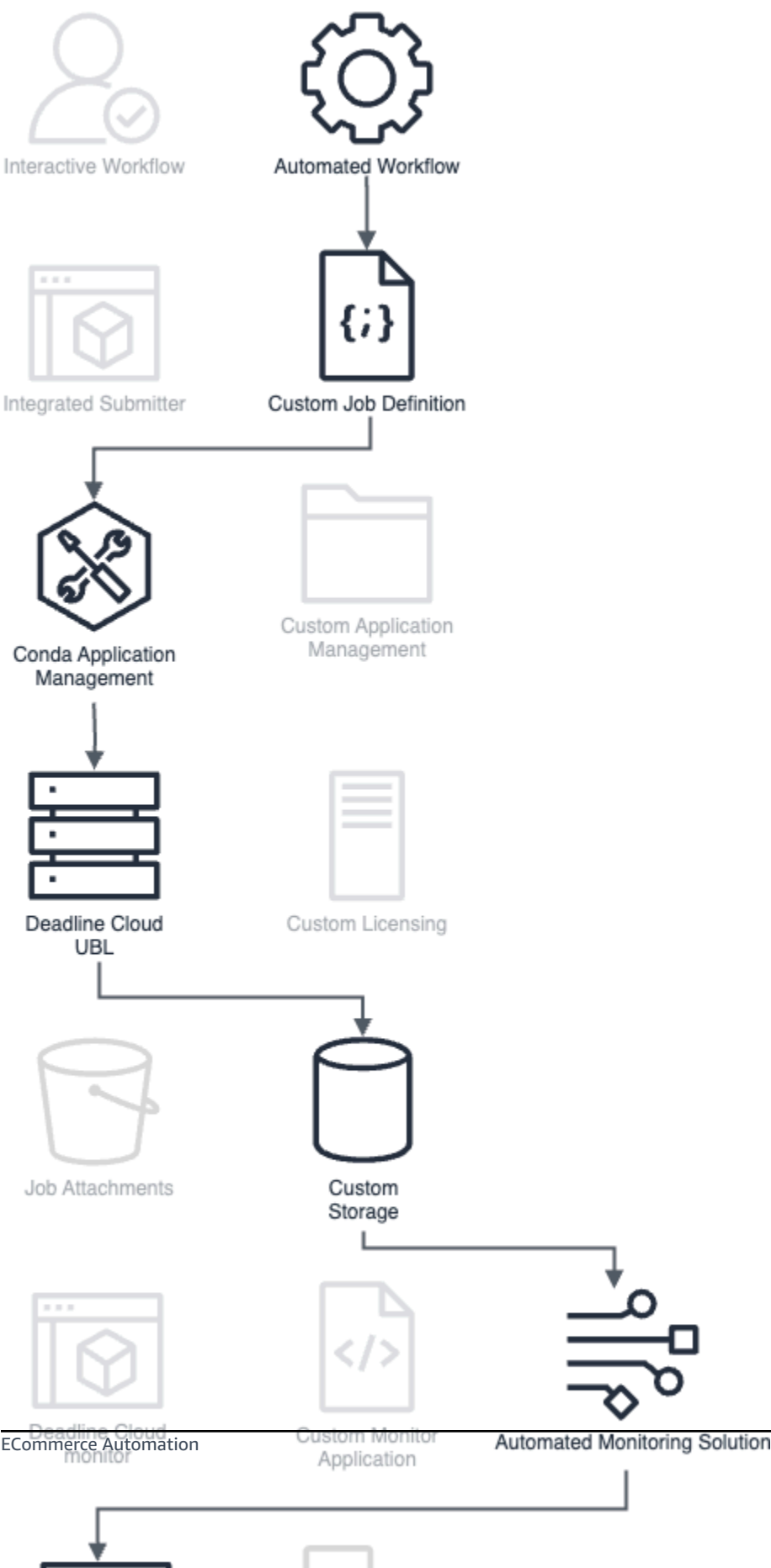
A Deadline Cloud deployment for ecommerce automation can be implemented using:

- Automated workflow job submission through custom API integration in the existing ecommerce ingestion application
- Custom job definitions tailored to standardized product visualization
- Application management via Deadline Cloud-managed conda channels
- Usage-based licensing automatically configured for supported software
- Direct Amazon S3 integration for asset management

- Custom monitoring application integrated with existing product management systems
- Service-managed fleets for elastic scaling

This approach enables processing of thousands of products per day, automatically generating standardized product visualizations like turntable animations. The service-managed infrastructure automatically scales to meet variable demand while maintaining cost efficiency through worker reuse and optimized application deployment.

eCommerce



Whitelabel/OEM/B2C Customer

Traditional digital content creation (DCC) software typically requires users to maintain their own rendering infrastructure or process renders locally on their workstation, leading to either significant hardware investments or long wait times that interrupt creative workflows. For software vendors, providing cloud rendering capabilities traditionally required building and maintaining complex infrastructure and billing systems.

A Deadline Cloud deployment integrated into B2C software enables cloud rendering directly within the user's familiar interface. This integration combines:

- Interactive workflow job submission embedded within the DCC application
- Deadline Cloud-managed conda channels for render application deployment
- Usage-based licensing configured automatically
- Asset management through job attachments with vendor-managed storage
- Custom monitoring integrated directly in the DCC interface
- Service-managed fleets shared across users

This approach allows end users to submit renders to the cloud with a single click from within their software, without managing accounts, infrastructure, or complex setup. The software vendor maintains a multi-tenant environment where:

- Users authenticate through their existing software credentials
- Jobs are automatically routed to dedicated per-user queues
- Assets are securely isolated using IAM-controlled storage prefixes
- Billing is handled through the vendor's existing systems
- Job status and outputs are streamed directly back to the user's application

The shared fleet approach ensures optimal performance by maintaining a warm pool of workers, minimizing startup times while maximizing resource utilization across the user base. This configuration allows software vendors to offer cloud rendering as a built-in product feature rather than a separate service requiring additional setup or accounts.

End users benefit from:

- One-click submission from their familiar interface

- Pay-as-you-go pricing without infrastructure management
- Fast job startup times through shared infrastructure
- Automatic download and organization of completed renders
- Consistent experience across all platforms

This integration pattern enables software vendors to provide enterprise-grade rendering capabilities to their entire user base while maintaining a simple, consumer-friendly experience that feels native to their application.

Whitelabel B2C Customer



Hybrid render farm

A studio that already runs a render farm on-premises can add cloud capacity to process peak workloads without buying more hardware. In a hybrid farm, your on-premises workers run as a customer-managed fleet, your cloud workers run as a service-managed fleet or a customer-managed fleet of Amazon EC2 instances, and both fleets share a queue. Because the setup differs across fleet types, you plan for job distribution, application environments, licensing, and asset access so that one queue works everywhere. For more information, see [Extend your on-premises render farm to the cloud](#).

What is a Deadline Cloud workload

With AWS Deadline Cloud, you can submit jobs to run your applications in the cloud and process data for the production of content or insights important to your business. Deadline Cloud uses Open Job Description (OpenJD) as the syntax for job templates, a specification designed for the needs of visual compute pipelines but applicable to many other use cases. For more information, see the [OpenJD specification](#) on the GitHub website. Some example workloads include computer graphics rendering, physics simulation, and photogrammetry.

Workloads scale from simple job bundles that users submit to a queue with either the CLI or an automatically generated GUI, to integrated submitter plugins that dynamically generate a job bundle for an application-defined workload.

How workloads arise from production

To understand workloads in production contexts and how to support them with Deadline Cloud, consider how they come to be. Production may involve creating visual effects, animation, games, product catalog imagery, 3D reconstructions for building information modeling (BIM), and more. This content is typically created by a team of artistic or technical specialists running a variety of software applications and custom scripting. Members of the team pass data between each other using a production pipeline. Many tasks performed by the pipeline involve intensive computations that would take days if run on a user's workstation.

Some examples of tasks in these production pipelines include:

- Using a photogrammetry application to process photographs taken of a movie set to reconstruct a textured digital mesh.
- Running a particle simulation in a 3D scene to add layers of detail to an explosion visual effect for a television show.
- Cooking data for a game level into the form needed for external release and applying optimization and compression settings.
- Rendering a set of images for a product catalog including variations in color, background, and lighting.
- Running a custom-developed script on a 3D model to apply a look that was custom-built and approved by a movie director.

These tasks involve many parameters to adjust to get an artistic result or to fine tune the output quality. Often there is a GUI to select those parameter values with a button or menu to run the process locally within the application. When a user runs the process, the application and possibly the host computer itself cannot be used to perform other operations because it uses the application state in memory and might consume all of the host computer's CPU and memory resources.

In many cases the process is quick. During the course of production, the speed of the process slows down when the requirements for quality and complexity go up. A character test that took 30 seconds during development can easily turn into 3 hours when it is applied to the final production character. Through this progression, a workload that began life inside a GUI can grow too large to fit. Porting it to Deadline Cloud can boost the productivity of users running these processes because they get back full control of their workstation and can keep track of more iterations from the Deadline Cloud monitor.

There are two levels of support to aim for when developing support for a workload in Deadline Cloud:

- Offloading the workload from the user workstation to a Deadline Cloud farm with no parallelism or speed-up. This approach might under-utilize the available compute resources in the farm, but the ability to shift long operations to a batch processing system enables users to get more done with their own workstation.
- Optimizing the parallelism of the workload so that it utilizes the Deadline Cloud farm's horizontal scale to complete quickly.

There are times that it is obvious how to make a workload run in parallel. For example, each frame of a computer graphics render can be done independently. It's important not to get stuck on this parallelism, however. Instead, understand that offloading a long-running workload to Deadline Cloud provides significant benefits, even when there is no obvious way to split the workload up.

The ingredients of a workload

To specify a Deadline Cloud workload, implement a job bundle that users submit to a queue with the Deadline Cloud CLI. Much of the work in creating a job bundle is to write the job template, but there are more factors like how to provide the applications that the workload requires. For more information about the CLI, see the [Deadline Cloud CLI repository](#) on the GitHub website. Here are the essential things to consider when defining a workload for Deadline Cloud:

- **The application to run.** The job must be able to launch application processes, and therefore needs an installation of the application available as well as any licensing the application uses, such as access to a floating license server. The installation and licensing are typically part of the farm configuration, and not embedded in the job bundle itself.
 - [Configure jobs using queue environments](#)
 - [Connect customer-managed fleets to a license endpoint](#)
- **Job parameter definitions.** The user experience of submitting the job is affected greatly by the parameters it provides. Example parameters include data files, directories, and application configuration.
 - [Parameter values elements for job bundles](#)
- **File data flow.** When a job runs, it reads input from files provided by the user, then writes its output as new files. To work with the job attachments and path mapping features, the job must specify the paths of the directories or specific files for these inputs and outputs.
 - [Using files in your jobs](#)
- **The step script.** The step script runs the application binary with the right command-line options to apply the provided job parameters. It also handles details like path mapping if the workload data files include absolute instead of relative path references.
 - [Job template elements for job bundles](#)

Workload portability

A workload is portable when it can run in multiple different systems without changing it each time you submit a job. For example, it might run on different render farms that have different shared file systems mounted, or on different operating systems like Linux or Windows. When you implement a portable job bundle, it's easier for users to run the job on their specific farm, or to adapt it for other use cases.

Here are some ways you can make your job bundle portable.

- Fully specify the input data files needed by a workload, using PATH job parameters and asset references in the job bundle. This approach makes the job portable to farms based on shared file systems and to farms that make copies of the input data, like the Deadline Cloud job attachments feature.
- Make file path references for the input files of the job relocatable and usable on different operating systems. For example when users submit jobs from Windows workstations to run on a Linux fleet.

- Use relative file path references, so if the directory containing them is moved to a different location, references still resolve. For information about Blender's support for relative paths, see [Relative paths](#) on the Blender website.
- If you can't use relative paths, support OpenJD path mapping metadata and translate the absolute paths according to how Deadline Cloud provides the files to the job. For more information, see [path mapping metadata](#) on the GitHub website.
- Implement commands in a job using portable scripts. Python and bash are two examples of scripting languages that can be used this way. You should consider providing them both on all the worker hosts of your fleets.
- Use the script interpreter binary, like python or bash, with the script file name as an argument. This approach works on all operating systems including Windows, compared to using a script file with its execute bit set on Linux.
- Write portable bash scripts by applying these practices:
 - Expand template path parameters in single quotes to handle paths with spaces and Windows path separators.
 - When running on Windows, watch for issues related to MinGW automatic path translation. For example, it transforms an AWS CLI command like `aws logs tail /aws/deadline/...` into a command similar to `aws logs tail "C:/Program Files/Git/aws/deadline/..."` and won't tail a log correctly. Set the variable `MSYS_NO_PATHCONV=1` to turn this behavior off.
 - In most cases, the same code works on all operating systems. When the code needs to be different use an `if/else` construct to handle the cases.

```
if [[ "$(uname)" == MINGW* || "$(uname -s)" == MSYS_NT* ]]; then
    : # Code for Windows
elif [[ "$(uname)" == Darwin ]]; then
    : # Code for MacOS
else
    : # Code for Linux and other operating systems
fi
```

- You can write portable Python scripts using `pathlib` to handle file system path differences and avoid operating-specific features. For information about which Python library features are Linux-specific, see the [signal library documentation](#) on the Python website. Linux-specific feature support is marked as "Availability: Linux."
- Use job parameters to specify application requirements. Use consistent conventions that the farm administrator can apply in [queue environments](#).

- For example, you can use the CondaPackages and/or RezPackages parameters in your job, with a default parameter value that lists the application package names and versions the job requires. Then, you can use one of the [sample conda or Rez queue environments](#) on the GitHub website to provide a virtual environment for the job.

Migrate from Deadline 10 to AWS Deadline Cloud

A Deadline 10 farm consists of a repository file share, a MongoDB database, and worker hosts that select their own jobs. Pools and groups organize the farm, limits control shared resources, and event plugins and job scripts extend it. AWS Deadline Cloud (Deadline Cloud) has equivalents for most of these concepts, but several of them moved, changed names, or split into more focused features.

Use the tables on this page to find the Deadline Cloud equivalent of each Deadline 10 concept, then follow the link for details.

Some equivalents depend on the fleet type. With a service-managed fleet (SMF), Deadline Cloud provisions, scales, and operates the workers. With a customer-managed fleet (CMF), you operate the worker hosts—in the cloud or on premises—and the service schedules work to them. For more information, see [Deadline Cloud fleets](#) in the *Deadline Cloud User Guide*.

Farm components you no longer operate

Deadline Cloud replaces the server components of a Deadline 10 farm with a managed service. There is no repository file share to mount, no database to back up, and no maintenance server to run. Your farm's state lives in the service, and every application communicates with it through authenticated AWS API endpoints over TLS. If your pipeline is built around a shared file system, you can keep using it for application installs and production data—the service doesn't require one.

Deadline 10 component	In Deadline Cloud
Repository (shared file system)	Nothing to operate. Job files move through job attachments in Amazon Simple Storage Service (Amazon S3). Job bundles and queue environments replace the plugins, scripts, and settings stored in the repository.
Database (MongoDB)	Nothing to operate. The service stores farm state, which you query through the Deadline Cloud API.
Remote Connection Server (RCS)	Nothing to operate. All clients connect to the service API endpoints using AWS Identity and Access Management (IAM) credentials over TLS.

Deadline 10 component	In Deadline Cloud
Pulse (house cleaning, pending job scan)	The service performs scheduling and maintenance. On a CMF, you're responsible for worker host health. A health check template is available in the CMF farm templates sample on the GitHub website.
Worker application and Launcher	The Deadline Cloud worker agent. An SMF runs it for you; on a CMF you install it on your own hosts.
Monitor (desktop application)	The AWS Deadline Cloud monitor , available as a desktop application and in the browser.
<code>deadlinecommand</code> and the REST API	The deadline CLI on the GitHub website, the Deadline Cloud API , and the AWS SDKs.

Map Deadline 10 concepts to Deadline Cloud

The following table maps the Deadline 10 concepts you use to organize and automate a farm to their Deadline Cloud equivalents.

Deadline 10	Deadline Cloud	For more information
Pools and groups	Queue–fleet associations select which fleets serve a queue. Host requirements and fleet capabilities select workers within those fleets.	the section called “Route jobs to specific workers (pools and groups)”
Job scheduling order (pool, priority, weighted, balanced)	Queue scheduling configurations: priority first-in-first-out, priority balanced, and weighted balanced.	the section called “Scheduling configurations”
Limits (license limits, resource limits, machine limits)	Resource limits, acquired per task and associated with queues.	the section called “Create resource limits for jobs”

Deadline 10	Deadline Cloud	For more information
Job dependencies and pending jobs	Step dependencies within a job. There is no built-in job-to-job dependency. To chain separate jobs, you build your own automation from Amazon EventBridge (EventBridge) events.	the section called “Step dependencies”, <i>Hooks and integration points</i>
Frame list and ChunkSize	Task parameter ranges and task chunking in the job template.	the section called “Task chunking”
Auxiliary files	Job attachments, stored in an Amazon S3 bucket in your account.	the section called “Job attachments”
Path mapping rules	Storage profiles describe each machine's file system locations, and the service derives path mapping rules from them.	the section called “Storage profiles and path mapping”
Application installation (on each host or on shared file storage)	On an SMF, many applications are available as conda packages, and you can package your own. On a CMF, you typically install applications on each host or on shared file storage, as in Deadline 10.	Deploy and configure custom software on workers
Application plugins and job files	Open Job Description (OpenJD) job templates, packaged as job bundles. Application adaptors wrap interactive applications such as Maya and Nuke.	the section called “Job bundles”, <i>adaptor runtime</i> on the GitHub website
Integrated submitters	Deadline Cloud submitters for supported digital content creation (DCC) applications.	Supported software

Deadline 10	Deadline Cloud	For more information
Event plugins, and pre and post job and task scripts	Integration points: submitter hooks, queue environments, job and step environments, dependent steps, and EventBridge events.	<i>Hooks and integration points</i>
AWS Portal	An SMF, which includes automatic scaling, usage-based licensing, and job attachments for asset transfer.	<i>Service-managed fleets</i>
Spot Event Plugin	An SMF scales automatically. If you need a CMF, use Amazon Elastic Compute Cloud (Amazon EC2) Auto Scaling driven by fleet size recommendation events.	<i>Service-managed fleets, the section called "Create fleet infrastructure"</i>
Usage-based licensing (UBL) and the License Forwarder	Usage-based licensing, built into an SMF and available to a CMF through license endpoints.	<i>Using software licenses</i>
User management, Power User mode, and Secrets Management	IAM policies, AWS IAM Identity Center users and groups, and per-resource access levels.	<i>Security</i>
Farm statistics and reports	Amazon CloudWatch (CloudWatch) metrics and logs, and session data aggregation for usage queries.	<i>Monitoring, Querying session statistics aggregated data</i>
Notifications (email, SQS)	No built-in user notifications. Deadline Cloud delivers EventBridge events to your account's event bus, and you build your own delivery, such as email through Amazon SNS.	<i>the section called "Managing events using EventBridge"</i>
Idle workstation rendering	CMF workers running on workstations alongside your existing farm.	<i>Hybrid rendering</i>

Route jobs to specific workers (pools and groups)

In Deadline 10, you assign workers to pools and groups, and each job targets a pool and a group. Deadline Cloud splits those responsibilities between two resources:

- **Queues organize work and control access.** A queue holds jobs, sets their scheduling order, and defines who can submit to it and view it. Use queues the way you used pools: to separate shows, departments, or clients.
- **Fleets and host requirements select workers.** A fleet declares the capabilities that its workers guarantee: hardware minimums and custom attributes such as installed software or a site label. Each step in a job states its host requirements. The service routes the step to a compatible fleet associated with the queue. Use fleets and host requirements the way you used groups: to make sure work only runs on appropriately equipped machines.

To dedicate workers to sensitive work, such as content under a security or privacy restriction, create a separate fleet for those workers. Associate that fleet only with the queues that are allowed to use it. The association is the enforcement point: AWS IAM Identity Center group memberships control who can view and manage a resource, and host requirements route work at the submitter's request, but neither keeps jobs off a fleet.

For more information about the scheduling mechanics and the access control distinction, see [the section called “Determine fleet compatibility”](#). For an end-to-end restricted fleet setup, see [Organize your farms, queues, and fleets](#) in the *Deadline Cloud User Guide*.

Features without a direct equivalent

Some Deadline 10 features don't have a named equivalent in Deadline Cloud. In most cases, you can express the same outcome with a job bundle:

- **Draft and Quick Draft** – To encode rendered frames into a movie, add a dependent step that runs an encoding tool. For a worked example, see [the section called “FFmpeg movie from job output”](#).
- **Jigsaw and tile rendering** – Split a frame into tiles with a parameterized step and assemble the result in a dependent step. For a worked example, see [the section called “Tile rendering”](#).
- **Power management** – An SMF starts and stops workers automatically. On a CMF, you're responsible for starting and stopping worker hosts—by scaling your Amazon EC2 fleet in the cloud, or with your own tooling on premises.

Choose a starting point

Use the following guidance to choose where to start your migration:

- To try Deadline Cloud with a small workload before committing, create a farm with an SMF and submit from a supported DCC application. See [Getting started](#).
- To keep rendering on hardware you already own, connect your render nodes or workstations to Deadline Cloud as a CMF. See [Hybrid rendering](#).
- To replace AWS Portal bursting, use an SMF. Automatic scaling, usage-based licensing, and asset transfer are built in. See [Service-managed fleets](#).
- To port your event plugins and job scripts, start with the integration point map. See [Hooks and integration points](#).

Getting started with Deadline Cloud resources

To start creating custom solutions for AWS Deadline Cloud, you must set up your resources. These include a farm, at least one queue for the farm, and at least one worker fleet to service the queue. You can create your resources using the Deadline Cloud console, or you can use the AWS Command Line Interface.

In this tutorial, you will use AWS CloudShell to create a simple developer farm and run the worker agent. You can then submit and run a simple job with parameters and attachments, add a service managed fleet, and clean up your farm resources when you're done.

The following sections introduce you to the different features of Deadline Cloud, and how they function and work together. Following these steps is useful for developing and testing new workloads and customizations.

For instructions to set up your farm using the console, see [Getting started](#) in the *Deadline Cloud User Guide*.

Topics

- [Create a Deadline Cloud farm](#)
- [Run the Deadline Cloud worker agent](#)
- [Submit with Deadline Cloud](#)
- [Submit jobs with job attachments in Deadline Cloud](#)
- [Add a service-managed fleet to your developer farm in Deadline Cloud](#)
- [Clean up your farm resources in Deadline Cloud](#)

Create a Deadline Cloud farm

To create your developer farm and queue resources in AWS Deadline Cloud, use the AWS Command Line Interface (AWS CLI), as shown in the following procedure. You will also create an AWS Identity and Access Management (IAM) role and a customer-managed fleet (CMF) and associate the fleet with your queue. Then you can configure the AWS CLI and confirm that your farm is set up and working as specified.

You can use this farm to explore the features of Deadline Cloud, then develop and test new workloads, customizations, and pipeline integrations.

To create a farm

1. [Open an AWS CloudShell session](#). You'll use the CloudShell window to enter AWS Command Line Interface (AWS CLI) commands to run the examples in this tutorial. Keep the CloudShell window open as you proceed.
2. Create a name for your farm, and add that farm name to `~/.bashrc`. Storing the name makes it available in other terminal sessions.

```
echo "DEV_FARM_NAME=DeveloperFarm" >> ~/.bashrc
source ~/.bashrc
```

3. Create the farm resource, and add its farm ID to `~/.bashrc`.

```
aws deadline create-farm \
  --display-name "$DEV_FARM_NAME"

echo "DEV_FARM_ID=$(aws deadline list-farms \
  --query \"farms[?displayName=='$DEV_FARM_NAME'].farmId \
  | [0]\" --output text)" >> ~/.bashrc
source ~/.bashrc
```

4. Create the queue resource, and add its queue ID to `~/.bashrc`.

```
aws deadline create-queue \
  --farm-id $DEV_FARM_ID \
  --display-name "$DEV_FARM_NAME Queue" \
  --job-run-as-user '{"posix": {"user": "job-user", "group": "job-group"},
  "runAs": "QUEUE_CONFIGURED_USER"}'

echo "DEV_QUEUE_ID=$(aws deadline list-queues \
  --farm-id $DEV_FARM_ID \
  --query \"queues[?displayName=='$DEV_FARM_NAME Queue'].queueId \
  | [0]\" --output text)" >> ~/.bashrc
source ~/.bashrc
```

5. Create an IAM role for the fleet. This role provides worker hosts in your fleet with the necessary security credentials to run jobs from your queue.

```
aws iam create-role \
  --role-name "${DEV_FARM_NAME}FleetRole" \
  --assume-role-policy-document \
    '{
```

```

        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Principal": {
                    "Service": "credentials.deadline.amazonaws.com"
                },
                "Action": "sts:AssumeRole"
            }
        ]
    }'
aws iam put-role-policy \
  --role-name "${DEV_FARM_NAME}FleetRole" \
  --policy-name WorkerPermissions \
  --policy-document \
    '{
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Action": [
            "deadline:AssumeFleetRoleForWorker",
            "deadline:UpdateWorker",
            "deadline>DeleteWorker",
            "deadline:UpdateWorkerSchedule",
            "deadline:BatchGetJobEntity",
            "deadline:AssumeQueueRoleForWorker"
          ],
          "Resource": "*",
          "Condition": {
            "StringEquals": {
              "aws:PrincipalAccount": "${aws:ResourceAccount}"
            }
          }
        },
        {
          "Effect": "Allow",
          "Action": [
            "logs>CreateLogStream"
          ],
          "Resource": "arn:aws:logs:*:*:*:/aws/deadline/*",
          "Condition": {
            "StringEquals": {
              "aws:PrincipalAccount": "${aws:ResourceAccount}"
            }
          }
        }
      ]
    }'

```



```

        }
    },
    {
        "Effect": "Allow",
        "Action": [
            "logs:PutLogEvents",
            "logs:GetLogEvents"
        ],
        "Resource": "arn:aws:logs:*:*:*:/aws/deadline/*",
        "Condition": {
            "StringEquals": {
                "aws:PrincipalAccount": "${aws:ResourceAccount}"
            }
        }
    }
]
}'

```

6. Create the customer-managed fleet (CMF), and add its fleet ID to ~/.bashrc.

```

FLEET_ROLE_ARN="arn:aws:iam::$(aws sts get-caller-identity \
    --query "Account" --output text):role/${DEV_FARM_NAME}FleetRole"
aws deadline create-fleet \
    --farm-id $DEV_FARM_ID \
    --display-name "$DEV_FARM_NAME CMF" \
    --role-arn $FLEET_ROLE_ARN \
    --max-worker-count 5 \
    --configuration \
        '{
            "customerManaged": {
                "mode": "NO_SCALING",
                "workerCapabilities": {
                    "vCpuCount": {"min": 1},
                    "memoryMiB": {"min": 512},
                    "osFamily": "linux",
                    "cpuArchitectureType": "x86_64"
                }
            }
        }'

echo "DEV_CMF_ID=$(aws deadline list-fleets \
    --farm-id \ $DEV_FARM_ID \
    --query \"fleets[?displayName=='\ $DEV_FARM_NAME CMF'].fleetId \

```

```
| [0]" --output text)" >> ~/.bashrc  
source ~/.bashrc
```

7. Associate the CMF with your queue.

```
aws deadline create-queue-fleet-association \  
  --farm-id $DEV_FARM_ID \  
  --queue-id $DEV_QUEUE_ID \  
  --fleet-id $DEV_CMF_ID
```

8. Install the Deadline Cloud command-line interface.

```
pip install deadline
```

9. To set the default farm to the farm ID and the queue to the queue ID that you created earlier, use the following command.

```
deadline config set defaults.farm_id $DEV_FARM_ID  
deadline config set defaults.queue_id $DEV_QUEUE_ID
```

10. (Optional) To confirm that your farm is set up according to your specifications, use the following commands:

- List all farms – **deadline farm list**
- List all queues in the default farm – **deadline queue list**
- List all fleets in the default farm – **deadline fleet list**
- Get the default farm – **deadline farm get**
- Get the default queue – **deadline queue get**
- Get all the fleets associated with the default queue – **deadline fleet get**

Next steps

After you create your farm, you can run the Deadline Cloud worker agent on the hosts in your fleet to process jobs. See [Run the Deadline Cloud worker agent](#).

Run the Deadline Cloud worker agent

Before you can run the jobs you submit to the queue on your developer farm, you must run the AWS Deadline Cloud worker agent in developer mode on a worker host.

Throughout the remainder of this tutorial, you will perform AWS CLI operations on your developer farm using two AWS CloudShell tabs. In the first tab, you can submit jobs. In the second tab, you can run the worker agent.

 Note

If you leave your CloudShell session idle for more than 20 minutes, it will timeout and stop the worker agent. To restart the worker agent, follow the instructions in the following procedure.

Before you can start a worker agent, you must set up a Deadline Cloud farm, queue, and fleet. See [Create a Deadline Cloud farm](#).

To run the worker agent in developer mode

1. With your farm still open in the first CloudShell tab, open a second CloudShell tab, then create the `demoenv-logs` and `demoenv-persist` directories.

```
mkdir ~/demoenv-logs
mkdir ~/demoenv-persist
```

2. Download and install the Deadline Cloud worker agent packages from PyPI:

 Note

On Windows, it is required that the agent files are installed into Python's global site-packages directory. Python virtual environments are not currently supported.

```
python -m pip install deadline-cloud-worker-agent
```

3. To allow the worker agent to create the temporary directories for running jobs, create a directory:

```
sudo mkdir /sessions
sudo chmod 750 /sessions
sudo chown cloudshell-user /sessions
```

4. Run the Deadline Cloud worker agent in developer mode with the variables `DEV_FARM_ID` and `DEV_CMF_ID` that you added to the `~/.bashrc`.

```
deadline-worker-agent \  
  --farm-id $DEV_FARM_ID \  
  --fleet-id $DEV_CMF_ID \  
  --run-jobs-as-agent-user \  
  --logs-dir ~/demoenv-logs \  
  --persistence-dir ~/demoenv-persist
```

As the worker agent initializes and then polls the `UpdateWorkerSchedule` API operation the following output is displayed:

```
INFO    Worker Agent starting  
[2024-03-27 15:51:01,292][INFO    ] # Worker Agent starting  
[2024-03-27 15:51:01,292][INFO    ] AgentInfo  
Python Interpreter: /usr/bin/python3  
Python Version: 3.9.16 (main, Sep  8 2023, 00:00:00) - [GCC 11.4.1 20230605 (Red  
  Hat 11.4.1-2)]  
Platform: linux  
...  
[2024-03-27 15:51:02,528][INFO    ] # API.Resp # [deadline:UpdateWorkerSchedule]  
(200) params={'assignedSessions': {}, 'cancelSessionActions': {},  
  'updateIntervalSeconds': 15} ...  
[2024-03-27 15:51:17,635][INFO    ] # API.Resp # [deadline:UpdateWorkerSchedule]  
(200) params=(Duplicate removed, see previous response) ...  
[2024-03-27 15:51:32,756][INFO    ] # API.Resp # [deadline:UpdateWorkerSchedule]  
(200) params=(Duplicate removed, see previous response) ...  
...
```

5. Select your first CloudShell tab, then list the workers in the fleet.

```
deadline worker list --fleet-id $DEV_CMF_ID
```

Output such as the following is displayed:

```
Displaying 1 of 1 workers starting at 0  
  
- workerId: worker-8c9af877c8734e89914047111f  
  status: STARTED  
  createdAt: 2023-12-13 20:43:06+00:00
```

In a production configuration, the Deadline Cloud worker agent requires setting up multiple users and configuration directories as an administrative user on the host machine. You can override these settings because you're running jobs in your own development farm, which only you can access.

Next steps

Now that a worker agent is running on your worker hosts, you can send jobs to your workers. You can:

- [Submit with Deadline Cloud](#) using a simple OpenJD job bundle.
- [Submit jobs with job attachments in Deadline Cloud](#) that share files between workstations using different operating systems.

Submit with Deadline Cloud

To run Deadline Cloud jobs on your worker hosts, you create and use an Open Job Description (OpenJD) job bundle to configure a job. The bundle configures the job, for example by specifying input files for a job and where to write the output of the job. This topic includes examples of ways that you can configure a job bundle.

Before you can follow the procedures in this section, you must complete the following:

- [Create a Deadline Cloud farm](#)
- [Run the Deadline Cloud worker agent](#)

To use AWS Deadline Cloud to run jobs, use the following procedures. Use the first AWS CloudShell tab to submit jobs to your developer farm. Use the second CloudShell tab to view the worker agent output.

Topics

- [Submit the simple_job sample](#)
- [Submit a simple_job with a parameter](#)
- [Create a simple_file_job job bundle with file I/O](#)
- [Next steps](#)

Submit the simple_job sample

After you create a farm and run the worker agent, you can submit the simple_job sample to Deadline Cloud.

To submit the simple_job sample to Deadline Cloud

1. Choose your first CloudShell tab.
2. Download the sample from GitHub.

```
cd ~  
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
```

3. Navigate to the job bundle samples directory.

```
cd ~/deadline-cloud-samples/job_bundles/
```

4. Submit the simple_job sample.

```
deadline bundle submit simple_job
```

5. Choose your second CloudShell tab to view the logging output about calling BatchGetJobEntities, getting a session, and running a session action.

```
...  
[2024-03-27 16:00:21,846][INFO    ] # Session.Starting  
# [session-053d77cef82648fe2] Starting new Session.  
[queue-3ba4ff683ff54db09b851a2ed8327d7b/job-d34cc98a6e234b6f82577940ab4f76c6]  
[2024-03-27 16:00:21,853][INFO    ] # API.Req # [deadline:BatchGetJobEntity]  
resource={'farm-id': 'farm-3e24cfc9bbcd423e9c1b6754bc1',  
'fleet-id': 'fleet-246ee60f46d44559b6cce010d05', 'worker-id':  
'worker-75e0fce9c3c344a69bff57fcd83'} params={'identifiers': [{'jobDetails':  
{'jobId': 'job-d34cc98a6e234b6f82577940ab4'}}]} request_url=https://  
scheduling.deadline.us-west-2.amazonaws.com/2023-10-12/farms/  
farm-3e24cfc9bbcd423e /fleets/fleet-246ee60f46d44559b1 /workers/worker-  
75e0fce9c3c344a69b /batchGetJobEntity  
[2024-03-27 16:00:22,013][INFO    ] # API.Resp # [deadline:BatchGetJobEntity](200)  
params={'entities': [{'jobDetails': {'jobId': 'job-d34cc98a6e234b6f82577940ab6',  
'jobRunAsUser': {'posix': {'user': 'job-user', 'group': 'job-group'},  
'runAs': 'QUEUE_CONFIGURED_USER'}, 'logGroupName': '/aws/deadline/  
farm-3e24cfc9bbcd423e9c1b6754bc1/queue-3ba4ff683ff54db09b851a2ed83', 'parameters':
```

```
'*REDACTED*', 'schemaVersion': 'jobtemplate-2023-09'}}], 'errors': []}
request_id=a3f55914-6470-439e-89e5-313f0c6
[2024-03-27 16:00:22,013][INFO    ] # Session.Add #
[session-053d77cef82648fea9c69827182] Appended new SessionActions.
(ActionIds: ['sessionaction-053d77cef82648fea9c69827182-0'])
[queue-3ba4ff683ff54db09b851a2ed8b/job-d34cc98a6e234b6f82577940ab6]
[2024-03-27 16:00:22,014][WARNING ] # Session.User #
[session-053d77cef82648fea9c69827182] Running as the Worker Agent's
user. (User: cloudshell-user) [queue-3ba4ff683ff54db09b851a2ed8b/job-
d34cc98a6e234b6f82577940ac6]
[2024-03-27 16:00:22,015][WARNING ] # Session.AWSCreds #
[session-053d77cef82648fea9c69827182] AWS Credentials are not available: Queue has
no IAM Role. [queue-3ba4ff683ff54db09b851a2ed8b/job-d34cc98a6e234b6f82577940ab6]
[2024-03-27 16:00:22,026][INFO    ] # Session.Logs #
[session-053d77cef82648fea9c69827182] Logs streamed to: AWS CloudWatch
Logs. (LogDestination: /aws/deadline/farm-3e24cfc9bbcd423e9c1b6754bc1/
queue-3ba4ff683ff54db09b851a2ed83/session-053d77cef82648fea9c69827181)
[queue-3ba4ff683ff54db09b851a2ed83/job-d34cc98a6e234b6f82577940ab4]
[2024-03-27 16:00:22,026][INFO    ] # Session.Logs #
[session-053d77cef82648fea9c69827182] Logs streamed to: local
file. (LogDestination: /home/cloudshell-user/demoenv-logs/
queue-3ba4ff683ff54db09b851a2ed8b/session-053d77cef82648fea9c69827182.log)
[queue-3ba4ff683ff54db09b851a2ed83/job-d34cc98a6e234b6f82577940ab4]
...
```

Note

Only the logging output from the worker agent is shown. There is a separate log for the session that runs the job.

6. Choose your first tab, then inspect the log files that the worker agent writes.

a. Navigate to the worker agent logs directory and view its contents.

```
cd ~/demoenv-logs
ls
```

b. Print the first log file that the worker agent creates.

```
cat worker-agent-bootstrap.log
```

This file contains worker agent output about how it called the Deadline Cloud API to create a worker resource in your fleet, and then assumed the fleet role.

- c. Print the log file output when the worker agent joins the fleet.

```
cat worker-agent.log
```

This log contains outputs about all the actions that the worker agent takes, but doesn't contain output about the queues it runs jobs from, except for the IDs of those resources.

- d. Print the log files for each session in a directory that is named the same as the queue resource id.

```
cat $DEV_QUEUE_ID/session-*.log
```

If the job is successful, the log file output will be similar to the following:

```
cat $DEV_QUEUE_ID/$(ls -t $DEV_QUEUE_ID | head -1)
```

```
2024-03-27 16:00:22,026 WARNING Session running with no AWS Credentials.
2024-03-27 16:00:22,404 INFO
2024-03-27 16:00:22,405 INFO =====
2024-03-27 16:00:22,405 INFO ----- Running Task
2024-03-27 16:00:22,405 INFO =====
2024-03-27 16:00:22,406 INFO -----
2024-03-27 16:00:22,406 INFO Phase: Setup
2024-03-27 16:00:22,406 INFO -----
2024-03-27 16:00:22,406 INFO Writing embedded files for Task to disk.
2024-03-27 16:00:22,406 INFO Mapping: Task.File.runScript -> /sessions/
session-053d77cef82648fea9c698271812a/embedded_files_gj55_/tmp2u9yqtsz
2024-03-27 16:00:22,406 INFO Wrote: runScript -> /sessions/
session-053d77cef82648fea9c698271812a/embedded_files_gj55_/tmp2u9yqtsz
2024-03-27 16:00:22,407 INFO -----
2024-03-27 16:00:22,407 INFO Phase: Running action
2024-03-27 16:00:22,407 INFO -----
2024-03-27 16:00:22,407 INFO Running command /sessions/
session-053d77cef82648fea9c698271812a/tmpzuzxpslm.sh
2024-03-27 16:00:22,414 INFO Command started as pid: 471
2024-03-27 16:00:22,415 INFO Output:
2024-03-27 16:00:22,420 INFO Welcome to AWS Deadline Cloud!
2024-03-27 16:00:22,571 INFO
2024-03-27 16:00:22,572 INFO =====
```



```
2024-03-27 16:00:22,572 INFO ----- Session Cleanup
2024-03-27 16:00:22,572 INFO =====
2024-03-27 16:00:22,572 INFO Deleting working directory: /sessions/
session-053d77cef82648fea9c698271812a
```

7. Print information about the job.

```
deadline job get
```

When you submit the job, the system saves it as the default so you don't have to enter the job ID.

Submit a simple_job with a parameter

You can submit jobs with parameters. In the following procedure, you edit the simple_job template to include a custom message, submit the simple_job, then print the session log file to view the message.

To submit the simple_job sample with a parameter

1. Select your first CloudShell tab, then navigate to the job bundle samples directory.

```
cd ~/deadline-cloud-samples/job_bundles/
```

2. Print the contents of the simple_job template.

```
cat simple_job/template.yaml
```

The parameterDefinitions section with the Message parameter should look like the following:

```
parameterDefinitions:
- name: Message
  type: STRING
  default: Welcome to AWS Deadline Cloud!
```

3. Submit the simple_job sample with a parameter value, then wait for the job to finish running.

```
deadline bundle submit simple_job \
```

```
-p "Message=Greetings from the developer getting started guide."
```

4. To see the custom message, view the most recent session log file.

```
cd ~/demoenv-logs
cat $DEV_QUEUE_ID/$(ls -t $DEV_QUEUE_ID | head -1)
```

Create a simple_file_job job bundle with file I/O

A render job needs to read the scene definition, render an image from it, and then save that image to an output file. You can simulate this action by making the job compute the hash of the input instead of rendering an image.

To create a simple_file_job job bundle with file I/O

1. Select your first CloudShell tab, then navigate to the job bundle samples directory.

```
cd ~/deadline-cloud-samples/job_bundles/
```

2. Make a copy of simple_job with the new name simple_file_job.

```
cp -r simple_job simple_file_job
```

3. Edit the job template as follows:

Note

We recommend that you use nano for these steps. If you prefer to use Vim, you must set its paste mode using `:set paste`.

- a. Open the template in a text editor.

```
nano simple_file_job/template.yaml
```

- b. Add the following type, objectType, and dataFlow parameterDefinitions.

```
- name: InFile
  type: PATH
  objectType: FILE
```

```
dataFlow: IN
- name: OutFile
  type: PATH
  objectType: FILE
dataFlow: OUT
```

- c. Add the following bash script command to the end of the file that reads from the input file and writes to the output file.

```
# hash the input file, and write that to the output
sha256sum "{{Param.InFile}}" > "{{Param.OutFile}}"
```

The updated `template.yaml` should exactly match the following:

```
specificationVersion: 'jobtemplate-2023-09'
name: Simple File Job Bundle Example
parameterDefinitions:
- name: Message
  type: STRING
  default: Welcome to AWS Deadline Cloud!
- name: InFile
  type: PATH
  objectType: FILE
  dataFlow: IN
- name: OutFile
  type: PATH
  objectType: FILE
  dataFlow: OUT
steps:
- name: WelcomeToDeadlineCloud
  script:
    actions:
      onRun:
        command: '{{Task.File.Run}}'
    embeddedFiles:
      - name: Run
        type: TEXT
        runnable: true
        data: |
          #!/usr/bin/env bash
          echo "{{Param.Message}}"

          # hash the input file, and write that to the output
```

```
sha256sum "{{Param.InFile}}" > "{{Param.OutFile}}"
```

Note

If you want to adjust the spacing in the `template.yaml`, make sure that you use spaces instead of indentations.

- d. Save the file, and exit the text editor.
4. Provide parameter values for the input and output files to submit the `simple_file_job`.

```
deadline bundle submit simple_file_job \  
-p "InFile=simple_job/template.yaml" \  
-p "OutFile=hash.txt"
```

5. Print information about the job.

```
deadline job get
```

- You will see output such as the following:

```
parameters:  
  Message:  
    string: Welcome to AWS Deadline Cloud!  
  InFile:  
    path: /local/home/cloudshell-user/BundleFiles/JobBundle-Examples/simple_job/  
template.yaml  
  OutFile:  
    path: /local/home/cloudshell-user/BundleFiles/JobBundle-Examples/hash.txt
```

- Although you only provided relative paths, the parameters have the full path set. The AWS CLI joins the current working directory to any paths that are provided as parameters when the paths have the type `PATH`.
- The worker agent running in the other terminal window picks up and runs the job. This action creates the `hash.txt` file, which you can view with the following command.

```
cat hash.txt
```

This command will print output similar to the following.

```
eea2df5d34b54be5ac34c56a24a8c237b8487231a607eaf530a04d76b89c9cd3 /local/home/  
cloudshell-user/BundleFiles/JobBundle-Examples/simple_job/template.yaml
```

Next steps

After learning how to submit simple jobs using the Deadline Cloud CLI, you can explore:

- [Submit jobs with job attachments in Deadline Cloud](#) to learn how to run jobs on hosts running different operating systems.
- [Add a service-managed fleet to your developer farm in Deadline Cloud](#) to run your jobs on hosts managed by Deadline Cloud.
- [Clean up your farm resources in Deadline Cloud](#) to shut down the resources that you used for this tutorial.

Submit jobs with job attachments in Deadline Cloud

Many farms use shared filesystems to share files between the hosts that submit jobs and those that run jobs. For example, in the previous `simple_file_job` example, the local filesystem is shared between the AWS CloudShell terminal windows, which run in tab one where you submit the job, and tab two where you run the worker agent.

A shared filesystem is advantageous when the submitter workstation and the worker hosts are on the same local area network. If you store your data on premises near the workstations that access it, then using a cloud-based farm means you have to share your filesystems over a high-latency VPN or synchronize your filesystems in the cloud. Neither of these options are easy to set up or operate.

AWS Deadline Cloud offers a simple solution with *job attachments*, which are similar to email attachments. With job attachments, you attach data to your job. Then, Deadline Cloud handles the details of transferring and storing your job data in Amazon Simple Storage Service (Amazon S3) buckets.

Content creation workflows are often iterative, meaning a user submits jobs with a small subset of modified files. Because Amazon S3 buckets store job attachments in a content-addressable storage, the name of each object is based on the hash of the object's data and the contents of a directory tree are stored in a manifest file format attached to a job.

Before you can follow the procedures in this section, you must complete the following:

- [Create a Deadline Cloud farm](#)
- [Run the Deadline Cloud worker agent](#)

To run jobs with job attachments, complete the following steps.

Topics

- [Add a job attachments configuration to your queue](#)
- [Submit simple_file_job with job attachments](#)
- [Understanding how job attachments are stored in Amazon S3](#)
- [Next steps](#)

Add a job attachments configuration to your queue

To enable job attachments in your queue, add a job attachments configuration to the queue resource in your account.

To add a job attachments configuration to your queue

1. Choose your first CloudShell tab, then enter one of the following commands to use an Amazon S3 bucket for job attachments.
 - If you don't have an existing private Amazon S3 bucket, you can create and use a new S3 bucket.

```
DEV_FARM_BUCKET=$(echo $DEV_FARM_NAME \  
    | tr '[:upper:]' '[:lower:]')-$(xxd -l 16 -p /dev/urandom)  
if [ "$AWS_REGION" == "us-east-1" ]; then LOCATION_CONSTRAINT=  
else LOCATION_CONSTRAINT="--create-bucket-configuration \  
    LocationConstraint=${AWS_REGION}"  
fi  
aws s3api create-bucket \  
    $LOCATION_CONSTRAINT \  
    --acl private \  
    --bucket ${DEV_FARM_BUCKET}
```

- If you already have a private Amazon S3 bucket, you can use it by replacing *MY_BUCKET_NAME* with the name of your bucket.

```
DEV_FARM_BUCKET=MY_BUCKET_NAME
```

2. After you create or choose your Amazon S3 bucket, add the bucket name to `~/.bashrc` to make the bucket available for other terminal sessions.

```
echo "DEV_FARM_BUCKET=$DEV_FARM_BUCKET" >> ~/.bashrc  
source ~/.bashrc
```

3. Create an AWS Identity and Access Management (IAM) role for the queue.

```
aws iam create-role --role-name "${DEV_FARM_NAME}QueueRole" \  
  --assume-role-policy-document \  
    '{  
      "Version": "2012-10-17",  
      "Statement": [  
        {  
          "Effect": "Allow",  
          "Principal": {  
            "Service": "credentials.deadline.amazonaws.com"  
          },  
          "Action": "sts:AssumeRole"  
        }  
      ]  
    }'  
aws iam put-role-policy \  
  --role-name "${DEV_FARM_NAME}QueueRole" \  
  --policy-name S3BucketsAccess \  
  --policy-document \  
    '{  
      "Version": "2012-10-17",  
      "Statement": [  
        {  
          "Action": [  
            "s3:GetObject*",  
            "s3:GetBucket*",  
            "s3:List*",  
            "s3:DeleteObject*",  
            "s3:PutObject",  
            "s3:PutObjectLegalHold",  
            "s3:PutObjectRetention",  
            "s3:PutObjectTagging",  
            "s3:PutObjectVersionTagging",
```

```

        "s3:Abort*"
      ],
      "Resource": [
        "arn:aws:s3:::'$DEV_FARM_BUCKET'",
        "arn:aws:s3:::'$DEV_FARM_BUCKET'/*"
      ],
      "Effect": "Allow"
    }
  ]
}'

```

4. Update your queue to include the job attachments settings and the IAM role.

```

QUEUE_ROLE_ARN="arn:aws:iam::$(aws sts get-caller-identity \
  --query "Account" --output text):role/${DEV_FARM_NAME}QueueRole"
aws deadline update-queue \
  --farm-id $DEV_FARM_ID \
  --queue-id $DEV_QUEUE_ID \
  --role-arn $QUEUE_ROLE_ARN \
  --job-attachment-settings \
    '{
      "s3BucketName": "'$DEV_FARM_BUCKET'",
      "rootPrefix": "JobAttachments"
    }'

```

5. Confirm that you updated your queue.

```
deadline queue get
```

Output such as the following is shown:

```

...
jobAttachmentSettings:
  s3BucketName: DEV_FARM_BUCKET
  rootPrefix: JobAttachments
roleArn: arn:aws:iam::ACCOUNT_NUMBER:role/DeveloperFarmQueueRole
...

```


Submit simple_file_job with job attachments

When you use job attachments, job bundles must give Deadline Cloud enough information to determine the job's data flow, such as using PATH parameters. In the case of the `simple_file_job`, you edited the `template.yaml` file to tell Deadline Cloud that the data flow is in the input file and output file.

After you've added the job attachments configuration to your queue, you can submit the `simple_file_job` sample with job attachments. After you do this, you can view the logging and job output to confirm that the `simple_file_job` with job attachments is working.

To submit the simple_file_job job bundle with job attachments

1. Choose your first CloudShell tab, then open the `JobBundle-Samples` directory.

2.

```
cd ~/deadline-cloud-samples/job_bundles/
```

3. Submit `simple_file_job` to the queue. When prompted to confirm the upload, enter `y`.

```
deadline bundle submit simple_file_job \  
-p InFile=simple_job/template.yaml \  
-p OutFile=hash-jobattachments.txt
```

4. To view the job attachments data transfer session log output, run the following command.

```
JOB_ID=$(deadline config get defaults.job_id)  
SESSION_ID=$(aws deadline list-sessions \  
    --farm-id $DEV_FARM_ID \  
    --queue-id $DEV_QUEUE_ID \  
    --job-id $JOB_ID \  
    --query "sessions[0].sessionId" \  
    --output text)  
cat ~/demoenv-logs/$DEV_QUEUE_ID/$SESSION_ID.log
```

5. List the session actions that were run within the session.

```
aws deadline list-session-actions \  
    --farm-id $DEV_FARM_ID \  
    --queue-id $DEV_QUEUE_ID \  
    --job-id $JOB_ID \  
    --session-id $SESSION_ID
```

Output such as the following is shown:

```
{
  "sessionactions": [
    {
      "sessionActionId": "sessionaction-123-0",
      "status": "SUCCEEDED",
      "startedAt": "<timestamp>",
      "endedAt": "<timestamp>",
      "progressPercent": 100.0,
      "definition": {
        "syncInputJobAttachments": {}
      }
    },
    {
      "sessionActionId": "sessionaction-123-1",
      "status": "SUCCEEDED",
      "startedAt": "<timestamp>",
      "endedAt": "<timestamp>",
      "progressPercent": 100.0,
      "definition": {
        "taskRun": {
          "taskId": "task-abc-0",
          "stepId": "step-def"
        }
      }
    }
  ]
}
```

The first session action downloaded the input job attachments, while the second action runs the task like in previous steps and then uploaded the output job attachments.

6. List the output directory.

```
ls *.txt
```

Output such as `hash.txt` exists in the directory, but `hash-jobattachments.txt` doesn't exist because the output file from the job hasn't been downloaded yet.

7. Download the output from the most recent job.

```
deadline job download-output
```

8. View the output of the downloaded file.

```
cat hash-jobattachments.txt
```

Output such as the following is shown:

```
eea2df5d34b54be5ac34c56a24a8c237b8487231a607eaf530a04d76b89c9cd3 /tmp/openjd/  
session-123/assetroot-abc/simple_job/template.yaml
```

Understanding how job attachments are stored in Amazon S3

You can use the AWS Command Line Interface (AWS CLI) to upload or download data for job attachments, which are stored in Amazon S3 buckets. Understanding how Deadline Cloud stores job attachments on Amazon S3 will help when you develop workloads and pipeline integrations.

To inspect how Deadline Cloud job attachments are stored in Amazon S3

1. Choose your first CloudShell tab, then open the job bundle samples directory.

```
cd ~/deadline-cloud-samples/job_bundles/
```

2. Inspect the job properties.

```
deadline job get
```

Output such as the following is shown:

```
parameters:  
  Message:  
    string: Welcome to AWS Deadline Cloud!  
  InFile:  
    path: /home/cloudshell-user/deadline-cloud-samples/job_bundles/simple_job/  
template.yaml  
  OutFile:  
    path: /home/cloudshell-user/deadline-cloud-samples/job_bundles/hash-  
jobattachments.txt
```

```

attachments:
  manifests:
    - rootPath: /home/cloudshell-user/deadline-cloud-samples/job_bundles/
      rootPathFormat: posix
      outputRelativeDirectories:
        - .
      inputManifestPath: farm-3040c59a5b9943d58052c29d907a645d/queue-
cde9977c9f4d4018a1d85f3e6c1a4e6e/Inputs/
f46af01ca8904cd8b514586671c79303/0d69cd94523ba617c731f29c019d16e8_input.xxh128
      inputManifestHash: f95ef91b5dab1fc1341b75637fe987ee
      fileSystem: COPIED

```

The attachments field contains a list of manifest structures that describe input and output data paths that the job uses when it runs. Look at `rootPath` to see the local directory path on the machine that submitted the job. To view the Amazon S3 object suffix that contains a manifest file, review the `inputManifestFile`. The manifest file contains metadata for a directory tree snapshot of the job's input data.

3. Pretty-print the Amazon S3 manifest object to see the input directory structure for the job.

```

MANIFEST_SUFFIX=$(aws deadline get-job \
  --farm-id $DEV_FARM_ID \
  --queue-id $DEV_QUEUE_ID \
  --job-id $JOB_ID \
  --query "attachments.manifests[0].inputManifestPath" \
  --output text)
aws s3 cp s3://$DEV_FARM_BUCKET/JobAttachments/Manifests/$MANIFEST_SUFFIX - | jq .

```

Output such as the following is shown:

```

{
  "hashAlg": "xxh128",
  "manifestVersion": "2023-03-03",
  "paths": [
    {
      "hash": "2ec297b04c59c4741ed97ac8fb83080c",
      "mtime": 16981861900000000,
      "path": "simple_job/template.yaml",
      "size": 445
    }
  ],
  "totalSize": 445
}

```

```
}
```

4. Construct the Amazon S3 prefix that holds manifests for the output job attachments and list the object under it.

```
SESSION_ACTION=$(aws deadline list-session-actions \
  --farm-id $DEV_FARM_ID \
  --queue-id $DEV_QUEUE_ID \
  --job-id $JOB_ID \
  --session-id $SESSION_ID \
  --query "sessionActions[?definition.taskRun != null] | [0]")
STEP_ID=$(echo $SESSION_ACTION | jq -r .definition.taskRun.stepId)
TASK_ID=$(echo $SESSION_ACTION | jq -r .definition.taskRun.taskId)
TASK_OUTPUT_PREFIX=JobAttachments/Manifests/$DEV_FARM_ID/$DEV_QUEUE_ID/$JOB_ID/
$STEP_ID/$TASK_ID/
aws s3api list-objects-v2 --bucket $DEV_FARM_BUCKET --prefix $TASK_OUTPUT_PREFIX
```

The output job attachments are not directly referenced from the job resource but are instead placed in an Amazon S3 bucket based on farm resource IDs.

5. Get the newest manifest object key for the specific session action id, then pretty-print the manifest objects.

```
SESSION_ACTION_ID=$(echo $SESSION_ACTION | jq -r .sessionActionId)
MANIFEST_KEY=$(aws s3api list-objects-v2 \
  --bucket $DEV_FARM_BUCKET \
  --prefix $TASK_OUTPUT_PREFIX \
  --query "Contents[*].Key" --output text \
  | grep $SESSION_ACTION_ID \
  | sort | tail -1)
MANIFEST_OBJECT=$(aws s3 cp s3://$DEV_FARM_BUCKET/$MANIFEST_KEY -)
echo $MANIFEST_OBJECT | jq .
```

You'll see properties of the file `hash-jobattachments.txt` in the output such as the following:

```
{
  "hashAlg": "xxh128",
  "manifestVersion": "2023-03-03",
  "paths": [
    {
      "hash": "f60b8e7d0fabf7214ba0b6822e82e08b",
```

```
    "mtime": 1698785252554950,  
    "path": "hash-jobattachments.txt",  
    "size": 182  
  },  
  ],  
  "totalSize": 182  
}
```

Your job will only have a single manifest object per task run, but in general it is possible to have more of objects per task run.

6. View content-addressable Amazon S3 storage output under the Data prefix.

```
FILE_HASH=$(echo $MANIFEST_OBJECT | jq -r .paths[0].hash)  
FILE_PATH=$(echo $MANIFEST_OBJECT | jq -r .paths[0].path)  
aws s3 cp s3://$DEV_FARM_BUCKET/JobAttachments/Data/$FILE_HASH -
```

Output such as the following is shown:

```
eea2df5d34b54be5ac34c56a24a8c237b8487231a607eaf530a04d76b89c9cd3  /tmp/openjd/  
session-123/assetroot-abc/simple_job/template.yaml
```

Next steps

After learning how to submit jobs with attachments using the Deadline Cloud CLI, you can explore:

- [Submit with Deadline Cloud](#) to learn how to run jobs using an OpenJD bundle on your worker hosts.
- [Add a service-managed fleet to your developer farm in Deadline Cloud](#) to run your jobs on hosts managed by Deadline Cloud.
- [Clean up your farm resources in Deadline Cloud](#) to shut down the resources that you used for this tutorial.

Add a service-managed fleet to your developer farm in Deadline Cloud

AWS CloudShell does not provide enough compute capacity to test larger workloads. It's also not configured to work with jobs that distribute tasks on multiple worker hosts.

Instead of using CloudShell, you can add an Auto Scaling service-managed fleet (SMF) to your developer farm. An SMF provides sufficient compute capacity for larger workloads and can handle jobs that need to distribute job tasks across multiple worker hosts.

Before you add an SMF, you must set up a Deadline Cloud farm, queue, and fleet. See [Create a Deadline Cloud farm](#).

To add a service-managed fleet to your developer farm

1. Choose your first AWS CloudShell tab, then create the service-managed fleet and add its fleet ID to `.bashrc`. This action makes it available for other terminal sessions.

```
FLEET_ROLE_ARN="arn:aws:iam::$(aws sts get-caller-identity \
    --query "Account" --output text):role/${DEV_FARM_NAME}FleetRole"
aws deadline create-fleet \
    --farm-id $DEV_FARM_ID \
    --display-name "$DEV_FARM_NAME SMF" \
    --role-arn $FLEET_ROLE_ARN \
    --max-worker-count 5 \
    --configuration \
        '{
            "serviceManagedEc2": {
                "instanceCapabilities": {
                    "vCpuCount": {
                        "min": 2,
                        "max": 4
                    },
                    "memoryMiB": {
                        "min": 512
                    },
                    "osFamily": "linux",
                    "cpuArchitectureType": "x86_64"
                },
                "instanceMarketOptions": {
                    "type": "spot"
                }
            }
        }'
```

```

    }
  }'

echo "DEV_SMF_ID=$(aws deadline list-fleets \
  --farm-id $DEV_FARM_ID \
  --query "fleets[?displayName=='$DEV_FARM_NAME SMF'].fleetId \
  | [0]" --output text)" >> ~/.bashrc

source ~/.bashrc

```

2. Associate the SMF with your queue.

```

aws deadline create-queue-fleet-association \
  --farm-id $DEV_FARM_ID \
  --queue-id $DEV_QUEUE_ID \
  --fleet-id $DEV_SMF_ID

```

3. Submit `simple_file_job` to the queue. When prompted to confirm the upload, enter `y`.

```

deadline bundle submit simple_file_job \
  -p InFile=simple_job/template.yaml \
  -p OutFile=hash-jobattachments.txt

```

4. Confirm the SMF is working correctly.

```
deadline fleet get
```

- The worker may take a few minutes to start. Repeat the `deadline fleet get` command until you can see that the fleet is running.
- The `queueFleetAssociationsStatus` for service-managed fleet will be `ACTIVE`.
- The `SMF autoScalingStatus` will change from `GROWING` to `STEADY`.

Your status will look similar to the following:

```

fleetId: fleet-2cc78e0dd3f04d1db427e7dc1d51ea44
farmId: farm-63ee8d77cdab4a578b685be8c5561c4a
displayName: DeveloperFarm SMF
description: ''
status: ACTIVE
autoScalingStatus: STEADY
targetWorkerCount: 0
workerCount: 0

```



```
minWorkerCount: 0
maxWorkerCount: 5
```

5. View the log for the job that you submitted. This log is stored in a log in Amazon CloudWatch Logs, not the CloudShell file system.

```
JOB_ID=$(deadline config get defaults.job_id)
SESSION_ID=$(aws deadline list-sessions \
  --farm-id $DEV_FARM_ID \
  --queue-id $DEV_QUEUE_ID \
  --job-id $JOB_ID \
  --query "sessions[0].sessionId" \
  --output text)
aws logs tail /aws/deadline/$DEV_FARM_ID/$DEV_QUEUE_ID \
  --log-stream-names $SESSION_ID
```

Next steps

After creating and testing a service-managed fleet, you should remove the resources that you created to avoid unnecessary charges.

- [Clean up your farm resources in Deadline Cloud](#) to shut down the resources that you used for this tutorial.

Clean up your farm resources in Deadline Cloud

To develop and test new workloads and pipeline integrations, you can continue to use the Deadline Cloud developer farm that you created for this tutorial. If you no longer need your developer farm, you can delete its resources including farm, fleet, queue, AWS Identity and Access Management (IAM) roles, and logs in Amazon CloudWatch Logs. After you delete these resources, you will need to begin the tutorial again to use the resources. For more information, see [Getting started with Deadline Cloud resources](#).

To clean up developer farm resources

1. Choose your first CloudShell tab, then stop all the queue-fleet associations for your queue.

```
FLEETS=$(aws deadline list-queue-fleet-associations \
  --farm-id $DEV_FARM_ID \
```

```

        --queue-id $DEV_QUEUE_ID \
        --query "queueFleetAssociations[].fleetId" \
        --output text)
for FLEET_ID in $FLEETS; do
    aws deadline update-queue-fleet-association \
        --farm-id $DEV_FARM_ID \
        --queue-id $DEV_QUEUE_ID \
        --fleet-id $FLEET_ID \
        --status STOP_SCHEDULING_AND_CANCEL_TASKS
done

```

2. List the queue fleet associations.

```

aws deadline list-queue-fleet-associations \
    --farm-id $DEV_FARM_ID \
    --queue-id $DEV_QUEUE_ID

```

You might need to rerun the command until the output reports "status": "STOPPED", then you can proceed to the next step. This process can take several minutes to complete.

```

{
  "queueFleetAssociations": [
    {
      "queueId": "queue-abcdefgh01234567890123456789012id",
      "fleetId": "fleet-abcdefgh01234567890123456789012id",
      "status": "STOPPED",
      "createdAt": "2023-11-21T20:49:19+00:00",
      "createdBy": "arn:aws:sts::123456789012:assumed-role/RoleToBeAssumed/MySessionName",
      "updatedAt": "2023-11-21T20:49:38+00:00",
      "updatedBy": "arn:aws:sts::123456789012:assumed-role/RoleToBeAssumed/MySessionName"
    },
    {
      "queueId": "queue-abcdefgh01234567890123456789012id",
      "fleetId": "fleet-abcdefgh01234567890123456789012id",
      "status": "STOPPED",
      "createdAt": "2023-11-21T20:32:06+00:00",
      "createdBy": "arn:aws:sts::123456789012:assumed-role/RoleToBeAssumed/MySessionName",
      "updatedAt": "2023-11-21T20:49:39+00:00",
      "updatedBy": "arn:aws:sts::123456789012:assumed-role/RoleToBeAssumed/MySessionName"
    }
  ]
}

```

```
    }  
  ]  
}
```

3. Delete all of the queue-fleet associations for your queue.

```
for FLEET_ID in $FLEETS; do  
    aws deadline delete-queue-fleet-association \  
        --farm-id $DEV_FARM_ID \  
        --queue-id $DEV_QUEUE_ID \  
        --fleet-id $FLEET_ID  
done
```

4. Delete all of the fleets associated with your queue.

```
for FLEET_ID in $FLEETS; do  
    aws deadline delete-fleet \  
        --farm-id $DEV_FARM_ID \  
        --fleet-id $FLEET_ID  
done
```

5. Delete the queue.

```
aws deadline delete-queue \  
    --farm-id $DEV_FARM_ID \  
    --queue-id $DEV_QUEUE_ID
```

6. Delete the farm.

```
aws deadline delete-farm \  
    --farm-id $DEV_FARM_ID
```

7. Delete other AWS resources for your farm.

- a. Delete the fleet AWS Identity and Access Management (IAM) role.

```
aws iam delete-role-policy \  
    --role-name "${DEV_FARM_NAME}FleetRole" \  
    --policy-name WorkerPermissions  
aws iam delete-role \  
    --role-name "${DEV_FARM_NAME}FleetRole"
```

- b. Delete the queue IAM role.

```
aws iam delete-role-policy \  
    --role-name "${DEV_FARM_NAME}QueueRole" \  
    --policy-name S3BucketsAccess  
aws iam delete-role \  
    --role-name "${DEV_FARM_NAME}QueueRole"
```

- c. Delete the Amazon CloudWatch Logs log groups. Each queue and fleet has their own log group.

```
aws logs delete-log-group \  
    --log-group-name "/aws/deadline/$DEV_FARM_ID/$DEV_QUEUE_ID"  
aws logs delete-log-group \  
    --log-group-name "/aws/deadline/$DEV_FARM_ID/$DEV_CMF_ID"  
aws logs delete-log-group \  
    --log-group-name "/aws/deadline/$DEV_FARM_ID/$DEV_SMF_ID"
```

Build jobs to submit to Deadline Cloud

You submit jobs to Deadline Cloud using job bundles. A job bundle is a collection of files, including a job template and any asset files needed to render the job. The job template follows the [Open Job Description \(OpenJD\) specification](#) on the GitHub website.

The job template describes how workers process and access the assets, and provides the script that the worker runs. Job bundles enable artists, technical directors, and pipeline developers to easily submit complex jobs to Deadline Cloud from their local workstations or on-premises infrastructure. Job bundles are useful for teams working on large-scale rendering, scientific simulations, machine learning training, or other workloads that require scalable, on-demand computing resources.

You can create the job bundle using the local file system to store files and a text editor to create the job template. After creating the bundle, submit the job to Deadline Cloud using either the Deadline Cloud CLI or a tool like a Deadline Cloud submitter

You can store your assets in a file system shared between your workers, or you can use Deadline Cloud job attachments to automate moving assets to S3 buckets where your workers can access them. Job attachments also help move the output from your jobs back to your workstations.

The following sections provide detailed instructions on creating and submitting job bundles to Deadline Cloud.

Topics

- [Open Job Description \(OpenJD\) templates for Deadline Cloud](#)
- [Using files in your jobs](#)
- [Use job attachments to share files](#)
- [Create resource limits for jobs](#)
- [How to submit a job to Deadline Cloud](#)
- [Share job bundles on your queue](#)
- [Host requirements and fleet capabilities](#)
- [Schedule jobs in Deadline Cloud](#)
- [Modify a job in Deadline Cloud](#)

Open Job Description (OpenJD) templates for Deadline Cloud

A *job bundle* is one of the tools you use to define jobs for AWS Deadline Cloud. They group an Open Job Description (OpenJD) template with additional information such as files and directories that your jobs use with job attachments. For more information, see the [Open Job Description specification](#) on the GitHub website. You use the Deadline Cloud command-line interface (CLI) to use a job bundle to submit jobs for a queue to run.

A job bundle is a directory structure that contains an OpenJD job template, other files that define the job, and job-specific files required as input for your job. You can specify the files that define your job as either YAML or JSON files.

The only required file is either `template.yaml` or `template.json`. You can also include the following files:

```
/template.yaml (or template.json)
/asset_references.yaml (or asset_references.json)
/parameter_values.yaml (or parameter_values.json)
/other job-specific files and directories
```

Use a job bundle for custom job submissions with the Deadline Cloud CLI and a job attachment, or you can use an graphical submission interface. For example, the following is the Blender sample from the [deadline-cloud-samples repository](#) on the GitHub website. To run the sample, use the following command in the Blender sample directory:

```
deadline bundle gui-submit blender_render
```

The screenshot shows a macOS-style window titled "Submit to AWS Deadline Cloud". It has three tabs: "Shared job settings" (selected), "Job-specific settings", and "Job attachments". The "Job Properties" section contains the following fields:

- Name:
- Description:
- Priority: (with up/down arrows)
- Initial state: (with a dropdown arrow)
- Maximum failed tasks count: (with up/down arrows)
- Maximum retries per task: (with up/down arrows)
- Maximum worker count:
☐ No max worker count
☒ Set max worker count
 (with up/down arrows)

The "Deadline Cloud settings" section shows:

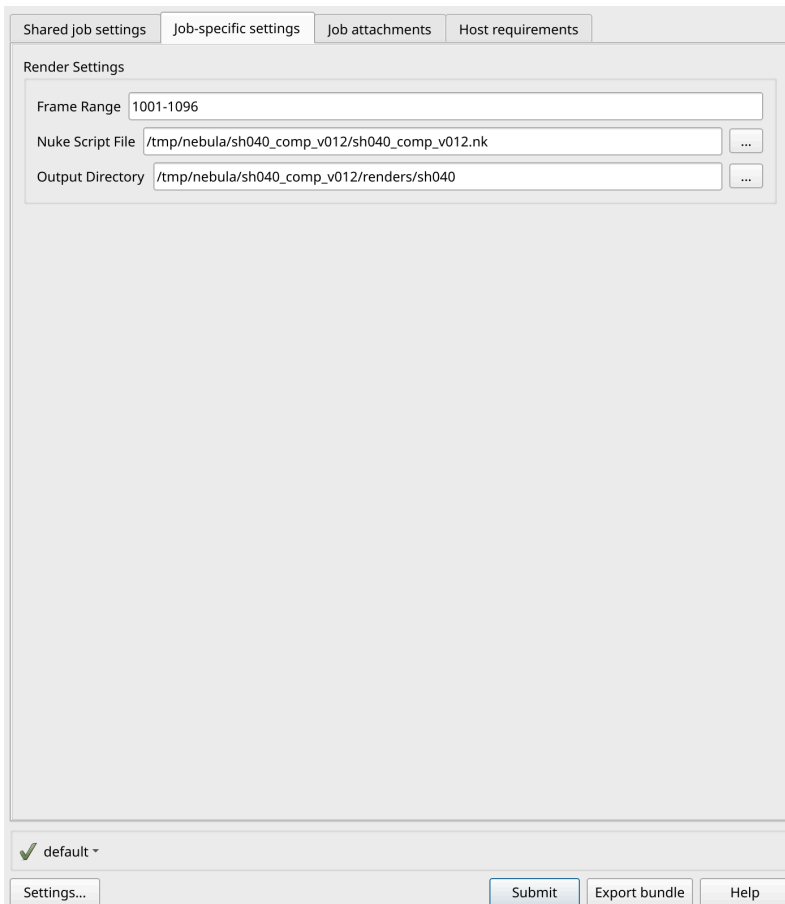
- Farm: TestFarm
- Queue: TestQueue2

At the bottom, there are three status boxes:

- Credential source: **HOST_PROVIDED**
- Authentication status: **AUTHENTICATED**
- AWS Deadline Cloud API: **AUTHORIZED**

Below these are buttons for "Login", "Logout", "Settings...", "Export bundle", and a blue "Submit" button.

The job-specific settings panel are generated from the `userInterface` properties of the job parameters defined in the job template.



Shared job settings | Job-specific settings | Job attachments | Host requirements

Render Settings

Frame Range 1001-1096

Nuke Script File /tmp/nebula/sh040_comp_v012/sh040_comp_v012.nk ...

Output Directory /tmp/nebula/sh040_comp_v012/renders/sh040 ...

✓ default ▾

Settings... Submit Export bundle Help

To submit a job using the command line, you can use a command similar to the following

```
deadline bundle submit \  
  --yes \  
  --name Demo \  
  -p BlenderSceneFile=location of scene file \  
  -p OutputDir=file pathe for job output \  
  blender_render/
```

Or you can use the `deadline.client.api.create_job_from_job_bundle` function in the deadline Python package.

All of the job submitter plugins provided with Deadline Cloud, such as the Autodesk Maya plugin, generate a job bundle for your submission and then use the Deadline Cloud Python package to submit your job to Deadline Cloud. You can see the job bundles submitted in the job history directory of your workstation or by using a submitter. You can find your job history directory with the following command:


```
deadline config get settings.job_history_dir
```

When your job is running on a Deadline Cloud worker, it has access to environment variables that provide it with information about the job. The environment variables are:

Variable name	Available
DEADLINE_FARM_ID	All actions
DEADLINE_FLEET_ID	All actions
DEADLINE_WORKER_ID	All actions
DEADLINE_QUEUE_ID	All actions
DEADLINE_JOB_ID	All actions
DEADLINE_STEP_ID	Task actions
DEADLINE_SESSION_ID	All actions
DEADLINE_TASK_ID	Task actions
DEADLINE_SESSIONACTION_ID	All actions

Topics

- [Job template elements for job bundles](#)
- [Task chunking for job templates](#)
- [Parameter values elements for job bundles](#)
- [Asset references elements for job bundles](#)

Job template elements for job bundles

The job template defines the runtime environment and the processes that run as part of a Deadline Cloud job. You can create parameters in a template so that it can be used to create jobs that differ only in input values, much like a function in a programming language.

When you submit a job to Deadline Cloud, it runs in any queue environments applied to the queue. Queue environments are built using the Open Job Description (OpenJD) external environments specification. For details, see the [OpenJD specification](#) on the GitHub website.

For an introduction to creating a job with an OpenJD job template, see [Introduction to creating a job](#) on the GitHub website. For how a job runs, see [How jobs are run](#) on the GitHub website.

You can define the job template in either YAML format (`template.yaml`) or JSON format (`template.json`). The examples in this section are shown in YAML format.

For example, the job template for the `blender_render` sample defines an input parameter `BlenderSceneFile` as a file path:

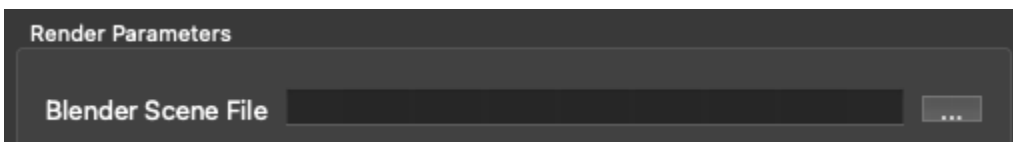
```
- name: BlenderSceneFile
  type: PATH
  objectType: FILE
  dataFlow: IN
  userInterface:
    control: CHOOSE_INPUT_FILE
    label: Blender Scene File
    groupLabel: Render Parameters
    fileFilters:
      - label: Blender Scene Files
        patterns: ["*.blend"]
      - label: All Files
        patterns: ["*"]
  description: >
    Choose the Blender scene file to render. Use the 'Job Attachments' tab
    to add textures and other files that the job needs.
```

The `userInterface` property defines the behavior of automatically generated user interfaces for both the command line using the `deadline bundle gui-submit` command and within the job submission plugins for applications like Autodesk Maya.

The `gui-submit` command opens a submission dialog with a **Shared job settings** tab for the name, priority, and target queue of the job, and a **Job-specific settings** tab generated from the parameter definitions in the job template.

The screenshot shows the 'Job Properties' dialog box in Deadline Cloud. It has four tabs: 'Shared job settings', 'Job-specific settings', 'Job attachments', and 'Host requirements'. The 'Job-specific settings' tab is active. It contains two sections: 'Job Properties' and 'Job submission settings'. The 'Job Properties' section includes fields for Name (sh040_comp_v012), Description, Priority (50), Initial state (READY), Maximum failed tasks count (20), Maximum retries per task (5), and Maximum worker count (No max worker count selected). The 'Job submission settings' section includes Farm ((us-west-2) Nebula Feature), Queue (Compositing), and Default storage profile (Linux workstations). Below these is a 'Loading Queue Environments...' section. At the bottom, there is a 'default' dropdown, a 'Settings...' button, and 'Submit', 'Export bundle', and 'Help' buttons.

In this example, the UI widget for inputting a value for the `BlenderSceneFile` parameter is a file-selection dialog that shows only `.blend` files.



For more examples of using the `userInterface` element, see the [gui_control_showcase sample](#) in the [deadline-cloud-samples repository](#) on the GitHub website.

The `objectType` and `dataFlow` properties control the behavior of job attachments when you submit a job from a job bundle. In this case, `objectType: FILE` and `dataFlow: IN` mean that the value of `BlenderSceneFile` is an input file for job attachments.

In contrast, the definition of the `OutputDir` parameter has `objectType: DIRECTORY` and `dataFlow: OUT`:

```
- name: OutputDir
  type: PATH
```

```
objectType: DIRECTORY
dataFlow: OUT
userInterface:
  control: CHOOSE_DIRECTORY
  label: Output Directory
  groupLabel: Render Parameters
default: "./output"
description: Choose the render output directory.
```

The value of the `OutputDir` parameter is used by job attachments as the directory where the job writes output files.

When you submit the bundle, the submission dialog's **Job attachments** tab lists the input files and directories it detected from these properties, along with the output directories the job writes to. Paths shown in *italics* were detected automatically; you can add others that the submitter didn't find.

The screenshot shows the 'Job attachments' tab of the Deadline Cloud submission dialog. It features four tabs at the top: 'Shared job settings', 'Job-specific settings', 'Job attachments' (selected), and 'Host requirements'. The main area is divided into three sections: 'General submission settings' with a checkbox for 'Require all input paths exist'; 'Attach input files' with a list of files including *luts/show_lut_v002.cube* and *sh040_comp_v012.nk*; and 'Attach input directories' with a list of directories including *elements* and *textures*. Below these is a 'Specify output directories' section with a list containing *renders/sh040*. At the bottom, there is a 'default' dropdown menu, a 'Settings...' button, and 'Submit', 'Export bundle', and 'Help' buttons.

For more information about the `objectType` and `dataFlow` properties, see [JobPathParameterDefinition](#) in the [Open Job Description specification](#) on the GitHub website.

The rest of the `blender_render` job template sample defines the job's workflow as a single step with each frame in the animation rendered as a separate task:

```
steps:
- name: RenderBlender
  parameterSpace:
    taskParameterDefinitions:
      - name: Frame
        type: INT
        range: "{{Param.Frames}}"
  script:
    actions:
      onRun:
        command: bash
        # Note: {{Task.File.Run}} is a variable that expands to the filename on the
worker host's
        # disk where the contents of the 'Run' embedded file, below, is written.
        args: ['{{Task.File.Run}}']
    embeddedFiles:
      - name: Run
        type: TEXT
        data: |
          # Configure the task to fail if any individual command fails.
          set -xeuo pipefail

          mkdir -p '{{Param.OutputDir}}'

          blender --background '{{Param.BlenderSceneFile}}' \
            --render-output '{{Param.OutputDir}}/{{Param.OutputPattern}}' \
            --render-format {{Param.Format}} \
            --use-extension 1 \
            --render-frame {{Task.Param.Frame}}
```

For example, if the value of the `Frames` parameter is 1-10, it defines 10 tasks. Each task has a different value for the `Frame` parameter. To run a task:

1. All of the variable references in the `data` property of the embedded file are expanded, for example `--render-frame 1`.
2. The contents of the `data` property is written to a file in the session working directory on disk.
3. The task's `onRun` command resolves to `bash location of embedded file` and then runs.

For more information about embedded files, sessions, and path-mapped locations, see the [Open Job Description specification](#) on the GitHub website.

There are more examples of job templates in the [deadline-cloud-samples repository](#) and the [OpenJD specifications repository](#) on the GitHub website.

Task chunking for job templates

Render and simulation jobs often spend more time loading the application and scene file than processing each frame. When a job runs one frame per task, that load time repeats for every frame. *Task chunking* groups multiple tasks into a single unit of work called a *chunk*, so the application loads once for each chunk instead of once for each frame. Chunking helps most when the load time is long relative to the time to process each frame, such as a scene that loads for minutes and then renders each frame in seconds.

If you're migrating from Deadline 10, task chunking replaces the frames per task setting, also called chunk size, in the Deadline 10 submitters.

Task chunking is an extension to Open Job Description (OpenJD) named TASK_CHUNKING. In a job template that uses the extension, you define a task parameter with type CHUNK[INT]. When the job runs, Deadline Cloud dispatches a range of values to each task instead of a single value, and the `{{Task.Param.Frame}}` variable in your script expands to a range expression such as 1-10. Your script passes that range to the application. For the full specification, see [RFC 0001: Task Chunking](#) on the GitHub website.

To use task chunking, you make two decisions:

- **Range constraint** – Choose CONTIGUOUS when your application accepts start and end frame arguments. Every chunk is then a consecutive range such as 1-10. Choose NONCONTIGUOUS when your application accepts arbitrary frame lists. Chunks can then cover sparse frame sets such as 1-3, 5, 7-20:2, which is useful when rendering pick-up frames.
- **Chunk size** – The defaultTaskCount field sets a fixed number of frames for each chunk. If chunks are too small, load time dominates and the job runs inefficiently. If chunks are too large, the job can't balance work across a larger fleet. To avoid tuning the value by hand, set the optional targetRuntimeSeconds field. Deadline Cloud then adjusts chunk sizes toward that target runtime based on the observed runtimes of completed chunks.

Chunk size also caps parallelism. A step produces one task for each chunk, which is approximately the frame count divided by the chunk size, and each task runs on one worker. The following table

shows how the chunk size limits the number of workers that can process a 100-frame job in parallel.

Frames in job	Chunk size	Tasks (maximum parallel workers)
100	1	100
100	10	10
100	50	2

If the chunk size is larger than the frame count divided by the number of available workers, some workers stay idle. When you set `targetRuntimeSeconds`, Deadline Cloud balances this trade-off for you by adjusting chunk sizes toward the target runtime.

Chunked jobs differ from one-frame-per-task jobs in two ways. In the Deadline Cloud monitor, the step's task list shows one task for each chunk, identified by its frame range. When you download the output for a single task in a chunked job, Deadline Cloud downloads the output for the entire chunk. For example, if Deadline Cloud processed frames 1-10 together as a chunk, downloading the output for frame 3 downloads all of frames 1-10. Downloading chunked output requires `deadline-cloud` version 0.53.3 or later.

For more information about task chunking, see the following topics:

- [Add task chunking to a job template](#) – Convert an existing job template to use chunking.
- [CHUNK\[INT\] task parameter reference](#) – Look up every field of the `CHUNK[INT]` parameter.
- [Group frames into chunks with task chunking on Deadline Cloud](#) – Run ready-made chunking samples.

Add task chunking to a job template

The following procedure converts a job template that renders one frame per task into one that renders chunks of frames. For background on chunking and how to choose a chunk size, see [Task chunking for job templates](#).

Before you begin, you need the following:

- A job bundle with a job template that defines a frame task parameter. For more information, see [Job template elements for job bundles](#).
- A queue associated with a fleet that supports chunking. Service-managed fleets always run a compatible worker agent. Customer-managed fleets require worker agent version 0.28.21 or later on the workers.

To add task chunking to a job template

1. Add the TASK_CHUNKING extension to the top of your `template.yaml` file:

```
specificationVersion: 'jobtemplate-2023-09'
extensions:
  - TASK_CHUNKING
```

If you skip this step, job creation fails with the error `The CHUNK[INT] task parameter requires the TASK_CHUNKING extension.`

2. In the step's `parameterSpace`, change the frame task parameter type from `INT` to `CHUNK[INT]` and add the `chunks` property:

```
taskParameterDefinitions:
  - name: Frame
    type: CHUNK[INT]
    range: "{{Param.Frames}}"
    chunks:
      defaultTaskCount: 10
      rangeConstraint: CONTIGUOUS
```

For all of the available fields, see [CHUNK\[INT\] task parameter reference](#).

3. Update the step's script to accept a frame range instead of a single frame. With `rangeConstraint: CONTIGUOUS`, the `{{Task.Param.Frame}}` variable always expands to a range in `start-end` format, so the script can split it with the `cut` command:

```
START_FRAME="$(echo '{{Task.Param.Frame}}' | cut -d- -f1)"
END_FRAME="$(echo '{{Task.Param.Frame}}' | cut -d- -f2)"
```

With `rangeConstraint: NONCONTIGUOUS`, the variable expands to an arbitrary range expression such as `1-3, 5, 7-20`: 2. Transform the expression into the syntax that your

application accepts. For a working transformation, see the [task chunking samples](#) on the GitHub website.

4. Submit the job bundle with the Deadline Cloud CLI:

```
deadline bundle submit my-job-bundle
```

5. To verify that chunking is in effect, open the job in the Deadline Cloud monitor and view the step's task list. Each task represents one chunk and shows a frame range such as 1-10 instead of a single frame number.

The following complete job template renders a Blender animation in chunks of 10 frames:

```
specificationVersion: 'jobtemplate-2023-09'
extensions:
  - TASK_CHUNKING
name: Blender Render with Contiguous Chunking
parameterDefinitions:
  - name: BlenderSceneFile
    type: PATH
    objectType: FILE
    dataFlow: IN
  - name: Frames
    type: STRING
    default: "1-100"
  - name: OutputDir
    type: PATH
    objectType: DIRECTORY
    dataFlow: OUT
    default: "./output"
steps:
  - name: RenderBlender
    parameterSpace:
      taskParameterDefinitions:
        - name: Frame
          type: CHUNK[INT]
          range: "{{Param.Frames}}"
          chunks:
            defaultTaskCount: 10
            rangeConstraint: CONTIGUOUS
    script:
      actions:
        onRun:
```

```

    command: bash
    args: ["{{Task.File.Run}}"]
  embeddedFiles:
  - name: Run
    type: TEXT
    data: |
      set -xeuo pipefail

      mkdir -p '{{Param.OutputDir}}'

      # Parse the chunk range (e.g., "1-10") into start and end frames
      START_FRAME="$(echo '{{Task.Param.Frame}}' | cut -d- -f1)"
      END_FRAME="$(echo '{{Task.Param.Frame}}' | cut -d- -f2)"

      blender --background '{{Param.BlenderSceneFile}}' \
        --render-output '{{Param.OutputDir}}/output_####' \
        --render-format PNG \
        --use-extension 1 \
        -s "$START_FRAME" \
        -e "$END_FRAME" \
        --render-anim

```

In this example, Deadline Cloud divides the 100 frames into chunks such as 1-10 and 11-20, and each task renders its chunk with a single Blender invocation.

For more information, see the following topics:

- [Group frames into chunks with task chunking on Deadline Cloud](#) – Ready-to-submit chunking samples, including a non-contiguous variant.
- [How to submit a job to Deadline Cloud](#) – Submit the job bundle to your queue.

CHUNK[INT] task parameter reference

The `CHUNK[INT]` task parameter type defines an integer task parameter that Deadline Cloud processes as chunks instead of as individual values. A chunked task parameter has the following structure:

```

taskParameterDefinitions:
  - name: identifier
    type: CHUNK[INT]
    range: integer range expression

```

```
chunks:
  defaultTaskCount: integer
  rangeConstraint: CONTIGUOUS | NONCONTIGUOUS
  targetRuntimeSeconds: integer
```

The following table describes each field.

Field	Required	Description
range	Yes	The full set of integer values to process, as a list of integers or an integer range expression such as "1-100" or "1,3,7-20:2" . Accepts a job parameter reference such as "{{Param.Frames}}" .
chunks.defaultTaskCount	Yes	The number of values to combine into a single chunk. The maximum value is 150. Accepts a job parameter reference such as "{{Param.ChunkSize}}" .
chunks.rangeConstraint	Yes	The shape of each chunk. With CONTIGUOUS , every chunk is a consecutive range such as 1-10. With NONCONTIGUOUS , a chunk can be an arbitrary set such as 1,3,7-10.
chunks.targetRuntimeSeconds	No	The target runtime for each chunk, in seconds. When set, Deadline Cloud adjusts chunk sizes toward the target based on the observed runtimes of completed chunks. When not set, every chunk uses defaultTaskCount values.

The CHUNK[INT] type has the following constraints:

- The job template must list TASK_CHUNKING in its extensions. Without the extension, job creation fails with the error The CHUNK[INT] task parameter requires the TASK_CHUNKING extension.
- A step's parameter space can contain at most one CHUNK[INT] parameter. When a chunk runs, all other task parameters take on a single value while the chunked parameter takes a set of values.

In a step's script, the `{{Task.Param.name}}` variable for a chunked parameter expands to an integer range expression string. With `rangeConstraint: CONTIGUOUS`, the expression is always in *start-end* format, such as 1-1 or 11-20. With `rangeConstraint: NONCONTIGUOUS`, the expression can be any integer range expression, such as 1-3, 5, 7-20:2.

For more information, see the following resources:

- [RFC 0001: Task Chunking](#) on the GitHub website – The full specification of the extension.
- [Add task chunking to a job template](#) – Convert a job template to use chunking.

Parameter values elements for job bundles

You can use the parameters file to set the values of some of the job parameters in the job template or [CreateJob](#) operation request arguments in the job bundle so that you don't need to set values when submitting a job. The UI for job submission enables you to modify these values.

You can define the job template in either YAML format (`parameter_values.yaml`) or JSON format (`parameter_values.json`). The examples in this section are shown in YAML format.

In YAML, the format of the file is:

```
parameterValues:
- name: <string>
  value: <integer>, <float>, or <string>
- name: <string>
  value: <integer>, <float>, or <string>ab
... repeating as necessary
```

Each element of the `parameterValues` list must be one of the following:

- A job parameter defined in the job template.
- A job parameter defined in a queue environment for the queue that you submit the job to..
- A special parameter passed to the `CreateJob` operation when creating a job.
 - `deadline:priority` – The value must be an integer. It is passed to the `CreateJob` operation as the [priority](#) parameter.
 - `deadline:targetTaskRunStatus` – The value must be a string. It is passed to the `CreateJob` operation as the [targetTaskRunStatus](#) parameter.

- `deadline:maxFailedTasksCount` – The value must be an integer. It is passed to the `CreateJob` operation as the [maxFailedTasksCount](#) parameter.
- `deadline:maxRetriesPerTask` – The value must be an integer. It is passed to the `CreateJob` operation as the [maxRetriesPerTask](#) parameter.
- `deadline:maxWorkerCount` – The value must be an integer. It is passed to the `CreateJob` operation as the [maxWorkerCount](#) parameter.

A job template is always a template rather than a specific job to run. A parameter values file enables a job bundle to either act as a template if some parameters don't have values defined in this file, or as a specific job submission if all parameters have values.

For example, the [blender_render sample](#) on the GitHub website doesn't have a parameters file and its job template defines parameters with no default values. This template must be used as a template to create jobs. After you create a job using this job bundle, Deadline Cloud writes a new job bundle to the job history directory.

For example, when you submit a job with the following command:

```
deadline bundle gui-submit blender_render/
```

The new job bundle contains a `parameter_values.yaml` file that contains the specified parameters:

```
% cat ~/.deadline/job_history/(default\)/2024-06/2024-06-20-01-JobBundle-Demo/
parameter_values.yaml
parameterValues:
- name: deadline:targetTaskRunStatus
  value: READY
- name: deadline:maxFailedTasksCount
  value: 10
- name: deadline:maxRetriesPerTask
  value: 5
- name: deadline:priority
  value: 75
- name: BlenderSceneFile
  value: /private/tmp/bundle_demo/bmw27_cpu.blend
- name: Frames
  value: 1-10
- name: OutputDir
```

```
value: /private/tmp/bundle_demo/output
- name: OutputPattern
  value: output_####
- name: Format
  value: PNG
- name: CondaPackages
  value: blender
- name: RezPackages
  value: blender
```

You can create the same job with the following command:

```
deadline bundle submit ~/.deadline/job_history/\(default\) /2024-06/2024-06-20-01-
JobBundle-Demo/
```

Note

The job bundle that you submit is saved to your job history directory. You can find the location of that directory with the following command:

```
deadline config get settings.job_history_dir
```

Asset references elements for job bundles

You can use Deadline Cloud [job attachments](#) to transfer files back and forth between your workstation and Deadline Cloud. The asset reference file lists input files and directories, as well as output directories for your attachments. If you don't list all of the files and directories in this file, you can select them when you submit a job with the `deadline bundle gui-submit` command.

This file has no effect if you are not using job attachments.

You can define the job template in either YAML format (`asset_references.yaml`) or JSON format (`asset_references.json`). The examples in this section are shown in YAML format.

In YAML, the format of the file is:

```
assetReferences:
  inputs:
```

```
# Filenames on the submitting workstation whose file contents are needed as
# inputs to run the job.
filenames:
- list of file paths
# Directories on the submitting workstation whose contents are needed as inputs
# to run the job.
directories:
- list of directory paths

outputs:
# Directories on the submitting workstation where the job writes output files
# if running locally.
directories:
- list of directory paths

# Paths referenced by the job, but not necessarily input or output.
# Use this if your job uses the name of a path in some way, but does not explicitly
need
# the contents of that path.
referencedPaths:
- list of directory paths
```

When selecting the input or output file to upload to Amazon S3, Deadline Cloud compares the file path against the paths listed in your storage profiles. Each SHARED-type file system location in a storage profile abstracts a network file share that is mounted on your workstations and worker hosts. Deadline Cloud uploads only files that are not on one of these file shares.

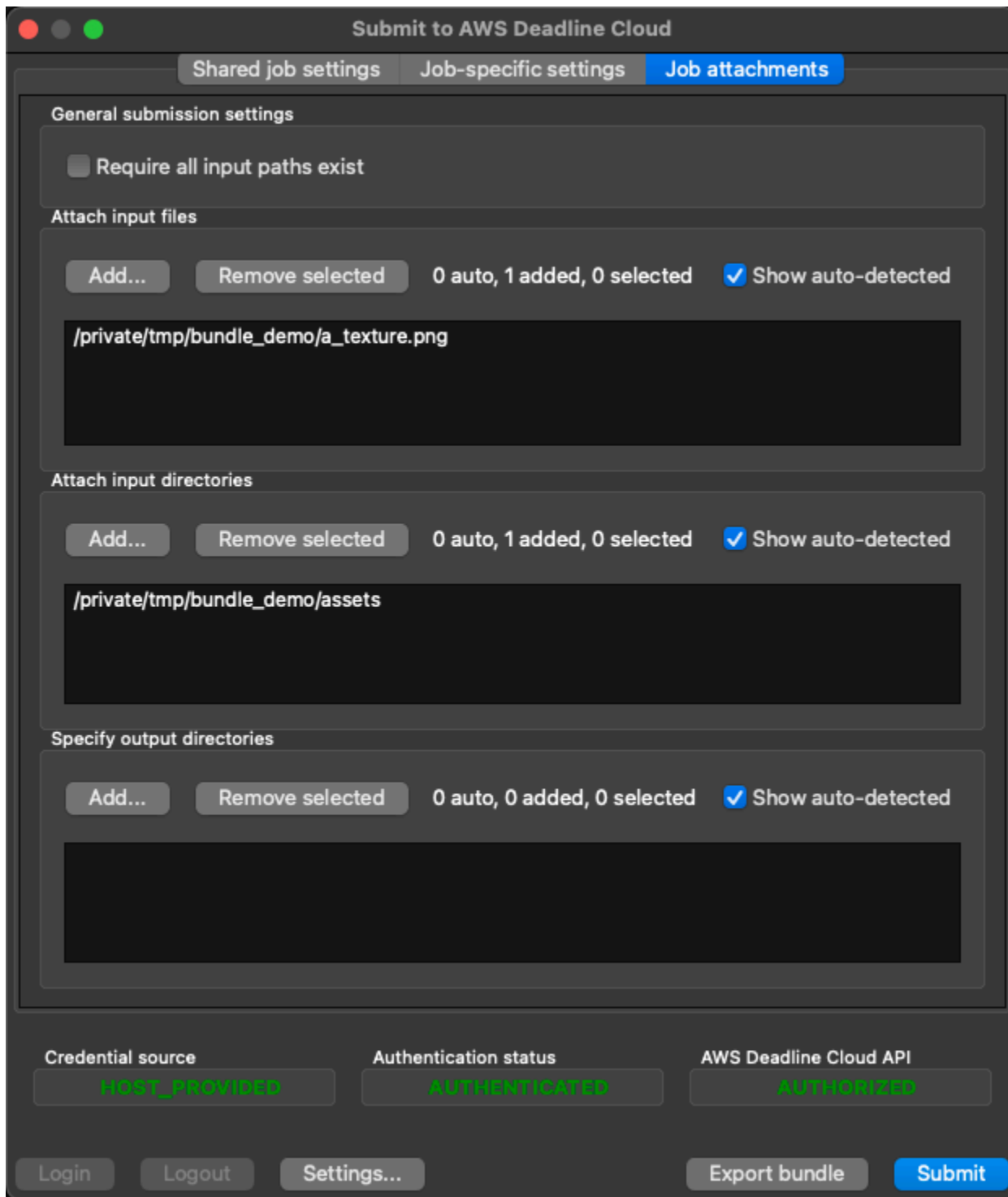
For more information about creating and using storage profiles, see [Shared storage in Deadline Cloud](#) in the *AWS Deadline Cloud User Guide*.

Example- The asset reference file created by the Deadline Cloud GUI

Use the following command to submit a job using the [blender_render sample](#) on the GitHub website:

```
deadline bundle gui-submit blender_render/
```

Add some additional files to the job on the **Job attachments** tab:



After you submit the job, you can look at the `asset_references.yaml` file in the job bundle in the job history directory to see the assets in the YAML file:

```
% cat ~/.deadline/job_history/(default\)/2024-06/2024-06-20-01-JobBundle-Demo/
asset_references.yaml
```



```
assetReferences:
  inputs:
    filenames:
      - /private/tmp/bundle_demo/a_texture.png
    directories:
      - /private/tmp/bundle_demo/assets
  outputs:
    directories: []
  referencedPaths: []
```

Using files in your jobs

Many of the jobs that you submit to AWS Deadline Cloud have input and output files. Your input files and output directories may be located on a combination of shared filesystems and local drives. Jobs need to locate the content in those locations. Deadline Cloud provides two features, [job attachments](#) and [storage profiles](#) that work together to help your jobs locate the files that they need.

Job attachments offer several benefits

- Move files between hosts using Amazon S3
- Transfer files from your work station to worker hosts and vice versa
- Available for jobs in queues where you enable the feature
- Primarily used with service-managed fleets, but also compatible with customer-managed fleets.

Use storage profiles to map the layout of shared filesystem locations on your workstation and worker hosts. This mapping helps your jobs locate shared files and directories when their locations differ between your workstation and worker hosts, such as cross-platform setups with Windows-based workstations and Linux-based worker hosts. Storage profile's map of your filesystem configuration is also used by job attachments to identify the files it needs to shuttle between hosts through Amazon S3.

If you are not using job attachments, and you don't need to remap file and directory locations between workstations and worker hosts then you don't need to model your fileshares with storage profiles.

Topics

- [Sample project infrastructure](#)

- [Storage profiles and path mapping](#)

Sample project infrastructure

To demonstrate using job attachments and storage profiles, set up a test environment with two separate projects. You can use the Deadline Cloud console to create the test resources.

1. If you haven't already, create a test farm. To create a farm, follow the procedure in [Create a farm](#).
2. Create two queues for jobs in each of the two projects. To create queues, follow the procedure in [Create a queue](#).
 - a. Create the first queue called **Q1**. Use the following configuration, use the defaults for all other items.
 - For job attachments, choose **Create a new Amazon S3 bucket**.
 - Select **Enable association with customer-managed fleets**.
 - For the run as user, enter **jobuser** for both the POSIX user and group.
 - For the queue service role, create a new role named **AssetDemoFarm-Q1-Role**
 - Clear the default conda queue environment checkbox.
 - b. Create the second queue called **Q2**. Use the following configuration, use the defaults for all other items.
 - For job attachments, choose **Create a new Amazon S3 bucket**.
 - Select **Enable association with customer-managed fleets**.
 - For the run as user, enter **jobuser** for both the POSIX user and group.
 - For the queue service role, create a new role named **AssetDemoFarm-Q2-Role**
 - Clear the default conda queue environment checkbox.
3. Create a single customer-managed fleet that runs the jobs from both queues. To create the fleet, follow the procedure in [Create a customer-managed fleet](#). Use the following configuration:
 - For **Name**, use **DemoFleet**.
 - For **Fleet type** choose **Customer managed**
 - For **Fleet service role**, create a new role named **AssetDemoFarm-Fleet-Role**.

- Don't associate the fleet with any queues.

The test environment assumes that there are three file systems shared between hosts using network file shares. In this example, the locations have the following names:

- FSCommon - contains input job assets that are common to both projects.
- FS1 - contains input and output job assets for project 1.
- FS2 - contains input and output job assets for project 2.

The test environment also assumes that there are three workstations, as follows:

- WSA11 - A Linux-based workstation used by developers for all projects. The shared file system locations are:
 - FSCommon: /shared/common
 - FS1: /shared/projects/project1
 - FS2: /shared/projects/project2
- WS1 - A Windows-based workstation used for project 1. The shared file system locations are:
 - FSCommon: S:\
 - FS1: Z:\
 - FS2: Not available
- WS1 - A macOS-based workstation used for project 2. The shared file system locations are:
 - FSCommon: /Volumes/common
 - FS1: Not available
 - FS2: /Volumes/projects/project2

Finally, define the shared file system locations for the workers in your fleet. The examples that follow refer to this configuration as `WorkerConfig`. The shared locations are:

- FSCommon: /mnt/common
- FS1: /mnt/projects/project1
- FS2: /mnt/projects/project2

You don't need to set up any shared file systems, workstations, or workers that match this configuration. The shared locations don't need to exist for the demonstration.

Storage profiles and path mapping

Use storage profiles to model the file systems on your workstation and worker hosts. Each storage profile describes the operating system and file system layout of one of your system configurations. This topic describes how to use storage profiles to model the file system configurations of your hosts so Deadline Cloud can generate path mapping rules for your jobs, and how those path mapping rules are generated from your storage profiles.

When you submit a job to Deadline Cloud you can provide an optional storage profile ID for the job. This storage profile describes the submitting workstation's file system. It describes the original file system configuration that the file paths in the job template use.

You can also associate a storage profile with a fleet. The storage profile describes the file system configuration of all worker hosts in the fleet. If you have workers with different file system configuration, those workers must be assigned to a different fleet in your farm.

Path mapping rules describe how paths should be remapped from how they are specified in the job to the path's actual location on a worker host. Deadline Cloud compares the file system configuration described in a job's storage profile with the storage profile of the fleet that is running the job to derive these path mapping rules.

Topics

- [Model shared file system locations with storage profiles](#)
- [Configure storage profiles for fleets](#)
- [Configure storage profiles for queues](#)
- [Derive path mapping rules from storage profiles](#)

Model shared file system locations with storage profiles

A storage profile models the file system configuration of one of your host configurations. There are four different host configurations in the [sample project infrastructure](#). In this example you create a separate storage profile for each. You can create a storage profile using any of the following:

- [CreateStorageProfile API](#)

- [AWS::Deadline::StorageProfile](#) CloudFormation resource
- [AWS console](#)

A storage profile is made up of a list of file system locations that each tell Deadline Cloud the location and type of a file system location that is relevant for jobs submitted from or run on a host. A storage profile should only model the locations that are relevant for jobs. For example, the shared FSCommon location is located on workstation WS1 at S:\, so the corresponding file system location is:

```
{
  "name": "FSCommon",
  "path": "S:\\",
  "type": "SHARED"
}
```

Use the following commands to create the storage profile for workstation configurations WS1, WS2, and WS3 and the worker configuration WorkerConfig using the [AWS CLI](#) in [AWS CloudShell](#):

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff

aws deadline create-storage-profile --farm-id $FARM_ID \
  --display-name WSA11 \
  --os-family LINUX \
  --file-system-locations \
  '[
    {"name": "FSCommon", "type": "SHARED", "path": "/shared/common"},
    {"name": "FS1", "type": "SHARED", "path": "/shared/projects/project1"},
    {"name": "FS2", "type": "SHARED", "path": "/shared/projects/project2"}
  ]'

aws deadline create-storage-profile --farm-id $FARM_ID \
  --display-name WS1 \
  --os-family WINDOWS \
  --file-system-locations \
  '[
    {"name": "FSCommon", "type": "SHARED", "path": "S:\\"},
    {"name": "FS1", "type": "SHARED", "path": "Z:\\"}
  ]'

aws deadline create-storage-profile --farm-id $FARM_ID \
```

```
--display-name WS2 \
--os-family MACOS \
--file-system-locations \
'[
  {"name": "FSCommon", "type":"SHARED", "path":"/Volumes/common"},
  {"name": "FS2", "type":"SHARED", "path":"/Volumes/projects/project2"}
]'

aws deadline create-storage-profile --farm-id $FARM_ID \
--display-name WorkerCfg \
--os-family LINUX \
--file-system-locations \
'[
  {"name": "FSCommon", "type":"SHARED", "path":"/mnt/common"},
  {"name": "FS1", "type":"SHARED", "path":"/mnt/projects/project1"},
  {"name": "FS2", "type":"SHARED", "path":"/mnt/projects/project2"}
]'
```

Note

You must refer to the file system locations in your storage profiles using the same values for the name property across all storage profiles in your farm. Deadline Cloud compares the names to determine that file system locations from different storage profiles are referring to the same location when generating path mapping rules.

Configure storage profiles for fleets

You can configure a fleet to include a storage profile that models the file system locations on all workers in the fleet. The host file system configuration of all workers in a fleet must match their fleet's storage profile. Workers with different file system configurations must be in separate fleets.

To set your fleet's configuration to use the WorkerConfig storage profile use the [AWS CLI](#) in [AWS CloudShell](#):

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of FLEET_ID to your fleet's identifier
FLEET_ID=fleet-00112233445566778899aabbccddeeff
# Change the value of WORKER_CFG_ID to your storage profile named WorkerConfig
WORKER_CFG_ID=sp-00112233445566778899aabbccddeeff
```

```

FLEET_WORKER_MODE=$( \
  aws deadline get-fleet --farm-id $FARM_ID --fleet-id $FLEET_ID \
    --query 'configuration.customerManaged.mode' \
)
FLEET_WORKER_CAPABILITIES=$( \
  aws deadline get-fleet --farm-id $FARM_ID --fleet-id $FLEET_ID \
    --query 'configuration.customerManaged.workerCapabilities' \
)

aws deadline update-fleet --farm-id $FARM_ID --fleet-id $FLEET_ID \
  --configuration \
  "{
    \"customerManaged\": {
      \"storageProfileId\": \"$WORKER_CFG_ID\",
      \"mode\": $FLEET_WORKER_MODE,
      \"workerCapabilities\": $FLEET_WORKER_CAPABILITIES
    }
  }"

```

Configure storage profiles for queues

A queue's configuration includes a list of case-sensitive names of the shared file system locations that jobs submitted to the queue require access to. For example, jobs submitted to queue Q1 require file system locations FSCommon and FS1. Jobs submitted to queue Q2 require file system locations FSCommon and FS2.

To set the queue's configurations to require these file system locations, use the following script:

```

# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff
# Change the value of QUEUE2_ID to queue Q2's identifier
QUEUE2_ID=queue-00112233445566778899aabbccddeeff

aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID \
  --required-file-system-location-names-to-add FSCommon FS1

aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE2_ID \
  --required-file-system-location-names-to-add FSCommon FS2

```

A queue's configuration also includes a list of allowed storage profiles that applies to jobs submitted to and fleets associated with that queue. Only storage profiles that define file system locations for all of the required file system locations for the queue are allowed in the queue's list of allowed storage profiles.

A job fails if you submit it with a storage profile that isn't in the list of allowed storage profiles for the queue. You can always submit a job with no storage profile to a queue. The workstation configurations labeled WSA11 and WS1 both have the required file system locations (FSCommon and FS1) for queue Q1. They need to be allowed to submit jobs to the queue. Similarly, workstation configurations WSA11 and WS2 meet the requirements for queue Q2. They need to be allowed to submit jobs to that queue. Update both queue configurations to allow jobs to be submitted with these storage profiles using the following script:

```
# Change the value of WSALL_ID to the identifier of the WSAll storage profile
WSALL_ID=sp-00112233445566778899aabbccddeeff
# Change the value of WS1 to the identifier of the WS1 storage profile
WS1_ID=sp-00112233445566778899aabbccddeeff
# Change the value of WS2 to the identifier of the WS2 storage profile
WS2_ID=sp-00112233445566778899aabbccddeeff

aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID \
  --allowed-storage-profile-ids-to-add $WSALL_ID $WS1_ID

aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE2_ID \
  --allowed-storage-profile-ids-to-add $WSALL_ID $WS2_ID
```

If you add the WS2 storage profile to the list of allowed storage profiles for queue Q1 it fails:

```
$ aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID \
  --allowed-storage-profile-ids-to-add $WS2_ID
```

```
An error occurred (ValidationException) when calling the UpdateQueue operation: Storage
profile id: sp-00112233445566778899aabbccddeeff does not have required file system
location: FS1
```

The reason is that the WS2 storage profile doesn't contain a definition for the file system location named FS1 that queue Q1 requires.

Associating a fleet that is configured with a storage profile that is not in the queue's list of allowed storage profiles also fails. For example:


```
$ aws deadline create-queue-fleet-association --farm-id $FARM_ID \  
--fleet-id $FLEET_ID \  
--queue-id $QUEUE1_ID
```

An error occurred (ValidationException) when calling the CreateQueueFleetAssociation operation: Mismatch between storage profile ids.

To fix the error, add the storage profile named `WorkerConfig` to the list of allowed storage profiles for both queue Q1 and queue Q2. Then, associate the fleet with these queues so that workers in the fleet can run jobs from both queues.

```
# Change the value of FLEET_ID to your fleet's identifier  
FLEET_ID=fleet-00112233445566778899aabbccddeeff  
# Change the value of WORKER_CFG_ID to your storage profile named WorkerCfg  
WORKER_CFG_ID=sp-00112233445566778899aabbccddeeff  
  
aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID \  
--allowed-storage-profile-ids-to-add $WORKER_CFG_ID  
  
aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE2_ID \  
--allowed-storage-profile-ids-to-add $WORKER_CFG_ID  
  
aws deadline create-queue-fleet-association --farm-id $FARM_ID \  
--fleet-id $FLEET_ID \  
--queue-id $QUEUE1_ID  
  
aws deadline create-queue-fleet-association --farm-id $FARM_ID \  
--fleet-id $FLEET_ID \  
--queue-id $QUEUE2_ID
```

Derive path mapping rules from storage profiles

Path mapping rules describe how paths should be remapped from the job to the path's actual location on a worker host. When a task is running on a worker, the storage profile from the job is compared to the storage profile of the worker's fleet to derive the path mapping rules for the task.

Deadline Cloud creates a mapping rule for each of the required file system locations in the queue's configuration. For example, a job submitted with the `WSA11` storage profile to queue Q1 has the path mapping rules:

- `FSCommon: /shared/common -> /mnt/common`

- FS1: /shared/projects/project1 -> /mnt/projects/project1

Deadline Cloud creates rules for the FSCommon and FS1 file system locations, but not the FS2 file system location even though both the WSAll and WorkerConfig storage profiles define FS2. The reason is that queue Q1's list of required file system locations is ["FSCommon", "FS1"].

You can confirm the path mapping rules available to jobs submitted with a particular storage profile by submitting a job that prints out the path mapping rules file, and then reading the session log after the job has completed. For more information about the file format, see [Path mapping](#) on the GitHub website.

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff
# Change the value of WSALL_ID to the identifier of the WSALL storage profile
WSALL_ID=sp-00112233445566778899aabbccddeeff

aws deadline create-job --farm-id $FARM_ID --queue-id $QUEUE1_ID \
  --priority 50 \
  --storage-profile-id $WSALL_ID \
  --template-type JSON --template \
  '{
    "specificationVersion": "jobtemplate-2023-09",
    "name": "DemoPathMapping",
    "steps": [
      {
        "name": "ShowPathMappingRules",
        "script": {
          "actions": {
            "onRun": {
              "command": "/bin/cat",
              "args": [ "{{Session.PathMappingRulesFile}}" ]
            }
          }
        }
      }
    ]
  }'
```

If you use the Deadline Cloud CLI to submit jobs, its configuration `settings.storage_profile_id` setting sets the storage profile that jobs submitted with the CLI will have. The CLI is available from [PyPI](#) on the PyPI website. To submit jobs with the `WSA11` storage profile, set:

```
deadline config set settings.storage_profile_id $WSALL_ID
```

To run a customer-managed worker as though it is running in the sample infrastructure, follow the procedure in [Run the worker agent](#) in the *Deadline Cloud User Guide* to run a worker with AWS CloudShell. If you followed those instructions before, delete the `~/demoenv-logs` and `~/demoenv-persist` directories first. Also, set the values of the `DEV_FARM_ID` and `DEV_CMF_ID` environment variables that the directions reference as follows before doing so:

```
DEV_FARM_ID=$FARM_ID
DEV_CMF_ID=$FLEET_ID
```

After the job runs, you can see the path mapping rules in the job's log file:

```
cat demoenv-logs/${QUEUE1_ID}/*.log
...
JJSON log results (see below)
...
```

The log contains mapping for both the FS1 and FSCommon file systems. Reformatted for readability, the log entry looks like this:

```
{
  "version": "pathmapping-1.0",
  "path_mapping_rules": [
    {
      "source_path_format": "POSIX",
      "source_path": "/shared/projects/project1",
      "destination_path": "/mnt/projects/project1"
    },
    {
      "source_path_format": "POSIX",
      "source_path": "/shared/common",
      "destination_path": "/mnt/common"
    }
  ]
}
```

```
}
```

You can submit jobs with different storage profiles to see how the path mapping rules change.

Use job attachments to share files

Use *job attachments* to make files not in shared directories available for your jobs, and to capture the output files if they are not written to shared directories. Job attachments uses Amazon S3 to shuttle files between hosts. Files are stored in S3 buckets, and you don't need to upload a file if its content hasn't changed.

You must use job attachments when running jobs on [service-managed fleets](#) because hosts don't share file system locations. Job attachments are also useful with [customer-managed fleets](#) when a job's input or output files stored on a shared network file system, such as when your [job bundle](#) contains shell or Python scripts.

When you submit a job bundle with either the [Deadline Cloud CLI](#) on the PyPI website or a Deadline Cloud submitter, job attachments use the job's storage profile and the queue's required file system locations to identify the input files that are not on a worker host and should be uploaded to Amazon S3 as part of job submission. These storage profiles also help Deadline Cloud identify the output files in worker host locations that must be uploaded to Amazon S3 so that they are available to your workstation.

The job attachments examples use the farm, fleet, queues, and storage profiles configurations from [Sample project infrastructure](#) and [Storage profiles and path mapping](#). You should go through those sections before this one.

In the following examples, you use a sample job bundle as a starting point, then modify it to explore job attachment's functionality. Job bundles are the best way for your jobs to use job attachments. They combine an Open Job Description job template in a directory with additional files that list the files and directories required by jobs using the job bundle. For more information about the [OpenJD specification](#) on the GitHub website, see the OpenJD wiki. For more information about job bundles, see [Open Job Description \(OpenJD\) templates for Deadline Cloud](#).

Submitting files with a job

With Deadline Cloud, you can enable job workflows to access input files that are unavailable in shared file system locations on worker hosts. Job attachments allow rendering jobs to access files residing only on a local workstation drive or a service-managed fleet environment. When

submitting a job bundle, you can include lists of input files and directories required by the job. Deadline Cloud identifies these non-shared files, uploads them from the local machine to Amazon S3, and downloads them to the worker host. It streamlines the process of transferring input assets to render nodes, ensuring all required files are accessible for distributed job execution.

You can specify the files for jobs directly in the job bundle, use parameters in the job template that you provide using environment variables or a script, and use the job's `assets_references` file. You can use one of these methods or a combination of all three. You can specify a storage profile for the bundle for the job so that it only uploads files that have changed on the local workstation.

This section uses an example job bundle from GitHub to demonstrate how Deadline Cloud identifies the files in your job to upload, how those files are organized in Amazon S3, and how they are made available to the worker hosts processing your jobs.

Topics

- [How Deadline Cloud uploads files to Amazon S3](#)
- [How Deadline Cloud chooses the files to upload](#)
- [How jobs find job attachment input files](#)

How Deadline Cloud uploads files to Amazon S3

This example shows how Deadline Cloud uploads files from your workstation or worker host to Amazon S3 so that they can be shared. It uses a sample job bundle from the [deadline-cloud-samples repository](#) on the GitHub website and the [Deadline Cloud CLI](#) on the PyPI website to submit jobs.

Start by cloning the `deadline-cloud-samples` repository into your [AWS CloudShell](#) environment, then copy the `job_attachments_devguide` job bundle into your home directory:

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
cp -r deadline-cloud-samples/job_bundles/job_attachments_devguide ~/
```

Install the Deadline Cloud CLI to submit job bundles:

```
pip install deadline --upgrade
```

The `job_attachments_devguide` job bundle has a single step with a task that runs a bash shell script whose file system location is passed as a job parameter. The job parameter's definition is:

```
...
- name: ScriptFile
  type: PATH
  default: script.sh
  dataFlow: IN
  objectType: FILE
...
```

The dataFlow property's IN value tells job attachments that the value of the ScriptFile parameter is an input to the job. The value of the default property is a relative location to the job bundle's directory, but it can also be an absolute path. This parameter definition declares the script.sh file in the job bundle's directory as an input file required for the job to run.

Next, make sure that the Deadline Cloud CLI does not have a storage profile configured then submit the job to queue Q1:

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff

deadline config set settings.storage_profile_id ''

deadline bundle submit --farm-id $FARM_ID --queue-id $QUEUE1_ID
job_attachments_devguide/
```

The output from the Deadline Cloud CLI after this command is run looks like:

```
Submitting to Queue: Q1
...
Hashing Attachments  [#####] 100%
Hashing Summary:
  Processed 1 file totaling 39.0 B.
  Skipped re-processing 0 files totaling 0.0 B.
  Total processing time of 0.0327 seconds at 1.19 KB/s.

Uploading Attachments [#####] 100%
Upload Summary:
  Processed 1 file totaling 39.0 B.
  Skipped re-processing 0 files totaling 0.0 B.
  Total processing time of 0.25639 seconds at 152.0 B/s.
```

```
Waiting for Job to be created...
Submitted job bundle:
  job_attachments_devguide/
Job creation completed successfully
job-74148c13342e4514b63c7a7518657005
```

When you submit the job, Deadline Cloud first hashes the `script.sh` file and then it uploads it to Amazon S3.

Deadline Cloud treats the S3 bucket as content-addressable storage. Files are uploaded to S3 objects. The object name is derived from a hash of the file's contents. If two files have identical contents they have the same hash value regardless of where the files are located or what they are named. This content-addressable storage enables Deadline Cloud to avoid uploading a file if it is already available.

You can use the [AWS CLI](#) to see the objects that were uploaded to Amazon S3:

```
# The name of queue `Q1`'s job attachments S3 bucket
Q1_S3_BUCKET=$(
  aws deadline get-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID \
    --query 'jobAttachmentSettings.s3BucketName' | tr -d '"'
)

aws s3 ls s3://$Q1_S3_BUCKET --recursive
```

Two objects were uploaded to S3:

- DeadlineCloud/Data/87cb19095dd5d78fc56384ef0e6241.xxh128 – The contents of `script.sh`. The value 87cb19095dd5d78fc56384ef0e6241 in the object key is the hash of the file's contents, and the extension xxh128 indicates that the hash value was calculated as a 128 bit hash using the [xxHash algorithm](#) on the xxHash website.
- DeadlineCloud/Manifests/<farm-id>/<queue-id>/Inputs/<guid>/a1d221c7fd97b08175b3872a37428e8c_input – The manifest object for the job submission. The values <farm-id>, <queue-id>, and <guid> are your farm identifier, queue identifier, and a random hexadecimal value. The value a1d221c7fd97b08175b3872a37428e8c in this example is a hash value calculated from the string `/home/cloudshell-user/job_attachments_devguide`, the directory where `script.sh` is located.

The manifest object contains the information for the input files on a specific root path uploaded to S3 as part of the job's submission. Download this manifest file (`aws s3 cp s3://$Q1_S3_BUCKET/<objectname>`). Its contents are similar to:

```
{
  "hashAlg": "xxh128",
  "manifestVersion": "2023-03-03",
  "paths": [
    {
      "hash": "87cb19095dd5d78fc56384ef0e6241",
      "mtime": 1721147454416085,
      "path": "script.sh",
      "size": 39
    }
  ],
  "totalSize": 39
}
```

This indicates that the file `script.sh` was uploaded, and the hash of that file's contents is `87cb19095dd5d78fc56384ef0e6241`. This hash value matches the value in the object name `DeadlineCloud/Data/87cb19095dd5d78fc56384ef0e6241.xhx128`. It is used by Deadline Cloud to know which object to download for this file's contents.

For the full schema for the manifest file, see the [deadline-cloud-job-attachments repository](#) on the GitHub website.

When you use the [CreateJob operation](#) you can set the location of the manifest objects. You can use the [GetJob operation](#) to see the location:

```
{
  "attachments": {
    "file system": "COPIED",
    "manifests": [
      {
        "inputManifestHash": "5b0db3d311805ea8de7787b64cbb8b3",
        "inputManifestPath": "<farm-id>/<queue-id>/Inputs/<guid>/a1d221c7fd97b08175b3872a37428e8c_input",
        "rootPath": "/home/cloudshell-user/job_attachments_devguide",
        "rootPathFormat": "posix"
      }
    ]
  },
}
```



```
...  
}
```

How Deadline Cloud chooses the files to upload

The files and directories that job attachments considers for upload to Amazon S3 as inputs to your job are:

- The values of all PATH-type job parameters defined in the job bundle's job template with a dataFlow value of IN or INOUT.
- The files and directories listed as inputs in the job bundle's asset references file.

If you submit a job with no storage profile, all of the files considered for uploading are uploaded. If you submit a job with a storage profile, files are not uploaded to Amazon S3 if they are located in the storage profile's SHARED-type file system locations that are also required file system locations for the queue. These locations are expected to be available on the worker hosts that run the job, so there is no need to upload them to S3.

In this example, you create SHARED file system locations in WSAll in your AWS CloudShell environment and then add files to those file system locations. Use the following command:

```
# Change the value of WSALL_ID to the identifier of the WSAll storage profile  
WSALL_ID=sp-00112233445566778899aabbccddeeff  
  
sudo mkdir -p /shared/common /shared/projects/project1 /shared/projects/project2  
sudo chown -R cloudshell-user:cloudshell-user /shared  
  
for d in /shared/common /shared/projects/project1 /shared/projects/project2; do  
    echo "File contents for $d" > ${d}/file.txt  
done
```

Next, add an asset references file to the job bundle that includes all the files that you created as inputs for the job. Use the following command:

```
cat > ${HOME}/job_attachments_devguide/asset_references.yaml << EOF  
assetReferences:  
  inputs:  
    filenames:  
      - /shared/common/file.txt
```

```
directories:
- /shared/projects/project1
- /shared/projects/project2
EOF
```

Next, configure the Deadline Cloud CLI to submit jobs with the WSAll storage profile, and then submit the job bundle:

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff
# Change the value of WSALL_ID to the identifier of the WSAll storage profile
WSALL_ID=sp-00112233445566778899aabbccddeeff

deadline config set settings.storage_profile_id $WSALL_ID

deadline bundle submit --farm-id $FARM_ID --queue-id $QUEUE1_ID
job_attachments_devguide/
```

Deadline Cloud uploads two files to Amazon S3 when you submit the job. You can download the manifest objects for the job from S3 to see the uploaded files:

```
for manifest in $( \
  aws deadline get-job --farm-id $FARM_ID --queue-id $QUEUE1_ID --job-id $JOB_ID \
    --query 'attachments.manifests[].inputManifestPath' \
    | jq -r '.[ ]'
); do
  echo "Manifest object: $manifest"
  aws s3 cp --quiet s3://$Q1_S3_BUCKET/DeadlineCloud/Manifests/$manifest /dev/stdout |
  jq .
done
```

In this example, there is a single manifest file with the following contents:

```
{
  "hashAlg": "xxh128",
  "manifestVersion": "2023-03-03",
  "paths": [
    {
      "hash": "87cb19095dd5d78fcdf56384ef0e6241",
```

```

        "mtime": 1721147454416085,
        "path": "home/cloudshell-user/job_attachments_devguide/script.sh",
        "size": 39
    },
    {
        "hash": "af5a605a3a4e86ce7be7ac5237b51b79",
        "mtime": 1721163773582362,
        "path": "shared/projects/project2/file.txt",
        "size": 44
    }
],
"totalSize": 83
}

```

Use the [GetJob operation](#) for the manifest to see that the rootPath is "/".

```
aws deadline get-job --farm-id $FARM_ID --queue-id $QUEUE1_ID --job-id $JOB_ID --query
'attachments.manifests[*]'
```

The root path for set of input files is always the longest common subpath of those files. If your job was submitted from Windows instead and there are input files with no common subpath because they were on different drives, you see a separate root path on each drive. The paths in a manifest are always relative to the root path of the manifest, so the input files that were uploaded are:

- /home/cloudshell-user/job_attachments_devguide/script.sh – The script file in the job bundle.
- /shared/projects/project2/file.txt – The file in a SHARED file system location in the WSAll storage profile that is **not** in the list of required file system locations for queue Q1.

The files in file system locations FSCommon (/shared/common/file.txt) and FS1 (/shared/projects/project1/file.txt) are not in the list. The reason is that those file system locations are SHARED in the WSAll storage profile and they both are in the list of required file system locations in queue Q1.

You can see the file system locations considered SHARED for a job that is submitted with a particular storage profile with the [GetStorageProfileForQueue operation](#). To query for storage profile WSAll for queue Q1 use the following command:

```
aws deadline get-storage-profile --farm-id $FARM_ID --storage-profile-id $WSALL_ID
```

```
aws deadline get-storage-profile-for-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID --
storage-profile-id $WSALL_ID
```

How jobs find job attachment input files

For a job to use the files that Deadline Cloud uploads to Amazon S3 using job attachments, your job needs those files available through the file system on the worker hosts. When a session for your job runs on a worker host, Deadline Cloud downloads the input files for the job into a temporary directory on the worker host's local drive and adds path mapping rules for each of the job's root paths to its file system location on the local drive. For more information about sessions, see [How jobs are run](#) on the GitHub website.

For this example, start the Deadline Cloud worker agent in an AWS CloudShell tab. Let any previously submitted jobs finish running, and then delete the job logs from the logs directory:

```
rm -rf ~/demoenv-logs/queue-*
```

The following script modifies the job bundle to show all files in the session's temporary working directory and the contents of the path mapping rules file, and then submits a job with the modified bundle:

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff
# Change the value of WSALL_ID to the identifier of the WSAll storage profile
WSALL_ID=sp-00112233445566778899aabbccddeeff

deadline config set settings.storage_profile_id $WSALL_ID

cat > ~/job_attachments_devguide/script.sh << EOF
#!/bin/bash

echo "Session working directory is: \$(pwd)"
echo
echo "Contents:"
find . -type f
echo
echo "Path mapping rules file: \$1"
jq . \$1
```

```

EOF

cat > ~/job_attachments_devguide/template.yaml << EOF
specificationVersion: jobtemplate-2023-09
name: "Job Attachments Explorer"
parameterDefinitions:
- name: ScriptFile
  type: PATH
  default: script.sh
  dataFlow: IN
  objectType: FILE
steps:
- name: Step
  script:
    actions:
      onRun:
        command: /bin/bash
        args:
          - "{{Param.ScriptFile}}"
          - "{{Session.PathMappingRulesFile}}"
EOF

deadline bundle submit --farm-id $FARM_ID --queue-id $QUEUE1_ID
job_attachments_devguide/

```

You can look at the log of the job's run after it has been run by the worker in your AWS CloudShell environment:

```
cat demoenv-logs/queue-*/session*.log
```

The log shows that the first thing that occurs in the session is the two input files for the job are downloaded to the worker:

```

2024-07-17 01:26:37,824 INFO =====
2024-07-17 01:26:37,825 INFO ----- Job Attachments Download for Job
2024-07-17 01:26:37,825 INFO =====
2024-07-17 01:26:37,825 INFO Syncing inputs using Job Attachments
2024-07-17 01:26:38,116 INFO Downloaded 142.0 B / 186.0 B of 2 files (Transfer rate:
0.0 B/s)
2024-07-17 01:26:38,174 INFO Downloaded 186.0 B / 186.0 B of 2 files (Transfer rate:
733.0 B/s)
2024-07-17 01:26:38,176 INFO Summary Statistics for file downloads:

```

```
Processed 2 files totaling 186.0 B.
Skipped re-processing 0 files totaling 0.0 B.
Total processing time of 0.09752 seconds at 1.91 KB/s.
```

Next is the output from `script.sh` run by the job:

- The input files uploaded when the job was submitted are located under a directory whose name begins with "assetroot" in the session's temporary directory.
- The input files' paths have been relocated relative to the "assetroot" directory instead of relative to the root path for the job's input manifest ("/").
- The path mapping rules file contains an additional rule that remaps "/" to the absolute path of the "assetroot" directory.

For example:

```
2024-07-17 01:26:38,264 INFO Output:
2024-07-17 01:26:38,267 INFO Session working directory is: /sessions/session-5b33f
2024-07-17 01:26:38,267 INFO
2024-07-17 01:26:38,267 INFO Contents:
2024-07-17 01:26:38,269 INFO ./tmp_xdhbsdo.sh
2024-07-17 01:26:38,269 INFO ./tmpdi00052b.json
2024-07-17 01:26:38,269 INFO ./assetroot-3751a/shared/projects/project2/file.txt
2024-07-17 01:26:38,269 INFO ./assetroot-3751a/home/cloudshell-user/
job_attachments_devguide/script.sh
2024-07-17 01:26:38,269 INFO
2024-07-17 01:26:38,270 INFO Path mapping rules file: /sessions/session-5b33f/
tmpdi00052b.json
2024-07-17 01:26:38,282 INFO {
2024-07-17 01:26:38,282 INFO   "version": "pathmapping-1.0",
2024-07-17 01:26:38,282 INFO   "path_mapping_rules": [
2024-07-17 01:26:38,282 INFO     {
2024-07-17 01:26:38,282 INFO       "source_path_format": "POSIX",
2024-07-17 01:26:38,282 INFO       "source_path": "/shared/projects/project1",
2024-07-17 01:26:38,283 INFO       "destination_path": "/mnt/projects/project1"
2024-07-17 01:26:38,283 INFO     },
2024-07-17 01:26:38,283 INFO     {
2024-07-17 01:26:38,283 INFO       "source_path_format": "POSIX",
2024-07-17 01:26:38,283 INFO       "source_path": "/shared/common",
2024-07-17 01:26:38,283 INFO       "destination_path": "/mnt/common"
2024-07-17 01:26:38,283 INFO     },
2024-07-17 01:26:38,283 INFO   ]
2024-07-17 01:26:38,283 INFO }
```

```

2024-07-17 01:26:38,283 INFO      "source_path_format": "POSIX",
2024-07-17 01:26:38,283 INFO      "source_path": "/",
2024-07-17 01:26:38,283 INFO      "destination_path": "/sessions/session-5b33f/
assetroot-3751a"
2024-07-17 01:26:38,283 INFO      }
2024-07-17 01:26:38,283 INFO    ]
2024-07-17 01:26:38,283 INFO  }
```

If the job you submit has multiple manifests with different root paths, the session has a separate "assetroot" directory for each root path. Job attachments groups a job's input files by root path. The root path for a group of files is their longest common subpath. Files on different Windows drives never have a common subpath, so each drive forms at least one separate root path. For example, suppose that you submit a job from Windows with the input files C:\show\project.mb, C:\show\plates\bg.png, and D:\audio\mix.wav. Job attachments places the files in two "assetroot" directories in the session's temporary directory:

```

C:\show\project.mb      -> assetroot-3751a/project.mb
C:\show\plates\bg.png   -> assetroot-3751a/plates/bg.png
D:\audio\mix.wav        -> assetroot-be92f/mix.wav
```

The path mapping rules file contains a separate rule for each root path. For the preceding example, a session running on a Linux-based worker host has the following rules:

```

{
  "version": "pathmapping-1.0",
  "path_mapping_rules": [
    {
      "source_path_format": "WINDOWS",
      "source_path": "C:\\show",
      "destination_path": "/sessions/session-5b33f/assetroot-3751a"
    },
    {
      "source_path_format": "WINDOWS",
      "source_path": "D:\\audio",
      "destination_path": "/sessions/session-5b33f/assetroot-be92f"
    }
  ]
}
```

To find the location of every root path in the session, read the file at `{{Session.PathMappingRulesFile}}` rather than assuming that the session has a single

"assetroot" directory. For more information about the path mapping rules file format, see [Path mapping](#) on the GitHub website.

Job attachments preserves the relative directory structure of the files within each root path. However, it doesn't combine multiple root paths into a single directory tree, and it doesn't modify paths stored inside your files. If an application locates files by using paths that are relative to a project file, those relative references resolve only for files in the same root path as the project file. To keep relative references working, store a project file and the files it references under a common parent directory on the same drive. Alternatively, remap the paths in your job by using the path mapping rules.

If you need to reference the relocated file system location of one of your input files, directories, or file system locations you can either process the path mapping rules file in your job and perform the remapping yourself, or add a PATH type job parameter to the job template in your job bundle and pass the value that you need to remap as the value of that parameter. For example, the following example modifies the job bundle to have one of these job parameters and then submits a job with the file system location /shared/projects/project2 as its value:

```
cat > ~/job_attachments_devguide/template.yaml << EOF
specificationVersion: jobtemplate-2023-09
name: "Job Attachments Explorer"
parameterDefinitions:
- name: LocationToRemap
  type: PATH
steps:
- name: Step
  script:
    actions:
      onRun:
        command: /bin/echo
        args:
        - "The location of {{RawParam.LocationToRemap}} in the session is
{{Param.LocationToRemap}}"
EOF

deadline bundle submit --farm-id $FARM_ID --queue-id $QUEUE1_ID
job_attachments_devguide/ \
-p LocationToRemap=/shared/projects/project2
```

The log file for this job's run contains its output:


```
2024-07-17 01:40:35,283 INFO Output:
2024-07-17 01:40:35,284 INFO The location of /shared/projects/project2 in the session
is /sessions/session-5b33f/assetroot-3751a
```

Getting output files from a job

This example shows how Deadline Cloud identifies the output files that your jobs generate, decides whether to upload those files to Amazon S3, and how you can get those output files on your workstation.

Use the `job_attachments_devguide_output` job bundle instead of the `job_attachments_devguide` job bundle for this example. Start by making a copy of the bundle in your AWS CloudShell environment from your clone of the Deadline Cloud samples GitHub repository:

```
cp -r deadline-cloud-samples/job_bundles/job_attachments_devguide_output ~/
```

The important difference between this job bundle and the `job_attachments_devguide` job bundle is the addition of a new job parameter in the job template:

```
...
parameterDefinitions:
...
- name: OutputDir
  type: PATH
  objectType: DIRECTORY
  dataFlow: OUT
  default: ./output_dir
  description: This directory contains the output for all steps.
...
```

The `dataFlow` property of the parameter has the value `OUT`. Deadline Cloud uses the value of `dataFlow` job parameters with a value of `OUT` or `INOUT` as outputs of your job. If the file system location passed as a value to these kinds of job parameters is remapped to a local file system location on the worker that runs the job, then Deadline Cloud will look for new files at the location and upload those to Amazon S3 as job outputs.

To see how this process works, first start the Deadline Cloud worker agent in an AWS CloudShell tab. Let any previously submitted jobs finish running. Then delete the job logs from the logs directory:

```
rm -rf ~/demoenv-logs/queue-*
```

Next, submit a job with this job bundle. After the worker running in your CloudShell runs, look at the logs:

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff
# Change the value of WSALL_ID to the identifier of the WSAll storage profile
WSALL_ID=sp-00112233445566778899aabbccddeeff

deadline config set settings.storage_profile_id $WSALL_ID

deadline bundle submit --farm-id $FARM_ID --queue-id $QUEUE1_ID ./
job_attachments_devguide_output
```

The log shows that a file was detected as output and uploaded to Amazon S3:

```
2024-07-17 02:13:10,873 INFO -----
2024-07-17 02:13:10,873 INFO Uploading output files to Job Attachments
2024-07-17 02:13:10,873 INFO -----
2024-07-17 02:13:10,873 INFO Started syncing outputs using Job Attachments
2024-07-17 02:13:10,955 INFO Found 1 file totaling 117.0 B in output directory: /
sessions/session-7efa/assetroot-3751a/output_dir
2024-07-17 02:13:10,956 INFO Uploading output manifest to
DeadlineCloud/Manifests/farm-0011/queue-2233/job-4455/step-6677/
task-6677-0/2024-07-17T02:13:10.835545Z_sessionaction-8899-1/
c6808439dfc59f86763aff5b07b9a76c_output
2024-07-17 02:13:10,988 INFO Uploading 1 output file to S3: s3BucketName/DeadlineCloud/
Data
2024-07-17 02:13:11,011 INFO Uploaded 117.0 B / 117.0 B of 1 file (Transfer rate: 0.0
B/s)
2024-07-17 02:13:11,011 INFO Summary Statistics for file uploads:
Processed 1 file totaling 117.0 B.
Skipped re-processing 0 files totaling 0.0 B.
Total processing time of 0.02281 seconds at 5.13 KB/s.
```

The log also shows that Deadline Cloud created a new manifest object in the Amazon S3 bucket configured for use by job attachments on queue Q1. The name of the manifest object is derived from the farm, queue, job, step, task, timestamp, and sessionaction identifiers of the task that

generated the output. Download this manifest file to see where Deadline Cloud placed the output files for this task:

```
# The name of queue `Q1`'s job attachments S3 bucket
Q1_S3_BUCKET=$(
  aws deadline get-queue --farm-id $FARM_ID --queue-id $QUEUE1_ID \
    --query 'jobAttachmentSettings.s3BucketName' | tr -d '"'
)

# Fill this in with the object name from your log
OBJECT_KEY="DeadlineCloud/Manifests/..."

aws s3 cp --quiet s3://$Q1_S3_BUCKET/$OBJECT_KEY /dev/stdout | jq .
```

The manifest looks like:

```
{
  "hashAlg": "xxh128",
  "manifestVersion": "2023-03-03",
  "paths": [
    {
      "hash": "34178940e1ef9956db8ea7f7c97ed842",
      "mtime": 1721182390859777,
      "path": "output_dir/output.txt",
      "size": 117
    }
  ],
  "totalSize": 117
}
```

This shows that the content of the output file is saved to Amazon S3 the same way that job input files are saved. Similar to input files, the output file is stored in S3 with an object name containing the hash of the file and the prefix `DeadlineCloud/Data`.

```
$ aws s3 ls --recursive s3://$Q1_S3_BUCKET | grep 34178940e1ef9956db8ea7f7c97ed842
2024-07-17 02:13:11          117 DeadlineCloud/
Data/34178940e1ef9956db8ea7f7c97ed842.xxx128
```

You can download the output of a job to your workstation using the Deadline Cloud monitor or the Deadline Cloud CLI:

```
deadline job download-output --farm-id $FARM_ID --queue-id $QUEUE1_ID --job-id $JOB_ID
```

The value of the `OutputDir` job parameter in the submitted job is `./output_dir`, so the output are downloaded to a directory called `output_dir` within the job bundle directory. If you specified an absolute path or different relative location as the value for `OutputDir`, then the output files would be downloaded to that location instead.

```
$ deadline job download-output --farm-id $FARM_ID --queue-id $QUEUE1_ID --job-id $JOB_ID
```

```
Downloading output from Job 'Job Attachments Explorer: Output'
```

```
Summary of files to download:
```

```
  /home/cloudshell-user/job_attachments_devguide_output/output_dir/output.txt (1 file)
```

```
You are about to download files which may come from multiple root directories. Here are a list of the current root directories:
```

```
[0] /home/cloudshell-user/job_attachments_devguide_output
```

```
> Please enter the index of root directory to edit, y to proceed without changes, or n to cancel the download (0, y, n) [y]:
```

```
Downloading Outputs [#####] 100%
```

```
Download Summary:
```

```
  Downloaded 1 files totaling 117.0 B.
```

```
  Total download time of 0.14189 seconds at 824.0 B/s.
```

```
  Download locations (total file counts):
```

```
    /home/cloudshell-user/job_attachments_devguide_output (1 file)
```

Using files from a step in a dependent step

This example shows how one step in a job can access the outputs from a step that it depends on in the same job.

To make the outputs of one step available to another, Deadline Cloud adds additional actions to a session to download those outputs before running tasks in the session. You tell it which steps to download the outputs from by declaring those steps as dependencies of the step that needs to use the outputs.

Use the `job_attachments_devguide_output` job bundle for this example. Start by making a copy in your AWS CloudShell environment from your clone of the Deadline Cloud samples GitHub

repository. Modify it to add a dependent step that only runs after the existing step and uses that step's output:

```
cp -r deadline-cloud-samples/job_bundles/job_attachments_devguide_output ~/

cat >> job_attachments_devguide_output/template.yaml << EOF
- name: DependentStep
  dependencies:
  - dependsOn: Step
  script:
    actions:
      onRun:
        command: /bin/cat
        args:
        - "{{Param.OutputDir}}/output.txt"
EOF
```

The job created with this modified job bundle runs as two separate sessions, one for the task in the step "Step" and then a second for the task in the step "DependentStep".

First start the Deadline Cloud worker agent in an CloudShell tab. Let any previously submitted jobs finish running, then delete the job logs from the logs directory:

```
rm -rf ~/demoenv-logs/queue-*
```

Next, submit a job using the modified `job_attachments_devguide_output` job bundle. Wait for it to finish running on the worker in your CloudShell environment. Look at the logs for the two sessions:

```
# Change the value of FARM_ID to your farm's identifier
FARM_ID=farm-00112233445566778899aabbccddeeff
# Change the value of QUEUE1_ID to queue Q1's identifier
QUEUE1_ID=queue-00112233445566778899aabbccddeeff
# Change the value of WSALL_ID to the identifier of the WSAll storage profile
WSALL_ID=sp-00112233445566778899aabbccddeeff

deadline config set settings.storage_profile_id $WSALL_ID

deadline bundle submit --farm-id $FARM_ID --queue-id $QUEUE1_ID ./
job_attachments_devguide_output
```

```
# Wait for the job to finish running, and then:
```

```
cat demoenv-logs/queue-*/session-*
```

In the session log for the task in the step named `DependentStep`, there are two separate download actions run:

```
2024-07-17 02:52:05,666 INFO =====
2024-07-17 02:52:05,666 INFO ----- Job Attachments Download for Job
2024-07-17 02:52:05,667 INFO =====
2024-07-17 02:52:05,667 INFO Syncing inputs using Job Attachments
2024-07-17 02:52:05,928 INFO Downloaded 207.0 B / 207.0 B of 1 file (Transfer rate: 0.0
B/s)
2024-07-17 02:52:05,929 INFO Summary Statistics for file downloads:
Processed 1 file totaling 207.0 B.
Skipped re-processing 0 files totaling 0.0 B.
Total processing time of 0.03954 seconds at 5.23 KB/s.

2024-07-17 02:52:05,979 INFO
2024-07-17 02:52:05,979 INFO =====
2024-07-17 02:52:05,979 INFO ----- Job Attachments Download for Step
2024-07-17 02:52:05,979 INFO =====
2024-07-17 02:52:05,980 INFO Syncing inputs using Job Attachments
2024-07-17 02:52:06,133 INFO Downloaded 117.0 B / 117.0 B of 1 file (Transfer rate: 0.0
B/s)
2024-07-17 02:52:06,134 INFO Summary Statistics for file downloads:
Processed 1 file totaling 117.0 B.
Skipped re-processing 0 files totaling 0.0 B.
Total processing time of 0.03227 seconds at 3.62 KB/s.
```

The first action downloads the `script.sh` file used by the step named "Step." The second action downloads the outputs from that step. Deadline Cloud determines which files to download by using the output manifest generated by that step as an input manifest.

Late in the same log, you can see the output from the step named "DependentStep":

```
2024-07-17 02:52:06,213 INFO Output:
2024-07-17 02:52:06,216 INFO Script location: /sessions/session-5b33f/assetroot-3751a/
script.sh
```

Create resource limits for jobs

Jobs submitted to Deadline Cloud may depend on resources that are shared between multiple jobs. For example, a farm may have more workers than floating licences for a specific resource. Or a shared file server may only be able to serve data to a limited number of workers at the same time. In some cases, one or more jobs can claim all of these resources, causing errors due to unavailable resources when new workers start.

To help solve this, you can use *limits* for these constrained resources. Deadline Cloud accounts for the availability of constrained resources and uses that information to ensure that resources are available as new workers start up so that jobs have a lower likelihood of failing due to unavailable resources.

Limits are created for the entire farm. Jobs submitted to a queue can only acquire limits associated with the queue. If you specify a limit for a job that is not associated with the queue, the job isn't compatible and won't run.

To use a limit, you

- [Create a limit](#)
- [Associate a limit and a queue](#)
- [Submit a job requiring limits](#)

Note

If you run a job that has constrained resources in a queue that is not associated with a limit, that job can consume all of the resources. If you have a constrained resource, make sure that all of the steps in jobs in queues that use the resource are associated with a limit.

For limits defined in a farm, associated with a queue, and specified in a job, one of four things can happen:

- If you create a limit, associate it with a queue, and specify the limit in a job's template, the job runs and uses only the resources defined in the limit.
- If you create a limit, specify it in a job template, but don't associate the limit with a queue, the job is marked incompatible and won't run.

- If you create a limit, don't associate it with a queue, and don't specify the limit in a job's template, the job runs but does not use the limit.
- If you don't use a limit at all, the job runs.

If you associate a limit to multiple queues, the queues share the resources constrained by the limit. For example, if you create a limit of 100, and one queue is using 60 resources, other queues can only use 40 resources. When a resource is released, it can be taken by a task from any queue.

Deadline Cloud provides two AWS CloudFormation metrics to help you monitor the resources provided by a limit. You can monitor the current number of resources in use and the maximum number of resources available in the limit. For more information, see [Resource limit metrics](#) in the *Deadline Cloud Developer Guide*.

You apply a limit to a job step in a job template. When you specify the amount requirement name of a limit in the `amounts` section of the `hostRequirements` of a step and a limit with the same `amountRequirementName` is associated with the job's queue, tasks scheduled for this step are constrained by the limit for the resource.

If a step requires a resource that is constrained by a limit that is reached, tasks in that step won't be picked up by additional workers.

You can apply more than one limit to a job step. For example, if the step uses two different software licenses, you can apply a separate limit for each license. If a step requires two limits and the limit for one of the resources is reached, tasks in that step won't be picked up by additional workers until the resources become available.

Stopping and deleting limits

When you stop or delete the association between a queue and a limit, a job using the limit stops scheduling tasks from steps that require this limit and blocks the creation of new sessions for a step.

Tasks that are in the `READY` state remain ready, and tasks automatically resume with the association between the queue and the limit becomes active again. You don't need to requeue any jobs.

When you stop or delete the association between a queue and a limit, you have two choices on how to stop running tasks:

- *Stop and cancel tasks* – Workers with sessions that acquired the limit cancel all tasks.
- *Stop and finish running tasks* – Workers with sessions that acquired the limit complete their tasks.

When you delete a limit using the console, workers first stop running tasks immediately or eventually when they complete. When the association is deleted, the following happens:

- Steps requiring the limit are marked not compatible.
- The entire job containing those steps is canceled, including steps that don't require the limit.
- The job is marked not compatible.

If the queue associated with the limit has an associated fleet with a fleet capability that matches the amount requirement name of the limit, that fleet will continue to process jobs with the specified limit.

Important

Do not add the limit's amount requirement name as a `customAmounts` entry in your fleet's capabilities. If a fleet declares the amount as a capability, the fleet treats it as a per-worker resource and the queue-level limit is bypassed. The queue-limit association alone provides scheduling compatibility for jobs that require the limit.

Create a limit

You create a limit using the Deadline Cloud console or the [CreateLimit operation in the Deadline Cloud API](#). Limits are defined for a farm, but associated with queues. After you create a limit, you can associate it with one or more queues.

To create a limit

1. From the Deadline Cloud console ([Deadline Cloud console](#)) dashboard, select the farm that you want to create a queue for.
2. Choose the farm to add the limit to, choose the **Limits** tab, and then choose **Create limit**.
3. Provide the details for the limit. The **Amount requirement name** is the name used in the job template to identify the limit. It must begin with the prefix **amount .** followed by the amount name. The amount requirement name must be unique in queues associated with the limit.

4. If you choose **Set a maximum amount**, that is the total number of resources allowed by this limit. If you choose **No maximum amount**, resource usage isn't limited. Even when resource usage isn't limited, the `CurrentCount` Amazon CloudWatch metric is emitted so that you can track usage. For more information, see [CloudWatch metrics](#) in the *Deadline Cloud Developer Guide*.
5. If you already know the queues that should use the limit, you can choose them now. You don't need to associate a queue to create a limit.
6. Choose **Create limit**.

Associate a limit and a queue

After you create a limit, you can associate one or more queues with the limit. Only queues that are associated with a limit use the values specified in the limit.

You create an association with a queue using the Deadline Cloud console or the [CreateQueueLimitAssociation operation in the Deadline Cloud API](#).

To associate a queue with a limit

1. From the Deadline Cloud console ([Deadline Cloud console](#)) dashboard, select the farm where you want to associate a limit with a queue.
2. Choose the **Limits** tab, choose the limit to associate a queue with, and then choose **Edit limit**.
3. In the **Associate queues** section, choose the queues to associate with the limit.
4. Choose **Save changes**.

Submit a job requiring limits

You apply a limit by specifying it as a host requirement for the job or job step. If you don't specify a limit in a step and that step uses an associated resource, the step's usage isn't counted against the limit when jobs are scheduled..

Some Deadline Cloud submitters enable you to set a host requirement. You can specify the limit's amount requirement name in the submitter to apply the limit.

If your submitter doesn't support adding host requirements, you can also apply a limit by editing the job template for the job.

To apply a limit to a job step in the job bundle

1. Open the job template for the job using a text editor. The job template is located in the job bundle directory for the job. For more information, see [Job bundles](#) in the *Deadline Cloud Developer Guide*.
2. Find the step definition for the step to apply the limit to.
3. Add the following to the step definition. Replace *amount.name* with the amount requirement name of your limit. For typical use, you should set the min value to 1.

YAML

```
hostRequirements:
  amounts:
    - name: amount.name
      min: 1
```

JSON

```
"hostRequirements": {
  "amounts": [
    {
      "name": "amount.name",
      "min": "1"
    }
  ]
}
```

You can add multiple limits to a job step as follows. Replace *amount.name_1* and *amount.name_2* with the amount requirement names of your limits.

YAML

```
hostRequirements:
  amounts:
    - name: amount.name_1
      min: 1
    - name: amount.name_2
      min: 1
```

JSON

```
"hostRequirements": {  
  "amounts": [  
    {  
      "name": "amount.name_1",  
      "min": "1"  
    },  
    {  
      "name": "amount.name_2",  
      "min": "1"  
    }  
  ]  
}
```

4. Save the changes to the job template.

Automate limits with submission hooks

If you want to enforce limits across all job submissions without requiring artists to manually edit job templates, you can use a pre-submission hook to automatically inject the host requirement into every job template at submission time. For more information, see [Submission hooks](#) on the GitHub website.

A pre-submission hook is a script that runs before the job is submitted. The hook can modify the job bundle's `template.yaml` to add the `hostRequirements` amounts entry for your limit. This approach ensures that every job submitted through the Deadline Cloud CLI or DCC submitters declares its need for the limited resource.

For a complete working example, see the [license limits submission hook sample](#) in the Deadline Cloud samples repository on the GitHub website.

End-to-end example: Enforce V-Ray license limits

This example shows how to set up a limit for 5 V-Ray floating licenses and verify that the scheduler enforces it.

To set up and test a V-Ray license limit

1. Create the limit on your farm:

```
aws deadline create-limit \
  --farm-id farm-XXXXXXXXXXXXXXXXXXXXXXXXXXXX \
  --display-name "VRay License" \
  --amount-requirement-name "amount.vray" \
  --max-count 5
```

2. Associate the limit with your queue:

```
aws deadline create-queue-limit-association \
  --farm-id farm-XXXXXXXXXXXXXXXXXXXXXXXXXXXX \
  --queue-id queue-XXXXXXXXXXXXXXXXXXXXXXXXXXXX \
  --limit-id limit-XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

3. Add the host requirement to your job template:

```
specificationVersion: jobtemplate-2023-09
name: My VRay Render
parameterDefinitions:
  - name: SceneFile
    type: PATH
    objectType: FILE
    dataFlow: IN
steps:
  - name: Render
    hostRequirements:
      amounts:
        - name: amount.vray
          min: 1
    script:
      actions:
        onRun:
          command: vray
          args: ["-scene", "{{Param.SceneFile}}"]
```

4. Submit the job. The scheduler allows at most 5 tasks with `amount.vray` to run concurrently across all jobs in the queue. Additional tasks remain in the READY state until a slot becomes available.

To verify the limit is working, temporarily set `maxCount` to 1 and submit two jobs. The first job runs while the second remains in the READY state until the first completes.

How to submit a job to Deadline Cloud

There are many different ways to submit jobs to AWS Deadline Cloud. This section describes some of the ways that you can submit jobs using the tools provided by Deadline Cloud or by creating your own custom tools for your workloads.

- From a terminal – for when you're first developing a job bundle, or when users submitting a job are comfortable using the command line
- From a script – for customizing and automating workloads
- From an application – for when the user's work is in an application, or when an application's context is important.

The following examples use the `deadline` Python library and the `deadline` command line tool. The library is available from [PyPI](#) on the PyPI website. The source code is available from the [deadline-cloud](#) repository on the GitHub website.

Topics

- [Submit a job to Deadline Cloud from a terminal](#)
- [Submit a job to Deadline Cloud using a script](#)
- [Submit a job within an application](#)

Submit a job to Deadline Cloud from a terminal

Using only a job bundle and the Deadline Cloud CLI, you or your more technical users can rapidly iterate on writing job bundles to test submitting a job. Use the following command to submit a job bundle:

```
deadline bundle submit <path-to-job-bundle>
```

If you submit a job bundle with parameters that do not have defaults in the bundle, you can specify them with the `-p / --parameter` option.

```
deadline bundle submit <path-to-job-bundle> -p <parameter-name>=<parameter-value> -  
p ...
```

For a complete list of the available options, run the help command:

```
deadline bundle submit --help
```

Submit a job to Deadline Cloud using a GUI

The Deadline Cloud CLI also comes with a graphical user interface that enables users to see the parameters they must provide before submitting a job. If your users prefer not to interact with the command line, you can write a desktop shortcut that opens a dialog to submit a specific job bundle:

```
deadline bundle gui-submit <path-to-job-bundle>
```

Use the `--browse` option so the user can select a job bundle:

```
deadline bundle gui-submit --browse
```

For a complete list of available options, run the help command:

```
deadline bundle gui-submit --help
```

Submit a job to Deadline Cloud using a script

To automate submitting jobs to Deadline Cloud, you can script them using tools such as bash, Powershell, and batch files.

You can add functionality like populating job parameters from environment variables or other applications. You can also submit multiple jobs in a row, or script the creation of a job bundle to submit.

Submit a job using Python

Deadline Cloud also has an open-source Python library to interact with the service. The source code is available from the [deadline-cloud](#) repository on the GitHub website.

The library is available on pypi via pip (`pip install deadline`). It's the same library used by the Deadline Cloud CLI tool:

```
from deadline.client import api
```

```
job_bundle_path = "/path/to/job/bundle"
job_parameters = [
    {
        "name": "parameter_name",
        "value": "parameter_value"
    },
]

job_id = api.create_job_from_job_bundle(
    job_bundle_path,
    job_parameters
)
print(job_id)
```

To create a dialog like the `deadline bundle gui-submit` command, you can use the `show_job_bundle_submitter` function. For more information, see [deadline.client.ui.job_bundle_submitter](#) on the GitHub website.

The following example starts a Qt application and shows the job bundle submitter:

```
# The GUI components must be installed with pip install "deadline[gui]"
import sys
from qtpy.QtWidgets import QApplication
from deadline.client.ui.job_bundle_submitter import show_job_bundle_submitter

app = QApplication(sys.argv)
submitter = show_job_bundle_submitter(browse=True)
submitter.show()
app.exec()
print(submitter.create_job_response)
```

To make your own dialog you can use the `SubmitJobToDeadlineDialog` class. For more information, see [deadline.client.ui.dialogs.submit_job_to_deadline_dialog](#) on the GitHub website. You can pass in values, embed your own job specific tab, and determine how the job bundle gets created (or passed in).

Submit a job within an application

To make it easy for users to submit jobs, you can use the scripting runtimes or plugin systems provided by an application. Users have a familiar interface and you can create tools that assist the users when submitting a workload.

Embed job bundles in an application

This example demonstrates submitting job bundles that you make available in the application.

To give a user access to these job bundles, create a script embedded in a menu item that launches the Deadline Cloud CLI.

The following script enables a user to select the job bundle:

```
deadline bundle gui-submit --install-gui
```

To use a specific job bundle in a menu item instead, use the following:

```
deadline bundle gui-submit </path/to/job/bundle> --install-gui
```

This opens a dialog where the user can modify the job parameters, inputs, and outputs, and then submit the job. You can have different menu items for different job bundles for a user to submit in an application.

If the job that you submit with a job bundle contains similar parameters and asset references across submissions, you can fill in the default values in the underlying job bundle.

Get information from an application

To pull information from an application so that users don't have to manually add it to the submission, you can integrate Deadline Cloud with the application so that your users can submit jobs using a familiar interface without needing exit the application or use command line tools.

If your application has a scripting runtime that supports Python and pyside/pyqt, you can use the GUI components from the Deadline Cloud client library to create a UI. For more information about the client library, see the [deadline-cloud](#) repository on the GitHub website. For an example integration, see the [deadline-cloud-for-maya](#) repository on the GitHub website.

The Deadline Cloud client library provides operations that do the following to help you provide a strong integrated user experience:

- Pull queue environment parameters, job parameters, and asset references from environment variables and by calling the application SDK.
- Set the parameters in the job bundle. To avoid modifying the original bundle, you should make a copy of the bundle and submit the copy.

If you use the `deadline bundle gui-submit` command to submit the job bundle, you must programmatically the `parameter_values.yaml` and `asset_references.yaml` files to pass the information from the application. For more information about these files see [Open Job Description \(OpenJD\) templates for Deadline Cloud](#).

If you need more complex controls than the ones offered by OpenJD, need to abstract the job from the user, or want to make the integration match the application's visual style, you can write your own dialog that calls the Deadline Cloud client library to submit the job.

Share job bundles on your queue

After you build a custom job bundle, you can share it on your queue. Everyone who submits jobs to that queue can then use it. AWS Deadline Cloud stores shared bundles in the queue's job attachments Amazon S3 bucket. Because of that, a teammate who can submit jobs to the queue can also browse, download, and submit the bundles shared on it. You don't need to create any additional infrastructure or permissions.

A shared bundle is a single `.ojd` archive file that packages the job bundle directory. It contains the Open Job Description template, asset references, and any script or data files. Deadline Cloud stores the archives in the `job-bundles/` folder under the queue's job attachments root prefix, for example `s3://amzn-s3-demo-bucket/DeadlineCloud/job-bundles/`. The upload and download commands create and unpack the archive for you.

Your teammates browse, preview, and submit shared bundles with the job bundle browser. For instructions on browsing and submitting shared bundles, see [Load and submit shared job bundles](#) in the *Deadline Cloud User Guide*. The rest of this page covers the sharing side: publishing bundles, working with them from the command line, and managing the bundles on a queue.

Before you begin, complete the following prerequisites:

- Create a job bundle. For more information, see [Open Job Description \(OpenJD\) templates for Deadline Cloud](#).
- Install the Deadline Cloud CLI and configure it with a default farm and queue. You can also pass the `--farm-id` and `--queue-id` options to each command.
- Configure a queue with job attachments. For more information, see [Use job attachments to share files](#).

The following topics describe how to publish, use, and manage the bundles shared on your queue.

Topics

- [Upload a job bundle](#)
- [Work with shared bundles from the command line](#)
- [Manage the bundles on your queue](#)

Upload a job bundle

To share a job bundle from the command line, upload the bundle directory:

```
deadline bundle upload /path/to/job/bundle
```

The command packages the directory as an .ojd archive and uploads it to the queue:

```
Archiving [#####] 100%
Uploading [#####] 100%
Uploaded bundle to s3://amzn-s3-demo-bucket/DeadlineCloud/job-bundles/my_render_job.ojd
```

By default, the shared bundle takes the name of the bundle directory. To use a different name, pass the `--name` option. You can also upload an .ojd archive that you exported earlier instead of a directory.

```
deadline bundle upload /path/to/job/bundle --name my_render_job_v2
```

You can also share a bundle while you're submitting a job. In the job submission dialog, choose **Save bundle as**, enter a name, then choose **Queue** as the save location. Choosing **Local** instead exports the bundle to a directory on your workstation.

Work with shared bundles from the command line

Use the CLI commands in this section to script bundle operations and to check the state of a queue's bundles while you develop them.

List the bundles shared on the queue:

```
deadline bundle list --queue
```

Print a bundle's name, description, steps, and parameters from its Open Job Description template:

```
deadline bundle info my_render_job --queue
```

Download a shared bundle to a directory. If the directory already contains a bundle with the same name, the download replaces it:

```
deadline bundle download my_render_job -o ~/job-bundles
```

Then submit it like any other job bundle:

```
deadline bundle submit ~/job-bundles/my_render_job
```

If you don't pass the `-o` option, the bundle downloads to a local cache and the command prints the path. Downloads are cached, so downloading a bundle that hasn't changed on the queue reuses the local copy.

Hide a bundle from your own listings and from the job bundle browser on your workstation. Hiding is local to your workstation, so the bundle stays on the queue and your teammates still see it. To include hidden bundles when listing, pass `--show-hidden`.

```
deadline bundle hide my_render_job  
deadline bundle unhide my_render_job
```

Manage the bundles on your queue

To publish an updated version of a bundle, upload it again with the same name. The command asks for confirmation before overwriting the existing archive. Teammates get the new version the next time they download or submit the bundle. Their cached copy is refreshed when the archive changes on the queue.

To remove a shared bundle from the queue for everyone, delete its `.ojd` file from the `job-bundles/` folder in the queue's job attachments bucket. Deleting the file requires Amazon S3 permissions on the bucket, so removing bundles is typically an administrator task.

You can also manage the folder in Amazon S3 directly, for example to seed a queue with a set of starter bundles from a deployment script. Place `.ojd` archives in the `job-bundles/` folder, using subfolders to organize them if you want. Only `.ojd` files are recognized as bundles, and the CLI commands only list bundles at the top level of the folder. Bundles in subfolders appear in the job bundle browser.

Host requirements and fleet capabilities

Capabilities and host requirements are complements. A fleet's capabilities advertise what its workers support, and a step's host requirements state what the step needs. Three concepts explain how they interact:

- **Requirements are matched to fleets, not to workers** – The scheduler compares a step's requirements against each fleet's declared capabilities. It never inspects individual workers.
- **Capabilities schedule; only service-managed fleet capabilities also configure hardware** – On a service-managed fleet, the instance capabilities additionally determine which Amazon EC2 instance types launch. On a customer-managed fleet, capabilities are a self-reported declaration and don't change your machines.
- **Requirements never provide hardware** – A host requirement filters which fleets can run the step. It doesn't launch larger instances or change auto scaling decisions.

Mechanism	Where you set it	What it does	What it doesn't do
Fleet instance capabilities (service-managed fleet)	Fleet configuration (instanceCapabilities)	Configures the hardware: the set of Amazon EC2 instance types that can launch. Also advertises the fleet's capabilities for scheduling	Doesn't pick one size within the set – any instance type that satisfies the configuration can launch
Fleet worker capabilities (customer-managed fleet)	Fleet configuration (workerCapabilities)	Advertises the fleet's capabilities for scheduling; a self-reported declaration of what every worker in the fleet provides	Doesn't configure or verify hardware – updating the declaration doesn't change your machines
Step host requirements	Job template (hostRequirements)	States what the step needs; matched against fleet capabilities when the job is created to select compatible fleets	Doesn't launch larger instances, change auto scaling decisions, or match against individual workers

Mechanism	Where you set it	What it does	What it doesn't do
Worker-advertised capabilities	Reported by the worker agent at startup	Records the hardware a worker reports, such as <code>amount.worker.memory</code> , for visibility and debugging	Not used for scheduling or hardware configuration

Fleet configuration provides the hardware

For a service-managed fleet, the `instanceCapabilities` configuration (memory, vCPUs, GPUs, and the allowed and excluded instance type lists) defines the set of instance types the fleet can launch. When the fleet scales out, any instance type in that set can launch, including the smallest. To guarantee a hardware floor, raise the configuration minimums, such as `memoryMiB.min`, or limit `allowedInstanceTypes`. For more information, see [Service-managed fleets](#) in the *Deadline Cloud User Guide*.

Service-managed instance capabilities are the only mechanism that configures hardware. On a customer-managed fleet, the `workerCapabilities` declaration is self-reported: it advertises the hardware that you state every worker in the fleet provides, and updating the declaration doesn't change your machines. For more information, see [Create a customer-managed fleet](#).

Host requirements select a fleet

When you create a job, the scheduler compares each step's `hostRequirements` against the capabilities of the fleets associated with the queue. A fleet that satisfies the requirements can run the step; if no fleet does, the step is marked `NOT_COMPATIBLE`.

Matching is fleet-level only. For example, if a step requires 16 GiB of memory and a fleet declares a memory capability of 8–32 GiB, the fleet is not compatible – even if a worker in that fleet has 32 GiB. The scheduler never inspects individual workers, so design fleets around guaranteed hardware tiers and let each step's requirements route it to the right fleet. For more information, see [Determine fleet compatibility](#) and [Fleet design best practices](#).

Worker capabilities are informational

Each worker reports the hardware it detected at startup, such as `amount.worker.memory`. These values are for visibility and debugging. The scheduler doesn't use them: compatibility comes from the fleet's declared capabilities, and hardware comes from the fleet configuration. In particular, a requirement such as `amount.worker.memory` doesn't cause a fleet to launch a larger instance. Treat the step requirement as a statement of need and the fleet configuration as the guarantee.

Auto scaling counts tasks

The auto scaler sizes a service-managed fleet based on two factors: the number of tasks ready for that fleet, and the fleet's minimum and maximum worker counts. It chooses how *many* workers to run; the fleet configuration determines what each worker is. For more information, see [Fleet scaling](#).

Host requirements are part of the Open Job Description specification. For the full field reference, see [Template Schemas](#) on the GitHub website.

Schedule jobs in Deadline Cloud

After you create a job, AWS Deadline Cloud schedules it to be processed on one or more of the fleets associated with a queue. The fleet that processes a particular task is chosen based on the scheduling configuration, the capabilities configured for the fleet, and the host requirements of a specific step.

The following sections provide details of the process of scheduling a job.

Scheduling configurations

You can configure how Deadline Cloud schedules jobs in a queue by setting a scheduling configuration on the queue. The scheduling configuration controls how workers are distributed across jobs.

You can set the scheduling configuration using the Deadline Cloud console or by calling the [CreateQueue](#) or [UpdateQueue](#) APIs.

There are three available scheduling configurations:

- **Priority, first-in-first-out** (`priorityFifo`) – Schedules the highest priority, earliest submitted job first (default).

- **Priority, balanced** (`priorityBalanced`) – Distributes workers evenly across jobs at the highest priority.
- **Weighted, balanced** (`weightedBalanced`) – Uses a weighted formula to determine how workers are distributed across jobs.

In all scheduling configurations, in-progress tasks run to completion before a new scheduling decision is made. If you change the scheduling configuration while tasks are running, the change applies only when workers are assigned next. Running tasks are not interrupted or reassigned.

Priority, first-in-first-out

Priority, first-in-first-out (`priorityFifo`) is the default scheduling configuration for new queues. Deadline Cloud assigns workers to the highest-priority job first. When multiple jobs share the same priority, the oldest (earliest submitted) job receives all available workers first.

Use priority FIFO when you want strict ordering of jobs. This configuration is appropriate when jobs should complete one at a time in the order they were submitted, such as sequential pipeline stages or batch processing where each job must finish before the next one starts.

This configuration has no additional parameters.

Priority, balanced

Priority, balanced (`priorityBalanced`) distributes workers evenly across all jobs at the highest priority level. When only one job exists at the highest priority, Deadline Cloud assigns all workers to that job. When multiple jobs share the highest priority, workers are split evenly among them. If the workers cannot be evenly divided, the extra workers are distributed among the highest priority jobs.

Use priority balanced when multiple artists or users submit jobs at the same priority and each user needs immediate feedback. This configuration ensures that no single job monopolizes all available workers, so that all users are allocated workers shortly after submission.

If a job has fewer remaining tasks than its share of workers, the surplus workers are redistributed to other jobs at the same priority level. If all jobs at the highest priority are fully allocated, surplus workers cascade to jobs at the next highest priority level.

This configuration has the following parameter:

renderingTaskBuffer

Controls worker stickiness. A worker switches from its current job to another job at the same priority only if the difference in rendering tasks exceeds the `renderingTaskBuffer` value. A higher value keeps workers on their current jobs longer, reducing context switching. The default value is 1.

Weighted, balanced

Weighted, balanced (`weightedBalanced`) uses a formula to calculate a weight for each job. Deadline Cloud assigns workers to the highest-weight job first. If multiple jobs have the same weight, workers are distributed among them.

Use weighted balanced when you need fine-grained control over how workers are distributed across jobs with varying priorities, error rates, and submission times. This configuration is appropriate for complex render farm environments where you want to tune the balance between job priority, job age, error handling, and worker stickiness.

The weight for each job is calculated as follows:

```
weight = (job.Priority * priorityWeight) +  
         (job.Errors * errorWeight) +  
         ((currentTimeInSeconds - job.SubmissionTime) * submissionTimeWeight) +  
         ((job.RenderingTasks - renderingTaskBuffer) * renderingTaskWeight)
```

The `renderingTaskBuffer` component is applied only if the worker is currently working on the job. The `renderingTaskWeight` is usually set to a negative value so that jobs with assigned workers receive a lower weight, bringing other jobs to the front of the queue. The `errorWeight` is also usually negative so that jobs with errors are deprioritized. You can use scheduling overrides for minimum and maximum priority jobs.

This configuration has the following parameters:

priorityWeight

The weight applied to a job's priority. A positive value means higher-priority jobs are scheduled first. The default value is 100.0. Range: 0 to 10000.

`errorWeight`

The weight applied to a job's error count. A negative value means jobs without errors are scheduled first. The default value is `-10.0`. Range: `-10000` to `10000`.

`submissionTimeWeight`

The weight applied to a job's submission time (in seconds). A positive value means earlier submitted jobs are scheduled first. The default value is `3.0`. Range: `0` to `10000`.

`renderingTaskWeight`

The weight applied to the number of tasks currently rendering for a job. A negative value means jobs with fewer workers are scheduled next. The default value is `-100.0`. Range: `-10000` to `10000`.

`renderingTaskBuffer`

The number of rendering tasks before the rendering task weight takes effect. A positive value keeps workers on their current jobs. The default value is `1`. Range: `0` to `1000`.

`maxPriorityOverride`

Optional. When set to `alwaysScheduleFirst`, jobs at the maximum priority (100) are always scheduled before other jobs, regardless of the weighted formula. When multiple jobs have the maximum priority, ties are broken using the standard weighted formula. When the override is absent, maximum priority jobs use the standard weighted formula with no special treatment.

`minPriorityOverride`

Optional. When set to `alwaysScheduleLast`, jobs at the minimum priority (0) are always scheduled after other jobs, regardless of the weighted formula. When multiple jobs have the minimum priority, ties are broken using the standard weighted formula. When the override is absent, minimum priority jobs use the standard weighted formula with no special treatment.

Determine fleet compatibility

Queues and fleets divide the work of routing jobs. A queue organizes jobs and controls who can submit and view them. Fleets and host requirements select which workers run each step.

To dedicate specific workers to certain jobs, create a separate fleet for those workers instead of a separate queue. For example, use a separate fleet for machines with particular hardware or for machines reserved for sensitive content. One queue can be associated with several fleets, and each

step's host requirements select a compatible fleet. If you're coming from Deadline 10, fleets and host requirements replace worker groups. For the full concept mapping, see [Migrate from Deadline 10](#).

After you create a job, Deadline Cloud checks the host requirements for each step in the job against the capabilities of the fleets associated with the queue the job was submitted to. If a fleet meets the host requirements, the job is put into the READY state.

If any step in the job has requirements that can't be met by a fleet associated with the queue, the step's status is set to NOT_COMPATIBLE. In addition, the rest of the steps in the job are canceled. If you associate a compatible fleet with the queue later, existing NOT_COMPATIBLE jobs don't automatically restart – to run them, requeue them. For more information, see [Modify a job in Deadline Cloud](#).

Capabilities for a fleet are set at the fleet level. Even if a worker in a fleet meets the job's requirements, it won't be assigned tasks from the job if its fleet doesn't meet the job's requirements.

Important

The fleet-level `workerCapabilities` declaration is the scheduling contract. The scheduler evaluates step compatibility exclusively against the fleet-level capabilities – it does not inspect individual workers. If a step's `hostRequirements` exceed the minimum values declared in the fleet's `workerCapabilities`, the fleet is evaluated as not compatible, even if some individual workers in the fleet would satisfy the requirements.

The following job template has a step that specifies host requirements for the step:

```
name: Sample Job With Host Requirements
specificationVersion: jobtemplate-2023-09
steps:
- name: Step 1
  script:
    actions:
      onRun:
        args:
          - '1'
        command: /usr/bin/sleep
  hostRequirements:
```

```

amounts:
  # Capabilities starting with "amount." are amount capabilities. If they start with
  "amount.worker.",
  # they are defined by the OpenJD specification. Other names are free for custom
  usage.
  - name: amount.worker.vcpu
    min: 4
    max: 8
attributes:
  - name: attr.worker.os.family
    anyOf:
      - linux

```

Submitters set the same requirements through the **Host requirements** tab. Choose **Run on worker hosts that meet the following requirements** to set an operating system, CPU architecture, and hardware ranges without editing the template.

Shared job settings | Job-specific settings | Job attachments | **Host requirements**

☐ Run on all available worker hosts
☒ Run on worker hosts that meet the following requirements
All fields below are optional

Operating system: linux
 CPU architecture: x86_64

Hardware requirements

Requirement	Min	Max
vCPUs	8	32
Memory (GiB)	16	64
GPUs	1	2
GPU memory (GiB)	16	48
Scratch space	100	500

Custom host requirements

☒ More info
 Add amount Add attribute

✓ default ▾
 Settings... Submit Export bundle Help

This job can be scheduled to a fleet with the following capabilities:

```
{
```

```
"vCpuCount": {"min": 4, "max": 8},  
"memoryMiB": {"min": 1024},  
"osFamily": "linux",  
"cpuArchitectureType": "x86_64"  
}
```

This job can't be scheduled to a fleet with any of the following capabilities:

```
{  
  "vCpuCount": {"min": 4},  
  "memoryMiB": {"min": 1024},  
  "osFamily": "linux",  
  "cpuArchitectureType": "x86_64"  
}
```

The vCpuCount has no maximum, so it exceeds the maximum vCPU host requirement.

```
{  
  "vCpuCount": {"max": 8},  
  "memoryMiB": {"min": 1024},  
  "osFamily": "linux",  
  "cpuArchitectureType": "x86_64"  
}
```

The vCpuCount has no minimum, so it doesn't satisfy the minimum vCPU host requirement.

```
{  
  "vCpuCount": {"min": 4, "max": 8},  
  "memoryMiB": {"min": 1024},  
  "osFamily": "windows",  
  "cpuArchitectureType": "x86_64"  
}
```

The osFamily doesn't match.

Fleet design best practices

Because the scheduler evaluates compatibility at the fleet level, design your customer-managed fleets so that the declared `workerCapabilities` represent the guaranteed minimum hardware characteristics of every worker in the fleet:

- Split workers into multiple fleets based on guaranteed minimum hardware characteristics (for example, GPU count, VRAM, or CPU count) rather than using one heterogeneous fleet with broad ranges.

- Ensure that each worker in a fleet meets or exceeds the declared minimums, even if the workers don't have identical hardware.
- Associate multiple fleets with a single queue so that the scheduler selects a compatible fleet based on each step's `hostRequirements`.

For example, if you have workers with 1 GPU (24 GiB VRAM) and workers with 4 GPUs (96 GiB VRAM), create two separate fleets: one declaring a minimum of 1 GPU and 24 GiB GPU memory, and another declaring a minimum of 4 GPUs and 96 GiB GPU memory. Then associate both fleets with the same queue. Steps that require 4 GPUs are automatically routed to the high-GPU fleet.

For more information about creating a customer-managed fleet, see [Create a customer-managed fleet](#).

Memory requirements and instance sizing

A step's `amount.worker.memory` host requirement selects fleets. It doesn't size instances. The scheduler matches the requirement against each fleet's declared capabilities, not against individual workers. As a result, the requirement doesn't cause a service-managed fleet to launch a larger instance. To guarantee the instance size, constrain the fleet configuration instead. The same applies to the other hardware requirements, such as `amount.worker.vcpu` and `amount.worker.gpu`. For more information about how requirements and fleet configuration divide this work, see [Host requirements and fleet capabilities](#).

Custom capabilities

In addition to the built-in worker capabilities (vCPU, memory, GPU, operating system, and CPU architecture), you can define custom amounts and custom attributes on a fleet to express additional scheduling constraints:

- **Custom amounts** – Numeric values such as available disk space or specialized hardware counters.
- **Custom attributes** – String values such as installed software, solver versions, or site labels. For example, you can define `attr.sw.solvers` with values like `["vray-6", "arnold-7"]` to route jobs that require specific software.

At the fleet level, declare software or hardware combinations in custom capabilities only if they are guaranteed to exist on all workers in that fleet.

Fleet-level custom capabilities have the following limits:

- Maximum of 15 custom amounts per fleet
- Maximum of 15 custom attributes per fleet

Names in the form `amount.worker.*` and `attr.worker.*` are reserved by the service for built-in capabilities. Use other prefixes for your custom capabilities.

The following example routes steps that need a specific solver to the fleet that provides it. The fleet declares the solvers installed on all of its workers in a custom attribute. This fleet configuration is part of the [CreateFleet](#) request:

```
"workerCapabilities": {
  "vCpuCount": {"min": 4},
  "memoryMiB": {"min": 16384},
  "osFamily": "linux",
  "cpuArchitectureType": "x86_64",
  "customAttributes": [
    {
      "name": "attr.sw.solvers",
      "values": ["vray-6", "arnold-7"]
    }
  ]
}
```

A step that requires one of the declared values states the requirement in its `hostRequirements` in the job template:

```
steps:
- name: RenderWithVray
  hostRequirements:
    attributes:
      - name: attr.sw.solvers
        anyOf:
          - vray-6
  script:
    actions:
      onRun:
        command: '{{Task.File.Render}}'
```

The requirement keeps the `RenderWithVray` step off fleets that do not declare `vray-6`. The reverse does not hold: a step with no `attr.sw.solvers` requirement is still compatible with this fleet and can be scheduled to it.

Note

Custom attributes and host requirements route work. They are not an access control. Host requirements are set in the job template by whoever submits the job, and a fleet that declares an attribute still accepts steps that do not mention it. To reserve workers for sensitive content, put them in their own fleet and associate that fleet only with approved queues. For more information, see [the section called “Access control and worker selection”](#) and [the section called “Isolate workloads”](#).

For more information about configuring custom capabilities when creating a fleet, see [Create a customer-managed fleet](#).

GPU memory reporting

When you configure fleets for GPU-intensive workloads that require a specific per-GPU VRAM threshold, understand how the worker agent reports GPU memory.

The Deadline Cloud worker agent reports the *minimum* GPU memory across all GPUs on the worker into `amount.worker.gpu.memory`, not the total sum. This behavior ensures that jobs requiring a specific per-GPU VRAM amount are routed to workers where every GPU meets that requirement.

For example, if a worker has two GPUs with 24 GiB and 48 GiB of VRAM respectively, the worker agent reports 24 GiB as the GPU memory value.

At the fleet level, set the `acceleratorTotalMemoryMiB` minimum to the lowest per-GPU VRAM that any worker in the fleet is guaranteed to have.

Per-worker capabilities

When you need to track operational state for individual workers without affecting scheduling decisions, use per-worker capabilities. For example, you might flag workers for maintenance, track software rollout state, or record health check indicators.

You can set capabilities on individual workers using the `UpdateWorker` API operation. Per-worker capabilities are *not* used as the scheduling contract. The scheduler evaluates compatibility exclusively against the fleet-level `workerCapabilities` declaration.

The `GetWorker` and `ListWorkers` operations don't return worker capabilities that are set through `UpdateWorker`.

For more information, see [UpdateWorker](#) in the *Deadline Cloud API Reference*.

Access control and worker selection

Deadline Cloud separates who can use a resource from where jobs run. IAM policies and AWS IAM Identity Center user and group memberships control who can view, submit to, and manage a farm, queue, or fleet. Queue–fleet associations and host requirements control which workers run each job.

Group membership on a fleet grants people access to view and manage that fleet in the monitor. Membership does not route jobs to the fleet or keep jobs off it. To direct work to or away from a fleet, use queue–fleet associations.

Host requirements are part of the job template, so whoever submits the job chooses them. They route work to capable fleets, and they are not an access control. The queue–fleet association is the enforcement point: the service schedules a step only to fleets associated with the job's queue, whatever the step's host requirements say. To keep unauthorized work off a restricted fleet, associate the fleet only with restricted queues and control who can submit to those queues. For more information about the security boundaries between queues and fleets that share workers, see [the section called "Isolate workloads"](#).

Fleet scaling

When a job is assigned to a compatible service-managed fleet, the fleet is auto scaled. The number of workers in the fleet changes based on the number of tasks available for the fleet to run.

When a job is assigned to a customer-managed fleet, workers might already exist or can be created using event-based auto scaling. For more information, see [Use EventBridge to handle auto scaling events](#) in the *Amazon EC2 Auto Scaling User Guide*.

Sessions

The tasks in a job are divided into one or more sessions. Workers run the sessions to set up the environment, run the tasks, and then tear down the environment. Each session is composed of one or more actions that a worker must take.

As a worker completes session actions, additional session actions can be sent to the worker. The worker reuses existing environments and job attachments in the session to complete tasks more efficiently.

On service-managed fleet workers, session directories are deleted after the session ends, but other directories are retained between sessions. This behavior allows you to implement caching strategies for data that can be reused across multiple sessions. To cache data between sessions, store it under the home directory of the user running the job. For example, conda packages are cached under the job user's home directory at `C:\Users\job-user\.conda-pkgs` on Windows workers and `/home/job-user/.conda-pkgs` on Linux workers. This data remains available until the worker shuts down.

Job attachments are created by the submitter that you use as part of your Deadline Cloud CLI job bundle. You can also create job attachments using the `--attachments` option for the `create-job` AWS CLI command. Environments are defined in two places: queue environments attached to a specific queue, and job and step environments defined in the job template.

There are four session action types:

- `syncInputJobAttachments` – Downloads the input job attachments to the worker.
- `envEnter` – Performs the `onEnter` actions for an environment.
- `taskRun` – Performs the `onRun` actions for a task.
- `envExit` – Performs the `onExit` actions for an environment.

The following job template has a step environment. It has an `onEnter` definition to set up the step environment, an `onRun` definition that defines the task to run, and an `onExit` definition to tear down the step environment. The sessions created for this job will include an `envEnter` action, one or more `taskRun` actions, and then an `envExit` action.

```
name: Sample Job with Maya Environment
specificationVersion: jobtemplate-2023-09
steps:
```

```
- name: Maya Step
stepEnvironments:
- name: Maya
description: Runs Maya in the background.
script:
  embeddedFiles:
  - name: initData
    filename: init-data.yaml
    type: TEXT
    data: |
      scene_file: MyAwesomeSceneFile
      renderer: arnold
      camera: persp
  actions:
    onEnter:
      command: MayaAdaptor
      args:
      - daemon
      - start
      - --init-data
      - file//{{Env.File.initData}}
    onExit:
      command: MayaAdaptor
      args:
      - daemon
      - stop
parameterSpace:
  taskParameterDefinitions:
  - name: Frame
    range: 1-5
    type: INT
script:
  embeddedFiles:
  - name: runData
    filename: run-data.yaml
    type: TEXT
    data: |
      frame: {{Task.Param.Frame}}
  actions:
    onRun:
      command: MayaAdaptor
      args:
      - daemon
      - run
```

```
- --run-data  
- file://{ Task.File.runData }
```

Session actions pipelining

Session actions pipelining lets a scheduler pre-assign multiple session actions to a worker. The worker can then run these actions sequentially, reducing or eliminating idle time between tasks.

To create an initial assignment, the scheduler creates a session with one task, the worker completes the task, and then the scheduler analyzes the task duration to determine future assignments.

For the scheduler to be effective, there are task duration rules. For tasks under one minute, the scheduler uses a power-of-2 growth pattern. For example, for a 1-second task, the scheduler assigns 2 new tasks, then 4, then 8. For tasks over one minute, the scheduler assigns only one new task and pipelining remains disabled.

To calculate pipeline size, the scheduler does the following:

- Uses average task duration from completed tasks
- Aims to keep the worker busy for one minute
- Considers only tasks within the same session
- Does not share duration data across workers

With session actions pipelining, workers start new tasks immediately and there's no waiting time between scheduler requests. It also provides improved worker efficiency and better task distribution for long-running processes.

Additionally, if there is a new higher priority job available, the worker will finish all of its previously assigned work before its current session ends and a new session from a higher priority job is assigned.

Step dependencies

Deadline Cloud supports defining dependencies between steps so that one step waits until another step is complete before starting. You can define more than one dependency for a step. A step with a dependency isn't scheduled until all of its dependencies are complete.

If the job template defines a circular dependency, the job is rejected and the job status is set to `CREATE_FAILED`.

The following job template creates a job with two steps. StepB depends on StepA. StepB only runs after StepA completes successfully.

After the job is created, StepA is in the READY state and StepB is in the PENDING state. After StepA finishes, StepB moves to the READY state. If StepA fails, or if StepA is canceled, StepB moves to the CANCELED state.

You can set a dependency on multiple steps. For example, if StepC depends on both StepA and StepB, StepC won't start until the other two steps finish.

Step dependencies have the following restrictions:

- **Dependencies per step** – A step can depend on a maximum of 128 other steps.
- **Consumers per step** – A maximum of 32 other steps can depend on a single step.

```
name: Step-Step Dependency Test
specificationVersion: 'jobtemplate-2023-09'
steps:
- name: A
  script:
    actions:
      onRun:
        command: bash
        args: ['{{ Task.File.run }}']
    embeddedFiles:
      - name: run
        type: TEXT
        data: |
          #!/bin/env bash

          set -euo pipefail

          sleep 1
          echo Task A Done!
- name: B
  dependencies:
    - dependsOn: A # This means Step B depends on Step A
  script:
    actions:
      onRun:
        command: bash
        args: ['{{ Task.File.run }}']
```

```
embeddedFiles:
  - name: run
    type: TEXT
    data: |
      #!/bin/env bash

      set -euo pipefail

      sleep 1
      echo Task B Done!
```

Modify a job in Deadline Cloud

You can use the following AWS Command Line Interface (AWS CLI) update commands to modify the configuration of a job, or to set the target status of a job, step, or task:

- `aws deadline update-job`
- `aws deadline update-step`
- `aws deadline update-task`

In the following examples of the update commands, replace each *user input placeholder* with your own information.

Example– Requeue a job

All tasks in the job switch to the READY status, unless there are step dependencies. Steps with dependencies switch to either READY or PENDING as they are restored.

```
aws deadline update-job \
--farm-id farmID \
--queue-id queueID \
--job-id jobID \
--target-task-run-status PENDING
```

Example– Cancel a job

All tasks in the job that don't have the status SUCCEEDED or FAILED are marked CANCELED.

```
aws deadline update-job \
--farm-id farmID \
```

```
--queue-id queueID \  
--job-id jobID \  
--target-task-run-status CANCELED
```

Example– Mark a job failed

All tasks in the job that have the status SUCCEEDED are left unchanged. All other tasks are marked FAILED.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--target-task-run-status FAILED
```

Example– Mark a job successful

All tasks in the job move to the SUCCEEDED state.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--target-task-run-status SUCCEEDED
```

Example– Suspend a job

Tasks in the job in the SUCCEEDED, CANCELED, or FAILED state don't change. All other tasks are marked SUSPENDED.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--target-task-run-status SUSPENDED
```

Example– Change the priority of a job

Updates the priority of a job in a queue to change the order that it is scheduled. Higher priority jobs are generally scheduled first.

```
aws deadline update-job \  

```

```
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--priority 100
```

Example– Change the number of failed tasks allowed

Updates the maximum number of failed tasks that the job can have before the remaining tasks are canceled.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--max-failed-tasks-count 200
```

Example– Change the number of task retries allowed

Updates the maximum number of retries for a task before the task fails. A task that has reached the maximum number of retries can't be requeued until this value is increased.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--max-retries-per-task 10
```

Example– Archive a job

Updates the job's lifecycle status to ARCHIVED. Archived jobs can't be scheduled or modified. You can only archive a job that is in the FAILED, CANCELED, SUCCEEDED, or SUSPENDED state.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--lifecycle-status ARCHIVED
```

Example– Change the name of a job

Updates the display name of a job. The job name can be up to 128 characters long.


```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--name "New Job Name"
```

Example– Change the description of a job

Updates the description of a job. The description can be up to 2048 characters long. To remove the existing description, pass an empty string.

```
aws deadline update-job \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--description "New Job Description"
```

Example– Requeue a step

All tasks in the step switch to the READY state, unless there are step dependencies. Tasks in steps with dependencies switch to either READY or PENDING, and the task is restored.

```
aws deadline update-step \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--step-id stepID \  
--target-task-run-status PENDING
```

Example– Cancel a step

All tasks in the step that don't have the status SUCCEEDED or FAILED are marked CANCELED.

```
aws deadline update-step \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--step-id stepID \  
--target-task-run-status CANCELED
```

Example– Mark a step failed

All tasks in the step that have the status SUCCEEDED are left unchanged. All other tasks are marked FAILED.

```
aws deadline update-step \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--step-id stepID \  
--target-task-run-status FAILED
```

Example– Mark a step successful

All tasks in the step are marked SUCCEEDED.

```
aws deadline update-step \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--step-id stepID \  
--target-task-run-status SUCCEEDED
```

Example– Suspend a step

Tasks in the step in the SUCCEEDED, CANCELED, or FAILED state don't change. All other tasks are marked SUSPENDED.

```
aws deadline update-step \  
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--step-id stepID \  
--target-task-run-status SUSPENDED
```

Example– Change the status of a task

When you use the update-task Deadline Cloud CLI command, the task switches to the specified status.

```
aws deadline update-task \  

```

```
--farm-id farmID \  
--queue-id queueID \  
--job-id jobID \  
--step-id stepID \  
--task-id taskID \  
--target-task-run-status SUCCEEDED | SUSPENDED | CANCELED | FAILED | PENDING
```

Hooks, events, and integration points for jobs

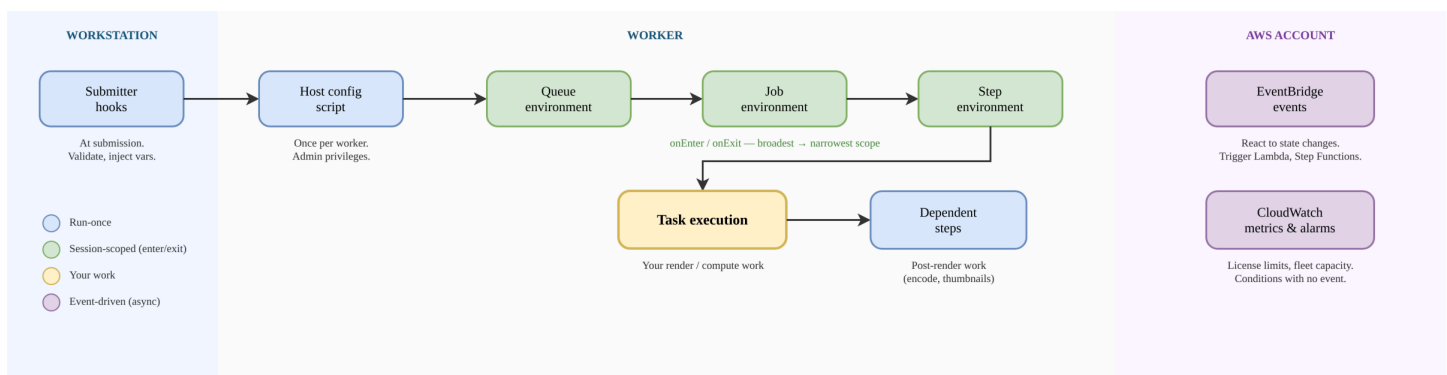
Deadline Cloud provides several *integration points* where you can run your own logic around a job. An integration point is a place in the job lifecycle where Deadline Cloud runs custom code or hands control to a service that you configure, such as a script that prepares a worker or a rule that reacts to a job event. You use integration points to adapt Deadline Cloud to your production pipeline: to install software, set environment variables, validate or change a job, process outputs, or trigger downstream automation.

The integration points differ on three questions:

- **Where the code runs** – on your workstation, on the worker, or in the AWS Cloud.
- **When it runs** – at submission, before or after the work, or in response to an event.
- **Where you configure it** – in a workstation setting, a job bundle, a queue or fleet property, or your own AWS account.

The following diagram shows where each integration point runs during the job lifecycle. Control flows from submission on the workstation, through worker execution, and into event-driven automation in your AWS account.

- **Workstation** – Submitter hooks and host configuration scripts run once.
- **Worker** – Queue, job, and step environments use session-scoped enter and exit actions. Task execution runs your work. Dependent steps run after the task completes.
- **AWS account** – EventBridge events and CloudWatch metrics and alarms respond asynchronously to state changes.



The following table summarizes the integration points. Choose one based on the answers to the preceding questions. The sections after the table describe each option and link to detailed instructions.

Integration point	Runs on	Configured in	When it runs	Use for
Submitter hooks	Workstation	Workstation configuration (all jobs) or a job bundle	During submission, before the job is created	Inject or override environment variables, validate or rewrite a job bundle, synchronize with production-tracking software at submission time
Host configuration scripts	Worker	Fleet property (service-managed fleet)	One time per worker, at startup, with administrator privileges	Network configuration, installing software that needs elevated privileges
Queue environments	Worker	Queue property	When a session enters or exits, for every job in the queue	Project-level configuration: non-admin software installs, shared environment variables, license or daemon setup
Job environments	Worker	Job bundle or job template	Before and after a job	Per-job setup and teardown
Step environments	Worker	Job bundle or step template	Before and after a step	Per-step setup and teardown
Dependent steps	Worker	Job bundle or step template (dependsOn)	After an earlier step in the same job succeeds	Post-render work that needs the output files locally, such as encoding frames into a movie or generating thumbnails

Integration point	Runs on	Configured in	When it runs	Use for
Amazon EventBridge (EventBridge) events	AWS Cloud	Your AWS account (rule and target)	On a job, step, task, fleet, or budget state change	Production tracking, notifications, cloud automation, chaining jobs across queues
Amazon CloudWatch (CloudWatch) metrics and alarms	AWS Cloud	Your AWS account	Continuously, against a threshold	Conditions that have no event, such as license availability (<code>LicensesInUse</code>) and fleet capacity

Run custom logic on the workstation

Submitter hooks run during submission, before the job reaches the service. You configure them per workstation, where they apply to all jobs submitted from that workstation, or inside a job bundle, where they apply to that bundle. Submitter hooks can alter the bundle at different points in the submission flow. Use them to inject environment variables, validate a submission, or synchronize with production-tracking software at submission time.

For more information, see [Submission hooks](#).

Run custom logic on the worker

The following integration points run on the rendering host. They are nested from the broadest scope to the narrowest:

- **Host configuration scripts** run one time when a worker starts, with administrator privileges. Host configuration scripts are specific to service-managed fleets (SMFs). On a customer-managed fleet (CMF), you bake your own image instead. Use host configuration scripts for network setup and administrator-level software installs.
- **Queue, job, and step environments** each define `onEnter` and `onExit` actions. The actions run when the worker's session enters a scope and, in reverse order, when the session exits the scope. Queue environments are a good place for project-level configuration that applies to every job in

a queue, such as shared software, environment variables, and license or daemon setup. Job and step environments narrow the scope to a single job or step.

- **Dependent steps** add a second step that uses `dependsOn` to depend on the render step. For example, you can render and then encode a movie, or render and then generate thumbnails. The step stays in the same job, so the outputs are already local. Dependent steps require a custom submitter or a hand-written job bundle. Job-to-job dependencies are not native. To chain separate jobs, use events in the cloud, as described in the following section.

Note

Environments are session-scoped, not per-task. An `onExit` action runs at session teardown, not after every task. If you are migrating a Deadline 10 post-task callback, use a dependent step instead of an environment `onExit` action.

For more information, see the following:

- [the section called “Host configuration scripts”](#)
- [Create a queue environment](#)
- [the section called “Provide applications for your jobs”](#)
- [the section called “Step dependencies”](#)
- [Open Job Description template schemas](#) on the GitHub website, for `onEnter` and `onExit` scoping

Respond to events in the cloud

EventBridge and CloudWatch let you react to job activity without consuming farm capacity. Use them when your logic responds to job outcomes rather than running as part of a job.

- Deadline Cloud sends events with the source `aws.deadline` to the default event bus in your AWS account. Events cover job, step, and task state changes, fleet size recommendations, and budget thresholds. You own the rule, the AWS Lambda (Lambda) function, and the dead-letter queue (DLQ). Wire a rule to a Lambda function for notifications, asset registration, or to chain jobs across queues (event to Lambda or `()` to `CreateJob`). Delivery is best-effort and events might arrive out of order, so make your consumers idempotent and configure retries and a DLQ.

- CloudWatch metrics and alarms cover conditions that EventBridge doesn't, most importantly license availability and fleet capacity. No event exists for license availability, so you alarm on the `LicensesInUse` metric and limit counts. The `Fleet Size Recommendation Change` event and the `RecommendedFleetSize` metric also drive customer-managed fleet auto scaling.

For more information, see the following:

- [the section called "Managing events using EventBridge"](#)
- [the section called "Events detail reference"](#)
- [the section called "CloudWatch metrics"](#)
- [the section called "Create fleet infrastructure"](#)
- [Deadline Cloud events](#) in the *EventBridge service reference*
- [Asynchronous invocation](#) in the *Lambda Developer Guide*
- [Event retry policy and using dead-letter queues](#) in the *EventBridge User Guide*

Choose an integration point

Use the following guidance to choose an integration point for a common need:

- To change a job before it's submitted, use a submitter hook.
- To run logic on the render host around the work, use a host configuration script or a queue, job, or step environment. Choose the broadest scope that fits, and use a queue environment for project-level configuration.
- To process job outputs while they're still local, use a dependent step.
- To react to job outcomes without consuming farm capacity, use an EventBridge rule with a Lambda function.
- For conditions that have no event, such as license or capacity limits, use a CloudWatch alarm.

Additional resources

For related concepts and reference material, see the following:

- [Build a job](#)
- [the section called "Configure jobs using queue environments"](#)
- [Monitoring](#)

Submission hooks

Submission hooks allow you to run custom scripts during the Deadline Cloud job submission workflow. Hooks run locally on the workstation that is submitting the job, before the job reaches the Deadline Cloud service. Hooks do not run on workers or in the AWS Cloud. Because hooks execute on your machine, they have access to local files, environment variables, and network resources available to the submitting user.

You can use hooks to validate job configurations, discover additional assets, modify submission parameters, or integrate with external systems such as production-tracking software. For other integration points that run on workers or in the cloud, see [Hooks and integration points](#).

There are two ways to configure hooks:

- **Bundle hooks** – Place a `hooks.yaml` (or `hooks.json`) file in your job bundle directory alongside `template.yaml`. Bundle hooks work well for CLI workflows where the bundle already exists before submission.
- **Environment hooks** – Point the `DEADLINE_HOOKS_DIR` environment variable to a directory that contains `hooks.yaml`. Environment hooks are useful for studios that want to enforce hooks across all submissions without modifying job bundles.

Both sources can be active at the same time. When both are present, environment hooks run first, then bundle hooks.

Note

In-application (DCC) submitters such as Maya, Nuke, or Blender run environment hooks for the pre-submission and post-submission phases only. The pre-GUI phase does not apply to DCC submitters. For more information, see [the section called “Pre-GUI hooks”](#).

Hook types

Deadline Cloud supports three hook types that correspond to different points in the submission workflow.

Pre-GUI hooks

Pre-GUI hooks run before the submission dialog opens. You can use pre-GUI hooks for the following tasks:

Important

Pre-GUI hooks run only with the standalone GUI submitter (`deadline bundle gui-submit`). They do not run in in-application (DCC) submitters such as Maya, Nuke, or Blender, because those applications build their submission dialog directly and do not invoke the pre-GUI phase. Pre-GUI hooks also do not apply to CLI submission (`deadline bundle submit`), which has no GUI phase. Pre-submission and post-submission hooks work across all submission methods.

- Pre-populate job name, description, and priority
- Set parameter defaults based on the current scene or pipeline context
- Query a project management system for task metadata

Pre-GUI hooks block the dialog from opening if they fail (non-zero exit code or timeout).

Output JSON to stdout to modify the initial dialog state. The following example shows the output format:

```
import json

output = {
    "name": "My Render - v042",
    "description": "Submitted via pipeline",
    "parameters": {
        "SceneFile": "/resolved/path/to/scene.ma",
        "OutputPath": "/shots/sh010/renders/",
        "deadline:priority": 75,
        "deadline:maxFailedTasksCount": 5,
        "deadline:maxRetriesPerTask": 3,
        "deadline:maxWorkerCount": 10,
        "deadline:targetTaskRunStatus": "READY"
    }
}
```

```
print(json.dumps(output))
```

The following table describes the output fields for pre-GUI hooks.

Field	Type	Description
name	String	Pre-fills the job name field.
description	String	Pre-fills the job description field.
parameters	Object	Pre-fills parameter values by name. Job template parameters use their name directly. Shared job properties use the <code>deadline:</code> prefix.

The following table describes the shared job properties that you can set with the `deadline:` prefix in the parameters object.

Key	Type	Description
<code>deadline:priority</code>	Integer	Priority (0–100).
<code>deadline:maxFailedTasksCount</code>	Integer	Maximum failed tasks before the job fails.
<code>deadline:maxRetriesPerTask</code>	Integer	Maximum retries per failed task.
<code>deadline:maxWorkerCount</code>	Integer	Maximum concurrent workers.
<code>deadline:targetTaskRunStatus</code>	String	Initial task status: <code>READY</code> or <code>SUSPENDED</code> .

 Note

CLI-supplied `--parameter` values take precedence over hook-supplied parameters.

Pre-submission hooks

Pre-submission hooks run before job attachments are hashed and uploaded. You can use pre-submission hooks for the following tasks:

- Validate job configuration
- Discover and add additional input files
- Modify job parameters such as priority
- Enforce studio policies

Pre-submission hooks block submission if they fail (non-zero exit code or timeout).

Output JSON to stdout to modify the submission. The hook output replaces asset references at the nested key level. If your hook outputs `inputFileNames`, the hook output replaces the entire `inputFileNames` list. Deadline Cloud preserves keys that you don't include in your output.

The following example adds discovered texture files to the submission:

```
import json
import os
import sys

metadata = json.load(sys.stdin)
bundle_dir = metadata["jobBundleDir"]

textures = []
for root, _, files in os.walk(bundle_dir):
    for f in files:
        if f.endswith(('.exr', '.png', '.jpg', '.tx')):
            textures.append(os.path.join(root, f))

if textures:
    print(json.dumps({
        "attachments": {
            "assetReferences": {
                "inputFileNames": textures
            }
        }
    }))
```

Pre-submission hooks can also modify job template parameter values by emitting a parameters map on stdout:

```
print(json.dumps({"parameters": {"SceneFile": "/resolved/scene.ma", "Quality":  
    "high"}}))
```

Parameter keys are job template parameter names. Values from a hook are applied on top of the bundle's parameter values, but CLI-supplied `--parameter` values still take precedence over hook-supplied ones.

Important

PATH parameters emitted on stdout must be absolute. A hook does not run from the submitting shell's working directory, so a relative PATH value on stdout is ambiguous and is rejected with an error. Emit an absolute path (for example, join with `DEADLINE_JOB_BUNDLE_DIR`), or write the value to `parameter_values.yaml/parameter_values.json` on disk instead, where a relative PATH is resolved against the job bundle directory.

Post-submission hooks

Post-submission hooks run after the `CreateJob` API call returns successfully. The job has been accepted by Deadline Cloud at this point. You can use post-submission hooks for the following tasks:

- Send notifications (Slack, email)
- Update tracking systems
- Log submission details

Post-submission hook failures are logged as warnings but do not affect the submitted job.

Configure submission hooks

Define hooks in a `hooks.yaml` or `hooks.json` file. Place the file in your job bundle directory alongside `template.yaml`, or in the directory specified by `DEADLINE_HOOKS_DIR`. If both formats exist in the same directory, the submitter reports an error.

The `version` field is required and must be `"1.0"`.

The following example shows a `hooks.yaml` configuration:

```
version: "1.0"
preGUI:
  - command: python3
    args: [scripts/prefill_from_shotgrid.py]
    timeout: 10

preSubmission:
  - command: python3
    args: [scripts/validate_assets.py]
    timeout: 60
    env:
      VALIDATION_LEVEL: strict

  - command: python3
    args: [scripts/discover_textures.py]

postSubmission:
  - command: python3
    args: [scripts/notify_slack.py]
    timeout: 15
    env:
      SLACK_WEBHOOK: https://hooks.slack.com/...
```

Hook definition fields

Each hook entry accepts the following fields.

Field	Required	Default	Description
<code>command</code>	Yes	–	Executable or interpreter (for example, <code>python3</code> or <code>bash</code>).
<code>args</code>	No	<code>[]</code>	Command-line arguments.
<code>timeout</code>	No	<code>60</code>	Maximum execution time in seconds.
<code>env</code>	No	<code>{}</code>	Additional environment variables. Hooks inherit the submitter's full environment. The

Field	Required	Default	Description
			DEADLINE_* variables and any hook-specific env values are layered on top.

Path resolution

Hook scripts are resolved according to the following rules:

- **Absolute paths** – Used as-is.
- **Relative paths** – Resolved relative to the job bundle directory.
- **Command names** – Searched in the system PATH.

Hook input

Hooks receive job metadata through JSON on stdin and through convenience environment variables.

Environment variables

The following environment variables are available to all hooks.

Variable	Description
DEADLINE_JOB_NAME	Job name.
DEADLINE_PRIORITY	Job priority.
DEADLINE_FARM_ID	Farm ID.
DEADLINE_QUEUE_ID	Queue ID.
DEADLINE_JOB_BUNDLE_DIR	Path to the job bundle directory.
DEADLINE_STORAGE_PROFILE_ID	Storage profile ID (if set).
DEADLINE_JOB_ID	Job ID (post-submission hooks only).

JSON on stdin

Complete metadata is provided as JSON on stdin. The following example shows the structure:

```
{
  "jobName": "My Render Job",
  "priority": 50,
  "farmId": "farm-abc123",
  "queueId": "queue-def456",
  "jobBundleDir": "/path/to/bundle",
  "parameters": {"SceneFile": "/path/to/scene.ma"},
  "submitterName": "Maya",
  "assetReferences": {
    "inputFileNames": ["/path/to/texture.exr"],
    "inputDirectories": [],
    "outputDirectories": ["/path/to/output"],
    "referencedPaths": []
  },
  "submissionPayload": {}
}
```

Security

Hooks are disabled by default. Each hook source has its own setting that you must enable.

Enabling bundle hooks

To allow hooks defined in `hooks.yaml` within job bundles, enable the bundle hooks setting.

To enable bundle hooks

- Run the following command:

```
deadline config set settings.allow_bundle_hooks true
```

Enabling environment hooks

To allow hooks from a directory specified by `DEADLINE_HOOKS_DIR`, enable the environment hooks setting and set the directory path.

To enable environment hooks

1. Enable the setting:

```
deadline config set settings.allow_environment_hooks true
```

2. Set the environment variable, typically in an application launcher script:

```
export DEADLINE_HOOKS_DIR=/studio/pipeline/hooks/blender
```

Confirmation prompt

When you enable hooks, the submitter prompts you to confirm before the hooks run. Pre-GUI hooks show the prompt before the dialog opens. Pre-submission and post-submission hooks show the prompt when you choose Submit.

The prompt shows which commands will run, allowing you to review before proceeding:

```
This job bundle contains submission hooks that will execute on your machine:
```

```
Pre-GUI hooks:
```

```
[1] python3 prefill_from_shotgrid.py
```

```
Pre-submission hooks:
```

```
[1] python3 validate_assets.py
```

```
Post-submission hooks:
```

```
[1] python3 notify.py
```

```
Bundle: /path/to/bundle
```

```
Do you want to run these hooks? [Y/n]
```

To skip confirmation prompts for CI/automation workflows, run the following command:

```
deadline config set settings.auto_accept true
```

Configuration settings summary

Setting	Default	Description
<code>settings.allow_bundle_hooks</code>	<code>false</code>	Specifies whether to allow hooks from the job bundle <code>hooks.yaml</code> file.
<code>settings.allow_environment_hooks</code>	<code>false</code>	Specifies whether to allow hooks from the <code>DEADLINE_HOOKS_DIR</code> directory.
<code>settings.auto_accept</code>	<code>false</code>	Specifies whether to skip confirmation prompts. Use with caution in CI/automation environments.

Studio deployment

Pipeline technical directors can configure hooks to run automatically for all artists by deploying environment hooks across workstations. Use this procedure when your studio has a shared network location for hook scripts and you have administrative access to configure artist workstations.

To deploy environment hooks for a studio

1. Configure workstations to allow environment hooks:

```
deadline config set settings.allow_environment_hooks true
```

2. Set `DEADLINE_HOOKS_DIR` in each application's launcher script:

```
# blender_launcher.sh
export DEADLINE_HOOKS_DIR=/studio/pipeline/hooks/blender
exec blender "$@"
```

3. Create the hooks at the specified location:

```
/studio/pipeline/hooks/blender/
### hooks.yaml
### validate_scene.py
```

Error handling

When a pre-submission or pre-GUI hook fails, the error output includes the following information:

- Which hook failed
- Exit code
- stdout and stderr output
- Timeout duration (if the hook timed out)

The submitter blocks submission until you resolve the issue. Post-submission hook failures are logged as warnings but do not affect the submitted job.

Best practices

- **Keep hooks fast.** Set appropriate timeouts and avoid long-running operations in hooks.
- **Log to stderr.** Reserve stdout for JSON output in pre-GUI and pre-submission hooks.
- **Handle errors gracefully.** Provide clear error messages on stderr so users can identify what went wrong.
- **Test with the CLI first.** CLI submission is easier to debug than GUI submission.
- **Use absolute paths in output.** When adding files to asset references, always use absolute paths.
- **Use environment hooks for studio-wide policies.** Environment hooks are more secure than bundle hooks because they are controlled by the studio rather than the bundle author.
- **Review bundle hooks before enabling.** Inspect hooks `.yaml` in bundles from untrusted sources before allowing bundle hooks to run.

Submission methods

Hooks work with the following submission methods:

- `deadline bundle submit` (CLI) – Pre-submission and post-submission hooks run. The CLI has no GUI phase, so pre-GUI hooks do not apply.
- `deadline bundle gui-submit` (standalone GUI) – All phases run, including pre-GUI hooks.
- In-application (DCC) submitters – Pre-submission and post-submission hooks run. DCC submitters do not invoke the pre-GUI phase.

The standalone GUI copies hooks .yaml to the job history bundle and resolves script paths back to your original bundle directory.

Additional resources

For more information about integration points and related topics, see the following:

- [*Hooks and integration points*](#)
- [the section called “Submit a job”](#)
- [*Build a job*](#)
- [deadline-cloud](#) repository on the GitHub website

Create and use Deadline Cloud customer-managed fleets

When you create a customer-managed fleet (CMF), you have full control over your processing pipeline. You define the network and software environment for each worker. Deadline Cloud acts as the repository and scheduler for your jobs. For more information about choosing a fleet type, see [Choose between service-managed and customer-managed fleets](#) in the *Deadline Cloud User Guide*.

A worker may be an Amazon Elastic Compute Cloud (Amazon EC2) instance, a worker in a co-location facility, or an on-premises worker. Each worker must run the Deadline Cloud worker agent. All workers must have access to [the Deadline Cloud service endpoint](#).

The following topics show you how to create a basic CMF using Amazon EC2 instances.

Topics

- [Create a customer-managed fleet](#)
- [Worker host setup and configuration](#)
- [Manage access to Windows job user secrets](#)
- [Install and configure software required for jobs](#)
- [Configuring AWS credentials](#)
- [Worker host data flow for customer-managed fleets](#)
- [Test the configuration of your worker host](#)
- [Create an Amazon Machine Image](#)
- [Create fleet infrastructure with an Amazon EC2 Auto Scaling group](#)

Create a customer-managed fleet

To create a customer-managed fleet (CMF), complete the following steps.

Deadline Cloud console

To use the Deadline Cloud console to create a customer-managed fleet

1. Open the Deadline Cloud [console](#).
2. Select **Farms**. A list of available farms displays.
3. Select the name of the **Farm** you want to work in.

4. Select the **Fleets** tab, and then choose **Create fleet**.
5. Enter a **Name** for your fleet.
6. (Optional) Enter a **Description** for your fleet.
7. Select **Customer managed** for **Fleet type**.
8. Select your fleet's service access.
 - a. We recommend using the **Create and use a new service role** option for each fleet for more granular permissions control. This option is selected by default.
 - b. You can also use an existing service role by selecting **Choose a service role**.
9. Review your selections, then choose **Next**.
10. Select an **operating system** for your fleet. All of a fleet's workers must have a common operating system.
11. Select the **host CPU architecture**.
12. Select the minimum and maximum vCPU and memory **Hardware capabilities** to meet the workload demands of your fleets.
13. Select an Auto Scaling type. For more information, see [Use EventBridge to handle Auto Scaling events](#).
 - **No scaling:** You are creating an on-premises fleet and want opt out of Deadline Cloud Auto Scaling.
 - **Scaling recommendations:** You are creating an Amazon Elastic Compute Cloud (Amazon EC2) fleet.
14. (Optional) Select the arrow to expand the Add capabilities section.
15. (Optional) Select the checkbox for **Add GPU capability - Optional**, then enter the minimum and maximum GPUs and memory.
16. Review your selections, then choose **Next**.
17. (Optional) Define custom worker capabilities, then choose **Next**.
18. Using the dropdown, select one or more **queues** to associate with the fleet.

 Note

We recommend associating a fleet only with queues that are all in the same trust boundary. This recommendation ensures a strong security boundary between running jobs on the same worker.

19. Review the queue associations, then choose **Next**.
20. (Optional) For Default conda queue environment, we'll create an environment for your queue that will install conda packages requested by jobs.

 Note

The conda queue environment is used to install conda packages requested by jobs. Typically, you should uncheck the conda queue environment on queues associated with CMFs because CMFs won't have the required conda commands installed by default.

21. (Optional) Add tags to your CMF. For more information, see [Tagging your AWS resources](#).
22. Review your fleet configuration and make any changes, then choose **Create fleet**.
23. Select the **Fleets** tab, then note the **Fleet ID**.

AWS CLI

To use the AWS CLI to create a customer-managed fleet

1. Open a terminal.
2. Create `fleet-trust-policy.json` in a new editor.
 - a. Add the following IAM policy, replacing the *ITALICIZED* text with your AWS account ID and Deadline Cloud farm ID.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.deadline.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        }
      }
    }
  ]
}
```

```

        },
        "ArnEquals": {
            "aws:SourceArn":
"arn:aws:deadline:*:111122223333:farm/FARM_ID"
        }
    }
}
]
}

```

b. Save your changes.

3. Create fleet-policy.json.

a. Add the following IAM policy.

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "deadline:AssumeFleetRoleForWorker",
        "deadline:UpdateWorker",
        "deadline>DeleteWorker",
        "deadline:UpdateWorkerSchedule",
        "deadline:BatchGetJobEntity",
        "deadline:AssumeQueueRoleForWorker"
      ],
      "Resource": "arn:aws:deadline:*:111122223333:*",
      "Condition": {
        "StringEquals": {
          "aws:PrincipalAccount": "${aws:ResourceAccount}"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs>CreateLogStream"
      ],

```



```

        "Resource": "arn:aws:logs:*:*:*://deadline/*",
        "Condition": {
            "StringEquals": {
                "aws:PrincipalAccount": "${aws:ResourceAccount}"
            }
        },
    },
    {
        "Effect": "Allow",
        "Action": [
            "logs:PutLogEvents",
            "logs:GetLogEvents"
        ],
        "Resource": "arn:aws:logs:*:*:*://aws/deadline/*",
        "Condition": {
            "StringEquals": {
                "aws:PrincipalAccount": "${aws:ResourceAccount}"
            }
        }
    }
]
}

```

b. Save your changes.

4. Add an IAM role for the workers in your fleet to use.

```

aws iam create-role --role-name FleetWorkerRoleName --assume-role-policy-
document file://fleet-trust-policy.json
aws iam put-role-policy --role-name FleetWorkerRoleName --policy-name
FleetWorkerPolicy --policy-document file://fleet-policy.json

```

5. Create `create-fleet-request.json`.

a. Add the following IAM policy, replacing the *ITALICIZED* text with your CMF's values.

 Note

You can find the *ROLE_ARN* in the `create-cmf-fleet.json`.
For the *OS_FAMILY*, you must choose one of `linux`, `macos` or `windows`.

```
{
```

```

    "farmId": "FARM_ID",
    "displayName": "FLEET_NAME",
    "description": "FLEET_DESCRIPTION",
    "roleArn": "ROLE_ARN",
    "minWorkerCount": 0,
    "maxWorkerCount": 10,
    "configuration": {
      "customerManaged": {
        "mode": "NO_SCALING",
        "workerCapabilities": {
          "vCpuCount": {
            "min": 1,
            "max": 4
          },
          "memoryMiB": {
            "min": 1024,
            "max": 4096
          },
          "osFamily": "OS_FAMILY",
          "cpuArchitectureType": "x86_64"
        }
      }
    }
  }
}

```

b. Save your changes.

6. Create your fleet.

```
aws deadline create-fleet --cli-input-json file://create-fleet-request.json
```

Worker host setup and configuration

A worker host refers to a host machine that runs a Deadline Cloud worker. This section explains how to set up the worker host and configure it for your specific needs. Each worker host runs a program called a *worker agent*. The worker agent is responsible for:

- Managing the worker life cycle.
- Synchronizing assigned work, its progress and results.
- Monitoring running work.
- Forwarding logs to configured destinations.

We recommend that you use the provided Deadline Cloud worker agent. The worker agent is open source and we encourage feature requests, but you can also develop and customize to fit your needs.

To complete the tasks in the following sections, you need the following:

Linux

- A Linux-based Amazon Elastic Compute Cloud (Amazon EC2) instance. We recommend Amazon Linux 2023.
- `sudo` privileges
- Python 3.9 or above

Windows

- A Windows-based Amazon Elastic Compute Cloud (Amazon EC2) instance. We recommend Windows Server 2022.
- Administrator access to the worker host
- Python 3.9 or above installed for all users

Create and configure a Python virtual environment

You can create a Python virtual environment on Linux if you have installed Python 3.9 or greater and placed it in your PATH.

Note

On Windows, agent files must be installed into Python's global site-packages directory. Python virtual environments are not currently supported.

To create and activate a Python virtual environment

1. Open a terminal as the `root` user (or use `sudo / su`).
2. Create and activate a Python virtual environment.

```
python3 -m venv /opt/deadline/worker
source /opt/deadline/worker/bin/activate
pip install --upgrade pip
```

Install Deadline Cloud worker agent

After you've set up your Python and created a virtual environment on Linux, install the Deadline Cloud worker agent Python packages.

To install the worker agent Python packages

Linux

1. Open a terminal as the root user (or use `sudo / su`).
2. Download and install the Deadline Cloud worker agent packages from PyPI:

```
/opt/deadline/worker/bin/python -m pip install deadline-cloud-worker-agent
```

Windows

1. Open an administrator command prompt or PowerShell terminal.
2. Download and install the Deadline Cloud worker agent packages from PyPI:

```
python -m pip install deadline-cloud-worker-agent
```

When your Windows worker host requires long path names (greater than 250 characters), you must enable long path names as follows:

To enable long paths for Windows worker hosts

1. Make sure that the long path registry key is enabled. For more information, see [Registry setting to enable long paths](#) on the Microsoft website.
2. Install the Windows SDK for Desktop C++ x86 Apps. For more information, see [Windows SDK](#) on the Microsoft website.

3. Open the Python installation location in your environment where the worker agent is installed. The default is `C:\Program Files\Python311`. There is an executable file named `python-service.exe`.
4. Create a new file called `python-service.exe.manifest` in the same location. Add the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity type="win32" name="python-service" processorArchitecture="x86"
    version="1.0.0.0"/>
  <application xmlns="urn:schemas-microsoft-com:asm.v3">
    <windowsSettings>
      <longPathAware xmlns="http://schemas.microsoft.com/SMI/2016/
WindowsSettings">true</longPathAware>
    </windowsSettings>
  </application>
</assembly>
```

5. Open a command prompt and run the following command in the location of the manifest file that you created:

```
"C:\Program Files (x86)\Windows Kits\10\bin\10.0.26100.0\x86\mt.exe" -manifest
python-service.exe.manifest -outputresource:python-service.exe;#1
```

You should see output similar to the following:

```
Microsoft (R) Manifest Tool
Copyright (c) Microsoft Corporation.
All rights reserved.
```

The worker is now able to access long paths. To clean up, remove the `python-service.exe.manifest` file and uninstall the SDK.

If jobs still fail with missing-file errors after you complete this procedure, the limit is being applied by the rendering application rather than by the worker agent. For more information, see [Why does my job fail on Windows when my file paths are long?](#) in the *AWS Deadline Cloud User Guide*.

Configure the Deadline Cloud worker agent

You can configure the Deadline Cloud worker agent settings in three ways. We recommend you use the operating system setup by running the `install-deadline-worker` tool.

Command line arguments — You can specify arguments when you run the Deadline Cloud worker agent from the command line. Some configuration settings are not available through command line arguments. To see all the available command line arguments, enter `deadline-worker-agent --help`.

Environment variables — You can configure the Deadline Cloud worker agent by setting environment variable beginning with `DEADLINE_WORKER_`. For example, to see all the available command line arguments you can use `export DEADLINE_WORKER_VERBOSE=true` to set the worker agent's output to verbose. For more examples and information, see `/etc/amazon/deadline/worker.toml.example` on Linux or `C:\ProgramData\Amazon\Deadline\Config\worker.toml.example` on Windows.

Configuration file — When you install the worker agent, it creates a configuration file located at `/etc/amazon/deadline/worker.toml` on Linux or `C:\ProgramData\Amazon\Deadline\Config\worker.toml` on Windows. The worker agent loads this configuration file when it starts. You can use the example configuration file (`/etc/amazon/deadline/worker.toml.example` on Linux or `C:\ProgramData\Amazon\Deadline\Config\worker.toml.example` on Windows) to tailor the default worker agent configuration file for your specific needs.

Finally, we recommend you enable auto shutdown for the worker agent *after* your software is deployed and working as expected. This allows the worker fleet to scale up when needed and to shut down when a job finishes. Auto scaling helps ensure you're only using the resources needed. To enable an instance started by the auto scaling group to shut down, you must add `shutdown_on_stop=true` to the `worker.toml` configuration file.

To enable auto shutdown

As a **root** user:

- Install the worker agent with parameters **`--allow-shutdown`**.

Linux

Enter:

```
/opt/deadline/worker/bin/install-deadline-worker \  
--farm-id FARM_ID \  
--fleet-id FLEET_ID \  
--region REGION \  
--allow-shutdown
```

Windows

Enter:

```
install-deadline-worker ^  
--farm-id FARM_ID ^  
--fleet-id FLEET_ID ^  
--region REGION ^  
--allow-shutdown
```

Create job users and groups

This section describes the required user and group relationship between the agent user and the `jobRunAsUser` defined on your queues.

The Deadline Cloud worker agent should run as a dedicated agent-specific user on the host. You should configure the `jobRunAsUser` property of Deadline Cloud queues so that workers will run the queue jobs as a specific operating system user and group. This configuration means you can control the shared filesystem permissions that your jobs have. It also provides as an important security boundary between your jobs and the worker agent user.

Linux job users and groups

To set up a local worker agent user and `jobRunAsUser`, ensure you meet the following requirements. If you are using a Linux Pluggable Authentication Module (PAM) such as Active Directory or LDAP, your procedure may be different.

The worker agent user and the shared `jobRunAsUser` group are set when you install the worker agent. The defaults are `deadline-worker-agent` and `deadline-job-users`, but you can change those when you install the worker agent.

```
install-deadline-worker \  

```

```
--user AGENT_USER_NAME \  
--group JOB_USERS_GROUP
```

Commands should be run as the root user.

- Each `jobRunAsUser` should have a matching primary group. Creating a user with the `adduser` command usually creates a matching primary group.

```
adduser -r -m jobRunAsUser
```

- The primary group of the `jobRunAsUser` is a secondary group for the worker agent user. The shared group allows the worker agent to make files available to the job as it is running.

```
usermod -a -G jobRunAsUser deadline-worker-agent
```

- The `jobRunAsUser` must be a member of the shared job group.

```
usermod -a -G deadline-job-users jobRunAsUser
```

- The `jobRunAsUser` must not belong to the worker agent user's primary group. Sensitive files written by the worker agent are owned by the agent's primary group. If a `jobRunAsUser` is part of this group, worker agent files may be accessible to jobs running on the worker.
- The default AWS Region must match the Region of the farm that the worker belongs to. Apply this setting to all `jobRunAsUser` accounts on the worker.

```
sudo -u jobRunAsUser aws configure set default.region aws-region
```

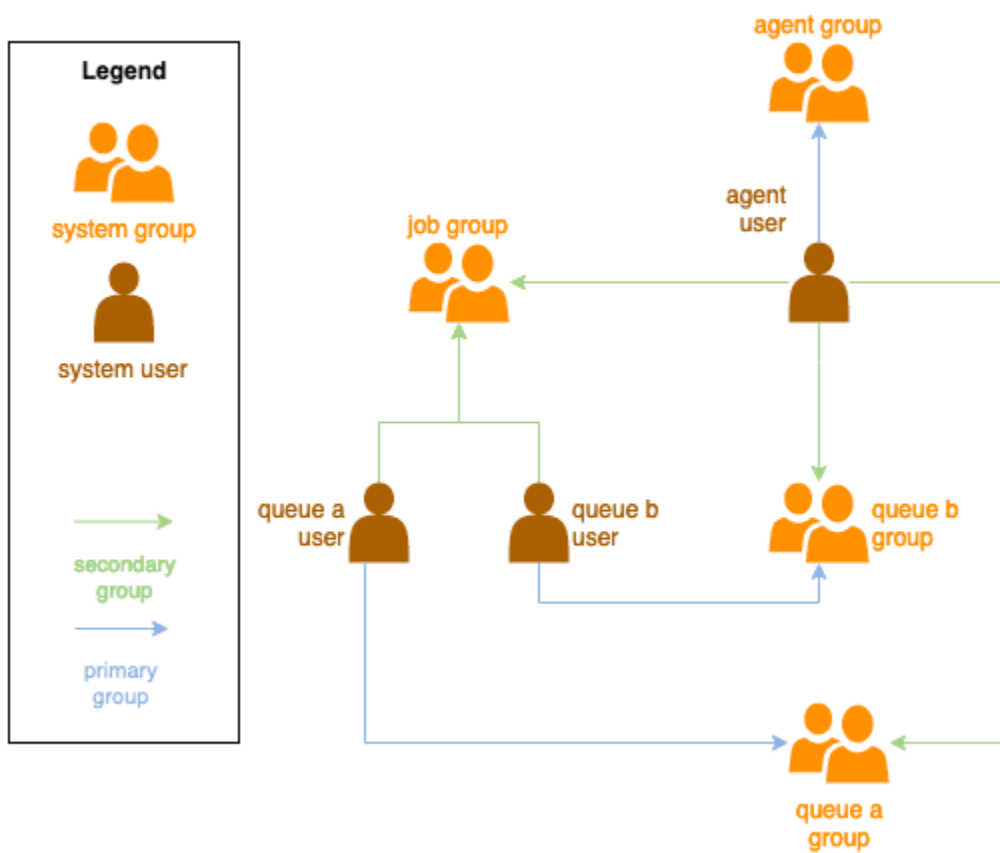
- The worker agent user must be able to run `sudo` commands as the `jobRunAsUser`. Run the following command to open an editor to create a new `sudoers` rule:

```
visudo -f /etc/sudoers.d/deadline-worker-job-user
```

Add the following to the file:

```
# Allows the Deadline Cloud worker agent OS user to run commands  
# as the queue OS user without requiring a password.  
deadline-worker-agent ALL=(jobRunAsUser) NOPASSWD:ALL
```


The following diagram illustrates the relationship between the agent user and the `jobRunAsUser` users and groups for queues associated with the fleet.



Windows users

To use a Windows user as the `jobRunAsUser`, it must meet the following requirements:

- All queue `jobRunAsUser` users must exist.
- Their passwords must match the value of the secret specified in their queue's `JobRunAsUser` field. For instructions, see step 7 in [Deadline Cloud queues](#) in the *AWS Deadline Cloud User Guide*.
- The agent-user must be able to log on as those users.

Securing your worker host

When setting up your worker host, follow security best practices to protect sensitive information and maintain proper access controls.

Configuring log folder permissions

The worker agent writes log files that may contain sensitive information from host configuration scripts and job execution. The `install-deadline-worker` command creates the log directory with secure permissions. If you need to create the directory manually before installation, use the following procedures to match the permissions used by service-managed fleets:

Linux

To configure log directory permissions on Linux

1. Create the log directory:

```
sudo mkdir -p /var/log/amazon/deadline
```

2. Set the owner and group to the worker agent user:

```
sudo chown -R deadline-worker-agent:deadline-worker-agent /var/log/amazon/deadline
```

3. Set permissions to 750:

```
sudo chmod -R 750 /var/log/amazon/deadline
```

These permissions ensure that only the worker agent user and group can access the log files, preventing job users and other unauthorized users from reading potentially sensitive information.

Windows

To configure log directory permissions on Windows

1. Open an administrator PowerShell terminal.
2. Create the log directory:

```
New-Item -ItemType Directory -Force -Path "$env:PROGRAMDATA\Amazon\Deadline\Logs"
```

3. Configure restricted ACLs to allow only the worker agent user and Administrators:

```
$acl = Get-Acl "$env:PROGRAMDATA\Amazon\Deadline\Logs"
$acl.SetAccessRuleProtection($true, $false)
$acl.Access | ForEach-Object { $acl.RemoveAccessRule($_) }
$agentRule = New-Object
System.Security.AccessControl.FileSystemAccessRule("deadline-worker",
"FullControl", "ContainerInherit,ObjectInherit", "None", "Allow")
$adminRule = New-Object
System.Security.AccessControl.FileSystemAccessRule("Administrators",
"FullControl", "ContainerInherit,ObjectInherit", "None", "Allow")
$acl.AddAccessRule($agentRule)
$acl.AddAccessRule($adminRule)
Set-Acl "$env:PROGRAMDATA\Amazon\Deadline\Logs" $acl
```

These commands restrict access to the log directory to only the worker agent user and Administrators group, preventing job users and other unauthorized users from reading potentially sensitive information.

Manage access to Windows job user secrets

When you configure a queue with a Windows `jobRunAsUser`, you must specify an AWS Secrets Manager secret. The value of this secret is expected to be JSON-encoded object of the form:

```
{
  "password": "JOB_USER_PASSWORD"
}
```

For Workers to run jobs as the queue's configured `jobRunAsUser`, the fleet's IAM role must have permissions to get the value of the secret. If the secret is encrypted using a customer-managed KMS key, then the fleet's IAM role must also have permissions to decrypt using the KMS key.

It is highly recommended to follow the principle of least-privilege for these secrets. Following least privilege means that access to fetch the secret value of a queue's `jobRunAsUser` → `windows` → `passwordArn` should be:

- granted to a fleet role when a queue-fleet association is created between the fleet and queue
- revoked from a fleet role when a queue-fleet association is deleted between the fleet and queue

Further, the AWS Secrets Manager secret containing the `jobRunAsUser` password should be deleted when it is no longer being used.

Grant access to a password secret

Deadline Cloud fleets require access to the `jobRunAsUser` password stored in the queue's password secret when the queue and fleet are associated. We recommend using the AWS Secrets Manager resource policy to grant access to the fleet roles. If you strictly adhere to this guideline, it is easier to determine which fleet roles have access to the secret.

To grant access to the secret

1. Open the AWS Secret Manager console to the secret.
2. In the Resource permissions section, add a policy statement of the form:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:role/FleetRole"
      },
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": "arn:aws:secretsmanager:us-east-1:111122223333:secret:YourSecretName-ABC123"
    }
  ]
}
```

Revoke access to a password secret

When a fleet no longer requires access to a queue, remove access to the password secret for the queue `jobRunAsUser`. We recommend using the AWS Secrets Manager resource policy to grant

access to the fleet roles. If you strictly adhere to this guideline, it is easier to determine which fleet roles have access to the secret.

To revoke access to the secret

1. Open the AWS Secret Manager console to the secret.
2. In the Resource permissions section, remove the policy statement of the form:

JSON

```
{
  "Version": "2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Principal" : {
        "AWS" : "arn:aws:iam::111122223333:role/FleetRole"
      },
      "Action" : [
        "secretsmanager:GetSecretValue"
      ],
      "Resource" : "arn:aws:secretsmanager:us-
east-1:111122223333:secret:YourSecretName-ABC123"
    }
  ]
}
```

Install and configure software required for jobs

After you set up the Deadline Cloud worker agent, you can prepare the worker host with any software that is required to run jobs.

When you submit a job to a queue with an associated `jobRunAsUser`, the job runs as that user. When a job is submitted with commands that are not an absolute path, that command must be found in the `PATH` of that user.

On Linux, you might specify the `PATH` for a user in one of the following:

- their `~/.bashrc` or `~/.bash_profile`

- system configuration files such as `/etc/profile.d/*` and `/etc/profile`
- shell startup scripts: `/etc/bashrc`.

On Windows, you might specify the PATH for a user in one of the following:

- their user-specific environment variables
- the system-wide environment variables

Install digital content creation tool adaptors

Deadline Cloud provides Open Job Description (OpenJD) adaptors for using popular digital content creation (DCC) applications. An adaptor is a command-line program that runs the DCC application on the worker host, keeps the application loaded between tasks, reports render progress to the job's logs, and applies path mapping. For more information, see [Adaptor packages](#).

To use these adaptors in a customer-managed fleet, you must install the DCC software and the application adaptors. Then, ensure the software's executable programs are available on the system search path (for example, in the PATH environment variable).

To install a digital content creation (DCC) adaptor on a customer-managed fleet

1. Open a terminal:
 - a. On Linux, open a terminal as the `root` user (or use `sudo` or `su`).
 - b. On Windows, open an administrator command prompt or PowerShell terminal.
2. Install the adaptor package for your DCC application. The following example installs the Maya adaptor:

```
pip install deadline-cloud-for-maya
```

The following table lists the adaptor package name and documentation link for each supported DCC application. All adaptor packages are available on the GitHub website:

DCC application	Adaptor package
Adobe After Effects	deadline-cloud-for-after-effects

DCC application	Adaptor package
Autodesk 3ds Max	deadline-cloud-for-3ds-max
Autodesk Maya	deadline-cloud-for-maya
Autodesk VRED	deadline-cloud-for-vred
Blender	deadline-cloud-for-blender
Chaos V-Ray for Maya	deadline-cloud-for-maya
Foundry Nuke	deadline-cloud-for-nuke
KeyShot Studio	deadline-cloud-for-keyshot
Maxon Cinema 4D	deadline-cloud-for-cinema-4d
Maxon Redshift for Maya	deadline-cloud-for-maya
SideFX Houdini	deadline-cloud-for-houdini
Unreal Engine	deadline-cloud-for-unreal-engine

Installing multiple adaptors on the same worker

If you install more than one DCC adaptor on the same worker, install each adaptor into its own Python virtual environment. Adaptors can pin different version ranges for shared Python packages (for example, `deadline` or `openjd-adaptor-runtime`). An environment with multiple adaptors might fail with a `ResolutionImpossible` error. It might also silently downgrade a shared package and break the adaptors that need the newer version.

Configuring AWS credentials

The initial phase of the worker life cycle is bootstrapping. In this phase the worker agent software creates a worker in your fleet, and obtains AWS credentials from your fleet's role for further operation.

AWS credentials for Amazon EC2

To create an IAM role for Amazon EC2 with Deadline Cloud worker host permissions

1. Open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, choose **Roles** in the navigation pane, then choose **Create role**.
3. Select **AWS service**.
4. Select **EC2** as the **Service or use case**, then select **Next**.
5. To grant the necessary permissions, attach the `AWSDeadlineCloud-WorkerHost` AWS managed policy.

On-premises AWS credentials

Your on-premises workers use credentials to access Deadline Cloud. For the most secure access, we recommend using IAM Roles Anywhere to authenticate your workers. For more information, see [IAM Roles Anywhere](#).

For testing, you can use IAM user access keys for AWS credentials. We recommend that you set an expiration for the IAM user by including a restrictive inline policy.

Important

Heed the following warnings:

- **Do NOT** use your account's root credentials to access AWS resources. These credentials provide unrestricted account access and are difficult to revoke.
- **Do NOT** put literal access keys or credential information in your application files. If you do, you create a risk of accidentally exposing your credentials if, for example, you upload the project to a public repository.
- **Do NOT** include files that contain credentials in your project area.
- Secure your access keys. Do not provide your access keys to unauthorized parties, even to help [find your account identifiers](#). By doing this, you might give someone permanent access to your account.
- Be aware that any credentials stored in the shared AWS credentials file are stored in plain text.

For more details, see [Best practices for managing AWS access keys in the AWS General Reference](#).

Create an IAM user

1. Open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, select **Users** and then select **Create user**.
3. Name the user. Clear the checkbox for **Provide user access to the AWS Management Console**, then choose **Next**.
4. Choose **Attach policies directly**.
5. From the list of permission policies, choose the **AWSDeadlineCloud-WorkerHost** policy and then choose **Next**.
6. Review the user details and then choose **Create user**.

Restrict user access to a limited time window

Any IAM user access keys that you create are long-term credentials. To ensure that these credentials expire in case they are mishandled, you can make these credentials time-bound by creating an inline policy that specifies a date after which the keys will no longer be valid.

1. Open the IAM user that you just created. In the Permissions tab, choose **Add permissions** and then choose **Create inline policy**.
2. In the JSON editor, specify the following permissions. To use this policy, replace the `aws:CurrentTime` timestamp value in the example policy with your own time and date.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "*",
      "Resource": "*",
      "Condition": {
        "DateGreaterThan": {
          "aws:CurrentTime": "2024-01-01T00:00:00Z"
        }
      }
    }
  ]
}
```

```
}  
}  
]  
}
```

Create an access key

1. On the user details page, select the **Security credentials** tab. In the **Access keys** section, choose **Create access key**.
2. Indicate that you want to use the key for Other, then choose **Next**, then choose **Create access key**.
3. On the **Retrieve access keys** page, choose **Show** to reveal the value of your user's secret access key. You can copy the credentials or download a .csv file.

Store the user access keys

- Store the user access keys in the agent user's AWS credentials file on the worker host system:
 - On Linux, the file is located at `~/.aws/credentials`
 - On Windows, the file is located at `%USERPROFILE\.aws\credentials`

Replace the following keys:

```
[default]  
aws_access_key_id=ACCESS_KEY_ID  
aws_secret_access_key=SECRET_ACCESS_KEY
```

Important

When you no longer need this IAM user, we recommend that you remove it to align with the [AWS security best practice](#). We recommend that you require your human users to use temporary credentials through [AWS IAM Identity Center](#) when accessing AWS.

Worker host data flow for customer-managed fleets

This topic describes the network connections that AWS Deadline Cloud (Deadline Cloud) worker hosts make during operation, including the endpoints contacted, protocols used, and data transmitted. This information applies to customer-managed fleet (CMF) workers, including both Amazon Elastic Compute Cloud (Amazon EC2) instances and on-premises workers. Use this information to configure firewall rules, create VPC endpoints, perform security audits, or plan network policies for your worker hosts. For information about service-managed fleet networking, see [Inter-network traffic privacy](#).

All worker communication is outbound only. Worker hosts initiate all connections—you don't need to allow any inbound connections. All connections use HTTPS (TLS 1.2 or later) over port 443.

This topic includes the following sections:

- [the section called “Endpoints and protocols”](#)
- [the section called “API operations used by workers”](#)
- [the section called “Other data transmitted”](#)
- [the section called “Private connectivity options”](#)

Endpoints and protocols

The following table lists the AWS service endpoints that worker hosts connect to during operation. For the complete list of regional endpoints for each service, see the [Service endpoints and quotas](#) in the *AWS General Reference*.

Worker host endpoint reference

AWS service	Endpoint	Port / Protocol	Purpose	Required
Deadline Cloud (scheduling)	scheduling.deadline. e. <i>[Region]</i> .amazonaws.com	443 / HTTPS	Worker registration, task polling, status updates, credential exchange, job entity retrieval. See the section called “API operations used by workers” .	Always

AWS service	Endpoint	Port / Protocol	Purpose	Required
Amazon CloudWatch Logs (CloudWatch Logs)	logs.[<i>Region</i>].amazonaws.com	443 / HTTPS	Worker agent and session log delivery.	Always
Amazon Simple Storage Service (Amazon S3)	s3.[<i>Region</i>].amazonaws.com	443 / HTTPS	Job attachment upload and download.	If using job attachments

If your jobs use other AWS services, you might also need to allow outbound connections to those service endpoints.

API operations used by workers

All of the following API operations use the `scheduling.deadline.[Region].amazonaws.com` endpoint. For the complete request and response schemas of each operation, see the [Deadline Cloud API Reference](#).

Bootstrap phase

When a worker host starts, the worker agent registers with the fleet. The bootstrap credentials require the permissions in the `AWSDeadlineCloud-WorkerHost` AWS managed policy, or equivalent custom permissions. The bootstrap phase uses the following API operations:

- [CreateWorker](#) – Registers the worker with the fleet. Sends the host name and IP addresses. Receives a worker ID.
- [AssumeFleetRoleForWorker](#) – Obtains fleet role credentials. Receives temporary AWS credentials that the worker agent uses for subsequent operations.

Operational phase

After bootstrap, the worker agent polls for work and processes sessions. The fleet role requires the permissions in the `AWSDeadlineCloud-FleetWorker` AWS managed policy, or equivalent custom permissions, and uses the following API operations:

- [UpdateWorker](#) – Updates the worker status, for example to STOPPED during shutdown.
- [UpdateWorkerSchedule](#) – Polls for work assignments. Sends session action status updates including completion status, progress percent, progress message, and output manifest hashes. Receives assigned sessions (job ID, queue ID, session actions, log configuration), cancellation requests, desired worker status, and the update interval.
- [BatchGetJobEntity](#) – Fetches job details for assigned work. Sends job entity identifiers. Receives job details, environment details, and job attachment details.
- [AssumeFleetRoleForWorker](#) – Periodically refreshes fleet role credentials.
- [AssumeQueueRoleForWorker](#) – Obtains queue role credentials scoped to a specific queue. The worker uses these credentials to access job attachments in Amazon S3.

Other data transmitted

In addition to the Deadline Cloud scheduling API operations, worker hosts transmit the following data to other AWS services:

Log data

The worker agent sends worker agent logs and session logs (stdout and stderr from job processes) to CloudWatch Logs using the `PutLogEvents` API operation.

Job attachments

Workers transfer input and output files through Amazon S3 using `GetObject` and `PutObject` API operations. The worker uses queue role credentials obtained through `AssumeQueueRoleForWorker` for this access.

Telemetry (optional)

The worker agent sends operational metrics such as crash reports. You can opt out of telemetry collection. For more information, see [Opt out](#).

Private connectivity options

You can use AWS PrivateLink to keep traffic between CMF worker hosts and Deadline Cloud within your VPC, without traversing the public internet. For on-premises workers, you can combine AWS PrivateLink with AWS Direct Connect (Direct Connect) or a VPN connection. For more information, see [Access AWS Deadline Cloud using an interface endpoint \(AWS PrivateLink\)](#).

Test the configuration of your worker host

After you have installed the worker agent, installed the software necessary to process your jobs, and configured the AWS credentials for the worker agent, you should test that the installation can process your jobs before creating an AMI for your fleet. You should test the following:

- The Deadline Cloud worker agent is properly configured to run as a system service.
- That the worker polls the associated queue for work.
- That the worker successfully processes jobs sent to the queue associated with the fleet.

After you test the configuration and can successfully process representative jobs, you can use the configured worker to create an AMI for Amazon EC2 workers, or as a model for your on-premises workers.

Note

If you are testing the worker host configuration of an auto scaling fleet, you may have difficulty testing your worker in the following situations:

- If there is no work in the queue, Deadline Cloud stops the worker agent shortly after the worker starts.
- If the worker agent is configured to shut down the host when stopping, the agent shuts down the machine if there is no work in the queue.

To avoid these issues, use a staging fleet that doesn't auto scale to configure and test your workers. After testing the worker host, be sure to set the correct fleet ID before baking an AMI.

To test your worker host configuration

1. Run the worker agent by starting the operating system service.

Linux

From a root shell run the following command:

```
systemctl start deadline-worker
```

Windows

From an administrator command prompt or PowerShell terminal, enter the following command:

```
sc.exe start DeadlineWorker
```

2. Monitor the worker to make sure it starts and polls for work.

Linux

From a root shell run the following command:

```
systemctl status deadline-worker
```

The command should return a response like:

```
Active: active (running) since Wed 2023-06-14 14:44:27 UTC; 7min ago
```

If the response doesn't look like that, inspect the log file using the following command:

```
tail -n 25 /var/log/amazon/deadline/worker-agent.log
```

Windows

From an administrator command prompt or PowerShell terminal, enter the following command:

```
sc.exe query DeadlineWorker
```

The command should return a response like:

```
STATE      : 4 RUNNING
```

If the response doesn't contain RUNNING, inspect the worker log file. Open and administrator PowerShell prompt and run the following command:

```
Get-Content -Tail 25 -Path $env:PROGRAMDATA\Amazon\Deadline\Logs\worker-agent.log
```

3. Submit jobs to queue associated with your fleet. The jobs should be representative of the jobs that the fleet processes.
4. Monitor the progress of the job [using the Deadline Cloud monitor](#) or CLI. If a job fails, check the session and worker logs.
5. Update the configuration of the worker host as needed until jobs complete successfully.
6. When the test jobs succeed you can stop the worker:

Linux

From a root shell run the following command:

```
systemctl stop deadline-worker
```

Windows

From an administrator command prompt or PowerShell terminal, enter the following command:

```
sc.exe stop DeadlineWorker
```

Create an Amazon Machine Image

To create an Amazon Machine Image (AMI) to use in an Amazon Elastic Compute Cloud (Amazon EC2) customer-managed fleet (CMF), complete the tasks in this section. You must create an Amazon EC2 instance before proceeding. For more information, see [Launch your instance](#) in the *Amazon EC2 User Guide for Linux Instances*.

⚠ Important

Creating an AMI creates a snapshot of the Amazon EC2 instance's attached volumes. Any software installed on the instance persists so instances, which are reused when you launch instances from the AMI. We recommend adopting a patching strategy and regularly updating any new AMI with updated software before applying to your fleet.

Prepare the Amazon EC2 instance

Before you build an AMI, you must delete the worker state. The worker state persists between worker agent launches. If this state persists onto the AMI, then all instances launched from it will share the same state.

We also recommend you delete any existing log files. Log files can remain on an Amazon EC2 instance when you prepare the AMI. Deleting these files minimizes confusion when diagnosing possible issue in worker fleets that use the AMI.

You should also enable the worker agent system service so the Deadline Cloud worker agent launch when the Amazon EC2 is started.

Finally, we recommend you enable the worker agent auto shutdown. Auto shutdown allows the worker fleet to scale up when needed and to shutdown when the rendering job finishes. This auto scaling helps ensure you're only using resources as needed.

To prepare the Amazon EC2 instance

1. Open the Amazon EC2 console.
2. Launch an Amazon EC2 instance. For more information, see [Launch your instance](#).
3. Set up the host to connect to your identity provider (IdP), then mount any shared filesystem it needs.
4. Follow the tutorials to [Install Deadline Cloud worker agent](#), then [Configure worker agent](#), and [Create job users and groups](#).
5. If you are preparing an AMI based on Amazon Linux 2023 to run software compatible with the VFX Reference Platform, you need to update several requirements. For information, see [VFX Reference Platform compatibility](#) in the *AWS Deadline Cloud User Guide*.
6. Open a terminal.

- a. On Linux, open a terminal as the root user (or use `sudo / su`)
 - b. On Windows, open an administrator command prompt or PowerShell terminal.
7. Ensure the worker service is not running and configured to start on boot:

- a. On Linux, run

```
systemctl stop deadline-worker  
systemctl enable deadline-worker
```

- b. On Windows, run

```
sc.exe stop DeadlineWorker  
sc.exe config DeadlineWorker start= auto
```

8. Delete the worker state.

- a. On Linux, run

```
rm -rf /var/lib/deadline/*
```

- b. On Windows, run

```
del /Q /S %PROGRAMDATA%\Amazon\Deadline\Cache\*
```

9. Delete the log files.

- a. On Linux, run

```
rm -rf /var/log/amazon/deadline/*
```

- b. On Windows, run

```
del /Q /S %PROGRAMDATA%\Amazon\Deadline\Logs\*
```

10. On Windows, it is recommended to run the Amazon EC2Launch Settings application found in the Start menu to complete the final host preparation and shutdown of the instance.

Note

You **MUST** choose **Shutdown without Sysprep** and never choose Shutdown with Sysprep. Shutting down with Sysprep will cause all local users to become unusable. For more information, see [Before you Begin section of the Create a custom AMI topic of the User Guide for Windows Instances](#).

Build the AMI

To build the AMI

1. Open the Amazon EC2 console.
2. Select **Instances** in the navigation pane, then select your instance.
3. Choose **Instance state**, then **Stop instance**.
4. After the instance is **Stopped**, choose **Actions**.
5. Choose **Image and templates**, then **Create image**.
6. Enter an **Image name**.
7. (Optional) Enter a description for your image.
8. Choose **Create image**.

Create fleet infrastructure with an Amazon EC2 Auto Scaling group

This section explains how to create an Amazon EC2 Auto Scaling fleet.

Use the CloudFormation YAML template below to create an Amazon EC2 Auto Scaling (Auto Scaling) group, an Amazon Virtual Private Cloud (Amazon VPC) with two subnets, an instance profile, and an instance access role. These resources are required to launch an instance using Auto Scaling in the subnets.

We recommend that you review and update the list of instance types to fit your rendering needs.

For a complete explanation of the resources and parameters used in the CloudFormation YAML template, see the [Deadline Cloud resource type reference](#) in the *AWS CloudFormation User Guide*.

To create an Amazon EC2 Auto Scaling fleet

1. Use the following example to create a CloudFormation template that defines the FarmID, FleetID, and AMIID parameters. Save the template to a .YAML file on your local computer.

```
AWSTemplateFormatVersion: 2010-09-09
Description: Amazon Deadline Cloud customer-managed fleet
Parameters:
  FarmId:
    Type: String
    Description: Farm ID
  FleetId:
    Type: String
    Description: Fleet ID
  AMIID:
    Type: String
    Description: AMI ID for launching workers
Resources:
  deadlineVPC:
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: 100.100.0.0/16
  deadlineWorkerSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: !Join
        - ' '
        - - Security group created for Deadline Cloud workers in the fleet
          - !Ref FleetId
      GroupName: !Join
        - ' '
        - - deadlineWorkerSecurityGroup-
          - !Ref FleetId
      SecurityGroupEgress:
        - CidrIp: 0.0.0.0/0
          IpProtocol: '-1'
      SecurityGroupIngress: []
      VpcId: !Ref deadlineVPC
  deadlineIGW:
    Type: 'AWS::EC2::InternetGateway'
    Properties: {}
  deadlineVPCGatewayAttachment:
    Type: 'AWS::EC2::VPCGatewayAttachment'
```

```

Properties:
  VpcId: !Ref deadlineVPC
  InternetGatewayId: !Ref deadlineIGW
deadlinePublicRouteTable:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref deadlineVPC
deadlinePublicRoute:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref deadlinePublicRouteTable
    DestinationCidrBlock: 0.0.0.0/0
    GatewayId: !Ref deadlineIGW
  DependsOn:
    - deadlineIGW
    - deadlineVPCGatewayAttachment
deadlinePublicSubnet0:
  Type: 'AWS::EC2::Subnet'
  Properties:
    VpcId: !Ref deadlineVPC
    CidrBlock: 100.100.16.0/22
    AvailabilityZone: !Join
      - ''
      - - !Ref 'AWS::Region'
        - a
deadlineSubnetRouteTableAssociation0:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    RouteTableId: !Ref deadlinePublicRouteTable
    SubnetId: !Ref deadlinePublicSubnet0
deadlinePublicSubnet1:
  Type: 'AWS::EC2::Subnet'
  Properties:
    VpcId: !Ref deadlineVPC
    CidrBlock: 100.100.20.0/22
    AvailabilityZone: !Join
      - ''
      - - !Ref 'AWS::Region'
        - c
deadlineSubnetRouteTableAssociation1:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    RouteTableId: !Ref deadlinePublicRouteTable
    SubnetId: !Ref deadlinePublicSubnet1

```

```

deadlineInstanceAccessAccessRole:
  Type: 'AWS::IAM::Role'
  Properties:
    RoleName: !Join
      - '-'
      - - deadline
        - InstanceAccess
        - !Ref FleetId
    AssumeRolePolicyDocument:
      Statement:
        - Effect: Allow
          Principal:
            Service: ec2.amazonaws.com
          Action:
            - 'sts:AssumeRole'
    Path: /
    ManagedPolicyArns:
      - 'arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy'
      - 'arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore'
      - 'arn:aws:iam::aws:policy/AWSDeadlineCloud-WorkerHost'
deadlineInstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'
  Properties:
    Path: /
    Roles:
      - !Ref deadlineInstanceAccessAccessRole
deadlineLaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateName: !Join
      - ''
      - - deadline-LT-
        - !Ref FleetId
    LaunchTemplateData:
      NetworkInterfaces:
        - DeviceIndex: 0
          AssociatePublicIpAddress: true
          Groups:
            - !Ref deadlineWorkerSecurityGroup
          DeleteOnTermination: true
      ImageId: !Ref AMIID
      InstanceInitiatedShutdownBehavior: terminate
      IamInstanceProfile:
        Arn: !GetAtt

```

```

        - deadlineInstanceProfile
        - Arn
    MetadataOptions:
        HttpTokens: required
        HttpEndpoint: enabled

deadlineAutoScalingGroup:
    Type: 'AWS::AutoScaling::AutoScalingGroup'
    Properties:
        AutoScalingGroupName: !Join
            - ''
            - - deadline-ASG-autoscalable-
              - !Ref FleetId
        MinSize: 0
        MaxSize: 10
        VPCZoneIdentifier:
            - !Ref deadlinePublicSubnet0
            - !Ref deadlinePublicSubnet1
        NewInstancesProtectedFromScaleIn: true
        MixedInstancesPolicy:
            InstancesDistribution:
                OnDemandBaseCapacity: 0
                OnDemandPercentageAboveBaseCapacity: 0
                SpotAllocationStrategy: capacity-optimized
                OnDemandAllocationStrategy: lowest-price
            LaunchTemplate:
                LaunchTemplateSpecification:
                    LaunchTemplateId: !Ref deadlineLaunchTemplate
                    Version: !GetAtt
                        - deadlineLaunchTemplate
                        - LatestVersionNumber
        Overrides:
            - InstanceType: m5.large
            - InstanceType: m5d.large
            - InstanceType: m5a.large
            - InstanceType: m5ad.large
            - InstanceType: m5n.large
            - InstanceType: m5dn.large
            - InstanceType: m4.large
            - InstanceType: m3.large
            - InstanceType: r5.large
            - InstanceType: r5d.large
            - InstanceType: r5a.large
            - InstanceType: r5ad.large

```

```
- InstanceType: r5n.large
- InstanceType: r5dn.large
- InstanceType: r4.large
MetricsCollection:
  - Granularity: 1Minute
  Metrics:
    - GroupMinSize
    - GroupMaxSize
    - GroupDesiredCapacity
    - GroupInServiceInstances
    - GroupTotalInstances
    - GroupInServiceCapacity
    - GroupTotalCapacity
```

2. Open the CloudFormation console at <https://console.aws.amazon.com/cloudformation>.

Use the CloudFormation console to create a stack using the instructions for uploading the template file that you created. For more information, see [Creating a stack on the CloudFormation console](#) in the *AWS CloudFormation User Guide*.

Note

- Credentials from the IAM role that are attached to your worker's Amazon EC2 instance are available to *all* processes running on that worker, which includes jobs. The worker should have the least privileges to operate: `deadline:CreateWorker` and `deadline:AssumeFleetRoleForWorker`.
- The worker agent obtains credentials for the queue role and configures them for use by running jobs. The Amazon EC2 instance profile role shouldn't include permissions that are needed by your jobs.

Auto scale your Amazon EC2 fleet with Deadline Cloud scale recommendation feature

Deadline Cloud uses an Amazon EC2 Auto Scaling (Auto Scaling) group to scale the Amazon EC2 customer-managed fleet (CMF) automatically. You need to configure the fleet mode as well as deploy the required infrastructure in your account to make your fleet auto scale. The infrastructure you deployed will work for all fleets, so you only need to set it up once.

Customer-managed fleets support two scaling modes:

- **NO_SCALING** – The fleet is treated as a static worker pool. Deadline Cloud does not emit scaling recommendations or fleet size events for the fleet. Use this mode for on-premises fleets or fleets that you scale manually.
- **EVENT_BASED_AUTO_SCALING** – Deadline Cloud emits Amazon EventBridge (EventBridge) events with fleet size recommendations whenever the recommended fleet size changes. The service does not directly start or stop workers. You must deploy infrastructure (such as an AWS Lambda (Lambda) function) to consume these events and adjust your Auto Scaling group capacity accordingly.

The basic workflow is: you configure your fleet mode to auto scale, and then Deadline Cloud will send out an EventBridge event for that fleet whenever recommended fleet size changes (one event contains fleet id, recommended fleet size, and other metadata). You will have an EventBridge rule to filter the relevant events and have a Lambda to consume them. The Lambda will integrate with Amazon EC2 Auto Scaling `AutoScalingGroup` to scale the Amazon EC2 fleet automatically.

Set fleet mode to **EVENT_BASED_AUTO_SCALING**

Configure your fleet mode to `EVENT_BASED_AUTO_SCALING`. You can use the console to do this, or use the AWS CLI to directly call the `CreateFleet` or `UpdateFleet` API. After the mode is configured, Deadline Cloud starts sending EventBridge events whenever the recommended fleet size changes.

- Example `UpdateFleet` command:

```
aws deadline update-fleet \  
  --farm-id FARM_ID \  
  --fleet-id FLEET_ID \  
  --configuration file://configuration.json
```

- Example `CreateFleet` command:

```
aws deadline create-fleet \  
  --farm-id FARM_ID \  
  --display-name "Fleet name" \  
  --max-worker-count 10 \  
  --configuration file://configuration.json
```

The following is an example of `configuration.json` used in the CLI commands above (-- configuration file://configuration.json).

- To enable Auto Scaling on your fleet, you should set the mode to `EVENT_BASED_AUTO_SCALING`.
- The `workerCapabilities` are the default values assigned to the CMF when you created it. You can change these values if you need to increase resources available to your CMF.

After you configure the fleet mode, Deadline Cloud starts emitting fleet size recommendation events for that fleet.

```
{
  "customerManaged": {
    "mode": "EVENT_BASED_AUTO_SCALING",
    "workerCapabilities": {
      "vCpuCount": {
        "min": 1,
        "max": 4
      },
      "memoryMiB": {
        "min": 1024,
        "max": 4096
      },
      "osFamily": "linux",
      "cpuArchitectureType": "x86_64"
    }
  }
}
```

Deploy Auto Scaling stack using the CloudFormation template

You can set up an EventBridge rule to filter events, a Lambda to consume the events and control Auto Scaling, and an SQS queue to store unprocessed events. Use the following CloudFormation template to deploy everything in a stack. After you deploy the resources successfully, you can submit a job and the fleet will automatically scale up.

```
Resources:
  AutoScalingLambda:
    Type: 'AWS::Lambda::Function'
    Properties:
      Code:
```

```

ZipFile: |-
"""

This lambda is configured to handle "Fleet Size Recommendation Change"
messages. It will handle all such events, and requires
that the ASG is named based on the fleet id. It will scale up/down the fleet
based on the recommended fleet size in the message.

Example EventBridge message:
{
    "version": "0",
    "id": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",
    "detail-type": "Fleet Size Recommendation Change",
    "source": "aws.deadline",
    "account": "111122223333",
    "time": "2017-12-22T18:43:48Z",
    "region": "us-west-1",
    "resources": [],
    "detail": {
        "farmId": "farm-1234567890000000000000000000000000",
        "fleetId": "fleet-1234567890000000000000000000000000",
        "oldFleetSize": 1,
        "newFleetSize": 5
    }
}
"""

import json
import boto3
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

auto_scaling_client = boto3.client("autoscaling")

def lambda_handler(event, context):
    logger.info(event)
    event_detail = event["detail"]
    fleet_id = event_detail["fleetId"]
    desired_capacity = event_detail["newFleetSize"]

    asg_name = f"deadline-ASG-autoscalable-{fleet_id}"
    auto_scaling_client.set_desired_capacity(
        AutoScalingGroupName=asg_name,

```

```

        DesiredCapacity=desired_capacity,
        HonorCooldown=False,
    )

    return {
        'statusCode': 200,
        'body': json.dumps(f'Successfully set desired_capacity for {asg_name}'
                           to {desired_capacity}')
    }
Handler: index.lambda_handler
Role: !GetAtt
    - AutoScalingLambdaServiceRole
    - Arn
Runtime: python3.11
DependsOn:
    - AutoScalingLambdaServiceRoleDefaultPolicy
    - AutoScalingLambdaServiceRole
AutoScalingEventRule:
    Type: 'AWS::Events::Rule'
    Properties:
        EventPattern:
            source:
                - aws.deadline
            detail-type:
                - Fleet Size Recommendation Change
        State: ENABLED
    Targets:
        - Arn: !GetAtt
            - AutoScalingLambda
            - Arn
        DeadLetterConfig:
            Arn: !GetAtt
                - UnprocessedAutoScalingEventQueue
                - Arn
        Id: Target0
        RetryPolicy:
            MaximumRetryAttempts: 15
AutoScalingEventRuleTargetPermission:
    Type: 'AWS::Lambda::Permission'
    Properties:
        Action: 'lambda:InvokeFunction'
        FunctionName: !GetAtt
            - AutoScalingLambda
            - Arn

```

```

Principal: events.amazonaws.com
SourceArn: !GetAtt
  - AutoScalingEventRule
  - Arn
AutoScalingLambdaServiceRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Statement:
        - Action: 'sts:AssumeRole'
          Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
      Version: 2012-10-17
    ManagedPolicyArns:
      - !Join
        - ''
        - - 'arn:'
          - !Ref 'AWS::Partition'
          - ':iam::aws:policy/service-role/AWSLambdaBasicExecutionRole'
AutoScalingLambdaServiceRoleDefaultPolicy:
  Type: 'AWS::IAM::Policy'
  Properties:
    PolicyDocument:
      Statement:
        - Action: 'autoscaling:SetDesiredCapacity'
          Effect: Allow
          Resource: '*'
      Version: 2012-10-17
    PolicyName: AutoScalingLambdaServiceRoleDefaultPolicy
    Roles:
      - !Ref AutoScalingLambdaServiceRole
UnprocessedAutoScalingEventQueue:
  Type: 'AWS::SQS::Queue'
  Properties:
    QueueName: deadline-unprocessed-autoscaling-events
    UpdateReplacePolicy: Delete
    DeletionPolicy: Delete
UnprocessedAutoScalingEventQueuePolicy:
  Type: 'AWS::SQS::QueuePolicy'
  Properties:
    PolicyDocument:
      Statement:
        - Action: 'sqs:SendMessage'

```

```
Condition:
  ArnEquals:
    'aws:SourceArn': !GetAtt
      - AutoScalingEventRule
      - Arn
  Effect: Allow
  Principal:
    Service: events.amazonaws.com
  Resource: !GetAtt
    - UnprocessedAutoScalingEventQueue
    - Arn
Version: 2012-10-17
Queues:
  - !Ref UnprocessedAutoScalingEventQueue
```

Perform a fleet health check

After creating your fleet, you should build a custom health check to ensure your fleet remains healthy and free of stalled instances to help prevent unnecessary costs. See [Deploying a Deadline Cloud fleet health check](#) on the GitHub website. A health check can lower the risk of an accidental change in your Amazon Machine Image, launch template, or network configuration running undetected.

Configure and use Deadline Cloud service-managed fleets

A service-managed fleet (SMF) is a collection of workers managed by Deadline Cloud. An SMF eliminates the need to manage fleet scaling for processing demands or reduce fleet size after task completion. For more information about choosing a fleet type, see [Choose between service-managed and customer-managed fleets](#) in the *Deadline Cloud User Guide*.

When an SMF is associated with a queue using the default conda queue environment, Deadline Cloud configures the workers in the fleet with the appropriate software package. For supported partner applications, see [Default conda queue environment](#) in the *AWS Deadline Cloud User Guide*.

In most cases, you don't need to change an SMF to process your workloads. However, some situations may require you make changes to your fleets.

Note

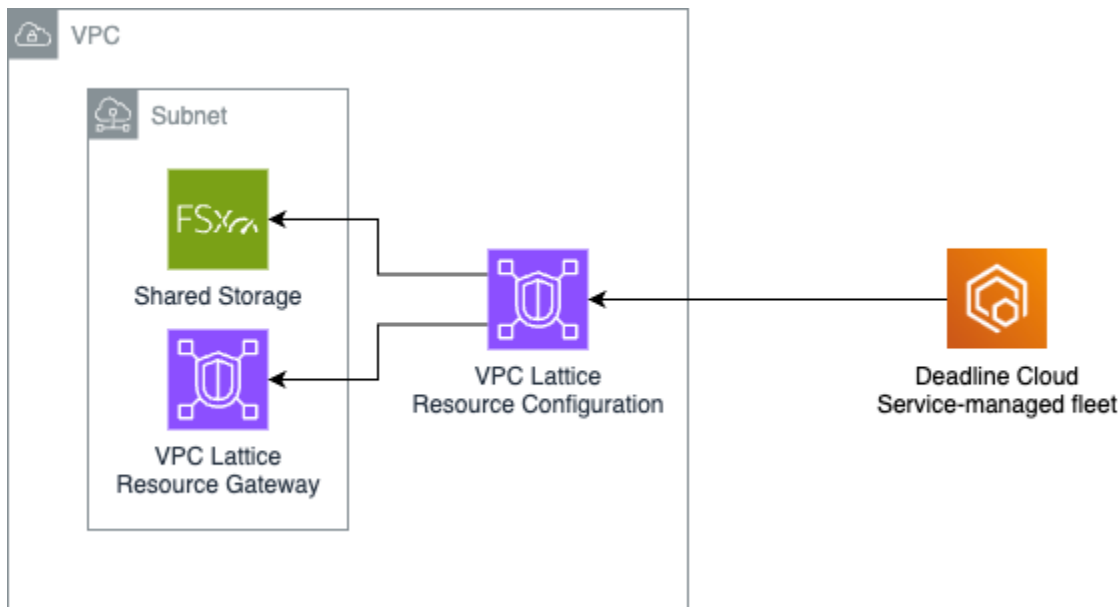
To install custom software on workers using host configuration scripts, see [Run host configuration scripts with administrator privileges](#).

Topics

- [Connect VPC resources to your SMF with VPC resource endpoints](#)
- [Use job attachments with service-managed fleets](#)
- [Persistent storage for service-managed fleets](#)

Connect VPC resources to your SMF with VPC resource endpoints

With Amazon VPC resource endpoints for Deadline Cloud service-managed fleets (SMF), you can connect your VPC resources such as network file systems (NFS), license servers, and databases with your Deadline Cloud workers. This feature lets you use the fully managed Deadline Cloud platform while integrating with your existing infrastructure within a VPC.

**Tip**

For a reference CloudFormation template that sets up an Amazon FSx cluster and connects it to a service-managed fleet, see [smf_vpc_fsx](#) in the Deadline Cloud samples repository on the GitHub website.

How VPC resource endpoints work

VPC resource endpoints use VPC Lattice to create a secure connection between your Deadline Cloud SMF workers and resources in your VPC. The connection is unidirectional, meaning workers can establish a connection to resources in your VPC and transfer data back and forth, but resources in your VPC cannot establish a connection to a worker.

When you connect a VPC resource to a Deadline Cloud service-managed fleet, your workers can access resources in your VPC using a private domain name. Additionally, traffic flows from workers to your VPC resources through VPC Lattice, and resources in your VPC see traffic coming from the VPC Lattice resource gateway.

To learn more, see the [VPC Lattice user guide](#).

Prerequisites

Before connecting VPC resources to your Deadline Cloud service-managed fleet, ensure you have the following:

- An AWS account with a VPC containing resources you want to connect.
- IAM permissions to create and manage VPC Lattice resources.
- A Deadline Cloud farm with at least one service-managed fleet.
- VPC resources you want to make accessible (FSx, NFS, license servers, etc.).

Set up a VPC resource endpoint

To set up a VPC resource endpoint, you need to create resources in [VPC Lattice](#) and [AWS RAM](#), and then connect those resources to your fleet in Deadline Cloud. To set up a VPC resource endpoint for your SMF, complete the following steps.

1. To create a resource gateway in VPC Lattice, see [Create a resource gateway](#) in the VPC Lattice user guide.
2. To create a resource configuration in VPC Lattice, see [Create a resource configuration](#) in the VPC Lattice user guide.
3. To share the resource with your Deadline Cloud fleet, create a resource share in AWS RAM. See [Creating a resource share](#) for instructions.

While creating a resource share, for **Principals**, select **Service principal** from the dropdown, and then enter **fleets.deadline.amazonaws.com**.

4. To connect the resource configuration with your Deadline Cloud fleet, complete the following steps.
 - a. If you haven't already, open the [Deadline Cloud console](#).
 - b. In the navigation pane, choose **Farms**, and then select your farm.
 - c. Choose the **Fleets** tab, and then select your fleet.
 - d. Choose the **Configurations** tab.
 - e. Under **VPC resource endpoints**, choose **Edit**.
 - f. Select the resource configuration you created, and then choose **Save changes**.

Accessing your VPC resources

After connecting your VPC resource to your fleet, workers can access it using a private domain name in the following format: `<resource_config_id>.resource-endpoints.deadline.<region>.amazonaws.com`

This domain is private and only accessible by workers (not from the internet or your workstation).

To mount or configure access to the VPC resource on your workers, use a [host configuration script](#). Host configuration scripts run with administrator privileges when workers start, allowing you to mount file systems, configure network settings, or perform other setup tasks.

Authentication and security

For resources requiring authentication, store credentials securely in AWS Secrets Manager, access secrets from your [host configuration scripts](#) or job scripts, and implement appropriate file system permissions to control access. Consider security implications when sharing resources across multiple fleets. For example, if two fleets are connected to the same shared storage, jobs that run on one fleet might be able to access assets created from the other fleet.

Technical considerations

When using VPC resource endpoints, consider the following:

- Connections can only be initiated from workers to VPC resources, not from VPC resources to workers.
- Once established, a connection persists until it is reset, even if the resource configuration is disconnected.
- The VPC Lattice connection handles connections between Availability Zones automatically with no additional charges. Your resource gateway must share an Availability Zone with your VPC resource, so we recommend configuring the resource gateway to span all Availability Zones.
- Traffic going through the VPC resource endpoint uses Network Address Translation (NAT), which is not compatible with all use cases. For example, Microsoft Active Directory (AD) cannot connect over NAT.

For more information about VPC Lattice quotas, see [Quotas for VPC Lattice](#).

Troubleshooting

If you encounter issues with VPC resource endpoints, check the following.

- If you receive an error message such as "mount.nfs: access denied by server while mounting," you might need to update the client configuration of your NFS volume.

- Verify your resource configuration setup by testing from an Amazon EC2 instance or AWS CloudShell in your VPC.
- Test your Deadline Cloud connection with simple CLI jobs. For more information, see [Deadline Cloud samples repository](#) on the GitHub website.
- Check the settings on the resource gateway's security group if you experience connection failures.
- Enable VPC access logs to monitor connections.

Use job attachments with service-managed fleets

Job attachments transfer files between your workstation and Deadline Cloud workers using Amazon Simple Storage Service (Amazon S3). You can use job attachments alone or together with shared storage to attach auxiliary data to jobs that isn't shared with other jobs, such as job scripts, configuration files, or project assets stored locally.

For information about how job attachments work, see [Job attachments](#) in the *Deadline Cloud User Guide*. For details about specifying input and output files in job bundles, see [Use job attachments to share files](#).

Choose a filesystem mode

When submitting a job with attachments, you can choose how workers load files from Amazon S3 by setting the `fileSystem` property:

- **COPIED** (default) – Downloads all files to local disk before tasks begin. Best when every task needs most input files.
- **VIRTUAL** – Mounts a virtual filesystem that downloads files on-demand. Best when tasks only need a subset of input files. Available on Linux SMF workers only.

Important

VIRTUAL mode caching can increase memory consumption and is not optimized for all workloads. We recommend that you test your workload before running production jobs.

For detailed information about configuring the filesystem mode, see [Virtual file system](#) in the *Deadline Cloud User Guide*.

Optimize transfer performance

The speed of synchronizing files from Amazon S3 to SMF workers depends on the Amazon Elastic Block Store (Amazon EBS) volume configuration of your fleet. By default, SMF workers use gp3 Amazon EBS volumes with baseline performance settings. For workloads with large input files or many small files, you can improve transfer speeds by increasing the Amazon EBS throughput and IOPS settings. You can update these settings using the AWS Command Line Interface (AWS CLI).

Throughput (MiB/s)

The rate at which data can be read from or written to the volume. Default is 125 MiB/s, maximum is 1,000 MiB/s for gp3 volumes. Increase for large sequential file transfers.

IOPS

Input/output operations per second. Default is 3,000 IOPS, maximum is 16,000 IOPS for gp3 volumes. Increase when transferring many small files.

Note

Increasing Amazon EBS throughput and IOPS increases fleet cost. For pricing information, see [Deadline Cloud pricing](#).

To update Amazon EBS settings on an existing fleet using the AWS CLI

- Run the following command:

```
aws deadline update-fleet \  
  --farm-id farm-0123456789abcdef0 \  
  --fleet-id fleet-0123456789abcdef0 \  
  --configuration '{  
    "serviceManagedEc2": {  
      "instanceCapabilities": {  
        "vCpuCount": {"min": 4},  
        "memoryMiB": {"min": 8192},  
        "osFamily": "linux",  
        "cpuArchitectureType": "x86_64",
```

```
    "rootEbsVolume": {
      "sizeGiB": 250,
      "iops": 6000,
      "throughputMiB": 500
    }
  },
  "instanceMarketOptions": {"type": "spot"}
}
```

Download job outputs

After your job completes, download output files using the Deadline Cloud CLI or AWS Deadline Cloud monitor (Deadline Cloud monitor):

```
deadline job download-output --job-id job-0123456789abcdef0
```

For automatic downloading of outputs as jobs complete, see [Automatic downloads](#) in the *Deadline Cloud User Guide*.

Persistent storage for service-managed fleets

AWS Deadline Cloud (Deadline Cloud) persistent storage attaches dedicated Amazon Elastic Block Store (Amazon EBS) gp3 volumes to service-managed fleet workers. These volumes preserve data such as conda package installations, application caches, and version control workspaces across worker lifecycle events. To enable persistent storage, include the `persistentVolumeConfiguration` object in the `serviceManagedEc2FleetConfiguration` when you call `CreateFleet` or `UpdateFleet`, or configure it in the console when you create or edit a fleet.

For the full documentation – how persistent storage works, when to use it, configuration parameters and examples, runtime integration with the `DEADLINE_PERSISTENT_MOUNT` environment variable, volume states, and considerations – see [Persistent storage for service-managed fleets](#) in the *AWS Deadline Cloud User Guide*.

For the API reference, see [CreateFleet](#) in the *Deadline Cloud API Reference*.

Manage Deadline Cloud infrastructure as code

You can define AWS Deadline Cloud (Deadline Cloud) farms, queues, and fleets as code. Deploy them with AWS CloudFormation (CloudFormation), the AWS Cloud Development Kit (AWS CDK), or Terraform. All three tools build on the same Deadline Cloud resource types, so they create equivalent infrastructure. Choose the tool your team already uses to manage the rest of your AWS infrastructure.

The following table compares the three tools. The sections after the table describe each tool and link to a working starter template in the [deadline-cloud-samples](#) repository on the GitHub website.

Aspect	CloudFormation	AWS CDK	Terraform
Authoring	YAML or JSON template	A programming language such as TypeScript or Python, compiled to a CloudFormation template	HCL configuration
Provider	AWS native	AWS native	HashiCorp AWSCC
Deadline Cloud resource names	<code>AWS::Deadline::*</code>	<code>CfnFarm</code> , <code>CfnQueue</code> , <code>CfnFleet</code> , and so on	<code>awscs_deadline_*</code>
State	Managed by AWS	Managed by AWS	Local or remote backend

CloudFormation

CloudFormation provides native resource types for Deadline Cloud in the `AWS::Deadline` namespace. For the properties and return values of each resource type, see [AWS Deadline Cloud](#) in the *CloudFormation Template Reference*.

For working templates that deploy a complete farm, connect fleets to private VPC resources, and manage capacity, see [the section called “CloudFormation templates”](#).

AWS CDK

With the AWS CDK, you define Deadline Cloud resources using L1 constructs, such as `CfnFarm`, `CfnQueue`, and `CfnFleet`, from the [aws-cdk-lib/aws-deadline](#) module. The constructs are generated from the CloudFormation resource types, and your AWS CDK app deploys as a CloudFormation stack.

For a working TypeScript app with four example farm stacks built from reusable constructs, see [the section called “AWS CDK app”](#).

Terraform

Terraform supports Deadline Cloud through the [AWS Cloud Control \(AWSCC\) provider](#) on the Terraform Registry website, which exposes resources such as `awsc_deadline_farm`, `awsc_deadline_queue`, and `awsc_deadline_fleet`. The AWSCC provider is generated from the CloudFormation resource types through AWS Cloud Control API, so it covers the same resources as CloudFormation. The `hashicorp/aws` provider doesn't include Deadline Cloud resources.

For working configurations, including a starter farm equivalent to the CloudFormation and AWS CDK versions, see [the section called “Terraform configurations”](#).

Resources outside your templates

The Deadline Cloud resource types cover farms, queues, fleets, queue environments, storage profiles, limits, license endpoints, metered products, monitors, and the associations between queues and fleets or limits. For the full list, see [AWS Deadline Cloud](#) in the *CloudFormation Template Reference*.

A few resources don't have resource types, so you manage them with the console, the AWS CLI, or the API after your deployment completes:

- **Budgets** – Create budgets for your farms and queues with budget manager. For more information, see [Managing budgets and usage](#) in the *Deadline Cloud User Guide*.
- **Memberships** – Grant users and groups access to farms, fleets, queues, and jobs. For more information, see [Managing users](#) in the *Deadline Cloud User Guide*.

Extend your on-premises render farm to the cloud

You already run a render farm on-premises, and you want extra capacity during peak demand without buying and maintaining more hardware. With AWS Deadline Cloud (Deadline Cloud), you can add cloud workers to the same farm as your on-premises workers and send jobs to either location. Sending overflow work to the cloud in periods of high demand is often called *cloud bursting*.

A hybrid farm combines two kinds of fleets in one farm:

- Your existing on-premises workers run as a *customer-managed fleet* (CMF). You control the hosts, operating system, and software. A worker can be an on-premises server, a server in a co-location facility, or an Amazon Elastic Compute Cloud (Amazon EC2) instance. For more information, see [Create and use Deadline Cloud customer-managed fleets](#).
- Your cloud capacity runs as a *service-managed fleet* (SMF), where Deadline Cloud manages the worker hosts, scaling, and patching for you. You can also run cloud capacity as a customer-managed fleet of Amazon EC2 instances when you need more control. For more information, see [Configure and use Deadline Cloud service-managed fleets](#).

Deadline Cloud sends work to your fleets through queues, so decide how to organize your queues before you extend your farm. Because service-managed fleets and customer-managed fleets install applications differently, you can give each fleet type its own queue or share one queue across both fleet types. For a comparison of the two approaches, see [Choose a queue strategy](#).

Whichever queue strategy you choose, plan for each of the following layers so that your jobs run correctly on every fleet. The rest of this topic describes each layer and links to detailed instructions.

The following table summarizes how each layer differs across the three kinds of workers you can combine in a hybrid farm. Your on-premises workers and your cloud Amazon EC2 instances are both customer-managed fleets, but they differ in how they reach your data and network, so the table lists them separately.

Layer	On-premises (customer-managed fleet)	Cloud customer-managed fleet (Amazon EC2)	Service-managed fleet
Worker hosts	You provision and manage your own hardware, operating system, and patching.	You provision and manage Amazon EC2 instances.	Deadline Cloud manages the hosts, operating system, and patching.
Applications	Preinstall applications on your workers, or use a conda channel in Amazon S3.	Preinstall applications, bake them into an Amazon Machine Image (AMI), or use a conda channel in Amazon S3.	Automatically installs popular DCCs from the Deadline Cloud-managed conda channel. To provide custom applications, use host configuration scripts or your own conda channel in Amazon S3. For the list of supported applications, see Supported software .
Licensing	Connect a license endpoint for UBL, or connect to your own license server for BYOL.	Connect a license endpoint for UBL, or connect to your own license server for BYOL.	UBL is configured automatically. For BYOL, forward the license ports to your server with SSM.
Asset access	Use job attachments, or shared storage that your workers already mount.	Use job attachments, or shared storage in your VPC.	Use job attachments, or connect to shared storage in your VPC with VPC resource endpoints.
Scaling and cost	Fixed capacity that you own.	Use an Amazon EC2 Auto Scaling group to grow and shrink capacity with demand.	Scales automatically between the minimum and maximum worker counts that you set.

Distribute jobs across your fleets

When you associate a queue with more than one fleet, Deadline Cloud distributes jobs across the fleets that are compatible with each job. The scheduler compares the host requirements of each step with the [worker capabilities](#) that each fleet declares, and it runs the step on a fleet that meets those requirements. For more information, see [Schedule jobs in Deadline Cloud](#).

To automatically balance capacity between fleets, you can use the capacity manager sample. It uses a Lambda function and EventBridge Scheduler to adjust fleet worker counts while holding a target total capacity. You can adapt the same pattern to shift capacity toward your cloud fleet during peak demand. For more information, see [Manage hybrid Wait and Save plus Spot fleet capacity with CloudFormation](#).

Note

Deadline Cloud does not provide a built-in setting to prefer one fleet over another, such as filling your on-premises fleet before your cloud fleet. To steer specific work to a specific location, define distinct worker capabilities on each fleet, and then set matching host requirements on the steps that must run in that location. To approximate fleet prioritization automatically, use the capacity manager sample described in the preceding paragraph.

Choose a queue strategy

Service-managed fleets and customer-managed fleets install and manage applications differently, so decide how to organize your queues before you associate fleets with them. On a service-managed fleet, the default conda queue environment automatically installs the conda packages that jobs request. Customer-managed fleets don't have the conda commands installed by default, so the same queue environment causes errors on your customer-managed workers.

You can resolve the difference in one of the following two ways:

- **Use a separate queue for each fleet type (recommended)** – Associate your service-managed fleet with a queue that uses the default conda queue environment, and associate your customer-managed fleet with a separate queue that doesn't use it. Each fleet type gets the application setup it expects, and you avoid conda errors. You submit each job to the queue for the location where you want the job to run.

- **Share one queue across both fleet types** – Associate both fleets with the same queue so that Deadline Cloud distributes each job to whichever location has available capacity, without changing where you submit the job. A shared queue takes more setup because you can't use the default conda queue environment. Instead, you provide applications in a way that works on both fleet types. Choose a shared queue when you want peak workloads to overflow to the cloud automatically.

To run applications on a shared queue, create your own conda channel in Amazon Simple Storage Service (Amazon S3) that both fleet types use, or install applications directly on your customer-managed workers and adjust the queue environment so that the applications appear on the job user's path. For more information, see [Create a conda channel using S3](#) and [Configure jobs using queue environments](#).

License applications on both fleets

Deadline Cloud provides usage-based licensing (UBL), and you can also bring your own license (BYOL). Both options work on cloud and on-premises workers. For an overview, see [Using software licenses with Deadline Cloud](#).

- UBL is configured automatically on service-managed fleets. To use UBL on a customer-managed fleet, including on-premises workers, connect a license endpoint. For more information, see [Connect customer-managed fleets to a license endpoint](#).
- To use your existing license server with a service-managed fleet, forward the license ports from the workers to your license server or proxy with Amazon EC2 Systems Manager (SSM). For more information, see [Connect service-managed fleets to a custom license server](#).
- To use your existing licenses first and fall back to UBL when they are exhausted, combine BYOL and UBL. For more information, see [Combining BYOL and UBL](#).

Give both fleets access to your assets

Both fleets need access to the data that your jobs read and write. Deadline Cloud supports two approaches, and you can use them together:

- **Job attachments** – Deadline Cloud uploads the files that a job needs to Amazon S3 and downloads the output afterward. Job attachments require no shared storage setup and work

with fleets of any size. If your data already resides on AWS, job attachments add another copy of your data, along with the associated transfer time and storage cost.

- **Shared storage** – Your workers mount a shared file system that holds your assets. To simplify how your on-premises and cloud workers manage assets, you can connect your service-managed fleet workers to shared storage in your virtual private cloud (VPC) with VPC resource endpoints. Both fleet types then read and write the same files. For more information, see [Connect VPC resources to your SMF with VPC resource endpoints](#).

Because your on-premises workstations and cloud workers can use different operating systems and file system layouts, use storage profiles to model each configuration. Storage profiles let Deadline Cloud map paths so that a worker can find the data no matter where a job is submitted. For more information, see [Storage profiles and path mapping](#).

Connect cloud workers to your network

Cloud workers often need to reach resources that run in your VPC or on-premises network, such as shared file systems, license servers, and databases. With VPC resource endpoints for service-managed fleets, your workers can connect to these resources over a private connection that uses VPC Lattice. For more information, see [Connect VPC resources to your SMF with VPC resource endpoints](#).

To mount a file system or configure other network access on a service-managed fleet worker, use a host configuration script. Host configuration scripts run with administrator privileges when a worker starts. For more information, see [Run host configuration scripts with administrator privileges](#).

Note

Traffic through a VPC resource endpoint uses Network Address Translation (NAT), which is not compatible with all resources. For example, Microsoft Active Directory cannot connect over NAT.

Match applications to each fleet type

Confirm that each digital content creation (DCC) application your jobs use can run on both fleet types. On service-managed fleets, the Deadline Cloud-managed conda channel provides popular

DCCs automatically. For applications that the channel doesn't provide, or for customer-managed workers that can't use the channel, package the applications yourself as described in [Choose a queue strategy](#). For the list of supported applications, see [Supported software](#) in the *AWS Deadline Cloud User Guide*.

Control cloud costs

To pay for cloud capacity only during peak demand, scale your cloud fleet down when it is idle. Service-managed fleets scale automatically between the minimum and maximum worker counts that you set. For a customer-managed fleet of Amazon EC2 instances, you can configure auto scaling so that the fleet grows and shrinks with demand. For more information, see [Create fleet infrastructure with an Amazon EC2 Auto Scaling group](#).

Related resources

For guidance on designing an end-to-end Deadline Cloud workflow across job submission, application management, licensing, asset access, and worker infrastructure, see [Deadline Cloud Architecture Guidance](#).

For runnable examples, including a reference template that connects a shared file system to a service-managed fleet, see the [deadline-cloud-samples](#) repository on the GitHub website.

Deploy and configure custom software on workers

AWS Deadline Cloud provides multiple methods to deploy and configure custom software, plugins, and tools on your workers. The method you choose depends on your requirements, such as whether you need administrator privileges, how often the software changes, and whether the software should be available to all jobs or only specific jobs.

Choose a deployment method

Use the following table to choose the right deployment method for your use case.

Criteria	Plugin sync	Queue environment	Host configuration	Custom conda package
Fleet support	Service-managed fleets; customer-managed fleets with a supporting conda package	Service-managed and customer-managed fleets	Service-managed fleet: host configuration script; customer-managed fleet: manual worker configuration	Service-managed and customer-managed fleets with a conda queue environment
Administrator privileges required	No	No	Yes	No
When it runs	Conda activation at session start	Session start	Service-managed fleet: worker startup; customer-managed fleet: before the worker joins the fleet	Session start

Criteria	Plugin sync	Queue environment	Host configuration	Custom conda package
Scope	Per queue, application, and version	Per queue or job	All workers in fleet	Per queue or job
Can be controlled by job submission	Application and version selection	Yes	No	Yes
Setup complexity	Low	Low	Medium	High
Best for	Simple file-based plugins for supported applications	Simple plugins, scripts, and environment variables	System drivers, containers, and storage mounts	Applications or plugins with dependencies

Quick decision guide:

- *Simple application plugin that plugin sync supports?* Start with [Use custom plugins with Deadline Cloud](#).
- *Need administrator or root privileges?* For a service-managed fleet, use a [host configuration script](#). For a customer-managed fleet, [configure the worker hosts manually](#).
- *Simple plugin or script without administrator privileges?* Use a [queue environment](#).
- *Application or plugin with dependencies and version constraints?* Create a [custom conda package](#).

Use custom plugins with Deadline Cloud

Choose among plugin sync, conda packages, queue environments, and worker host configuration based on the plugin's dependencies, required access, and fleet type. The following table compares these delivery methods.

Plugin requirement	Delivery method	Fleet support	When to choose it
Simple file-based plugins	Deliver custom plugins with plugin sync (simplest option)	Service-managed fleets; customer-managed fleets that use a supporting application conda package	Choose this method when the plugin can run after its files are copied to the application's plugin search path.
Dependencies, installation steps, or per-job version selection	Package a plugin	Service-managed and customer-managed fleets with a conda queue environment	Choose this method when conda must install dependencies, register the plugin, or resolve compatible versions.
Setup commands that do not require administrator access	Configure jobs using queue environments	Service-managed and customer-managed fleets	Choose this method for scripts or environment variables that run at the start of each session.
Drivers, system libraries, or administrator-level installation on a service-managed fleet	Run host configuration scripts with administrator privileges	Service-managed fleets	Choose this method to install system software when a worker starts.
Drivers, system libraries, or administrator-level installation on a customer-managed fleet	Install and configure software required for jobs	Customer-managed fleets	Install the software in the worker image or on each worker host before the worker joins the fleet.

Plugin delivery does not provide a plugin license. If the plugin requires a license, configure the worker network and license client so that jobs can reach your license server. For more information, see [Using software licenses with Deadline Cloud](#).

Deliver custom plugins with plugin sync

Plugin sync delivers custom plugins and add-ons without requiring you to build a conda package. Upload the plugin files to a prescribed prefix in the queue's job attachments bucket. When a job uses a supported application package, the package copies the files to the worker and configures the application to load them.

Use plugin sync when the plugin can run after its files are copied to an application search path. If the plugin requires dependency resolution, installation commands, or a system change, return to [Use custom plugins with Deadline Cloud](#) to choose another delivery method.

How plugin sync works

The supported digital content creation (DCC) conda packages use activation and deactivation scripts for plugin sync. The following events occur for each worker session:

1. The queue environment creates a conda environment containing the DCC and adaptor packages selected by the job.
2. When conda activates the environment, the DCC package's plugin sync script reads the job attachments bucket and root prefix from the session environment.
3. The script downloads files from the operating system, DCC, and DCC version prefix in Amazon S3. It then sets application-specific environment variables or copies the files to an application plugin directory.
4. When the session ends, the deactivation script removes the downloaded plugin files and configuration. The next session starts from the packaged DCC environment.

The script writes messages that begin with `Plugin Sync:` to the session log. You can use these messages to confirm the source prefix, destination, and files processed.

Prerequisites

Before you upload a plugin, confirm the following requirements:

- The application, version, and worker operating system appear in the [supported applications table](#).

- The plugin files are compatible with the worker operating system, application version, and processor architecture.
- You know the plugin vendor's required directory structure. Plugin sync preserves the directory structure that you upload.
- If the plugin requires a license, the worker can reach the license server.

Upload plugin files to Amazon S3

Upload the plugin files to the job attachments Amazon S3 bucket associated with your queue. Use the following prefix:

```
s3://<job-attachments-bucket>/<root-prefix>/plugins/<os>/<dcc-name>/<dcc-version>/
```

Important

The Amazon S3 prefix is case-sensitive. Use lowercase for the operating system and DCC folder names.

Replace the variables as follows:

<job-attachments-bucket>

The job attachments bucket associated with your queue. The queue details page displays the bucket.

<root-prefix>

The job attachments root prefix for your queue. The queue details page displays the prefix.

<os>

The worker operating system, either `linux` or `windows`.

<dcc-name>

The exact folder name from the [supported applications table](#).

<dcc-version>

The version folder from the supported applications table. For example, Maya 2025.3 uses `2025`, and Blender 5.1 uses `5.1`.

For example, the prefix for a Maya 2025 plugin on Linux is:

```
s3://<job-attachments-bucket>/<root-prefix>/plugins/linux/maya/2025/
```

To upload plugin files using the Amazon S3 console

1. Open the Deadline Cloud console, and then open the queue details page.
2. Choose the job attachments bucket link to open the bucket in the Amazon S3 console.
3. Open the root prefix for the queue.
4. Create the prefix `plugins/<os>/<dcc-name>/<dcc-version>/`. For example, use `plugins/linux/maya/2025/` for Maya 2025.
5. Upload the plugin files and directories to the version prefix. Preserve the directory structure that the plugin vendor requires.

Verify plugin delivery

To verify that plugin sync delivered the files

1. Submit a job that selects the supported DCC package and version.
2. In the AWS Deadline Cloud monitor (Deadline Cloud monitor), open the session log for a task.
3. Search the session log for `Plugin Sync:`. Confirm that the log shows the expected Amazon S3 prefix and destination.
4. Confirm in the task log that the application loaded the plugin. The message depends on the application and plugin.

Applications that support plugin sync

The following table lists applications and versions that contain plugin sync scripts. It also identifies whether a conda recipe sample is available on the GitHub website and whether the `deadline-cloud` channel for service-managed fleets includes the scripts.

DCC application	S3 folder name	Version folders	Worker operating system	Conda recipe sample	Service-managed fleet package
Blender	blender	5.0, 5.1	Linux	Yes	Yes
Autodesk Maya	maya	2024, 2025, 2026, 2027	Linux	Yes	Yes
Foundry Nuke	nuke	15.0, 16.0, 17.0	Linux	Yes	Yes
SideFX Houdini	houdini	19.5, 20.0, 20.5, 21.0, 22.0	Linux	Yes	Yes
Maxon Cinema 4D	cinema4d	Linux: 2024.5.1, 2025.3.1, and 2026.3.4; Windows: 2025.3.3 and 2026.3.4	2024: Linux; 2025 and 2026: Linux and Windows	Yes	Yes
Adobe After Effects	aftereffects	25.6 or 25.6.4; 26.0 or 26.0.0	Windows	Yes	Yes

If your application or version does not appear in the table, see [Package a plugin](#) or [Configure jobs using queue environments](#), or use a customer-managed fleet. The [conda recipe samples](#) on the GitHub website include plugin sync scripts that you can use with a self-hosted channel.

Configure jobs using queue environments

AWS Deadline Cloud uses *queue environments* to configure the software on your workers. An environment enables you to perform time-consuming tasks, such as set up and tear-down, once for all the tasks in a session. It defines the actions to run on a worker when starting or stopping a session. You can configure an environment for a queue, jobs that run in the queue, and the individual steps for a job.

You define environments as queue environments or job environments. Create queue environments with the Deadline Cloud console or with the [deadline:CreateQueueEnvironment](#) operation and define job environments in the job templates of the jobs you submit. They follow the Open Job Description (OpenJD) specification for environments. For details, see [<Environment>](#) in the OpenJD specification on the GitHub website.

In addition to a name and description, each environment contains two fields that define the environment on the host. They are:

- **script** – The action taken when this environment is run on a worker.
- **variables** – A set of environment variable name/value pairs that are set when entering the environment.

You must set at least one of **script** or **variables**.

You can define more than one environment in your job template. Each environment is applied in the order that they are listed in the template. You can use this to help manage the complexity of your environments.

The default queue environment for Deadline Cloud uses the conda package manager to load software into the environment, but you can use other package managers. The default environment defines two parameters to specify the software that should be loaded. These variables are set by submitters provided by Deadline Cloud, though you can set them in your own scripts and applications that use the default environment. They are:

- **CondaPackages** – A space-separated list of conda package match specifications to install for the job. For example, the Blender submitter would add `blender=3.6` to render frames in Blender 3.6.

- `CondaChannels` – A space-separated list of conda channels to install packages from. For service-managed fleets, packages are installed from the `deadline-cloud` channel. You can add other channels.

Control the job environment with OpenJD queue environments

You can define customized environments for your rendering jobs using *queue environments*. A queue environment is a template that controls the environment variables, file mappings, and other settings for jobs running in a specific queue. It enables you to tailor the execution environment for the jobs submitted to a queue to the requirements of your workloads. AWS Deadline Cloud provides three nested levels where you can apply Open Job Description (OpenJD) environments: queue, job, and step. For more information, see [Environments](#) on the GitHub website. By defining queue environments, you can ensure consistent and optimized performance for different types of jobs, streamline resource allocation, and simplify queue management.

The queue environment is a template that you attach to a queue in your AWS account from the AWS management console or using the AWS CLI. You can create one environment for a queue, or you can create multiple queue environments that applied in order to create the execution environment. This approach enables you to create and test an environment in steps to help ensure that it works correctly for you jobs.

Job and step environments are defined in the job template you use to create a job in your queue. The OpenJD syntax is the same in these different forms of environments. In this section we will show them inside of job templates.

Topics

- [Set environment variables in a queue environment](#)
- [Set the path in a queue environment](#)
- [Run a background daemon process from the queue environment](#)

Set environment variables in a queue environment

Many applications and frameworks use environment variables to control feature settings, logging levels, and display configuration. You can use Open Job Description (OpenJD) environments to set environment variables that every task command within their scope inherits. For more information, see [Environments](#) on the GitHub website.

Environment variable scope

AWS Deadline Cloud applies environment variables from queue environments that you attach to a queue. Within a job template, you can also define environment variables at the job and step levels using OpenJD environments. For more information, see [Environments](#) on the GitHub website. Variables defined at a narrower scope override variables with the same name from a broader scope.

- **Queue environment** – A template that you attach to a queue in Deadline Cloud. Variables apply to all jobs submitted to the queue. You can set variables with a `variables` map for fixed values, or use scripts for dynamic values.
- **Job environment** – Defined under `jobEnvironments` in a job template. Variables apply to all steps and tasks in the job. A job-level variable overrides a queue-level variable with the same name.
- **Step environment** – Defined under `stepEnvironments` in a job template. Variables apply only to the tasks in that step. A step-level variable overrides a job-level or queue-level variable with the same name.

Setting variables in a queue environment

You can set environment variables in a queue environment using a `variables` map for fixed values, or using a script with an `onEnter` action for dynamic values.

The following queue environment template uses a `variables` map to set the `QT_QPA_PLATFORM` variable to `offscreen`, which allows applications that use the Qt Framework to run on worker hosts without an interactive display. For more information, see the [Qt Framework](#) on the Qt website.

```
specificationVersion: 'environment-2023-09'
environment:
  name: QtOffscreen
  variables:
    QT_QPA_PLATFORM: offscreen
```

For dynamic values, such as modifying `PATH` or activating virtual environments, use a script that prints lines in the format `openjd_env: VAR=value` to stdout. The `openjd_env:` prefix is required. Using `echo`, `export`, or other shell mechanisms without the prefix does not propagate variables to jobs and tasks.

The following queue environment template sets the `QT_QPA_PLATFORM` variable using a script.

```
specificationVersion: 'environment-2023-09'
environment:
  name: QtOffscreen
  script:
    actions:
      onEnter:
        command: bash
        args:
          - "{{Env.File.Enter}}"
  embeddedFiles:
    - name: Enter
      type: TEXT
      data: |
        #!/bin/env bash
        set -euo pipefail
        echo "openjd_env: QT_QPA_PLATFORM=offscreen"
```

To attach a queue environment to your queue, use the Deadline Cloud console or the AWS CLI. For more information, see [Create a queue environment](#) in the AWS Deadline Cloud User Guide. The following AWS CLI command creates a queue environment from a template file.

```
aws deadline create-queue-environment \
  --farm-id FARM_ID \
  --queue-id QUEUE_ID \
  --priority 1 \
  --template-type YAML \
  --template file://my-queue-env.yaml
```

For more complex examples, such as creating and activating conda virtual environments, see the [Deadline Cloud queue environment samples](#) on the GitHub website.

Setting variables in a job template

In a job template, add a `variables` map to a `jobEnvironments` or `stepEnvironments` entry. Each entry is a key-value pair where the key is the variable name and the value is the variable value.

The following job template sets the `QT_QPA_PLATFORM` environment variable to `offscreen`, which allows applications that use the Qt Framework to run on worker hosts without an interactive display. For more information, see the [Qt Framework](#) on the Qt website.


```
specificationVersion: 'jobtemplate-2023-09'
name: MyJob
jobEnvironments:
- name: JobEnv
  variables:
    QT_QPA_PLATFORM: offscreen
```

You can set multiple variables in a single environment definition.

```
jobEnvironments:
- name: JobEnv
  variables:
    JOB_VERBOSEITY: MEDIUM
    JOB_PROJECT_ID: my-project-id
    JOB_ENDPOINT_URL: https://my-host-name/my/path
    QT_QPA_PLATFORM: offscreen
```

You can reference job parameters in variable values by using the `{{Param.ParameterName}}` syntax.

```
jobEnvironments:
- name: JobEnv
  variables:
    JOB_EXAMPLE_PARAM: "{{Param.ExampleParam}}"
```

To override a job-level variable for a specific step, define a `stepEnvironments` entry with the same variable name. The following example defines `JOB_PROJECT_ID` at the job level with the value `project-12`, and then overrides the value at the step level with `step-project-12`. Tasks in the step use the step-level value.

```
specificationVersion: 'jobtemplate-2023-09'
name: MyJob
jobEnvironments:
- name: JobEnv
  variables:
    JOB_PROJECT_ID: project-12
steps:
- name: MyStep
  stepEnvironments:
  - name: StepEnv
```

```
variables:  
  JOB_PROJECT_ID: step-project-12
```

Try it: Running the environment variable sample

The Deadline Cloud samples repository includes a [job bundle that demonstrates setting and viewing environment variables](#) on the GitHub website. The sample job template defines variables at both the job and step levels, then runs a task that prints the merged result. Use the following procedure to run the sample and inspect the results.

Prerequisites

1. If you do not have a Deadline Cloud farm with a queue and associated Linux fleet, follow the guided onboarding experience in the [Deadline Cloud console](#) to create one with default settings.
2. If you do not have the Deadline Cloud CLI and AWS Deadline Cloud monitor on your workstation, follow the steps in [Set up Deadline Cloud submitters](#).
3. Use `git` to clone the [Deadline Cloud samples GitHub repository](#) on the GitHub website.

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git  
cd deadline-cloud-samples/job_bundles
```

Running the sample

1. Use the Deadline Cloud CLI to submit the `job_env_vars` sample.

```
deadline bundle submit job_env_vars
```

2. In the Deadline Cloud monitor, select the new job to monitor its progress. After the Linux fleet associated with the queue has a worker available, the job completes in a few seconds. Select the task, then choose **View logs** in the top right menu of the tasks panel.

Comparing session actions with their definitions

The log view shows three session actions. Open the file [job_env_vars/template.yaml](#) on the GitHub website in a text editor to compare each action with its definition in the job template.

1. Select the **Launch JobEnv** session action. The log output shows the job-level environment variables being set.

```
Setting: JOB_VERBOSITY=MEDIUM
Setting: JOB_EXAMPLE_PARAM=An example parameter value
Setting: JOB_PROJECT_ID=project-12
Setting: JOB_ENDPOINT_URL=https://internal-host-name/some/path
Setting: QT_QPA_PLATFORM=offscreen
```

The following lines from the job template define this environment.

```
jobEnvironments:
- name: JobEnv
  variables:
    JOB_VERBOSITY: MEDIUM
    JOB_EXAMPLE_PARAM: "{{Param.ExampleParam}}"
    JOB_PROJECT_ID: project-12
    JOB_ENDPOINT_URL: https://internal-host-name/some/path
    QT_QPA_PLATFORM: offscreen
```

2. Select the **Launch StepEnv** session action. The log output shows the step-level variables, including the overridden JOB_PROJECT_ID.

```
Setting: STEP_VERBOSITY=HIGH
Setting: JOB_PROJECT_ID=step-project-12
```

The following lines from the job template define this environment.

```
stepEnvironments:
- name: StepEnv
  variables:
    STEP_VERBOSITY: HIGH
    JOB_PROJECT_ID: step-project-12
```

3. Select the **Task run** session action. The log output shows the merged environment variables available to the task. Notice that JOB_PROJECT_ID uses the step-level value step-project-12.

```
Environment variables starting with JOB_*:
JOB_ENDPOINT_URL=https://internal-host-name/some/path
JOB_EXAMPLE_PARAM='An example parameter value'
JOB_PROJECT_ID=step-project-12
JOB_VERBOSITY=MEDIUM
```

```
Environment variables starting with STEP_*:  
STEP_VERBOSITY=HIGH
```

Set the path in a queue environment

Use OpenJD environments to provide new commands in an environment. First you create a directory containing script files, and then add that directory to the PATH environment variables so that executables in your script can run them without needing to specify the directory path each time. The list of variables in an environment definition doesn't provide a way to modify the variable, so you do this by running a script instead. After the script sets things up and modifies the PATH, it exports the variable to the OpenJD runtime with the command `echo "openjd_env: PATH=$PATH"`.

Prerequisites

Perform the following steps to run the [sample job bundle with environment variables](#) from the Deadline Cloud samples repository on the GitHub website.

1. If you do not have a Deadline Cloud farm with a queue and associated Linux fleet, follow the guided onboarding experience in the [Deadline Cloud console](#) to create one with default settings.
2. If you do not have the Deadline Cloud CLI and Deadline Cloud monitor on your workstation, follow the steps in [Set up Deadline Cloud submitters](#) from the user guide.
3. Use `git` to clone the [Deadline Cloud samples GitHub repository](#) on the GitHub website.

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git  
Cloning into 'deadline-cloud-samples'...  
...  
cd deadline-cloud-samples/job_bundles
```

Run the path sample

1. Use the Deadline Cloud CLI to submit the `job_env_with_new_command` sample.

```
$ deadline bundle submit job_env_with_new_command  
Submitting to Queue: MySampleQueue  
...
```

2. In the Deadline Cloud monitor, you will see the new job and can monitor its progress. Once the Linux fleet associated with the queue has a worker available to run the job's task, the job completes in a few seconds. Select the task, then choose the **View logs** option in the top right menu of the tasks panel.

On the right are two session actions, **Launch RandomSleepCommand** and **Task run**. The log viewer in the center of the window corresponds to the selected session action on the right.

Compare session actions with their definitions

In this section you use the Deadline Cloud monitor to compare the session actions with where they are defined in the job template. It continues from the previous section.

Open the file [job_env_with_new_command/template.yaml](#) from the Deadline Cloud samples repository on the GitHub website in a text editor. Compare the session actions to where they are defined in the job template.

1. Select the **Launch RandomSleepCommand** session action in the Deadline Cloud monitor. You will see log output as follows.

```
2024/07/16 17:25:32-07:00
2024/07/16 17:25:32-07:00 =====
2024/07/16 17:25:32-07:00 ----- Entering Environment: RandomSleepCommand
2024/07/16 17:25:32-07:00 =====
2024/07/16 17:25:32-07:00 -----
2024/07/16 17:25:32-07:00 Phase: Setup
2024/07/16 17:25:32-07:00 -----
2024/07/16 17:25:32-07:00 Writing embedded files for Environment to disk.
2024/07/16 17:25:32-07:00 Mapping: Env.File.Enter -> /sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/embedded_filesf3tq_1os/tmpbt8j_c3f
2024/07/16 17:25:32-07:00 Mapping: Env.File.SleepScript -> /sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/embedded_filesf3tq_1os/tmperastlp4
2024/07/16 17:25:32-07:00 Wrote: Enter -> /sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/embedded_filesf3tq_1os/tmpbt8j_c3f
2024/07/16 17:25:32-07:00 Wrote: SleepScript -> /sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/embedded_filesf3tq_1os/tmperastlp4
2024/07/16 17:25:32-07:00 -----
2024/07/16 17:25:32-07:00 Phase: Running action
2024/07/16 17:25:32-07:00 -----
2024/07/16 17:25:32-07:00 Running command sudo -u job-user -i setsid -w /sessions/
session-ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/tmpbwrquq5u.sh
```

```

2024/07/16 17:25:32-07:00 Command started as pid: 2205
2024/07/16 17:25:32-07:00 Output:
2024/07/16 17:25:33-07:00 openjd_env: PATH=/sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/bin:/opt/conda/condabin:/home/job-
user/.local/bin:/home/job-user/bin:/usr/local/sbin:/usr/local/bin:/usr/
bin:/sbin:/bin:/var/lib/snapd/snap/bin
No newer logs at this moment.

```

The following lines from the job template specified this action.

```

jobEnvironments:
- name: RandomSleepCommand
  description: Adds a command 'random-sleep' to the environment.
  script:
    actions:
      onEnter:
        command: bash
        args:
          - "{{Env.File.Enter}}"
    embeddedFiles:
      - name: Enter
        type: TEXT
        data: |
          #!/bin/env bash
          set -euo pipefail

          # Make a bin directory inside the session's working directory for providing
new commands
          mkdir -p '{{Session.WorkingDirectory}}/bin'

          # If this bin directory is not already in the PATH, then add it
          if ! [[ ":$PATH:" == *'{{Session.WorkingDirectory}}/bin:*' ]]; then
            export "PATH={{Session.WorkingDirectory}}/bin:$PATH"

            # This message to Open Job Description exports the new PATH value to the
environment
            echo "openjd_env: PATH=$PATH"
          fi

          # Copy the SleepScript embedded file into the bin directory
          cp '{{Env.File.SleepScript}}' '{{Session.WorkingDirectory}}/bin/random-
sleep'

          chmod u+x '{{Session.WorkingDirectory}}/bin/random-sleep'

```

```
- name: SleepScript
  type: TEXT
  runnable: true
  data: |
    ...
```

2. Select the **Launch StepEnv** session action in the Deadline Cloud monitor. You see log output as follows.

```
2024/07/16 17:25:33-07:00
2024/07/16 17:25:33-07:00 =====
2024/07/16 17:25:33-07:00 ----- Running Task
2024/07/16 17:25:33-07:00 =====
2024/07/16 17:25:33-07:00 -----
2024/07/16 17:25:33-07:00 Phase: Setup
2024/07/16 17:25:33-07:00 -----
2024/07/16 17:25:33-07:00 Writing embedded files for Task to disk.
2024/07/16 17:25:33-07:00 Mapping: Task.File.Run -> /sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/embedded_filesf3tq_1os/tmpdrwuehjf
2024/07/16 17:25:33-07:00 Wrote: Run -> /sessions/session-
ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/embedded_filesf3tq_1os/tmpdrwuehjf
2024/07/16 17:25:33-07:00 -----
2024/07/16 17:25:33-07:00 Phase: Running action
2024/07/16 17:25:33-07:00 -----
2024/07/16 17:25:33-07:00 Running command sudo -u job-user -i setsid -w /sessions/
session-ab132a51b9b54d5da22cbe839dd946baaw1c8hk5/tmpz81iaqfw.sh
2024/07/16 17:25:33-07:00 Command started as pid: 2256
2024/07/16 17:25:33-07:00 Output:
2024/07/16 17:25:34-07:00 + random-sleep 12.5 27.5
2024/07/16 17:26:00-07:00 Sleeping for duration 26.90
2024/07/16 17:26:00-07:00 -----
2024/07/16 17:26:00-07:00 Uploading output files to Job Attachments
2024/07/16 17:26:00-07:00 -----
```

3. The following lines from the job template specified this action.

```
steps:
- name: EnvWithCommand
  script:
    actions:
      onRun:
        command: bash
        args:
```

```
- '{{Task.File.Run}}'
embeddedFiles:
- name: Run
  type: TEXT
  data: |
    set -xeuo pipefail

    # Run the script installed into PATH by the job environment
    random-sleep 12.5 27.5
hostRequirements:
  attributes:
  - name: attr.worker.os.family
    anyOf:
    - linux
```

Run a background daemon process from the queue environment

In many rendering use cases, loading the application and scene data can take a significant amount of time. If a job reloads them for every frame, it will spend most of its time on overhead. It's often possible to load the application once as a background daemon process, have it load the scene data, and then send it commands via inter-process communication (IPC) to perform the renders.

Many of the open source Deadline Cloud integrations use this pattern. The Open Job Description project provides an [adaptor runtime library](#) on the GitHub website with reliable IPC patterns on all supported operating systems.

To demonstrate this pattern, there is a [self-contained sample job bundle](#) on the GitHub website that uses Python and bash code to implement a background daemon and the IPC for tasks to communicate with it. The daemon is implemented in Python, and listens for a POSIX SIGUSR1 signal for when to process a task. The task details are passed to the daemon in a specific JSON file, and the results of running the task are returned as another JSON file.

Prerequisites

Perform the following steps to run the [sample job bundle with a daemon process](#) from the Deadline Cloud samples repository on the GitHub website.

1. If you do not have a Deadline Cloud farm with a queue and associated Linux fleet, follow the guided onboarding experience in the [Deadline Cloud console](#) to create one with default settings.
2. If you do not have the Deadline Cloud CLI and Deadline Cloud monitor on your workstation, follow the steps in [Set up Deadline Cloud submitters](#) from the user guide.

3. Use git to clone the [Deadline Cloud samples GitHub repository](#) on the GitHub website.

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
Cloning into 'deadline-cloud-samples'...
...
cd deadline-cloud-samples/job_bundles
```

Run the daemon sample

1. Use the Deadline Cloud CLI to submit the `job_env_daemon_process` sample.

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
Cloning into 'deadline-cloud-samples'...
...
cd deadline-cloud-samples/job_bundles
```

2. In the Deadline Cloud monitor application, you will see the new job and can monitor its progress. Once the Linux fleet associated with the queue has a worker available to run the job's task, it completes in about a minute. With one of the tasks selected, choose the **View logs** option in the top right menu of the tasks panel.

On the right there are two session actions, **Launch DaemonProcess** and **Task run**. The log viewer in the center of the window corresponds to the selected session action on the right.

Select the option **View logs for all tasks**. The timeline shows the rest of the tasks that ran as part of the session, and the Shut down DaemonProcess action that exited the environment.

View the daemon logs

1. In this section you use the Deadline Cloud monitor to compare the session actions with where they are defined in the job template. It continues from the previous section.

Open the file [job_env_daemon_process/template.yaml](#) from the Deadline Cloud samples repository on the GitHub website in a text editor. Compare the session actions to where they are defined in the job template.

2. Select the Launch DaemonProcess session action in Deadline Cloud monitor. You will see log output as follows.

```
2024/07/17 16:27:20-07:00
```

```

2024/07/17 16:27:20-07:00 =====
2024/07/17 16:27:20-07:00 ----- Entering Environment: DaemonProcess
2024/07/17 16:27:20-07:00 =====
2024/07/17 16:27:20-07:00 -----
2024/07/17 16:27:20-07:00 Phase: Setup
2024/07/17 16:27:20-07:00 -----
2024/07/17 16:27:20-07:00 Writing embedded files for Environment to disk.
2024/07/17 16:27:20-07:00 Mapping: Env.File.Enter -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/enter-daemon-
process-env.sh
2024/07/17 16:27:20-07:00 Mapping: Env.File.Exit -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/exit-daemon-
process-env.sh
2024/07/17 16:27:20-07:00 Mapping: Env.File.DaemonScript -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/daemon-
script.py
2024/07/17 16:27:20-07:00 Mapping: Env.File.DaemonHelperFunctions -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/daemon-
helper-functions.sh
2024/07/17 16:27:20-07:00 Wrote: Enter -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/enter-daemon-
process-env.sh
2024/07/17 16:27:20-07:00 Wrote: Exit -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/exit-daemon-
process-env.sh
2024/07/17 16:27:20-07:00 Wrote: DaemonScript -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/daemon-
script.py
2024/07/17 16:27:20-07:00 Wrote: DaemonHelperFunctions -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/daemon-
helper-functions.sh
2024/07/17 16:27:20-07:00 -----
2024/07/17 16:27:20-07:00 Phase: Running action
2024/07/17 16:27:20-07:00 -----
2024/07/17 16:27:20-07:00 Running command sudo -u job-user -i setsid -w /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/tmp_u8slys3.sh
2024/07/17 16:27:20-07:00 Command started as pid: 2187
2024/07/17 16:27:20-07:00 Output:
2024/07/17 16:27:21-07:00 openjd_env: DAEMON_LOG=/sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/daemon.log
2024/07/17 16:27:21-07:00 openjd_env: DAEMON_PID=2223

```

```
2024/07/17 16:27:21-07:00 openjd_env: DAEMON_BASH_HELPER_SCRIPT=/sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/daemon-
helper-functions.sh
```

The following lines from the job template specified this action.

```
stepEnvironments:
- name: DaemonProcess
  description: Runs a daemon process for the step's tasks to share.
  script:
    actions:
      onEnter:
        command: bash
        args:
          - "{{Env.File.Enter}}"
      onExit:
        command: bash
        args:
          - "{{Env.File.Exit}}"
    embeddedFiles:
      - name: Enter
        filename: enter-daemon-process-env.sh
        type: TEXT
        data: |
          #!/bin/env bash
          set -euo pipefail

          DAEMON_LOG='{{Session.WorkingDirectory}}/daemon.log'
          echo "openjd_env: DAEMON_LOG=${DAEMON_LOG}"
          nohup python {{Env.File.DaemonScript}} > $DAEMON_LOG 2>&1 &
          echo "openjd_env: DAEMON_PID=$!"
          echo "openjd_env:
DAEMON_BASH_HELPER_SCRIPT={{Env.File.DaemonHelperFunctions}}"

          echo 0 > 'daemon_log_cursor.txt'
          ...
```

3. Select one of the Task run: N session action in Deadline Cloud monitor. You will see log output as follows.

```
2024/07/17 16:27:22-07:00
2024/07/17 16:27:22-07:00 =====
2024/07/17 16:27:22-07:00 ----- Running Task
```

```

2024/07/17 16:27:22-07:00 =====
2024/07/17 16:27:22-07:00 Parameter values:
2024/07/17 16:27:22-07:00 Frame(INT) = 2
2024/07/17 16:27:22-07:00 -----
2024/07/17 16:27:22-07:00 Phase: Setup
2024/07/17 16:27:22-07:00 -----
2024/07/17 16:27:22-07:00 Writing embedded files for Task to disk.
2024/07/17 16:27:22-07:00 Mapping: Task.File.Run -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/run-task.sh
2024/07/17 16:27:22-07:00 Wrote: Run -> /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/embedded_fileswy00x5ra/run-task.sh
2024/07/17 16:27:22-07:00 -----
2024/07/17 16:27:22-07:00 Phase: Running action
2024/07/17 16:27:22-07:00 -----
2024/07/17 16:27:22-07:00 Running command sudo -u job-user -i setsid -w /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/tmpv4obfkhn.sh
2024/07/17 16:27:22-07:00 Command started as pid: 2301
2024/07/17 16:27:22-07:00 Output:
2024/07/17 16:27:23-07:00 Daemon PID is 2223
2024/07/17 16:27:23-07:00 Daemon log file is /sessions/
session-972e21d98dde45e59c7153bd9258a64dohwg4yg1/daemon.log
2024/07/17 16:27:23-07:00
2024/07/17 16:27:23-07:00 === Previous output from daemon
2024/07/17 16:27:23-07:00 ===
2024/07/17 16:27:23-07:00
2024/07/17 16:27:23-07:00 Sending command to daemon
2024/07/17 16:27:23-07:00 Received task result:
2024/07/17 16:27:23-07:00 {
2024/07/17 16:27:23-07:00   "result": "SUCCESS",
2024/07/17 16:27:23-07:00   "processedTaskCount": 1,
2024/07/17 16:27:23-07:00   "randomValue": 0.2578537967668988,
2024/07/17 16:27:23-07:00   "failureRate": 0.1
2024/07/17 16:27:23-07:00 }
2024/07/17 16:27:23-07:00
2024/07/17 16:27:23-07:00 === Daemon log from running the task
2024/07/17 16:27:23-07:00 Loading the task details file
2024/07/17 16:27:23-07:00 Received task details:
2024/07/17 16:27:23-07:00 {
2024/07/17 16:27:23-07:00   "pid": 2329,
2024/07/17 16:27:23-07:00   "frame": 2
2024/07/17 16:27:23-07:00 }
2024/07/17 16:27:23-07:00 Processing frame number 2
2024/07/17 16:27:23-07:00 Writing result
2024/07/17 16:27:23-07:00 Waiting until a USR1 signal is sent...

```

```

2024/07/17 16:27:23-07:00 ===
2024/07/17 16:27:23-07:00
2024/07/17 16:27:23-07:00 -----
2024/07/17 16:27:23-07:00 Uploading output files to Job Attachments
2024/07/17 16:27:23-07:00 -----

```

The following lines from the job template are what specified this action. `` steps:

```

steps:
- name: EnvWithDaemonProcess
  parameterSpace:
    taskParameterDefinitions:
      - name: Frame
        type: INT
        range: "{{Param.Frames}}"

  stepEnvironments:
    ...

  script:
    actions:
      onRun:
        timeout: 60
        command: bash
        args:
          - '{{Task.File.Run}}'
    embeddedFiles:
      - name: Run
        filename: run-task.sh
        type: TEXT
        data: |
          # This bash script sends a task to the background daemon process,
          # then waits for it to respond with the output result.

          set -euo pipefail

          source "$DAEMON_BASH_HELPER_SCRIPT"

          echo "Daemon PID is $DAEMON_PID"
          echo "Daemon log file is $DAEMON_LOG"

          print_daemon_log "Previous output from daemon"

```

```
send_task_to_daemon "{\"pid\": $$, \"frame\": {{Task.Param.Frame}} }"
wait_for_daemon_task_result

echo Received task result:
echo "$TASK_RESULT" | jq .
```

```
print_daemon_log "Daemon log from running the task"
```

```
hostRequirements:
  attributes:
    - name: attr.worker.os.family
      anyOf:
        - linux
```

Provide applications for your jobs

You can use a queue environment to load applications to process your jobs. When you create a service-managed fleet using the Deadline Cloud console, you have the option of creating a queue environment that uses the conda package manager to load applications.

If you want to use a different package manager, you can create a queue environment for that manager. For an example using Rez, see [Use a different package manager](#).

Deadline Cloud provides a conda channel to load a selection of rendering applications into your environment. They support the submitters that Deadline Cloud provides for digital content creation applications.

You can also load software for conda-forge to use in your jobs. The following examples show job templates using the queue environment provided by Deadline Cloud to load applications before running the job.

Topics

- [Getting an application from a conda channel](#)
- [Use a different package manager](#)
- [Use pip for Python-only packages](#)

Getting an application from a conda channel

You can create a custom queue environment for your Deadline Cloud workers that installs the software of your choice. This example queue environment has the same behavior as the environment used by the console for service-managed fleets. It runs conda directly to create the environment.

The environment creates a new conda virtual environment for every Deadline Cloud session that runs on a worker, and then deletes the environment when it is done.

Conda caches the downloaded packages so that they don't need to be downloaded again, but each session must link all of the packages into the environment.

The environment defines three scripts that run when Deadline Cloud starts a session on a worker. The first script runs when the `onEnter` action is called. It calls the other two to set up environment variables. When the script finishes running, the conda environment is available with all of the specified environment variables set.

For the latest version of the example, see [conda_queue_env_from_console.yaml](#) in the [deadline-cloud-samples](#) repository on the GitHub website.

If you want to use an application that is not available in the conda channel, you can create a conda channel in Amazon S3 and then build your own packages for that application. See [Create a conda channel using S3](#) to learn more.

Get open source libraries from conda-forge

This section describes how to use open source libraries from the conda-forge channel. The following example is a job template that uses the `polars` Python package.

The job sets the `CondaPackages` and `CondaChannels` parameters defined in the queue environment that tell Deadline Cloud where to get the package.

The section of the job template that sets the parameters is:

```
- name: CondaPackages
  description: A list of conda packages to install. The job expects a Queue Environment to handle this.
  type: STRING
  default: polars
```

```
- name: CondaChannels
  description: A list of conda channels to get packages from. The job expects a Queue
  Environment to handle this.
  type: STRING
  default: conda-forge
```

For the latest version of the complete example job template, see [stage_1_self_contained_template/template.yaml](#) on the GitHub website. For the latest version of the queue environment that loads the conda packages, see [conda_queue_env_from_console.yaml](#) in the [deadline-cloud-samples](#) repository on the GitHub website.

Get Blender from the deadline-cloud channel

The following example shows a job template that gets Blender from the deadline-cloud conda channel. This channel supports the submitters that Deadline Cloud provides for digital content creation software, though you can use the same channel to load software for your own use.

For a list of the software provided by the deadline-cloud channel, see [Default queue environment](#) in the *AWS Deadline Cloud User Guide*.

This job sets the CondaPackages parameter defined in the queue environment to tell Deadline Cloud to load Blender into the environment.

The section of the job template that sets the parameter is:

```
- name: CondaPackages
  type: STRING
  userInterface:
    control: LINE_EDIT
    label: Conda Packages
    groupLabel: Software Environment
  default: blender
  description: >
    Tells the queue environment to install Blender from the deadline-cloud conda
    channel.
```

For the latest version of the complete example job template, see [blender_render/template.yaml](#) on the GitHub website. For the latest version of the queue environment that loads the conda packages, see [conda_queue_env_from_console.yaml](#) in the [deadline-cloud-samples](#) repository on the GitHub website.

Use a different package manager

The default package manager for Deadline Cloud is conda. If you need to use a different package manager, such as Rez, you can create a custom queue environment that contains scripts that use your package manager instead.

This example queue environment provides the same behavior as the environment used by the console for service-managed fleets. It replaces the conda package manager with Rez.

The environment defines three scripts that run when Deadline Cloud starts a session on a worker. The first script runs when the `onEnter` action is called. It calls the other two to set up environment variables. When the script finishes running, the Rez environment is available with all of the specified environment variables set.

The example assumes that you have a customer-managed fleet that uses a shared file system for the Rez packages.

For the latest version of the example, see [rez_queue_env.yaml](#) in the [deadline-cloud-samples](#) repository on the GitHub website.

Use pip for Python-only packages

If the applications or libraries your jobs need are pure Python packages available on PyPI, you can use pip and the Python standard library venv module instead of conda or Rez. This approach works well when a package isn't published to a conda channel, or when you want a smaller virtual environment than conda creates. Unlike conda and Rez, pip and venv are included with Python itself, so worker hosts only need a python3 interpreter on the PATH. Deadline Cloud service-managed fleets provide one.

The [the section called "Pip queue environment"](#) creates a Python virtual environment in the session working directory when a job sets the `PipPackages` parameter, installs the requested packages into it with pip, and activates it so that the job's steps run with those packages available. Because the environment is created in the session working directory, Deadline Cloud cleans it up automatically when the session ends, and you don't need to write cleanup logic of your own. If a job doesn't set `PipPackages`, the queue environment does nothing, so you can add it to a queue that also runs jobs that don't use it.

To install packages from a private index such as an [AWS CodeArtifact](#) repository instead of the public PyPI index, set the `PipIndexUrl` and `PipExtraIndexUrls` parameters that the queue environment provides.

If you'd rather not configure a queue environment, you can define the same pip environment inline in a job bundle instead. For an example of each approach, see the [pip_package_job](#) and [pip_self_contained_job](#) job bundles in the [deadline-cloud-samples](#) repository on the GitHub website.

For production queues, pin package versions in the `PipPackages` parameter value instead of installing the latest release each time. Version pinning gives you reproducible environments and avoids unexpected breakages when a package publishes a new version. If a job depends on matching submitter and adaptor versions, pin both to the same release.

If your customer-managed fleet workers run jobs for more than one digital content creation (DCC) application, and you install Deadline Cloud adaptors with pip, install each adaptor into its own virtual environment. Installing multiple adaptors together in the same virtual environment can cause pip to resolve a shared dependency to a version that conflicts with one of the adaptors.

Worker hosts need network access to a package source, such as PyPI or a private mirror, to install packages when a session starts. If your customer-managed fleet workers run in a private subnet without internet access, use a VPC endpoint to your package source, a private pip mirror, or install the packages into your worker AMI instead.

The following table summarizes the trade-offs of each package manager to help you choose which one to use.

Package manager	Best for	Trade-offs
Conda (default)	DCC applications, complex native dependencies, cross-platform packages	Larger environments; packages must be published to a conda channel
Rez	Studio pipelines with existing Rez infrastructure	Requires Rez installed on worker hosts and a shared file system for the package repository
Pip	Python-only packages, PyPI-hosted libraries, lightweight installs	Limited to Python packages; no native dependency resolution

After you choose a queue environment, attach it to your queue. You can either add the YAML template from the **Queue environments** tab for your queue in the Deadline Cloud console, or use

the [deadline>CreateQueueEnvironment](#) operation. For console instructions, see [Create a queue environment](#) in the *AWS Deadline Cloud User Guide*. To attach a queue environment with the AWS CLI:

```
aws deadline create-queue-environment \  
  --farm-id FARM_ID \  
  --queue-id QUEUE_ID \  
  --priority 1 \  
  --template-type YAML \  
  --template file://queue-environment.yaml
```

Create a conda channel using S3

If your jobs need to run applications not available on the [deadline-cloud](#) channel or the [conda-forge](#) channel on the conda-forge website, you can host a custom conda channel to serve your own packages. When you create a queue in the AWS Deadline Cloud (Deadline Cloud) console, the console adds a conda queue environment by default. To make your packages available to jobs, add the custom channel to the queue environment.

A conda channel is static hosted content that you can host in [a variety of ways](#) on the prefix.dev website, including on a filesystem or in an Amazon Simple Storage Service (Amazon S3) bucket. If your Deadline Cloud farm uses a shared filesystem for assets, you can use any path on it as a channel name. You can host the channel in an Amazon S3 bucket for broader access using AWS Identity and Access Management (IAM) permissions.

You can [build and test packages locally](#), then [publish them to a channel](#). Building packages locally is an easy way to start iterating on package build recipes with no infrastructure setup. You can also use a Deadline Cloud [package building queue](#) to build packages and publish them to a channel. A package building queue simplifies maintaining packages for multiple operating systems and accelerator configurations. You can update versions and submit full sets of package builds from anywhere.

You can configure channels for your studio and your Deadline Cloud farm in multiple ways. You can have one Amazon S3 channel and configure all your workstations and farm hosts to use it. You can also have more than one channel and set up mirroring with AWS DataSync (DataSync). For example, your Deadline Cloud package building queue can publish to an Amazon S3 channel that gets mirrored on premises for workstations and on-premises farm hosts.

Topics

- [Build and test packages locally](#)
- [Publish packages to an Amazon S3 conda channel](#)
- [Configure production queue permissions for custom conda packages](#)
- [Add a conda channel to a queue environment](#)
- [Create a conda package for an application or plugin](#)
- [Create a conda build recipe for Blender](#)
- [Create a conda build recipe for Autodesk Maya](#)
- [Create a conda build recipe for the Maya adaptor](#)
- [Create a conda build recipe for Autodesk Maya to Arnold \(MtoA\) plugin](#)
- [Automate package builds with Deadline Cloud](#)

Build and test packages locally

Before publishing packages to Amazon S3 or setting up CI/CD automation on your Deadline Cloud farm, you can build and test conda packages on your workstation using a local filesystem channel. This approach lets you rapidly iterate locally on recipes and verify packages.

The `rattler-build` `publish` command builds a recipe, copies the resulting package to a channel, and indexes the channel in one step. When you target a local filesystem directory, `rattler-build` creates and initializes the channel automatically if the directory does not exist.

The following instructions use the Blender 4.5 sample recipe from the [Deadline Cloud samples repository](#) on the GitHub website. You can substitute a different recipe from the samples repository or use your own recipe.

Prerequisites

Before you begin, install the following tools on your workstation:

- **pixi** – A package manager that you use to install `rattler-build` and to test packages. For installation instructions, see [pixi.sh](#) on the Pixi website.
- **rattler-build** – The package build tool used by Deadline Cloud conda recipes. After you install `pixi`, run the following command to install `rattler-build`.

```
pixi global install rattler-build
```

- **git** – Required to clone the samples repository. On Windows, [Git for Windows](#) on the Git for Windows website also provides a bash shell, which some of the Windows sample recipes require.

Building and publishing a package to a local channel

In this procedure, you clone the Deadline Cloud samples repository and use `rattler-build` to build and publish the package to a local filesystem channel.

Note

Large applications can require tens of GB of free disk space for the source archive, extracted files, and build output. Make sure that you use a disk with enough available space for the package build output.

To build and publish a package to a local channel

1. Clone the Deadline Cloud samples repository.

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
```

2. Change to the `conda_recipes` directory.

```
cd deadline-cloud-samples/conda_recipes
```

3. Run the following command to build the Blender 4.5 recipe and publish the package to a local channel directory.

On Linux and macOS, run the following command.

```
rattler-build publish blender-4.5/recipe/recipe.yaml \  
  --to file://$HOME/my-conda-channel \  
  --build-number=+1
```

On Windows (cmd), run the following command.

```
rattler-build publish blender-4.5/recipe/recipe.yaml ^  
  --to file://%USERPROFILE%/my-conda-channel ^  
  --build-number=+1
```

The `rattler-build publish` command performs the following actions:

- Builds the package from the recipe.
- Creates the channel directory if the directory does not exist.
- Copies the package file to the channel.
- Indexes the channel so that package managers can find the package.

If your package recipe depends on packages from a particular channel, such as [conda-forge](#) on the conda-forge website, add `-c conda-forge` to the command.

About build numbers

The `--build-number=+1` option automatically picks the next build number based on what already exists in the destination channel. The best practice is to never overwrite a package in a channel. Always build to a new build number if the package would otherwise have the same filename. Using `--build-number=+1` achieves this when you build to a production channel or a staging channel that mirrors production.

If you want to control the build number directly, you can set it with a specific value such as `--build-number=7`. If you omit the option, `rattler-build` uses the build number defined in the `recipe.yaml` file.

For more information about `rattler-build publish`, see the [rattler-build publish documentation](#) on the prefix.dev website.

Debugging builds

If a build fails, `rattler-build` preserves the build directory so you can investigate. Run the following command to open an interactive shell in the build environment with all environment variables set up as they were during the build.

```
rattler-build debug shell
```

From the debug shell, you can modify files, run individual build commands, and add dependencies to isolate the issue. For more information, see [Debugging builds](#) on the prefix.dev website.

Testing the package

After you build and publish the package, create a temporary pixi project. Use the project to install the package from the local channel and verify that it works correctly.

To test the package

1. Create a temporary test directory and initialize a pixi project with the local channel.

On Linux and macOS, run the following commands.

```
mkdir package-test-env
cd package-test-env
pixi init --channel file://$HOME/my-conda-channel
```

On Windows (cmd), run the following commands.

```
mkdir package-test-env
cd package-test-env
pixi init --channel file://%USERPROFILE%/my-conda-channel
```

2. Add the package to the project.

```
pixi add blender=4.5
```

3. Verify that the package works correctly.

```
pixi run blender --version
```

The `pixi run` command activates the conda environment for the project directory and runs the specified command within it. The environment persists in the project directory, so you can use the same `pixi run` command from other terminals. For more information, see the [pixi run command](#) on the Pixi website.

When you are satisfied with the package, you can publish the package to an Amazon S3 conda channel so that Deadline Cloud workers can install the package. See [Publish packages to an S3 conda channel](#).

Removing packages from the channel

Avoid removing packages from channels that you use for production, because lockfiles reference specific packages by hash. Removing a package prevents re-creating environments from those lockfiles. For development and testing channels, you can remove a specific package by deleting the `.conda` file from the channel directory and then re-indexing the channel. First, install `rattler-index`.

```
pixi global install rattler-index
```

Then delete the package file and re-index the channel.

On Linux and macOS, run the following commands.

```
rm $HOME/my-conda-channel/linux-64/blender-4.5.0-hb0f4dca_1.conda
rattler-index fs $HOME/my-conda-channel
```

On Windows (cmd), run the following commands.

```
del %USERPROFILE%\my-conda-channel\win-64\blender-4.5.0-hb0f4dca_1.conda
rattler-index fs %USERPROFILE%\my-conda-channel
```

Package files are stored in platform-specific subdirectories such as `linux-64`, `win-64`, or `osx-arm64`. List the contents of these subdirectories to find the exact filename of the package you want to remove.

Cleaning up

After testing, you can remove the test project and the local channel.

To clean up test resources

1. Remove the test project directory.

On Linux and macOS, run the following command.

```
rm -rf package-test-env
```

On Windows (cmd), run the following command.


```
rmdir /s /q package-test-env
```

2. Remove the local conda channel directory.

On Linux and macOS, run the following command.

```
rm -rf $HOME/my-conda-channel
```

On Windows (cmd), run the following command.

```
rmdir /s /q %USERPROFILE%\my-conda-channel
```

3. (Optional) Remove the rattler-build output directory that contains the built package file.

On Linux and macOS, run the following command.

```
rm -rf deadline-cloud-samples/conda_recipes/output
```

On Windows (cmd), run the following command.

```
rmdir /s /q deadline-cloud-samples\conda_recipes\output
```

For supported plugins with plugin sync examples, see [Deliver custom plugins with plugin sync](#).

Publish packages to an Amazon S3 conda channel

You can publish conda packages to an Amazon Simple Storage Service (Amazon S3) bucket so that AWS Deadline Cloud (Deadline Cloud) workers can install them for running jobs. The `rattler-build publish` command works with Amazon S3 the same way as with a local filesystem channel. The command can build a recipe and publish the result, or publish a package file that you already built. In both cases, the command uploads the package to the bucket and indexes the channel in one step.

The `rattler-build publish` command authenticates with AWS using the standard credential chain, so it uses your AWS configuration like any AWS tool. For more information about configuring credentials, see [Configuration and credential file settings](#) in the *AWS Command Line Interface (AWS CLI) User Guide*.

Prerequisites

Before you publish packages to Amazon S3, complete the following prerequisites:

- **pixi and rattler-build** – Install pixi from pixi.sh on the Pixi website, then install rattler-build.

```
pixi global install rattler-build
```

- **git** – Required to clone the samples repository. On Windows, [git for Windows](#) on the Git for Windows website also provides a bash shell, which some of the Windows sample recipes require.
- **Amazon S3 bucket** – An Amazon S3 bucket to use as the conda channel. You can use the job attachments bucket from your Deadline Cloud farm or create a separate bucket.
- **AWS credentials** – Configure credentials on your workstation using the `aws configure` command or the `aws login` command. For more information, see [Setting up the AWS CLI](#) in the *AWS Command Line Interface User Guide*.
- **IAM permissions** – (Optional) To reduce the scope of permissions your credentials have, you can use an AWS Identity and Access Management (IAM) policy that only grants the following permissions on the Amazon S3 bucket and the channel prefix you use (for example, `/Conda/*`):
 - `s3:GetObject`
 - `s3:PutObject`
 - `s3:DeleteObject`
 - `s3:ListBucket`
 - `s3:GetBucketLocation`

Publishing a package to an Amazon S3 channel

Use `rattler-build publish` with an `s3://` target to publish a package to your Amazon S3 conda channel. If the channel does not exist in the bucket, `rattler-build` initializes the channel automatically. Before you begin, make sure that you have completed the [prerequisites](#).

The following example publishes the Blender 4.5 sample recipe from the [deadline-cloud-samples](#) repository on the GitHub website. You can substitute a different recipe from the samples repository or use your own recipe.

Note

Large applications can require tens of GB of free disk space for the source archive, extracted files, and build output. Make sure that you use a disk with enough available space for the package build output.

To publish a package to an Amazon S3 channel

1. Clone the Deadline Cloud samples repository.

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
```

2. Change to the `conda_recipes` directory.

```
cd deadline-cloud-samples/conda_recipes
```

3. Run the following command. Replace *amzn-s3-demo-bucket* with your bucket name.

```
rattler-build publish blender-4.5/recipe/recipe.yaml --to s3://amzn-s3-demo-bucket/  
Conda/Default --build-number=+1
```

The `/Conda/Default` prefix organizes the channel within the bucket. You can use a different prefix, but the prefix must be consistent across all commands and queue configurations that reference the channel.

About build numbers

The `--build-number=+1` option automatically picks the next build number based on what already exists in the destination channel. The best practice is to never overwrite a package in a channel. Always build to a new build number if the package would otherwise have the same filename. Using `--build-number=+1` achieves this when you build to a production channel or a staging channel that mirrors production.

If you want to control the build number directly, you can set it with a specific value such as `--build-number=7`. If you omit the option, `rattler-build` uses the build number defined in the `recipe.yaml` file.

If your package recipe depends on packages from a particular channel, such as [conda-forge](#) on the conda-forge website, add `-c conda-forge` to the command.

You can also publish a package file that you already built, for example, a `.conda` file from a local build. Replace `amzn-s3-demo-bucket` with your bucket name.

```
rattler-build publish output/linux-64/blender-4.5.0-hb0f4dca_0.conda \  
--to s3://amzn-s3-demo-bucket/Conda/Default
```

Initializing or reindexing a channel

When you use `rattler-build publish` to publish a package, the command initializes the channel automatically if the channel does not already exist. In most cases, you don't need to initialize or reindex the channel manually.

You might need to manually initialize or reindex a channel in the following situations:

- You want to create an empty channel before publishing any packages, for example, to verify that your Deadline Cloud queue environment can connect to the channel.
- You uploaded or deleted `.conda` files directly with Amazon S3 tools instead of using `rattler-build publish`, and the channel index is out of date.

Initializing an empty channel

To initialize an empty channel, create a `repodata.json` file and upload it to the `noarch` subdirectory of the channel prefix. Replace `amzn-s3-demo-bucket` with your bucket name.

```
echo '{"info":{"subdir":"noarch"},"packages":{},"packages.conda":{},"removed":  
[],"repodata_version":1}' > empty_channel_repodata.json  
aws s3api put-object --body empty_channel_repodata.json --key Conda/Default/noarch/  
repodata.json --bucket amzn-s3-demo-bucket
```

The `/Conda/Default` prefix must match the channel prefix that your queue environment uses. After you initialize the channel, you can publish packages to the channel by using `rattler-build publish`.

Reindexing a channel

If the channel index is out of date, use `rattler-index` to rebuild the index from the package files in the channel. First, install `rattler-index`.

```
pixi global install rattler-index
```

Then reindex the channel. Replace *amzn-s3-demo-bucket* with your bucket name.

```
rattler-index s3 s3://amzn-s3-demo-bucket/Conda/Default
```

Testing the package

After you publish the package, create a temporary pixi project to verify that the package works correctly. The project installs the package from the Amazon S3 channel.

To test the package

1. Create a temporary test directory and initialize a pixi project with the Amazon S3 channel. Replace *amzn-s3-demo-bucket* with your bucket name.

```
mkdir package-test-env  
cd package-test-env  
pixi init --channel s3://amzn-s3-demo-bucket/Conda/Default
```

2. Add the package to the project.

```
pixi add blender=4.5
```

3. Verify that the package works correctly.

```
pixi run blender --version
```

The `pixi run` command activates the conda environment for the project directory and runs the specified command within it. The environment persists in the project directory, so you can use the same `pixi run` command from other terminals. For more information, see the [pixi run command](#) on the Pixi website.

Removing packages from the channel

Avoid removing packages from channels that you use for production, because lockfiles reference specific packages by hash. Removing a package prevents re-creating environments from those lockfiles. For development and testing channels, you can remove a specific package by deleting the `.conda` file from the bucket and then reindexing the channel.

Delete the package file and then reindex the channel. Replace *amzn-s3-demo-bucket* with your bucket name.

```
aws s3 rm s3://amzn-s3-demo-bucket/Conda/Default/linux-64/blender-4.5.0-hb0f4dca_1.conda
```

After you delete the file, reindex the channel to update the channel metadata. For instructions, see [Reindexing a channel](#).

Package files are stored in platform-specific subdirectories such as `linux-64`, `win-64`, or `osx-arm64`. To list the packages in a subdirectory, run the following command.

```
aws s3 ls s3://amzn-s3-demo-bucket/Conda/Default/linux-64/
```

Cleaning up

After testing, remove the test project directory.

To clean up test resources

- Remove the test project directory.

On Linux and macOS, run the following command.

```
rm -rf package-test-env
```

On Windows (cmd), run the following command.

```
rmdir /s /q package-test-env
```

Debugging builds

If a build fails, `rattler-build` preserves the build directory so you can investigate. Run the following command to open an interactive shell in the build environment with all environment variables set up as they were during the build.

```
rattler-build debug shell
```

From the debug shell, you can modify files, run individual build commands, and add dependencies to isolate the issue. For more information, see [Debugging builds](#) on the prefix.dev website.

Building packages for other platforms

The `rattler-build publish` command builds packages for the operating system of the workstation where the command runs. If your Deadline Cloud fleet uses a different operating system than your workstation, or if your package has other host requirements, you have the following options:

- Run `rattler-build publish` on a host that matches the target operating system. For example, use an Amazon Elastic Compute Cloud (Amazon EC2) instance running Linux to build packages for a Linux fleet.
- Use a Deadline Cloud package building queue to automate builds on the target platform. See [Create a package building queue](#).
- (Advanced) Use cross-compilation to build packages for a different platform from your workstation. For more information, see [Cross-compilation](#) on the prefix.dev website.

Next steps

After you publish packages to your Amazon S3 conda channel, configure your Deadline Cloud queues to use the channel:

- [Configure production queue permissions for custom conda packages](#) – Grant your production queues read-only access to the Amazon S3 conda channel.
- [Add a conda channel to a queue environment](#) – Configure the queue environment to install packages from the Amazon S3 conda channel.

Configure production queue permissions for custom conda packages

Your production queue needs read-only permissions to the `/Conda` prefix in the queue's S3 bucket. Open the AWS Identity and Access Management (IAM) page for the role associated with the production queue and modify the policy with the following:

1. Open the Deadline Cloud console and navigate to the queue details page for the package build queue.
2. Choose the queue service role, then choose **Edit queue**.

3. Scroll to the **Queue service role** section, then choose **View this role in the IAM console**.
4. From the list of permission policies, choose the **AmazonDeadlineCloudQueuePolicy** for your queue.
5. From the **Permissions** tab, choose **Edit**.
6. Add a new section to the queue service role like the following. Replace *amzn-s3-demo-bucket* and *111122223333* with your own bucket and account.

```
{
  "Effect": "Allow",
  "Sid": "CustomCondaChannelReadOnly",
  "Action": [
    "s3:GetObject",
    "s3:ListBucket"
  ],
  "Resource": [
    "arn:aws:s3:::amzn-s3-demo-bucket",
    "arn:aws:s3:::amzn-s3-demo-bucket/Conda/*"
  ],
  "Condition": {
    "StringEquals": {
      "aws:ResourceAccount": "111122223333"
    }
  }
},
```

Add a conda channel to a queue environment

To use the S3 conda channel, you need to add the `s3://amzn-s3-demo-bucket/Conda/Default` channel location to the `CondaChannels` parameter of jobs that you submit to Deadline Cloud. The submitters provided with Deadline Cloud provide fields to specify custom conda channels and package.

You can avoid modifying every job by editing the conda queue environment for your production queue. Use the following procedure:

1. Open the Deadline Cloud console and navigate to the queue details page for the production queue.
2. Choose the environments tab.

3. Select the **Conda** queue environment, and then choose **Edit**.
4. Choose the **JSON editor**, and then in the script, find the parameter definition for `CondaChannels`.
5. Edit the line `default: "deadline-cloud"` so that it starts with the newly created S3 conda channel:

```
default: "s3://amzn-s3-demo-bucket/Conda/Default deadline-cloud"
```

Service-managed fleets enable flexible channel priority for conda by default. For a job requesting `blender=4.5` if Blender 4.5 is in both the new channel and the `deadline-cloud` channel, the package will be pulled from whichever channel is first in the channel list. If a specified package version is not found in the first channel then subsequent channels will be checked in order for the package version.

For customer-managed fleets, you can enable the use of conda packages by using one of the conda queue environment samples in the Deadline Cloud samples repository. For more information, see the [conda queue environment samples](#) on the GitHub website.

Create a conda package for an application or plugin

A conda package is a compressed archive of software written in any language. Conda supports a variety of operating system and architecture combinations, so you can package full applications like Blender, Maya, and Nuke alongside libraries for Python and other languages. For more information about conda packages, see the [Packages](#) documentation on the conda website.

To use a conda package, you install it into a virtual environment. A conda virtual environment has a *prefix directory* where packages are installed. Installing a package uses hardlinking or reflinking of files when supported, so creating multiple environments with the same packages does not use significant additional disk space. To use a virtual environment, you activate it to set environment variables. Activation runs scripts that packages provide, giving each package the opportunity to modify `PATH` or other environment variables. Conda packages typically contain applications or libraries, but the flexible activation means they can also point to applications installed on a shared filesystem.

Making a custom package involves three stages: a *recipe* contains the build instructions, a *package* is the built artifact (`.conda` or `.tar.bz2` file), and a *channel* hosts packages for installation. The

`rattler-build` `publish` command handles all three steps—it can build a recipe into a package and publish to a channel, or it can take a package artifact directly to publish it.

The [conda-forge](#) community maintains package recipes for a broad set of open source software, and hosts package artifacts in the `conda-forge` channel on the conda-forge website. You can configure your queue to include conda-forge as a package source, and then build custom packages that depend on conda-forge packages to run. For Linux, conda-forge hosts a full compiler toolchain including CUDA support, with consistent compiling and linking options selected. You can use conda-forge packages as dependencies in your own recipes, or install them alongside your custom packages in the same environment.

You can combine an entire application, including dependencies, into a conda package. The packages Deadline Cloud provides in the [deadline-cloud channel](#) for service-managed fleets use this binary repackaging approach. This organizes the same files as an installation to fit the conda virtual environment.

Note

Large applications can require tens of GB of free disk space for the source archive, extracted files, and build output. Make sure that you use a disk with enough available space for the package build output.

Package an application

When repackaging an application for conda, there are two goals:

- Most files for the application should be separate from the primary conda virtual environment structure. Environments can then mix the application with packages from other sources like [conda-forge](#) on the conda-forge website.
- When a conda virtual environment is activated, the application should be available from the `PATH` environment variable.

To repackage an application for conda

1. Write conda build recipes that install the application into a subdirectory like `$CONDA_PREFIX/opt/<application-name>`. This separates it from the standard prefix directories like `bin` and `lib`.

2. Add symlinks or launch scripts to `$CONDA_PREFIX/bin` to run the application binaries.

Alternatively, create `activate.d` scripts that the `conda activate` command will run to add the application binary directories to the `PATH`. On Windows, where symlinks are not supported everywhere environments can be created, use application launch or `activate.d` scripts instead.

3. Some applications depend on libraries not installed by default on Deadline Cloud service-managed fleets. For example, the X11 window system is usually unnecessary for non-interactive jobs, but some applications still require it to run without a graphical interface. You must provide those dependencies within the package you create.
4. If the application supports plugins, provide a clear convention that plugin packages should follow to integrate with the application in a virtual environment. For example, the [Maya 2026 sample recipe](#) on the GitHub website documents conventions for Maya plugins.
5. Ensure you follow the copyright and license agreements for the applications you package. We recommend using a private Amazon S3 bucket for your conda channel to control distribution and limit package access to your farm.

Sample recipes for the packages in the `deadline-cloud` channel are available in the [Deadline Cloud samples repository](#) on the GitHub website.

Package a plugin

Application plugins can be packaged as their own conda packages. When creating a plugin package, follow these guidelines:

- Include the host application package as both a build and a run dependency in the build recipe `recipe.yaml`. Use a version constraint so that the build recipe is only installed with compatible packages.
- Follow the host application package conventions for registering the plugin.

Adaptor packages

Some Deadline Cloud application integrations use an *adaptor* that extends the application interface to simplify [writing job templates](#). An adaptor is a command-line interface with support for running a background daemon, reporting status, and applying path mapping. For example, the [deadline-cloud-for-maya](#) repository on the GitHub website includes an integrated job submission GUI and a Maya adaptor that is available as the `maya-openjd` package on service-managed fleets.

For more information about adaptors and the adaptor runtime, see the [Open Job Description Adaptor Runtime](#) on the GitHub website.

Job submissions from Deadline Cloud submitter GUIs include a `CondaPackages` parameter value that specifies the conda packages to include in a virtual environment for running the job. The `CondaPackages` parameter value for Maya typically looks like `maya=2026.* maya-openjd=0.15.* maya-mtoa` and might contain alternative entries for plugin packages. When the queue environment sets up a conda virtual environment for running the job, it resolves these package names and version constraints to be compatible and adds all the dependency packages they need to run. Each adaptor and plugin package specifies what it is compatible with, including which versions of Maya, which versions of Python, and other dependencies.

To build your own adaptor packages, you can use sample recipes such as the [maya-openjd recipe](#) on the GitHub website. Build on the packages for Python and other dependencies provided by [conda-forge](#) on the conda-forge website. You might need to build the [deadline](#) and [openjd-adaptor-runtime](#) recipes on the GitHub website first to satisfy dependencies.

Create a conda build recipe for Blender

Blender is free to use and simple to package with conda, which makes it a good starting point for learning how to create conda packages for AWS Deadline Cloud (Deadline Cloud). The Blender Foundation provides [application archives](#) on the Blender website for multiple operating systems. The [Blender 4.5 sample recipe](#) in the Deadline Cloud samples repository on the GitHub website packages these archives into a conda package.

Understanding the recipe

The [recipe.yaml](#) file on the GitHub website defines the package metadata, source URLs, and build options in [rattler-build template syntax](#) on the prefix.dev website. The recipe specifies the version number once and provides different source URLs based on the operating system.

The `build` section in `recipe.yaml` turns off binary relocation and dynamic shared object (DSO) linking checks. These options control how the package works when installed into a conda virtual environment at any directory prefix. The default values used in the `build` section are designed for packaging each dependency library separately, but when binary repackaging an application, you need to change them. Blender does not require any `RPATH` adjustment because the application archives are built with relocatability in mind. See [Create a conda recipe for Maya](#) for an example of adding relocatability.

During the package build, the [build.sh](#) or [build_win.sh](#) script on the GitHub website runs to install files into the environment. These scripts copy the installation files into `$PREFIX/opt/blender`, create symlinks from `$PREFIX/bin` (on Linux), and set up activation scripts that configure environment variables such as `BLENDER_LOCATION`. On Windows, the activation script adds the Blender directory to the `PATH` instead of creating symlinks.

The Windows build script uses `bash` instead of a `cmd.exe` `.bat` file for consistency across platforms. You can install [git for Windows](#) on the Git for Windows website to provide `bash` for package building.

The recipe also includes a `deadline-cloud.yaml` file that specifies the conda platforms and metadata for submitting automated package build jobs to Deadline Cloud. For more information, see [Submit a package build job](#).

Building the Blender package

Use `rattler-build publish` to build the Blender 4.5 recipe and publish the package to a channel. You can publish to a local filesystem channel for testing or directly to an Amazon S3 channel for production use. If you completed the setup in [Build and test packages locally](#), run the following command from the `conda_recipes` directory.

```
rattler-build publish blender-4.5/recipe/recipe.yaml \
  --to file://$HOME/my-conda-channel \
  --build-number=+1
```

For other publishing options:

- To publish to an Amazon S3 channel, see [Publish packages to an S3 conda channel](#).
- To automate builds using a Deadline Cloud package building queue, see [Automate package builds with Deadline Cloud](#).

Test your package with a Blender render job

After you build the Blender 4.5 package, you can test it with a render job. If you do not have a Blender scene, download the Blender 3.5 - Cozy Kitchen scene from the [Blender demo files page](#) on the Blender website. The Deadline Cloud samples repository contains a `blender_render` job bundle and a conda queue environment that you can use for both local and cloud testing.

Testing locally

You can run the job template on your workstation using the [Open Job Description CLI](#) on the GitHub website. Install the CLI with pip.

```
pip install openjd-cli
```

From the `job_bundles` directory in the samples repository, run the following command. Replace */path/to/scene.blend* with the path to your Blender scene file.

```
openjd run blender_render/template.yaml \  
  --environment ../queue_environments/conda_queue_env_pyrattler.yaml \  
  -p CondaPackages=blender=4.5 \  
  -p CondaChannels=file://$HOME/my-conda-channel \  
  -p BlenderSceneFile=/path/to/scene.blend \  
  -p Frames=1
```

The `--environment` option applies the conda queue environment, which creates a conda virtual environment with the packages specified in `CondaPackages`. The `CondaChannels` parameter tells the queue environment where to find the packages. If you published to an Amazon S3 channel instead of a local channel, replace the `file://` path with your `s3://` channel URL.

Testing on Deadline Cloud

After you configure your production queue to use the Amazon S3 conda channel, you can submit the render job to Deadline Cloud. From the `job_bundles` directory in the samples repository, run the following command.

```
deadline bundle submit blender_render \  
  -p CondaPackages=blender=4.5 \  
  -p BlenderSceneFile=/path/to/scene.blend \  
  -p Frames=1
```

Use the Deadline Cloud monitor to track the progress of the job. In the monitor, select the task for the job and choose **View logs**. Select the **Launch conda** session action to verify that the package was found in the Amazon S3 channel.

Create a conda build recipe for Autodesk Maya

Commercial applications like Autodesk Maya introduce additional packaging requirements compared to open source applications like Blender. The [Blender recipe](#) packages a simple

relocatable archive under an open source license. Commercial applications are often distributed through installers and require license management configuration.

Considerations for commercial applications

The following considerations apply when packaging commercial applications. The details illustrate how each applies to Maya.

- **Licensing** – Understand the licensing rights and restrictions of the application. You might need to configure a license management system. Read the [Autodesk Subscription Benefits FAQ about Cloud Rights](#) and [Thin Client Licensing for Maya and MotionBuilder](#) on the Autodesk website to understand the cloud rights for Maya. Autodesk products rely on a `ProductInformation.pit` file that typically requires administrator access to configure. Product features for thin clients provide a relocatable alternative.
- **System library dependencies** – Some applications depend on libraries not installed on service-managed fleet worker hosts. Maya depends on libraries including freetype and fontconfig. When these libraries are available in the system package manager, such as `dnf` for AL2023, you can use the package manager as a source. Because RPM packages are not built to be relocatable, you need to use tools such as `patchelf` to resolve dependencies within the Maya installation prefix.
- **Administrator access for installation** – Some installers require administrator access. Service-managed fleets do not provide administrator access, so you need to install the application on a separate system and create an archive of the files for the package build. The Windows installer for Maya requires this approach. The [README.md](#) in the `maya-2025` recipe on the GitHub website documents a repeatable procedure using a freshly launched Amazon Elastic Compute Cloud (Amazon EC2) instance.
- **Plugin integration** – The sample Maya package defines `MAYA_NO_HOME=1` to isolate the application from user-level configuration, and adds module search paths to `MAYA_MODULE_PATH` so that plugin packages can place `.mod` files within the virtual environment. See the [Maya 2026 sample recipe](#) on the GitHub website for the full plugin integration convention.

Understanding the recipe

The [recipe.yaml](#) file on the GitHub website defines the package metadata in [rattler-build template syntax](#) on the prefix.dev website. Review the following sections of the file:

- **source** – References the installer archives, including the sha256 hash. On Linux, the source is the Autodesk installer archive. On Windows, the source includes both the installer archive and

a `cleanMayaForCloud.py` script from Autodesk that prepares Maya for cloud deployment. Update the hashes when you change the source files, for example when packaging a new version.

- **build** – Turns off the default binary relocation and DSO linking checks because the automatic mechanisms do not work correctly for the library and binary directories that Maya uses. On Linux, the recipe includes `patchelf` as a build dependency to manually set relative RPATHs.
- **about** – Metadata about the application for browsing or processing the contents of a conda channel.

The build scripts ([build.sh](#) for Linux, [build_win.sh](#) in the maya-2025 recipe for Windows) on the GitHub website include comments explaining each step. The scripts perform the following key tasks:

- **Extract the installer** – Extracts the Maya installation files into the conda prefix. The Linux and Windows scripts handle this differently due to the installer formats. See the build scripts for details.
- **Install system library dependencies** – On Linux, the script downloads and extracts system libraries that Maya needs but that are not present on service-managed fleet hosts. The script copies these libraries into the Maya `lib` directory so they are available within the conda environment.
- **Set relative RPATHs with patchelf** – On Linux, the script uses `patchelf --add-rpath` to add `$ORIGIN`-relative paths to the shared libraries. This approach follows the conda recommendation to never use `LD_LIBRARY_PATH` in conda environments. The script patches libraries at multiple directory levels (`lib`, `lib/python*/site-packages`, `lib/python*/lib-dynload`) so that each library can find its dependencies relative to its own location. The recipe follows the best practice of setting `DT_RUNPATH` instead of `DT_RPATH`, which allows `LD_LIBRARY_PATH` to override the search path when needed for debugging.
- **Configure thin client licensing** – The script sets up thin client licensing so that the `ProductInformation.pit` file can be located within the conda environment rather than requiring system-level administrator access. For more information, see [Thin Client Licensing for Maya and MotionBuilder](#) on the Autodesk website.
- **Set up activation scripts** – The scripts create activate and deactivate scripts that set environment variables including `MAYA_LOCATION`, `MAYA_VERSION`, `MAYA_NO_HOME`, and `MAYA_MODULE_PATH`. On Windows, the scripts produce both `.sh` and `.bat` activation files because the Deadline Cloud sample queue environments use `bash` to activate environments on Windows.

Building the Maya package

Before you build the Maya package, download the Maya installer from your Autodesk account. For Linux, place the archive directly into the `conda_recipes/archive_files` directory. For Windows, follow the procedure in the [maya-2025 README.md](#) on the GitHub website to create the archive.

Use `rattler-build publish` to build and publish the package. The Maya recipe requires `patchelf` as a build dependency on Linux, which is available from [conda-forge](#) on the conda-forge website. Add `-c conda-forge` to make the dependency available during the build. From the `conda_recipes` directory, run the following command.

```
rattler-build publish maya-2026/recipe/recipe.yaml \
  --to file://$HOME/my-conda-channel \
  --build-number=+1 \
  -c conda-forge
```

For other publishing options:

- To publish to an Amazon S3 channel, see [Publish packages to an S3 conda channel](#).
- To automate builds using a Deadline Cloud package building queue, see [Automate package builds with Deadline Cloud](#). To build both Linux and Windows packages, use the `--all-platforms` option with the `submit-package-job` script.

To render the turntable sample with Maya and Arnold, build both the [MtoA plugin](#) and [Maya adaptor](#) packages. After you publish all three packages, you can submit a test render job using the [turntable with Maya/Arnold job bundle](#) from the Deadline Cloud samples repository on the GitHub website. See [Test your packages with a Maya render job](#).

Create a conda build recipe for the Maya adaptor

The `maya-openjd` package provides the adaptor that integrates Maya with AWS Deadline Cloud (Deadline Cloud) job submissions. When you submit a Maya render job using a Deadline Cloud submitter GUI, the `CondaPackages` parameter includes `maya-openjd` alongside the `maya` package. The adaptor handles launching Maya, communicating render parameters, and managing the application lifecycle during a job session. For more information about adaptors, see [Adaptor packages](#).

Understanding the recipe

The [maya-openjd sample recipe](#) on the GitHub website builds the adaptor from the [deadline-cloud-for-maya source package](#) published to PyPI. The [recipe.yaml](#) on the GitHub website installs the package using pip into the conda environment.

The recipe depends on Python and two other packages from the Deadline Cloud samples repository that you need to build first. These packages are available on the GitHub website:

- [deadline](#) – The Deadline Cloud client library.
- [openjd-adaptor-runtime](#) – The Open Job Description adaptor runtime.

Python and other dependencies are available from [conda-forge](#) on the conda-forge website, so add `-c conda-forge` to the `rattler-build` publish command when you build the adaptor package.

Building the adaptor package

The `maya-openjd` package depends on two other packages from the Deadline Cloud samples repository. Build all three packages in order from the `conda_recipes` directory. The `-c conda-forge` option on each command is to satisfy recipe dependencies for Python and Python libraries.

Build the deadline package.

```
rattler-build publish deadline/recipe/recipe.yaml \  
  --to file://$HOME/my-conda-channel \  
  --build-number=+1 \  
  -c conda-forge
```

Build the `openjd-adaptor-runtime` package.

```
rattler-build publish openjd-adaptor-runtime/recipe/recipe.yaml \  
  --to file://$HOME/my-conda-channel \  
  --build-number=+1 \  
  -c conda-forge
```

Build the `maya-openjd` package.

```
rattler-build publish maya-openjd/recipe/recipe.yaml \  
  --to file://$HOME/my-conda-channel \  
  --build-number=+1
```

```
--build-number=+1 \  
-c conda-forge
```

For other publishing options:

- To publish to an Amazon S3 channel, see [Publish packages to an S3 conda channel](#).
- To automate builds using a Deadline Cloud package building queue, see [Automate package builds with Deadline Cloud](#).

Create a conda build recipe for Autodesk Maya to Arnold (MtoA) plugin

The Maya to Arnold (MtoA) plugin adds the Arnold renderer as an option within Maya. The [MtoA sample recipe](#) on the GitHub website demonstrates how to package a plugin as a separate conda package that integrates with the host application package.

Understanding the recipe

The [recipe.yaml](#) on the GitHub website specifies a dependency on the maya package for both build and run requirements. The dependency uses a version constraint so that the plugin is only installed with a compatible Maya version.

The recipe uses the same source archives as the Maya recipe. The build script installs MtoA and creates a `mtoa.mod` file in the `$PREFIX/usr/autodesk/maya$MAYA_VERSION/modules` directory that the Maya package configures in `MAYA_MODULE_PATH`. Arnold and Maya use the same licensing technology, so the Maya package already includes the licensing information that Arnold needs.

Building the MtoA package

Build the Maya package before you build the MtoA package, because MtoA depends on Maya at build time. Use `rattler-build publish` to build and publish the package. From the `conda_recipes` directory, run the following command.

```
rattler-build publish maya-mtoa-2026/recipe/recipe.yaml \  
--to file://$HOME/my-conda-channel \  
--build-number=+1
```

The `rattler-build publish` command uses the target channel as the highest priority channel when resolving dependencies, so the maya package you published earlier is available automatically.

For other publishing options:

- To publish to an Amazon S3 channel, see [Publish packages to an S3 conda channel](#).
- To automate builds using a Deadline Cloud package building queue, see [Automate package builds with Deadline Cloud](#).

Test your packages with a Maya render job

After you build the Maya, MtoA, and maya-openjd packages, you can test them with a render job. The Deadline Cloud samples repository contains a [turntable with Maya/Arnold job bundle](#) on the GitHub website that renders an animation using Maya and Arnold. The job bundle also uses FFmpeg to encode a video, which is available from the conda-forge channel.

Testing locally

You can run the job template on your workstation using the [Open Job Description CLI](#) on the GitHub website. Install the CLI with pip.

```
pip install openjd-cli
```

From the `job_bundles` directory in the samples repository, run the following command. The `ErrorOnArnoldLicenseFail=false` parameter tells Arnold to render with watermarks instead of failing when no license is available.

```
openjd run turntable_with_maya_arnold/template.yaml \  
  --environment ../queue_environments/conda_queue_env_pyrattler.yaml \  
  -p CondaPackages="maya maya-mtoa maya-openjd ffmpeg" \  
  -p CondaChannels="file://$HOME/my-conda-channel conda-forge" \  
  -p ErrorOnArnoldLicenseFail=false \  
  -p FrameRange=1-5
```

The `--environment` option applies the conda queue environment, which creates a conda virtual environment with the packages specified in `CondaPackages`. The `CondaChannels` parameter includes both the local channel for your custom packages and `conda-forge` for `ffmpeg`. If you published to an Amazon S3 channel instead of a local channel, replace the `file://` path with your `s3://` channel URL.

When the job completes, the rendered output is in the `turntable_with_maya_arnold/output/` directory.

Testing on Deadline Cloud

After you configure your production queue to use the Amazon S3 conda channel, submit the render job to Deadline Cloud. Add the conda-forge channel to the CondaChannels parameter in your conda queue environment to provide a source for ffmpeg and the Python dependencies that the adaptor requires. From the job_bundles directory in the samples repository, run the following command.

```
deadline bundle submit turntable_with_maya_arnold
```

Use the Deadline Cloud monitor to track the progress of the job. In the monitor, select the task for the job and choose **View logs**. Select the **Launch conda** session action to verify that the maya, maya-mtoa, and maya-openjd packages were found in the Amazon S3 channel.

Automate package builds with Deadline Cloud

For CI/CD workflows or when you need to build packages for multiple operating systems, you can create a Deadline Cloud package building queue. The queue schedules build jobs on your fleet, which build the packages and publish them to your Amazon Simple Storage Service (Amazon S3) conda channel. This automation simplifies maintaining continuous package builds for software releases across all your required configurations.

You can create a package building queue using an AWS CloudFormation (CloudFormation) template, or manually from the Deadline Cloud console. The CloudFormation template deploys a complete farm with a production queue and a package building queue already configured. Creating the queue from the console gives you more control over individual settings.

Create a package building queue with CloudFormation

You can use a CloudFormation template to create a Deadline Cloud farm that includes a package building queue. The template configures a production queue and a package building queue with a private Amazon S3 conda channel.

Before you deploy the template, create an Amazon S3 bucket to hold job attachments and your conda channel. You can create a bucket from the [Amazon S3 console](#). You need the bucket name when you deploy the template.

To deploy the CloudFormation template

1. Download the [deadline-cloud-starter-farm-template.yaml](#) template from the [Deadline Cloud samples](#) directory on the GitHub website.
2. From the [CloudFormation console](#), choose **Create Stack**, then **With new resources (standard)**.
3. Select the option to upload a template file, then upload the `deadline-cloud-starter-farm-template.yaml` file.
4. Enter a name for the stack, such as **StarterFarm**, and provide the name of an Amazon S3 bucket for job attachments and the conda channel.
5. Follow the CloudFormation console steps to complete stack creation.

For more information about the template parameters and customization options, see the [starter farm README](#) in the Deadline Cloud samples repository on the GitHub website.

Create a package building queue from the console

Follow the instructions in [Create a queue](#) in the *Deadline Cloud User Guide*. Make the following changes:

- In step 5, choose an existing Amazon S3 bucket. Specify a root folder name such as **DeadlineCloudPackageBuild** so that build artifacts stay separate from your normal Deadline Cloud attachments.
- In step 6, you can associate the package building queue with an existing fleet, or you can create an entirely new fleet if your current fleet is unsuitable.
- In step 9, create a new service role for your package building queue. You will modify the permissions to give the queue the permissions required for uploading packages and reindexing a conda channel.

Configure the package building queue permissions

To allow the package building queue to access the `/Conda` prefix in the queue's Amazon S3 bucket, you must modify the queue's role to give it read/write access. The role needs the following permissions so that package build jobs can upload new packages and reindex the channel.

- `s3:GetObject`
- `s3:PutObject`

- `s3:ListBucket`
 - `s3:GetBucketLocation`
 - `s3:DeleteObject`
1. Open the Deadline Cloud console and navigate to the queue details page for the package build queue.
 2. Choose the queue service role, then choose **Edit queue**.
 3. Scroll to the **Queue service role** section, then choose **View this role in the IAM console**.
 4. From the list of permission policies, choose the **AmazonDeadlineCloudQueuePolicy** for your queue.
 5. From the **Permissions** tab, choose **Edit**.
 6. Add a new section to the queue service role like the following. Replace *amzn-s3-demo-bucket* and *111122223333* with your own bucket and account.

```
{
  "Effect": "Allow",
  "Sid": "CustomCondaChannelReadWrite",
  "Action": [
    "s3:GetObject",
    "s3:PutObject",
    "s3:DeleteObject",
    "s3:ListBucket",
    "s3:GetBucketLocation"
  ],
  "Resource": [
    "arn:aws:s3:::amzn-s3-demo-bucket",
    "arn:aws:s3:::amzn-s3-demo-bucket/Conda/*"
  ],
  "Condition": {
    "StringEquals": {
      "aws:ResourceAccount": "111122223333"
    }
  }
},
```

Submit a package build job

After you create a package building queue and configure the queue permissions, you can submit jobs to build conda packages. The `submit-package-job` script in the [Deadline Cloud samples repository](#) on the GitHub website submits a build job for a conda recipe.

You need the following:

- The Deadline Cloud CLI installed on your workstation. For installation instructions, see the [deadline-cloud](#) repository on the GitHub website.
- An active [AWS Deadline Cloud monitor \(Deadline Cloud monitor\)](#) login session.
- A clone of the [Deadline Cloud samples repository](#) on the GitHub website.

To submit a package build job

1. Open the Deadline Cloud configuration GUI and set the default farm and queue to your package building queue.

```
deadline config gui
```

2. Change to the `conda_recipes` directory in the samples repository.

```
cd deadline-cloud-samples/conda_recipes
```

3. Run the `submit-package-job` script with the recipe directory. The following example builds the Blender 4.5 recipe.

```
./submit-package-job blender-4.5/
```

If the recipe requires a source archive that you have not yet downloaded, the script provides download instructions. Download the archive and run the script again.

After you submit the job, use the Deadline Cloud monitor to view the progress and status of the job.

Home > Conda Blog Farm > Package Build Queue

Job monitor Info

Reset to default layout

Jobs (1/1) Info

Find jobs Any User (default) Status

Job name	Progress	Status	Duration	Priority	Failed tasks	Create time	Start time	End time
CondaBuild: blender-4.1		100% (2/2) ✓ Succeeded	00:22:05	50	0	45m 43s ago	43m 15s ago	21m 9s ago

Steps (1/2) Info

Find steps

Step name	Progress	Status	Duration	Failed ta...	Sta
PackageBuild		100% (1/1) ✓ Succeeded	00:20:53	0	43m
ReindexCo...		100% (1/1) ✓ Succeeded	00:00:54	0	22m

Tasks (1/1) Info

Find tasks

Status	Duration	Retries / Ma...	Start time	End time
✓ Succeeded	00:19:55	0/1	42m 18s ago	22m 22s ago

The monitor shows the two steps of the job: building the package and then reindexing the conda channel. When you right-click on the task for the package building step and choose **View logs**, the monitor shows the session actions:

- **Sync attachments** – Copies the input job attachments or mounts a virtual file system.
- **Launch conda** – The queue environment action. The build job doesn't specify conda packages, so this action finishes quickly.
- **Launch CondaBuild Env** – Creates a conda virtual environment with the software needed to build a conda package and reindex a channel.
- **Task run** – Builds the package and uploads the results to Amazon S3.

As the actions run, they send logs to Amazon CloudWatch (CloudWatch). When a job is complete, select **View logs for all tasks** to see additional logs about the setup and teardown of the environment.

Run host configuration scripts with administrator privileges

Host configuration scripts allow you to perform administrative tasks, such as software installation, on your service-managed fleet workers. These scripts run with elevated privileges (sudo on Linux, Administrator on Windows), giving you the flexibility to configure your workers for your system.

Deadline Cloud runs the script after the worker enters the STARTING state and before it runs any tasks.

The script runs after Deadline Cloud sets up usage-based licensing (UBL). Deadline Cloud sets the UBL license environment variables when the worker instance launches, before the worker agent starts, and the worker agent then runs your host configuration script. The automatic UBL setup doesn't overwrite environment variables that your script sets: license variables that your script sets system-wide take precedence for the jobs that the worker runs. For more information about licensing options, see [Using software licenses with Deadline Cloud](#).

Important

The script runs with elevated permissions. It is your responsibility to ensure that the script does not introduce any security issues.

When you use a host configuration script you are responsible for monitoring the health of your fleet.

Common uses for host configuration scripts include:

- Installing software that requires administrator access
- Installing Docker containers
- Installing third-party cloud storage solutions such as LucidLink. For a walkthrough, see [Set up LucidLink with service managed fleet scripts for Deadline Cloud](#) on the AWS for M&E Blog.

You can create and update a host configuration script using the console or using the AWS CLI.

Console

1. On the **Fleet details** page, choose the **Configurations** tab.
2. In the **Script** field, enter the script to run with elevated permissions. You can choose **Import** to load a script from your workstation.
3. Set a timeout period in seconds for running the script. The default is 300 seconds (5 minutes).
4. Choose **Save changes** to save the script.

Create with CLI

Use the following AWS CLI command to create a fleet with a host configuration script. Replace the *placeholder* text with your information.

```
aws deadline create-fleet \  
--farm-id farm-12345 \  
--display-name "fleet-name" \  
--max-worker-count 1 \  
--configuration '{  
  "serviceManagedEc2": {  
    "instanceCapabilities": {  
      "vCpuCount": {"min": 2},  
      "memoryMiB": {"min": 4096},  
      "osFamily": "linux",  
      "cpuArchitectureType": "x86_64"  
    },  
    "instanceMarketOptions": {"type": "spot"}  
  },  
  "instanceMarketOptions": {"type": "spot"}  
}' \  
--role-arn arn:aws:iam::111122223333:role/role-name \  
--host-configuration '{ "scriptBody": "script body", "scriptTimeoutSeconds": timeout value}'
```

Update with CLI

Use the following AWS CLI command to update a fleet's host configuration script. Replace the *placeholder* text with your information.

```
aws deadline update-fleet \  
--farm-id farm-12345 \  
--fleet-id fleet-455678 \  
--host-configuration '{ "scriptBody": "script body", "scriptTimeoutSeconds": timeout value}'
```

The following scripts demonstrate:

- The environment variables available to the script
- That AWS credentials are working in the shell
- That the script is running in an elevated shell

Linux

Use the following script to show that a script is running with `root` privileges:

```
# Print environment variables
set
# Check AWS Credentials
aws sts get-caller-identity
```

Windows

Use the following PowerShell script to show that a script is running with Administrator privileges:

```
Get-ChildItem env: | ForEach-Object { "$($_.Name)=$($_.Value)" }
aws sts get-caller-identity
function Test-AdminPrivileges {
    $currentUser = New-Object
    Security.Principal.WindowsPrincipal([Security.Principal.WindowsIdentity]::GetCurrent())
    $isAdmin =
    $currentUser.IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)

    return $isAdmin
}

if (Test-AdminPrivileges) {
    Write-Host "The current PowerShell session is elevated (running as
Administrator)."
} else {
    Write-Host "The current PowerShell session is not elevated (not running as
Administrator)."
}
exit 0
```

Troubleshoot host configuration scripts

When you run the host configuration script:

- On success: The worker runs the job
- On failure (non-zero exit code or crash):
 - The worker shuts down

The fleet automatically launches a new worker using the latest host configuration script

To monitor the script:

1. Open the fleet page in the Deadline Cloud console.
2. Choose **View workers** to open the Deadline Cloud monitor.
3. View the worker status in the monitor page.

 Tip

When testing host configuration scripts, set the fleet's maximum worker count to 1 to avoid starting multiple workers while iterating on the script.

Important notes:

- Workers that shut down due to an error are not available in the list of workers in the monitor. Use CloudWatch Logs to view the worker logs in the following log group:

```
/aws/deadline/farm-XXXXX/fleet-YYYYY
```

Within that log group, look for a stream named `worker-ZZZZZ`.

- CloudWatch Logs retains worker logs according to your configured retention period.

Monitor host configuration script execution

With host configuration scripts, you can take full control of a Deadline Cloud worker. You can install any software package, reconfigure operating system parameters, or mount shared file systems. With this advanced feature and Deadline Cloud's capability to scale to thousands of workers, you can monitor when configuration scripts are executed successfully or failed.

We recommend the following solutions for monitoring host configuration script execution.

CloudWatch Logs monitoring

All fleet host configuration logs are streamed to the fleet's CloudWatch log group, and specifically to a worker's CloudWatch log stream. For example, `/aws/deadline/farm-123456789012/fleet-777788889999` is the log group for farm 123456789012, fleet 777788889999.

Each worker provisions a dedicated log stream, for example `worker-123456789012`. Host configuration logs include log banners such as *Running Host Configuration Script* and *Finished running Host Configuration Script, exit code: 0*. The exit code of the script is included in the finished banner and can be queried using CloudWatch tools.

CloudWatch Logs Insights

CloudWatch Logs Insights offers advanced capabilities to analyze log information. For example, the following Log Insights query parses for the host configuration exit code, sorted by time:

```
fields @timestamp, @message, @logStream, @log
| filter @message like /Finished running Host Configuration Script/
| parse @message /exit code: (?<exit_code>\d+)/
| display @timestamp, exit_code
| sort @timestamp desc
```

For more information about CloudWatch Logs Insights, see [Analyzing log data with CloudWatch Logs Insights](#) in the *Amazon CloudWatch Logs User Guide*.

Worker agent structured logging

The Deadline Cloud worker agent publishes structured JSON logs to CloudWatch. The worker agent offers many structured logs for analyzing worker health. For more information, see [Deadline Cloud worker agent logging](#) on the GitHub website.

The attributes of the structured logs are unpacked to fields in Log Insights. You can use this CloudWatch capability to count and analyze host configuration startup failures. For example, a count and bin query can be used to determine how often failures occur:

```
fields @timestamp, @message, @logStream, @log
| sort @timestamp desc
| filter message like /Worker Agent host configuration failed with exit code/
| stats count(*) by exit_code, bin(1h)
```

CloudWatch metric filters for metrics and alarming

You can set up CloudWatch metric filters to generate CloudWatch metrics from logs. Metric filters let you create alarms and dashboards for monitoring host configuration script execution.

To create a metric filter

1. Open the CloudWatch console.

2. In the navigation pane, choose **Logs**, then **Log groups**.
3. Select your fleet's log group.
4. Choose **Create metric filter**.
5. Define your filter pattern using one of the following:

- **For success metrics:**

```
{$.message = "*Worker Agent host configuration succeeded.*"}
```

- **For failure metrics:**

```
{$.exit_code != 0 && $.message = "*Worker Agent host configuration failed with  
exit code*"}
```

6. Choose **Next** to create a metric with the following values:
 - **Metric namespace:** Your metric namespace (for example, **MyDeadlineFarm**)
 - **Metric name:** Your requested metric name (for example, **host_config_failure**)
 - **Metric value:** **1** (each instance is a count of 1)
 - **Default value:** Leave empty
 - **Unit:** **Count**

After creating metric filters, you can configure standard CloudWatch alarms to take action on elevated host configuration failure rates, or add the metrics to a CloudWatch dashboard for day-to-day operations and monitoring.

For more details, see [Filter and pattern syntax](#) in the *Amazon CloudWatch Logs User Guide*.

Using software licenses with Deadline Cloud

With Deadline Cloud, you can license the applications and renderers that process your jobs in two ways:

- *Usage-based licensing (UBL)* – Deadline Cloud provides on-demand licenses for supported third-party applications and renderers, billed for the hours that your fleet uses them to process a job. There is no set number of licenses, so your fleet can scale as needed.
- *Bring your own license (BYOL)* – your workers check out licenses from a license server that you run, so you can use your existing license agreements or software that UBL doesn't cover.

Both methods work on both fleet types. The setup difference is how workers reach the license source: service-managed fleet workers run in infrastructure that AWS manages, outside your VPC, so they reach your license server through a connectivity option that you configure. Customer-managed fleet workers run on your own compute and use your existing network.

The following table summarizes the options. The sections after the table describe each option and link to setup instructions.

Licensing	Fleet type	How workers get a license	Setup
UBL	Service-managed	Automatic for supported products	None
UBL	Customer-managed	License endpoint in your VPC, for a subset of UBL products	Connect customer-managed fleets to a license endpoint
BYOL	Service-managed	Your license server, reached through a VPC resource endpoint, SSM port forwarding, or the internet	Connect service-managed fleets to a custom license server
BYOL	Customer-managed	Your license server, over your existing network	Configure license environment variables on your workers

Usage-based licensing

UBL licenses cover processing jobs on workers. They don't license the interactive DCC applications on your workstations. For the list of supported products and the per-hour rate for each, see [AWS Deadline Cloud pricing](#).

On service-managed fleets, UBL is available with no setup: supported products are licensed automatically when they run on a worker. Deadline Cloud sets the license environment variables on each worker when the instance launches, before the worker agent starts and before any host configuration script runs. On customer-managed fleets, you create a *license endpoint* in your VPC to route license requests to Deadline Cloud, and a subset of the UBL products is available. For the product list and setup steps, see [Connect customer-managed fleets to a license endpoint](#).

Bring your own license

With BYOL, workers request licenses from a license server that you run, the same way your on-premises render farm does. On a customer-managed fleet, your workers are on your own network, so you point each application at your license server with its usual environment variable or configuration setting.

Service-managed fleet workers run outside your VPC, so you connect them to your license server in one of three ways:

- **VPC resource endpoint** – connect your fleet to a license server in your VPC through VPC Lattice. Workers reach the server using a private domain name. For more information, see [Connect VPC resources to your SMF with VPC resource endpoints](#).
- **SSM port forwarding** – a queue environment forwards license ports from each worker to an Amazon EC2 instance in your account that hosts your license server or acts as a proxy for it. For more information, see [Connect service-managed fleets to a custom license server](#).
- **Internet** – workers connect directly to a license server that is reachable over the internet, such as a software vendor's cloud license service.

The license server doesn't have to be in the same VPC or account as the resource gateway or proxy instance. The gateway or proxy only needs network access to the server, for example through VPC peering, a transit gateway, or a VPN connection. With this approach, service-managed fleet workers can use license servers in a shared networking account.

Your license count doesn't limit how many workers Deadline Cloud starts. To keep concurrent tasks within your available seats, set up a limit. For more information, see [Create resource limits for jobs](#) and the [Enforce fixed license limits with a Deadline Cloud submission hook](#) sample. To turn off UBL entirely for a queue so that jobs only use your license server, see the [Disconnect Deadline Cloud usage-based licensing with a queue environment](#) sample.

Combining BYOL and UBL

You can list your license server ahead of the UBL endpoint in an application's license configuration so that workers consume your existing licenses first and fall back to UBL when those licenses run out. For more information, see [Combining BYOL and UBL](#).

Connect service-managed fleets to a custom license server

You can bring your own license server to use with a Deadline Cloud service-managed fleet. To bring your own license, you can configure a license server using a queue environment in your farm. To configure your license server, you should already have a farm and queue set up.

Service-managed fleet workers run in infrastructure that AWS manages, outside your VPC, so they reach a license server in one of three ways:

- Through a VPC resource endpoint. Workers connect to a license server in your VPC through VPC Lattice using a private domain name, without a queue environment or port forwarding. For setup steps, see [Connect VPC resources to your SMF with VPC resource endpoints](#). Set the application's license environment variable to the resource endpoint's domain name the same way as for a server address in [Step 3: Connect a rendering application to an endpoint](#).
- Through Amazon EC2 Systems Manager (SSM) port forwarding. A queue environment forwards license ports from each worker to an Amazon EC2 instance in your account that hosts the license server, or that proxies traffic to it. The rest of this page covers this option.
- Directly over the internet. Your workers obtain a license from the software vendor's license server using the internet.

With the resource endpoint and port forwarding options, the license server doesn't have to be in the same VPC or account as the resource gateway or proxy instance. The gateway or proxy only needs network access to the server, for example through VPC peering, a transit gateway, or a VPN connection.

With the following instructions, you use SSM to forward ports from a worker instance to your license server or proxy instance. In this example, if your license server is unable to provide a license, Deadline Cloud usage-based licensing is used as a fallback. Remove the sections that don't apply to your pipeline or products for which you don't want to use usage based licensing after exhausting your licenses.

Topics

- [Step 1: Configure the queue environment](#)
- [Step 2: \(Optional\) License proxy instance setup](#)
- [Step 3: CloudFormation template setup](#)

Step 1: Configure the queue environment

You can configure a queue environment in your queue to access your license server. First, ensure that you have an AWS instance configured with license server access using one of the following methods:

- License server – The instance hosts the license servers directly.
- License proxy – The instance has network access to the license server, and forwards license server ports to the license server. For details on how to configure a license proxy instance, see [Step 2: \(Optional\) License proxy instance setup](#).

For information about configuring license environment variables, see [Step 3: Connect a rendering application to an endpoint](#). For a custom license server setup, the license server address remains localhost instead of the Amazon VPC endpoint.

To add required permissions to the queue role

1. From the [Deadline Cloud console](#), choose **Go to Dashboard**.
2. From the dashboard, select the farm, and then the queue you want to configure.
3. From queue details > service role, select the role.
4. Choose **Add permission**, and then choose **Create inline policy**.
5. Select the JSON policy editor, and then copy and paste the following text into the editor.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Action": [
        "ssm:StartSession"
      ],
      "Resource": [
        "arn:aws:ssm:us-east-1::document/AWS-StartPortForwardingSession",
        "arn:aws:ec2:us-east-1:111122223333:instance/instance_id"
      ]
    }
  ]
}
```

6. Before saving the new policy, replace the following values in the policy text:
 - Replace `region` with the AWS Region where your farm is located
 - Replace `instance_id` with the instance ID for the license server or proxy instance you're using
 - Replace `account_id` with the AWS account number containing your farm
7. Choose **Next**.
8. For the Policy name, enter **LicenseForwarding**.
9. Choose **Create policy** to save your changes and create the policy with the required permissions.

To add a new queue environment to the queue

1. From the [Deadline Cloud console](#), choose **Go to Dashboard** if you haven't already.
2. From the dashboard, select the farm, and then the queue you want to configure.
3. Choose **Queue Environments > Actions > Create new with YAML**.
4. Copy and paste the following text into the YAML script editor.

Windows

```
specificationVersion: "environment-2023-09"
parameterDefinitions:
  - name: LicenseInstanceId
    type: STRING
    description: >
      The Instance ID of the license server/proxy instance
    default: ""
  - name: LicenseInstanceRegion
    type: STRING
    description: >
      The region containing this farm
    default: ""
  - name: LicensePorts
    type: STRING
    description: >
      Comma-separated list of ports to be forwarded to the license server/proxy
      instance. Example: "2701,2702,7075,2703,6101,1715,1716,1717,7054,7055,30304"
    default: "2701,2702,7075,2703,6101,1715,1716,1717,7054,7055,30304"
environment:
  name: BYOL License Forwarding
  variables:
    example_LICENSE: 2701@localhost
  script:
    actions:
      onEnter:
        command: bash
        args: [ "{{Env.File.Enter}}" ]
      onExit:
        command: bash
        args: [ "{{Env.File.Exit}}" ]
    embeddedFiles:
      - name: Enter
        type: TEXT
        runnable: True
        data: |
          curl "https://s3.amazonaws.com/session-manager-downloads/plugin/latest/
          windows/SessionManagerPlugin.zip" -o "{{Session.WorkingDirectory}}/ssm-
          plugin.zip"
          powershell -Command "Expand-Archive -Path '{{Session.WorkingDirectory}}/
          ssm-plugin.zip' -DestinationPath '{{Session.WorkingDirectory}}/ssm-plugin'
```

```

-Force; Expand-Archive -Path '{{Session.WorkingDirectory}}/ssm-plugin/
package.zip' -DestinationPath '{{Session.WorkingDirectory}}/ssm-plugin/package'
-Force"
    conda activate
    python "{{Env.File.StartSession}}" "{{Session.WorkingDirectory}}/ssm-
plugin/package/bin/session-manager-plugin.exe"
- name: Exit
  type: TEXT
  runnable: True
  data: |
    echo Killing SSM Manager Plugin PIDs: $BYOL_SSM_PIDS
    for pid in ${BYOL_SSM_PIDS//,/ }; do kill $pid; done
- name: StartSession
  type: TEXT
  data: |
    import boto3
    import json
    import subprocess
    import sys
    import os
    import tempfile

    instance_id = "{{Param.LicenseInstanceId}}"
    region = "{{Param.LicenseInstanceRegion}}"
    license_ports_list = "{{Param.LicensePorts}}".split(",")

    ssm_client = boto3.client("ssm", region_name=region)
    pids = []

    for port in license_ports_list:
        session_response = ssm_client.start_session(
            Target=instance_id,
            DocumentName="AWS-StartPortForwardingSession",
            Parameters={"portNumber": [port], "localPortNumber": [port]}
        )

    cmd = [
        sys.argv[1],
        json.dumps(session_response),
        region,
        "StartSession",
        "",
        json.dumps({"Target": instance_id}),
        f"https://{ssm.{region}.amazonaws.com"

```

```

]

    process = subprocess.Popen(cmd, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
    pids.append(process.pid)
    print(f"SSM Port Forwarding Session started for port {port}")

    print(f"openjd_env: BYOL_SSM_PIDS='{','.join(str(pid) for pid in pids)}")

    # Enabling UBL after the BYOL has run out requires prepending the BYOL
configuration to the existing license setup
    # Remove the sections that do not apply to your pipeline, or you do not
want to use UBL after exhausting the BYOL licenses.
    # The port numbers used may not match what your license server is serving.

    # Arnold
    os.environ["ADSKFLEX_LICENSE_FILE"] = f"2701@localhost;
{os.environ.get('ADSKFLEX_LICENSE_FILE', '')}"
    print(f"openjd_env:
ADSKFLEX_LICENSE_FILE={os.environ['ADSKFLEX_LICENSE_FILE']}")

    # Cinema4D
    os.environ["g_licenseServerRLM"] = f"localhost:7057;
{os.environ.get('g_licenseServerRLM', '')}"
    print(f"openjd_env:
g_licenseServerRLM={os.environ['g_licenseServerRLM']}")

    # Nuke
    os.environ["foundry_LICENSE"] = f"6101@localhost;
{os.environ.get('foundry_LICENSE', '')}"
    print(f"openjd_env: foundry_LICENSE={os.environ['foundry_LICENSE']}")

    # SideFX
    os.environ["SESI_LMHOST"] = f"localhost:1715;
{os.environ.get('SESI_LMHOST', '')}"
    print(f"openjd_env: SESI_LMHOST={os.environ['SESI_LMHOST']}")

    # Redshift and Red Giant
    os.environ["redshift_LICENSE"] = f"7054@localhost;7055@localhost;
{os.environ.get('redshift_LICENSE', '')}"
    print(f"openjd_env: redshift_LICENSE={os.environ['redshift_LICENSE']}")

    # V-Ray doesn't support multiple license servers in a single environment
variable

```

```

# See https://documentation.chaos.com/space/LIC5/125050770/Sharing+a
+License+Configuration+in+a+Network
vray_license = os.environ.get('VRAY_AUTH_CLIENT_SETTINGS', '')
xml_content = """<VRLClient>
    <LicServer>
        <Host>localhost</Host>
        <Port>30304</Port>"""

if vray_license and vray_license.startswith('licset://'):
    server_parts = vray_license.removeprefix('licset://').split(':')
    if len(server_parts) >= 2:
        xml_content += f"""
        <Host1>{server_parts[0]}</Host1>
        <Port1>{server_parts[1]}</Port1>"""

xml_content += """
    <User></User>
    <Pass></Pass>
</LicServer>
</VRLClient>"""

temp_dir = tempfile.gettempdir()
xml_path = os.path.join(temp_dir, 'vrlclient.xml')

with open(xml_path, 'w') as f:
    f.write(xml_content)

os.environ["VRAY_AUTH_CLIENT_FILE_PATH"] = temp_dir
print(f"openjd_env:
VRAY_AUTH_CLIENT_FILE_PATH={os.environ['VRAY_AUTH_CLIENT_FILE_PATH']}")

# Clear the existing VRAY_AUTH_CLIENT_SETTINGS so only the vrlclient.xml
file is used.
os.environ["VRAY_AUTH_CLIENT_SETTINGS"] = ''
print(f"openjd_env:
VRAY_AUTH_CLIENT_SETTINGS={os.environ['VRAY_AUTH_CLIENT_SETTINGS']}")

# Print out the created xml file's contents
print(f"V-Ray configuration file: {xml_path}")
with open(xml_path, 'r') as f:
    print(f"{f.read()}")

```


Linux

```
specificationVersion: "environment-2023-09"
parameterDefinitions:
  - name: LicenseInstanceId
    type: STRING
    description: >
      The Instance ID of the license server/proxy instance
    default: ""
  - name: LicenseInstanceRegion
    type: STRING
    description: >
      The region containing this farm
    default: ""
  - name: LicensePorts
    type: STRING
    description: >
      Comma-separated list of ports to be forwarded to the license server/proxy
      instance. Example: "2701,2702,7075,2703,6101,1715,1716,1717,7054,7055,30304"
    default: "2701,2702,7075,2703,6101,1715,1716,1717,7054,7055,30304"
environment:
  name: BYOL License Forwarding
  variables:
    example_LICENSE: 2701@localhost
  script:
    actions:
      onEnter:
        command: bash
        args: [ "{{Env.File.Enter}}" ]
      onExit:
        command: bash
        args: [ "{{Env.File.Exit}}" ]
    embeddedFiles:
      - name: Enter
        type: TEXT
        runnable: True
        data: |
          curl https://s3.amazonaws.com/session-manager-downloads/plugin/
latest/linux_64bit/session-manager-plugin.rpm -Ls | rpm2cpio - | cpio -iv
--to-stdout ./usr/local/sessionmanagerplugin/bin/session-manager-plugin >
{{Session.WorkingDirectory}}/session-manager-plugin
          chmod +x {{Session.WorkingDirectory}}/session-manager-plugin
```

```

conda activate
python {{Env.File.StartSession}} {{Session.WorkingDirectory}}/session-
manager-plugin
- name: Exit
  type: TEXT
  runnable: True
  data: |
    echo Killing SSM Manager Plugin PIDs: $BYOL_SSM_PIDS
    for pid in ${BYOL_SSM_PIDS//,/ }; do kill $pid; done
- name: StartSession
  type: TEXT
  data: |
    import boto3
    import json
    import subprocess
    import sys
    import os
    import tempfile

    instance_id = "{{Param.LicenseInstanceId}}"
    region = "{{Param.LicenseInstanceRegion}}"
    license_ports_list = "{{Param.LicensePorts}}".split(",")

    ssm_client = boto3.client("ssm", region_name=region)
    pids = []

    for port in license_ports_list:
        session_response = ssm_client.start_session(
            Target=instance_id,
            DocumentName="AWS-StartPortForwardingSession",
            Parameters={"portNumber": [port], "localPortNumber": [port]}
        )

        cmd = [
            sys.argv[1],
            json.dumps(session_response),
            region,
            "StartSession",
            "",
            json.dumps({"Target": instance_id}),
            f"https://ssm.{region}.amazonaws.com"
        ]

```

```

        process = subprocess.Popen(cmd, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
        pids.append(process.pid)
        print(f"SSM Port Forwarding Session started for port {port}")

    print(f"openjd_env: BYOL_SSM_PIDS='{','.join(str(pid) for pid in pids)}")

    # Enabling UBL after the BYOL has run out requires prepending the BYOL
configuration to the existing license setup
    # Remove the sections that do not apply to your pipeline, or you do not
want to use UBL after exhausting the BYOL licenses.
    # The port numbers used may not match what your license server is serving.

    # Arnold
    os.environ["ADSKFLEX_LICENSE_FILE"] = f"2701@localhost:
{os.environ.get('ADSKFLEX_LICENSE_FILE', '')}"
    print(f"openjd_env:
ADSKFLEX_LICENSE_FILE={os.environ['ADSKFLEX_LICENSE_FILE']}")

    # Nuke
    os.environ["foundry_LICENSE"] = f"6101@localhost:
{os.environ.get('foundry_LICENSE', '')}"
    print(f"openjd_env: foundry_LICENSE={os.environ['foundry_LICENSE']}")

    # SideFX
    os.environ["SESI_LMHOST"] = f"localhost:1715;
{os.environ.get('SESI_LMHOST', '')}"
    print(f"openjd_env: SESI_LMHOST={os.environ['SESI_LMHOST']}")

    # Redshift and Red Giant
    os.environ["redshift_LICENSE"] = f"7054@localhost:7055@localhost:
{os.environ.get('redshift_LICENSE', '')}"
    print(f"openjd_env: redshift_LICENSE={os.environ['redshift_LICENSE']}")

    # V-Ray doesn't support multiple license servers in a single environment
variable
    # See https://documentation.chaos.com/space/LIC5/125050770/Sharing+a
+License+Configuration+in+a+Network
    vray_license = os.environ.get('VRAY_AUTH_CLIENT_SETTINGS', '')
    xml_content = """<VRLClient>
    <LicServer>
        <Host>localhost</Host>
        <Port>30304</Port>"""

```

```

if vray_license and vray_license.startswith('licset://'):
    server_parts = vray_license.removeprefix('licset://').split(':')
    if len(server_parts) >= 2:
        xml_content += f"""
        <Host1>{server_parts[0]}</Host1>
        <Port1>{server_parts[1]}</Port1>"""

xml_content += """
    <User></User>
    <Pass></Pass>
  </LicServer>
</VRLClient>"""

temp_dir = tempfile.gettempdir()
xml_path = os.path.join(temp_dir, 'vrlclient.xml')

with open(xml_path, 'w') as f:
    f.write(xml_content)

os.environ["VRAY_AUTH_CLIENT_FILE_PATH"] = temp_dir
print(f"openjd_env:
VRAY_AUTH_CLIENT_FILE_PATH={os.environ['VRAY_AUTH_CLIENT_FILE_PATH']}")

# Clear the existing VRAY_AUTH_CLIENT_SETTINGS so only the vrlclient.xml
file is used.
os.environ["VRAY_AUTH_CLIENT_SETTINGS"] = ''
print(f"openjd_env:
VRAY_AUTH_CLIENT_SETTINGS={os.environ['VRAY_AUTH_CLIENT_SETTINGS']}")

# Print out the created xml file's contents
print(f"V-Ray configuration file: {xml_path}")
with open(xml_path, 'r') as f:
    print(f"{f.read()}")

```

5. Before saving the queue environment, make the following changes to the environment text as needed:

- Update the default values for the following parameters to reflect your environment:
 - **LicenseInstanceID** – The Amazon EC2 instance ID of your license server or proxy instance
 - **LicenseInstanceRegion** – The AWS Region containing your farm

- **LicensePorts** – A comma-separated list of ports to be forwarded to the license server or proxy instance (for example 2700,2701)
- If you want to use usage based licensing (UBL) after Bring your own license (BYOL) is exhausted be sure the port is correct for your license server. If you do not want to use UBL after running out of BYOL, add any required licensing environment variables to the variables section.

These variables should direct the DCCs to localhost on the license server port. For example, if your Foundry license server is listening on port 6101, you would add the variable as **foundry_LICENSE: 6101@localhost**.

6. (Optional) You can leave **Priority** set to **0**, or you can change it to order the priority differently among multiple queue environments.
7. Choose **Create queue environment** to save the new environment.

With the queue environment set, jobs submitted to this queue will retrieve licenses from the configured license server.

Step 2: (Optional) License proxy instance setup

As an alternative to using a license server, you can use a license proxy. To create a license proxy, create a new Amazon Linux 2023 instance that has network access to the license server. If needed, you can configure this access using a VPN connection. For more information, see [VPN connections](#) in the *Amazon VPC User Guide*.

To set up a license proxy instance for Deadline Cloud, follow the steps in this procedure. Perform the following configuration steps on this new instance to enable forwarding of license traffic to your license server

1. To install the HAProxy package, enter

```
sudo yum install haproxy
```

2. Update the listen license-server section of the **/etc/haproxy/haproxy.cfg** configuration file with the following:
 - a. Replace **LicensePort1** and **LicensePort2** with the port numbers to be forwarded to the license server. Add or remove comma-separated values to accommodate the required number of ports.

- b. Replace **LicenseServerHost** with the host name or IP address of the license server.

```
global
    log          127.0.0.1 local2
    chroot       /var/lib/haproxy
    user         haproxy
    group        haproxy
    daemon

defaults
    timeout queue      1m
    timeout connect    10s
    timeout client     1m
    timeout server     1m
    timeout http-keep-alive 10s
    timeout check      10s

listen license-server
    bind *:LicensePort1, *:LicensePort2
    server license-server LicenseServerHost
```

3. To enable and start the HAProxy service, run the following commands:

```
sudo systemctl enable haproxy
sudo service haproxy start
```

After completing the steps, license requests sent to localhost from the forwarding queue environment should be forwarded to the specified license server.

Step 3: CloudFormation template setup

You can use a CloudFormation template to configure an entire farm to use your own licensing.

1. Modify the template provided in the next step to add any required licensing environment variables to the **variables** section under **BYOLQueueEnvironment**.
2. Use the following CloudFormation template.

```
AWSTemplateFormatVersion: 2010-09-09
Description: "Create Deadline Cloud resources for BYOL"
```

Parameters:**LicenseInstanceId:**

Type: AWS::EC2::Instance::Id

Description: Instance ID for the license server/proxy instance

LicensePorts:

Type: String

Description: Comma-separated list of ports to forward to the license instance

Resources:**JobAttachmentBucket:**

Type: AWS::S3::Bucket

Properties:

BucketName: !Sub byol-example-ja-bucket-\${AWS::AccountId}-\${AWS::Region}

BucketEncryption:

ServerSideEncryptionConfiguration:

- ServerSideEncryptionByDefault:

SSEAlgorithm: AES256

Farm:

Type: AWS::Deadline::Farm

Properties:

DisplayName: BYOLFarm

QueuePolicy:

Type: AWS::IAM::ManagedPolicy

Properties:

ManagedPolicyName: BYOLQueuePolicy

PolicyDocument:

Version: 2012-10-17

Statement:

- Effect: Allow

Action:

- s3:GetObject

- s3:PutObject

- s3:ListBucket

- s3:GetBucketLocation

Resource:

- !Sub \${JobAttachmentBucket.Arn}

- !Sub \${JobAttachmentBucket.Arn}/job-attachments/*

Condition:**StringEquals:**

aws:ResourceAccount: !Sub \${AWS::AccountId}

- Effect: Allow

```

        Action: logs:GetLogEvents
        Resource: !Sub arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-
group:/aws/deadline/${Farm.FarmId}/*
        - Effect: Allow
        Action:
            - s3:ListBucket
            - s3:GetObject
        Resource:
            - "*"
        Condition:
            ArnLike:
                s3:DataAccessPointArn:
                    - arn:aws:s3:*:*:accesspoint/deadline-software-*
            StringEquals:
                s3:AccessPointNetworkOrigin: VPC

```

BYOLSSMPolicy:

```

Type: AWS::IAM::ManagedPolicy
Properties:
    ManagedPolicyName: BYOLSSMPolicy
    PolicyDocument:
        Version: 2012-10-17
        Statement:
            - Effect: Allow
            Action:
                - ssm:StartSession
            Resource:
                - !Sub arn:aws:ssm:${AWS::Region}::document/AWS-
StartPortForwardingSession
                - !Sub arn:aws:ec2:${AWS::Region}:${AWS::AccountId}:instance/
${LicenseInstanceId}

```

WorkerPolicy:

```

Type: AWS::IAM::ManagedPolicy
Properties:
    ManagedPolicyName: BYOLWorkerPolicy
    PolicyDocument:
        Version: 2012-10-17
        Statement:
            - Effect: Allow
            Action:
                - logs:CreateLogStream

```



```

    Resource: !Sub arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-
group:/aws/deadline/${Farm.FarmId}/*
    Condition:
      ForAnyValue:StringEquals:
        aws:CalledVia:
          - deadline.amazonaws.com
    - Effect: Allow
    Action:
      - logs:PutLogEvents
      - logs:GetLogEvents
    Resource: !Sub arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-
group:/aws/deadline/${Farm.FarmId}/*

```

QueueRole:

```

Type: AWS::IAM::Role
Properties:
  RoleName: BYOLQueueRole
  ManagedPolicyArns:
    - !Ref QueuePolicy
    - !Ref BYOLSSMPolicy
  AssumeRolePolicyDocument:
    Version: 2012-10-17
    Statement:
      - Effect: Allow
        Action:
          - sts:AssumeRole
        Principal:
          Service:
            - credentials.deadline.amazonaws.com
            - deadline.amazonaws.com
        Condition:
          StringEquals:
            aws:SourceAccount: !Sub ${AWS::AccountId}
          ArnEquals:
            aws:SourceArn: !Ref Farm

```

WorkerRole:

```

Type: AWS::IAM::Role
Properties:
  RoleName: BYOLWorkerRole
  ManagedPolicyArns:
    - arn:aws:iam::aws:policy/AWSDeadlineCloud-FleetWorker
    - !Ref WorkerPolicy

```

```
AssumeRolePolicyDocument:
  Version: 2012-10-17
  Statement:
    - Effect: Allow
      Action:
        - sts:AssumeRole
      Principal:
        Service: credentials.deadline.amazonaws.com
```

```
Queue:
  Type: AWS::Deadline::Queue
  Properties:
    DisplayName: BYOLQueue
    FarmId: !GetAtt Farm.FarmId
    RoleArn: !GetAtt QueueRole.Arn
    JobRunAsUser:
      Posix:
        Group: ""
        User: ""
      RunAs: WORKER_AGENT_USER
    JobAttachmentSettings:
      RootPrefix: job-attachments
      S3BucketName: !Ref JobAttachmentBucket
```

```
Fleet:
  Type: AWS::Deadline::Fleet
  Properties:
    DisplayName: BYOLFleet
    FarmId: !GetAtt Farm.FarmId
    MinWorkerCount: 1
    MaxWorkerCount: 2
    Configuration:
      ServiceManagedEc2:
        InstanceCapabilities:
          VCpuCount:
            Min: 4
            Max: 16
          MemoryMiB:
            Min: 4096
            Max: 16384
          OsFamily: LINUX
          CpuArchitectureType: x86_64
        InstanceMarketOptions:
```

```
Type: on-demand
RoleArn: !GetAtt WorkerRole.Arn
```

QFA:

```
Type: AWS::Deadline::QueueFleetAssociation
Properties:
  FarmId: !GetAtt Farm.FarmId
  FleetId: !GetAtt Fleet.FleetId
  QueueId: !GetAtt Queue.QueueId
```

CondaQueueEnvironment:

```
Type: AWS::Deadline::QueueEnvironment
Properties:
```

```
  FarmId: !GetAtt Farm.FarmId
  Priority: 5
  QueueId: !GetAtt Queue.QueueId
  TemplateType: YAML
  Template: |
```

```
    specificationVersion: 'environment-2023-09'
```

```
    parameterDefinitions:
```

```
      - name: CondaPackages
```

```
        type: STRING
```

```
        description: >
```

This is a space-separated list of conda package match specifications to install for the job.

E.g. "blender=3.6" for a job that renders frames in Blender 3.6.

See <https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/pkg-specs.html#package-match-specifications>

```
    default: ""
```

```
    userInterface:
```

```
      control: LINE_EDIT
```

```
      label: Conda Packages
```

```
      - name: CondaChannels
```

```
        type: STRING
```

```
        description: >
```

This is a space-separated list of conda channels from which to install packages. Deadline Cloud SMF packages are

installed from the "deadline-cloud" channel that is configured by Deadline Cloud.

Add "conda-forge" to get packages from the <https://conda-forge.org/> community, and "defaults" to get packages

```

        from Anaconda Inc (make sure your usage complies with https://
www.anaconda.com/terms-of-use).
        default: "deadline-cloud"
        userInterface:
            control: LINE_EDIT
            label: Conda Channels
    environment:
        name: Conda
        script:
            actions:
                onEnter:
                    command: "conda-queue-env-enter"
                    args: ["{{Session.WorkingDirectory}}/.env", "--packages",
"{{Param.CondaPackages}}", "--channels", "{{Param.CondaChannels}}"]
                onExit:
                    command: "conda-queue-env-exit"

BYOLQueueEnvironment:
    Type: AWS::Deadline::QueueEnvironment
    Properties:
        FarmId: !GetAtt Farm.FarmId
        Priority: 10
        QueueId: !GetAtt Queue.QueueId
        TemplateType: YAML
        Template: |
            specificationVersion: "environment-2023-09"
            parameterDefinitions:
                - name: LicenseInstanceId
                  type: STRING
                  description: >
                    The Instance ID of the license server/proxy instance
                  default: ""
                - name: LicenseInstanceRegion
                  type: STRING
                  description: >
                    The region containing this farm
                  default: ""
                - name: LicensePorts
                  type: STRING
                  description: >
                    Comma-separated list of ports to be forwarded to the license server/
proxy
                    instance. Example:
                    "2701,2702,7075,2703,6101,1715,1716,1717,7054,7055,30304"

```

```

    default: "2701,2702,7075,2703,6101,1715,1716,1717,7054,7055,30304"
environment:
  name: BYOL License Forwarding
  variables:
    example_LICENSE: 2701@localhost
  script:
    actions:
      onEnter:
        command: bash
        args: [ "{{Env.File.Enter}}" ]
      onExit:
        command: bash
        args: [ "{{Env.File.Exit}}" ]
    embeddedFiles:
      - name: Enter
        type: TEXT
        runnable: True
        data: |
          curl https://s3.amazonaws.com/session-manager-downloads/plugin/
latest/linux_64bit/session-manager-plugin.rpm -Ls | rpm2cpio - | cpio -iv
--to-stdout ./usr/local/sessionmanagerplugin/bin/session-manager-plugin >
{{Session.WorkingDirectory}}/session-manager-plugin
          chmod +x {{Session.WorkingDirectory}}/session-manager-plugin
          conda activate
          python {{Env.File.StartSession}} {{Session.WorkingDirectory}}/
session-manager-plugin
      - name: Exit
        type: TEXT
        runnable: True
        data: |
          echo Killing SSM Manager Plugin PIDs: $BYOL_SSM_PIDS
          for pid in ${BYOL_SSM_PIDS//,/ }; do kill $pid; done
      - name: StartSession
        type: TEXT
        data: |
          import boto3
          import json
          import subprocess
          import sys
          import os
          import tempfile

          instance_id = "{{Param.LicenseInstanceId}}"
          region = "{{Param.LicenseInstanceRegion}}"

```

```

license_ports_list = "{{Param.LicensePorts}}".split(",")

ssm_client = boto3.client("ssm", region_name=region)
pids = []

for port in license_ports_list:
    session_response = ssm_client.start_session(
        Target=instance_id,
        DocumentName="AWS-StartPortForwardingSession",
        Parameters={"portNumber": [port], "localPortNumber": [port]}
    )

    cmd = [
        sys.argv[1],
        json.dumps(session_response),
        region,
        "StartSession",
        "",
        json.dumps({"Target": instance_id}),
        f"https://ssm.{region}.amazonaws.com"
    ]

    process = subprocess.Popen(cmd, stdout=subprocess.DEVNULL,
stderr=subprocess.DEVNULL)
    pids.append(process.pid)
    print(f"SSM Port Forwarding Session started for port {port}")

print(f"openjd_env: BYOL_SSM_PIDS='{','.join(str(pid) for pid in
pids)}")

# Enabling UBL after the "bring your own license" (BYOL) has run
out requires prepending the BYOL configuration to the existing license setup
# Remove the sections that do not apply to your pipeline, or you do
not want to use UBL after exhausting the BYOL licenses.
# The port numbers used may not match what your license server is
serving.

# Arnold
os.environ["ADSKFLEX_LICENSE_FILE"] = f"2701@localhost:
{os.environ.get('ADSKFLEX_LICENSE_FILE', '')}"
print(f"openjd_env:
ADSKFLEX_LICENSE_FILE={os.environ['ADSKFLEX_LICENSE_FILE']}")

# Nuke

```

```

        os.environ["foundry_LICENSE"] = f"6101@localhost:
{os.environ.get('foundry_LICENSE', '')}"
        print(f"openjd_env:
foundry_LICENSE={os.environ['foundry_LICENSE']}")

        # SideFX
        os.environ["SESI_LMHOST"] = f"localhost:1715;
{os.environ.get('SESI_LMHOST', '')}"
        print(f"openjd_env: SESI_LMHOST={os.environ['SESI_LMHOST']}")

        # Redshift and Red Giant
        os.environ["redshift_LICENSE"] = f"7054@localhost:7055@localhost:
{os.environ.get('redshift_LICENSE', '')}"
        print(f"openjd_env:
redshift_LICENSE={os.environ['redshift_LICENSE']}")

        # V-Ray doesn't support multiple license servers in a single
        environment variable
        # See https://documentation.chaos.com/space/LIC5/125050770/Sharing
+a+License+Configuration+in+a+Network
        vray_license = os.environ.get('VRAY_AUTH_CLIENT_SETTINGS', '')
        xml_content = """<VRLClient>
        <LicServer>
        <Host>localhost</Host>
        <Port>30304</Port>"""

        if vray_license and vray_license.startswith('licset://'):
            server_parts =
vray_license.removeprefix('licset://').split(':')
            if len(server_parts) >= 2:
                xml_content += f"""
                <Host1>{server_parts[0]}</Host1>
                <Port1>{server_parts[1]}</Port1>"""

            xml_content += """
            <User></User>
            <Pass></Pass>
            </LicServer>
            </VRLClient>"""

        temp_dir = tempfile.gettempdir()
        xml_path = os.path.join(temp_dir, 'vrlclient.xml')

        with open(xml_path, 'w') as f:

```

```
f.write(xml_content)

os.environ["VRAY_AUTH_CLIENT_FILE_PATH"] = temp_dir
print(f"openjd_env:
VRAY_AUTH_CLIENT_FILE_PATH={os.environ['VRAY_AUTH_CLIENT_FILE_PATH']}")

# Clear the existing VRAY_AUTH_CLIENT_SETTINGS so only the
vrlclient.xml file is used.
os.environ["VRAY_AUTH_CLIENT_SETTINGS"] = ''
print(f"openjd_env:
VRAY_AUTH_CLIENT_SETTINGS={os.environ['VRAY_AUTH_CLIENT_SETTINGS']}")

# Print out the created xml file's contents
print(f"V-Ray configuration file: {xml_path}")
with open(xml_path, 'r') as f:
    print(f"{f.read()}")
```

3. When deploying the CloudFormation template, provide the following parameters:
 - Update the **LicenseInstanceID** with the Amazon EC2 Instance ID of your license server or proxy instance
 - Update the **LicensePorts** with a comma-separated list of ports to be forwarded to the license server or proxy instance (for example 2700,2701)
 - Add the license environment variables by replacing **example_LICENSE: 2700@localhost** in the template
4. Deploy the template to setup your farm with bring your own license capability.

Connect customer-managed fleets to a license endpoint

The AWS Deadline Cloud usage-based license server provides on-demand licenses for select third-party products. With usage-based licenses, you can pay as you go. You are only charged for the time you use. Usage-based licensing provides licenses for your Deadline Cloud workers to render, it doesn't provide licenses for your DCC applications.

Some usage-based license products are available on both service-managed and customer-managed fleets, while others are only available on service-managed fleets. The license server is automatically set up in service-managed fleets. The following setup is only needed for customer-managed fleets and covers only the products supported on that fleet type.

To create the license server, you need a security group for your farm's VPC that allows traffic for third-party licenses.

Topics

- [Step 1: Create a security group](#)
- [Step 2: Set up the license endpoint](#)
- [Step 3: Connect a rendering application to an endpoint](#)
- [Step 4: Delete a license endpoint](#)

Step 1: Create a security group

Use the [Amazon VPC Console](#) to create a security group for your farm's VPC. Configure the security group to allow the following inbound rules:

Note

The following applications are the only applications supported with usage-based licensing on customer-managed fleets.

- Arnold – 2701, TCP, IPv4, IPv6
- Cinema 4D – 7057, TCP, IPv4, IPv6
- Foundry Nuke – 6101, TCP, IPv4, IPv6
- Red Giant – 7055, TCP, IPV4
- Redshift – 7054, TCP, IPv4, IPv6
- SideFX Houdini, Mantra, and Karma – 1715 - 1717, TCP, IPv4, IPv6
- V-Ray – 30304, TCP, IPV4

The source for each inbound rule is the fleet's worker security group.

For more information about creating a security group, see [Create a security group](#) in the *Amazon Virtual Private Cloud user guide*.

Step 2: Set up the license endpoint

A *license endpoint* provides access to license servers for third-party products. License requests are sent to the license endpoint. The endpoint routes them to the appropriate license server. The license server tracks usage limits and entitlements. Creating a license endpoint in Deadline Cloud provisions an AWS PrivateLink interface endpoint in your VPC. These endpoints are billed according to standard AWS PrivateLink pricing. For more information, see [AWS PrivateLink pricing](#).

With the appropriate permissions, you can create your license endpoint. For the required policy to create a license endpoint, see [Policy to allow creating a license endpoint](#).

You can create your license endpoint from your dashboard in the Deadline Cloud [console](#).

1. From the left navigation pane, choose **License endpoints**, then choose **Create license endpoint**.
2. From the Create license endpoint page, complete the following:
 - Select a VPC.
 - Select the subnets that contain your Deadline Cloud workers. You can select up to 10 subnets.
 - Select the security group you created in step 1. You can select up to 10 security groups for more complicated scenarios.
 - (Optional) Choose **Add new tag** and add one or more tags. You can add up to 50 tags.
3. Choose **Create license endpoint**. When the license endpoint creates, it displays on the license endpoints page.
4. From the metered products section, choose **Add products**, and then select the products you want to add to your license endpoint. Choose **Add**.

To remove a product from a license endpoint, in the metered products section, select the product and then choose **Remove**. In the confirmation, choose **Remove** again.

Step 3: Connect a rendering application to an endpoint

After the license endpoint is set up, applications use it the same as they use a third-party license server. You typically configure the license server for the application by setting an environment variable or other system setting, such as a Microsoft Windows registry key, to a license server port and address.

To get the license endpoint DNS name, select the license endpoint in the console and then choose the copy icon in the DNS Name section.

Configuration examples

Example– Arnold

Set the environment variable `ADSKFLEX_LICENSE_FILE` to:

```
2701@VPC_Endpoint_DNS_Name
```

Example– Cinema 4D

Set the environment variable `g_licenseServerRLM` to:

```
VPC_Endpoint_DNS_Name:7057
```

After you create the environment variable, you should be able to render a an image using a command line similar to this one:

```
"C:\Program Files\Maxon Cinema 4D 2025\Commandline.exe" -render ^  
  "C:\Users\User\MyC4DFileWithRedshift.c4d" -frame 0 ^  
  -oimage "C:\Users\Administrator\User\MyOutputImage.png"
```

Example– Foundry Nuke

Set the environment variable `foundry_LICENSE` to:

```
6101@VPC_Endpoint_DNS_Name
```

To test that licensing is working properly, you can run Nuke in a terminal:

```
~/nuke/Nuke14.0v5/Nuke14.0 -x
```

Example– Red Giant

Set the environment variable `redshift_LICENSE` to:

```
7055@VPC_Endpoint_DNS_Name
```

Note that Red Giant and Redshift have the same `redshift_LICENSE` environment variable. If you want to use both applications, you can set the environment variable to:

```
7054@VPC_Endpoint_DNS_Name:7055@VPC_Endpoint_DNS_Name
```

Note

For Windows workers, use a semi-colon (;) instead of a colon (:) to separate endpoints.

To test that licensing is working properly, ensure you have After Effects and Red Giant installed. Then, you can render a project using a command similar to this one:

```
C:\Program Files\Adobe\Adobe After Effects 2025\Support Files\aerender.exe -comp "Comp 1" -project  
C:\Users\MyUser\myAfterEffectsProjectUsingRedGiant.aep -output  
C:\Users\MyUser\myMovieWithRedGiant.mp4
```

Example– Redshift

Set the environment variable `redshift_LICENSE` to:

```
7054@VPC_Endpoint_DNS_Name
```

After you create the environment variable, you should be able to render an image using a command line similar to this one:

```
C:\ProgramData\redshift\bin\redshiftCmdLine.exe ^  
C:\demo\proxy\RS_Proxy_Demo.rs ^  
-oip C:\demo\proxy\images
```

Example– SideFX Houdini, Mantra, and Karma

Run the following command:

```
/opt/hfs19.5.640/bin/hserver -S  
"http://VPC_Endpoint_DNS_Name:1715;http://VPC_Endpoint_DNS_Name:1716;http://  
VPC_Endpoint_DNS_Name:1717;"
```

To test that licensing is working properly, you can render a Houdini scene via this command:

```
/opt/hfs19.5.640/bin/hython ~/forpentest.hip -c "hou.node('/out/mantra1').render()"
```

Example– V-Ray

Set the environment variable VRAY_AUTH_CLIENT_SETTINGS to:

```
licset://VPC_Endpoint_DNS_Name:30304
```

Set the environment variable VRAY_AUTH_CLIENT_FILE_PATH to:

```
/null
```

To test that licensing is working properly, you can render an image in V-Ray using a command similar to this one:

```
/usr/Chaos/V-Ray/bin/vray -sceneFile=/root/my_scene.vrscene -display=0
```

Step 4: Delete a license endpoint

When deleting your customer-managed fleet, remember to delete your license endpoint. If you don't delete the license endpoint, you will continue to be charged for AWS PrivateLink fixed costs.

You can delete your license endpoint from your dashboard in the Deadline Cloud [console](#).

1. From the left navigation pane, choose **License endpoints**.
2. Select the endpoint you want to delete and choose **delete**, then choose **delete** again to confirm.

Combining BYOL and UBL

You can combine BYOL and UBL so that your workers use your existing licenses first and automatically fall back to Deadline Cloud usage-based licenses when your BYOL licenses are exhausted. This approach is useful when you have a limited number of existing licenses but need to scale beyond that capacity during peak workloads.

How combined licensing works

When you configure combined licensing, the queue environment sets up license environment variables so that the BYOL license server is listed before the UBL license endpoint. Most third-party applications check license servers in the order they appear in the environment variable. When a worker requests a license, the application first contacts your BYOL license server. If no BYOL license is available, the application falls back to the UBL license endpoint.

The BYOL queue environment template provided in [Connect service-managed fleets to a custom license server](#) configures this fallback behavior automatically. The Python script in the queue environment prepends your BYOL license server address to the existing UBL license environment variables. To use combined licensing, keep the UBL sections in the queue environment script for the products where you want fallback behavior.

To use only BYOL without UBL fallback for a specific product, remove the UBL section for that product from the script and add the license environment variable directly to the `variables` section of the queue environment. For example, to use only BYOL for Cinema 4D, remove the Cinema 4D section from the script and add `g_licenseServerRLM: 127.0.0.1:7057` to the `variables` section.

Example: Using BYOL Cinema 4D licenses with UBL fallback

Consider a studio that has existing Cinema 4D licenses on a license server in their on-premises network. The studio wants to use those licenses for Deadline Cloud rendering, but also wants to scale beyond their license count by falling back to UBL when all BYOL licenses are in use.

To configure this setup, follow the steps in [Connect service-managed fleets to a custom license server](#) and make the following changes to the queue environment template:

To configure BYOL Cinema 4D licenses with UBL fallback

1. Set the `LicenseInstanceId` parameter to the Amazon Elastic Compute Cloud (Amazon EC2) instance ID of the license server or proxy that has access to the Cinema 4D license server.
2. Set the `LicensePorts` parameter to include port 7057 (the Cinema 4D RLM license port).
3. In the Python script, keep the Cinema 4D section that prepends the BYOL server to the UBL configuration:

```
# Cinema4D
```

```
os.environ["g_licenseServerRLM"] = f"127.0.0.1:7057;  
{os.environ.get('g_licenseServerRLM', '')}"  
print(f"openjd_env: g_licenseServerRLM={os.environ['g_licenseServerRLM']}")
```

This configuration sets `g_licenseServerRLM` to `127.0.0.1:7057`; *UBL_endpoint*:7057. Cinema 4D checks the BYOL server at `127.0.0.1:7057` first. If no license is available, Cinema 4D falls back to the UBL endpoint.

4. Remove the sections for products that you don't use (for example, Arnold, Nuke, or SideFX) to keep the configuration clean.

If you also have other products that use only BYOL without UBL fallback, add those license environment variables directly to the `variables` section of the queue environment and remove the corresponding sections from the Python script.

Considerations for combined licensing

Keep the following considerations in mind when you use combined licensing:

- Some applications don't support multiple license servers in a single environment variable. For example, V-Ray uses an XML configuration file instead. The queue environment template handles V-Ray configuration separately. For more information, see the V-Ray section in the queue environment template in [Connect service-managed fleets to a custom license server](#).
- The order of license servers in the environment variable determines the priority. List the BYOL server first so that your existing licenses are consumed before UBL licenses.
- On Windows workers, separate license server entries in environment variables with a semicolon (;) instead of a colon (:). For more information about configuring license environment variables, see [Connect customer-managed fleets to a license endpoint](#).
- To use UBL fallback with customer-managed fleets, set up a license endpoint. For more information, see [Connect customer-managed fleets to a license endpoint](#).

Using AI agents with Deadline Cloud

Use AI agents to write job bundles, develop conda packages, and troubleshoot jobs in Deadline Cloud. This topic explains what AI agents are, key points for working with them effectively, and resources to help agents understand Deadline Cloud.

An AI agent is a software tool that uses a large language model (LLM) to perform tasks autonomously. AI agents can read and write files, run commands, and iterate on solutions based on feedback. Examples include command-line tools like [Kiro](#) on the Kiro website and IDE-integrated assistants.

Key points for working with AI agents

The following key points help you get better results when you use AI agents with Deadline Cloud.

- **Provide grounding** – AI agents perform best when they have access to relevant documentation, specifications, and examples. You can provide grounding by pointing the agent to specific documentation pages, sharing existing example code as references, cloning relevant open source repositories into the local workspace, and providing documentation for third-party applications.
- **Specify success criteria** – Define the expected outcome and technical requirements for the agent. For example, when you ask an agent to develop a job bundle, specify the job inputs, parameters, and expected outputs. If you're unsure about the specifications, ask the agent to propose options first, then refine the requirements together.
- **Enable a feedback loop** – AI agents iterate more effectively when they can test their solutions and receive feedback. Instead of expecting a working solution on the first attempt, give the agent the ability to run its solution and review the results. This approach works well when the agent can access status updates, logs, and validation errors. For example, when you develop a job bundle, allow the agent to submit the job and review the logs.
- **Expect to iterate** – Even with good context, agents can get off track or make assumptions that don't match your environment. Observe how the agent approaches the task and provide guidance along the way. Add missing context if the agent struggles, help find errors by pointing to specific log files, refine requirements as you discover them, and add negative requirements to explicitly state what the agent should avoid.

Resources for agent context

The following resources on the GitHub website help AI agents understand Deadline Cloud concepts and produce accurate output.

- **Deadline Cloud Model Context Protocol (MCP) server** – For agents that support the Model Context Protocol, the [deadline-cloud](#) repository contains the Deadline Cloud client which includes an MCP server for interacting with jobs.
- **AWS Documentation MCP server** – For agents that support MCP, configure the [AWS Documentation MCP server](#) to give the agent direct access to AWS documentation, including the Deadline Cloud User Guide and Developer Guide.
- **Open Job Description specification** – The [Open Job Description specification](#) defines the schema for job templates. Reference the repository when agents need to understand the structure and syntax of job templates.
- **deadline-cloud-samples** – The [deadline-cloud-samples](#) repository contains sample job bundles, conda recipes, and CloudFormation templates for common applications and use cases.
- **aws-deadline GitHub organization** – The [aws-deadline](#) GitHub organization contains reference plugins for many third-party applications that you can use as examples for other integrations.

Example prompt: Writing a job bundle

The following example prompt demonstrates how to use an AI agent to create job bundles that train a LoRA (Low-Rank Adaptation) adapter for generating AI images. The prompt illustrates the key points discussed earlier: it provides grounding by pointing to relevant repositories, defines success criteria for the job bundle outputs, and outlines a feedback loop for iterative development.

Write a pair of job bundles for Deadline Cloud that use the diffusers Python library to train a LoRA adapter on a set of images and then generate images from it.

Requirements:

- The training job takes a set of JPEG images as input, uses an image description, LoRA rank, learning rate, batch size, and number of training steps as parameters, and outputs a `.safetensors` file.
- The generation job takes the `.safetensors` file as input and the number of images to generate, then outputs JPEG images. The jobs use Stable Diffusion 1.5 as the base model.
- The jobs run `diffusers` as a Python script. Install the necessary packages using conda by setting the job parameters:
 - `CondaChannels`: `conda-forge`
 - `CondaPackages`: list of conda packages to install

For context, clone the following repositories to your workspace and review their documentation and code:

- OpenJobDescription specification: <https://github.com/OpenJobDescription/openjd-specifications/blob/mainline/wiki/2023-09-Template-Schemas.md>
- Deadline Cloud sample job bundles: https://github.com/aws-deadline/deadline-cloud-samples/tree/mainline/job_bundles
- diffusers library: <https://github.com/huggingface/diffusers>

Read through the provided context before you start. To develop a job bundle, iterate with the following steps until the submitted job succeeds. If a step fails, update the job bundle and restart the loop:

1. Create a job bundle.
2. Validate the job template syntax: ``openjd check``
3. Submit the job to Deadline Cloud: ``deadline bundle submit``
4. Wait for the job to complete: ``deadline job wait``
5. View the job status and logs: ``deadline job logs``
6. Download the job output: ``deadline job download-output``

To verify the training and generation jobs work together, iterate with the following steps until the generation job produces images that resemble the dog in the training data:

1. Develop and submit a training job using the training images in ``./exdog``
2. Wait for the job to succeed then download its output.
3. Develop and submit a generation job using the LoRA adapter from the training job.
4. Wait for the job to succeed then download its output.
5. Inspect the generated images. If they resemble the dog in the training data, you're done. Otherwise, review the job template, job parameters, and job logs to identify and fix the issue.

Monitoring AWS Deadline Cloud

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS Deadline Cloud (Deadline Cloud) and your AWS solutions. Collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. Before you start monitoring Deadline Cloud, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- Which resources will you monitor?
- How often will you monitor these resources?
- Which monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

AWS and Deadline Cloud provide tools that you can use to monitor your resources and respond to potential incidents. Some of these tools do the monitoring for you, some of the tools require manual intervention. You should automate monitoring tasks as much as possible.

- *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](#).

Deadline Cloud has three CloudWatch metrics.

- *Amazon CloudWatch Logs* enables you to monitor, store, and access your log files from Amazon EC2 instances, CloudTrail, and other sources. CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see the [Amazon CloudWatch Logs User Guide](#).
- *Amazon EventBridge* can be used to automate your AWS services and respond automatically to system events, such as application availability issues or resource changes. Events from AWS services are delivered to EventBridge in near real time. You can write simple rules to indicate

which events are of interest to you and which automated actions to take when an event matches a rule. For more information, see [Amazon EventBridge 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](#).

Topics

- [Logging Deadline Cloud API calls using AWS CloudTrail](#)
- [Monitoring with CloudWatch](#)
- [Managing Deadline Cloud events using Amazon EventBridge](#)

Logging Deadline Cloud API calls using AWS CloudTrail

Deadline Cloud is integrated with [AWS CloudTrail](#), a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all API calls for Deadline Cloud as events. The calls captured include calls from the Deadline Cloud console and code calls to the Deadline Cloud API operations. Using the information collected by CloudTrail, you can determine the request that was made to Deadline Cloud, the IP address from which the request was made, when it was made, and additional details.

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 user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see [Working with CloudTrail Event history](#) in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a [CloudTrail Lake](#) event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see [Creating a trail for your AWS account](#) and [Creating a trail for an organization](#) in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For more information about CloudTrail pricing, see [AWS CloudTrail Pricing](#). For information about Amazon S3 pricing, see [Amazon S3 Pricing](#).

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to [Apache ORC](#) format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying [advanced event selectors](#). The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see [Working with AWS CloudTrail Lake](#) in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the [pricing option](#) you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see [AWS CloudTrail Pricing](#).

Deadline Cloud data events in CloudTrail

[Data events](#) provide information about the resource operations performed on or in a resource (for example, reading or writing to an Amazon S3 object). These operations are also known as data

plane operations. Data events are often high-volume activities. By default, CloudTrail doesn't log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see [AWS CloudTrail Pricing](#).

You can log data events for the Deadline Cloud resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations. For more information about how to log data events, see [Logging data events with the AWS Management Console](#) and [Logging data events with the AWS Command Line Interface](#) in the *AWS CloudTrail User Guide*.

The following table lists the Deadline Cloud resource types for which you can log data events. The **Data event type (console)** column shows the value to choose from the **Data event type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
Deadline Fleet	AWS::Deadline::Fleet	• SearchWorkers
Deadline Queue	AWS::Deadline::Fleet	• SearchJobs
Deadline Worker	AWS::Deadline::Worker	• GetWorker • ListSessionsForWorker • UpdateWorkerSchedule • BatchGetJobEntity • ListWorkers
Deadline Job	AWS::Deadline::Job	• ListStepConsumers • UpdateTask • ListJobs • GetStep • ListSteps • GetJob

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
		• GetTask • GetSession • ListSessions • CreateJob • ListSessionActions • ListTasks • CopyJobTemplate • UpdateSession • UpdateStep • UpdateJob • ListJobParameterDefinitions • GetSessionAction • ListStepDependencies • SearchTasks • SearchSteps

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.ARN` fields to log only those events that are important to you. For more information about these fields, see [AdvancedFieldSelector](#) in the *AWS CloudTrail API Reference*.

Deadline Cloud management events in CloudTrail

[Management events](#) provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

AWS Deadline Cloud logs the following Deadline Cloud control plane operations to CloudTrail as *management events*.

- [associate-member-to-farm](#)

- [associate-member-to-fleet](#)
- [associate-member-to-job](#)
- [associate-member-to-queue](#)
- [assume-fleet-role-for-read](#)
- [assume-fleet-role-for-worker](#)
- [assume-queue-role-for-read](#)
- [assume-queue-role-for-user](#)
- [assume-queue-role-for-worker](#)
- [create-budget](#)
- [create-farm](#)
- [create-fleet](#)
- [create-license-endpoint](#)
- [create-limit](#)
- [create-monitor](#)
- [create-queue](#)
- [create-queue-environment](#)
- [create-queue-fleet-association](#)
- [create-queue-limit-association](#)
- [create-storage-profile](#)
- [create-worker](#)
- [delete-budget](#)
- [delete-farm](#)
- [delete-fleet](#)
- [delete-license-endpoint](#)
- [delete-limit](#)
- [delete-metered-product](#)
- [delete-monitor](#)
- [delete-queue](#)
- [delete-queue-environment](#)
- [delete-queue-fleet-association](#)

- [delete-queue-limit-association](#)
- [delete-storage-profile](#)
- [delete-worker](#)
- [disassociate-member-from-farm](#)
- [disassociate-member-from-fleet](#)
- [disassociate-member-from-job](#)
- [disassociate-member-from-queue](#)
- [get-application-version](#)
- [get-budget](#)
- [get-farm](#)
- [get-feature-map](#)
- [get-fleet](#)
- [get-license-endpoint](#)
- [get-limit](#)
- [get-monitor](#)
- [get-queue](#)
- [get-queue-environment](#)
- [get-queue-fleet-association](#)
- [get-queue-limit-association](#)
- [get-sessions-statistics-aggregation](#)
- [get-storage-profile](#)
- [get-storage-profile-for-queue](#)
- [list-available-metered-products](#)
- [list-budgets](#)
- [list-farm-members](#)
- [list-farms](#)
- [list-fleet-members](#)
- [list-fleets](#)
- [list-job-members](#)
- [list-license-endpoints](#)

- [list-limit](#)
- [list-metered-products](#)
- [list-monitors](#)
- [list-queue-environments](#)
- [list-queue-fleet-associations](#)
- [list-queue-limit-associations](#)
- [list-queue-members](#)
- [list-queues](#)
- [list-storage-profiles](#)
- [list-storage-profiles-for-queue](#)
- [list-tags-for-resource](#)
- [put-metered-product](#)
- [start-sessions-statistics-aggregation](#)
- [tag-resource](#)
- [untag-resource](#)
- [update-budget](#)
- [update-farm](#)
- [update-fleet](#)
- [update-limit](#)
- [update-monitor](#)
- [update-queue](#)
- [update-queue-environment](#)
- [update-queue-fleet-association](#)
- [update-queue-limit-association](#)
- [update-storage-profile](#)
- [update-worker](#)

Deadline Cloud event examples

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log

files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows a CloudTrail event that demonstrates the CreateFarm operation.

```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE-PrincipalID:EXAMPLE-Session",
    "arn": "arn:aws:sts::111122223333:assumed-role/EXAMPLE-UserName/EXAMPLE-Session",
    "accountId": "111122223333",
    "accessKeyId": "EXAMPLE-accessKeyId",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "EXAMPLE-PrincipalID",
        "arn": "arn:aws:iam::111122223333:role/EXAMPLE-UserName",
        "accountId": "111122223333",
        "userName": "EXAMPLE-UserName"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2021-03-08T23:25:49Z"
      }
    }
  },
  "eventTime": "2021-03-08T23:25:49Z",
  "eventSource": "deadline.amazonaws.com",
  "eventName": "CreateFarm",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "192.0.2.0",
  "userAgent": "EXAMPLE-userAgent",
  "requestParameters": {
    "displayName": "example-farm",
    "kmsKeyArn": "arn:aws:kms:us-west-2:111122223333:key/111122223333",
    "X-Amz-Client-Token": "12abc12a-1234-1abc-123a-1a11bc1111a",
    "description": "example-description",
    "tags": {
      "purpose_1": "e2e",
      "purpose_2": "tag_test"
    }
  }
}
```

```
    },
    "responseElements": {
      "farmId": "EXAMPLE-farmID"
    },
    "requestID": "EXAMPLE-requestID",
    "eventID": "EXAMPLE-eventID",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
  }
```

The JSON example shows the AWS Region, IP address, and other "requestParameters" such as the "displayName" and "kmsKeyArn" that can help you identify the event.

For information about CloudTrail record contents, see [CloudTrail record contents](#) in the *AWS CloudTrail User Guide*.

Monitoring with CloudWatch

Amazon CloudWatch (CloudWatch) collects raw data and processes it into readable, near real-time metrics. You can open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/> to view and filter Deadline Cloud metrics.

These statistics are kept for 15 months so you can access historical information to 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](#).

Deadline Cloud has two kinds of logs – task logs and worker logs. A task log is when you run execution logs as a script or as DCC runs. A task log might show events such as assets loading, tiles rendering, or textures not being found.

A worker log shows worker agent processes. These might include things such as when the worker agents starts up, registers itself, reports progress, loads configurations, or completes tasks.

The namespace for these logs is `/aws/deadline/*`.

For Deadline Cloud, workers upload these logs to CloudWatch Logs. By default, logs never expire. If a job outputs a high volume of data, you can incur extra costs. For more information, see [Amazon CloudWatch pricing](#).

You can adjust the retention policy for each log group. A shorter retention removes old logs and can help reduce storage costs. To keep logs, you can archive them to Amazon Simple Storage Service before removing the log. For more information, see [Export log data to Amazon S3 using the console](#) in the *Amazon CloudWatch user guide*.

Note

CloudWatch log reads are limited by AWS. If you plan to onboard many artists, we suggest you contact AWS customer support and request an increase for the GetLogEvents quota in CloudWatch. Additionally, we recommend you close the log tailing portal when you are not debugging.

For more information, see [CloudWatch Logs quotas](#) in the *Amazon CloudWatch user guide*.

CloudWatch metrics

Deadline Cloud sends metrics to Amazon CloudWatch. You can use the AWS Management Console, the AWS CLI, or an API to list the metrics that Deadline Cloud sends to CloudWatch. By default, each data point covers the 1 minute that follows the start time of activity. For information about how to view the available metrics using the AWS Management Console or the AWS CLI, see [View available metrics](#) in the *Amazon CloudWatch User Guide*.

Customer-managed fleet metrics

The AWS/DeadlineCloud namespace contains the following metrics for your customer-managed fleets:

Metric	Description	Unit
RecommendedFleetSize	The number of workers that Deadline Cloud recommends that you use to process jobs. You can use this metric to expand or contract the	Count

Metric	Description	Unit
	number of workers from your fleet.	
UnhealthyWorkerCount	The number of workers assigned to process jobs that are not healthy.	Count

You can use the following dimensions to refine the customer-managed fleet metrics:

Dimension	Description
FarmId	This dimension filters the data that you request to the specified farm.
FleetId	This dimension filters the data that you request to the specified worker fleet.

Licensing metrics

The AWS/DeadlineCloud namespace contains the following metrics for licensing:

Metric	Description	Unit
LicensesInUse	The number of license sessions in use.	Count

You can use the following dimensions to refine the licensing metrics:

Dimension	Description
FleetId	Use this dimension to filter the data to the specified service-managed fleet. For customer-managed fleets, use the LicenseEndpointId dimension instead.

Dimension	Description
LicenseEndpointId	Use this dimension to filter the data to the specified license endpoint.
Product	Use this dimension to filter the data to the specified metered product.

Resource limit metrics

The AWS/DeadlineCloud namespace contains the following metrics for resource limits:

Metric	Description	Unit
CurrentCount	The number of resources modeled by this limit in use.	Count
MaxCount	The maximum number of resources modeled by this limit. If you set the maxCount value to -1 using the API, Deadline Cloud doesn't emit the MaxCount metric.	Count

You can use the following dimensions to refine the concurrent limit metrics:

Dimension	Description
FarmId	This dimension filters the data that you request to the specified farm.
LimitId	This dimension filters the data that you request to the specified limit.

Recommended alarms

With CloudWatch, you can create alarms that watch metrics and send you a notification or perform another action when a threshold is breached. For more information on configuring CloudWatch alarms, see the [Amazon CloudWatch User Guide](#).

We recommend that you set alarms for the following Deadline Cloud metrics:

LicensesInUse

Dimensions: FleetId, LicenseEndpointId

Alarm description: This alarm detects when the active license sessions for a service-managed fleet or license endpoint are approaching your account quota. If this error occurs, you can raise the account quota for license sessions. See your current quotas and request increases using Service Quotas. To learn more, see the [Service Quotas User Guide](#).

Intent: Prevent license checkout failures by monitoring usage before it reaches the quota limit.

Statistic: Maximum

Recommended threshold: 90% of your license session quota

Threshold justification: Set the threshold to a percentage of your quota, so that you can take action before it reaches the limit.

Period: 1 minute

Datapoints to alarm: 1

Evaluation periods: 1

Comparison Operator: GREATER_THAN_THRESHOLD

Additional resources

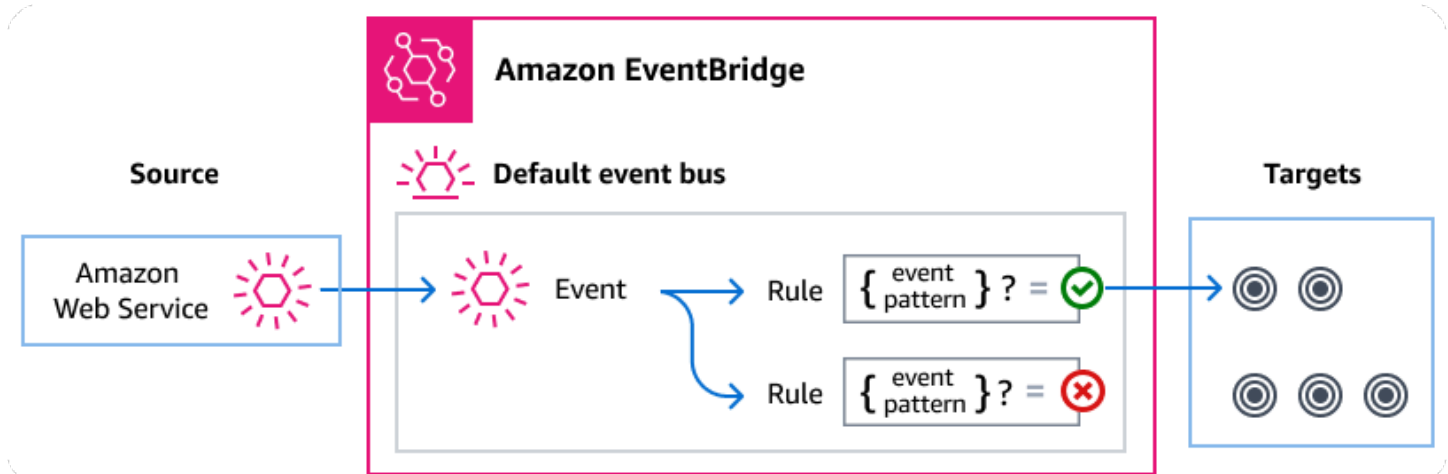
- [Amazon CloudWatch User Guide](#)
- [Service Quotas User Guide](#)

Managing Deadline Cloud events using Amazon EventBridge

Amazon EventBridge is a serverless service that uses events to connect application components together, making it easier for you to build scalable event-driven applications. Event-driven architecture is a style of building loosely-coupled software systems that work together by emitting and responding to events. Events represent a change in a resource or environment.

Here's how it works:

As with many AWS services, Deadline Cloud generates and sends events to the EventBridge default event bus. (The default event bus is automatically provisioned in every AWS account.) An event bus is a router that receives events and delivers them to zero or more destinations, or *targets*. Rules you specify for the event bus evaluate events as they arrive. Each rule checks whether an event matches the rule's *event pattern*. If the event does match, the event bus sends the event to the specified target(s).



Topics

- [Deadline Cloud events](#)
- [Delivering Deadline Cloud events using EventBridge rules](#)
- [Deadline Cloud events detail reference](#)

Deadline Cloud events

Deadline Cloud sends the following events to the default EventBridge event bus automatically. Events that match a rule's event pattern are delivered to the specified targets on a [best-effort basis](#). Events might be delivered out of order.

For more information, see [EventBridge events](#) in the *Amazon EventBridge User Guide*.

Event detail type	Description
Budget Threshold Reached	Sent when a queue reaches a percentage of its assigned budget.
Job Lifecycle Status Change	Sent when there is a change to the lifecycle status of a job.
Job Run Status Change	Sent when the overall status of the tasks in a job changes.
Step Lifecycle Status Change	Sent when there is a change to the lifecycle status of a step in a job.
Step Run Status Change	Sent when the overall status of the tasks in a step changes.
Task Run Status Change	Sent when the status of a task changes.

Delivering Deadline Cloud events using EventBridge rules

To have the EventBridge default event bus send Deadline Cloud events to a target, you must create a rule. Each rule contains an event pattern, which EventBridge matches against each event received on the event bus. If the event data matches the specified event pattern, EventBridge delivers that event to the rule's target(s).

For comprehensive instructions on creating event bus rules, see [Creating rules that react to events](#) in the *EventBridge User Guide*.

Creating event patterns that match Deadline Cloud events

Each event pattern is a JSON object that contains:

- A `source` attribute that identifies the service sending the event. For Deadline Cloud events, the source is `aws.deadline`.
- (Optional): A `detail-type` attribute that contains an array of the event types to match.
- (Optional): A `detail` attribute containing any other event data on which to match.

For example, the following event pattern matches against all Fleet Size Recommendation Change events for the specified farmId for Deadline Cloud:

```
{
  "source": ["aws.deadline"],
  "detail-type": ["Fleet Size Recommendation Change"],
  "detail": {
    "farmId": ["farm-1234567890000000000000000000000000"]
  }
}
```

For more information on writing event patterns, see [Event patterns](#) in the *EventBridge User Guide*.

Deadline Cloud events detail reference

All events from AWS services have a common set of fields containing metadata about the event, such as the AWS service that is the source of the event, the time the event was generated, the account and region in which the event took place, and others. For definitions of these general fields, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

In addition, each event has a `detail` field that contains data specific to that particular event. The reference below defines the detail fields for the various Deadline Cloud events.

When using EventBridge to select and manage Deadline Cloud events, it's useful to keep the following in mind:

- The `source` field for all events from Deadline Cloud is set to `aws.deadline`.
- The `detail-type` field specifies the event type.

For example, Fleet Size Recommendation Change.

- The `detail` field contains the data that is specific to that particular event.

For information on constructing event patterns that enable rules to match Deadline Cloud events, see [Event patterns](#) in the *Amazon EventBridge User Guide*.

For more information on events and how EventBridge processes them, see [Amazon EventBridge events](#) in the *Amazon EventBridge User Guide*.

Topics

- [Budget Threshold Reached event](#)
- [Fleet Size Recommendation Change event](#)
- [Job Lifecycle Status Change event](#)
- [Job Run Status Change event](#)
- [Step Lifecycle Status Change event](#)
- [Step Run Status Change event](#)
- [Task Run Status Change event](#)

Budget Threshold Reached event

You can use the Budget Threshold Reached event to monitor the percentage of a budget that has been used. Deadline Cloud sends events when the percentage used passes the following thresholds:

- 10, 20, 30, 40, 50, 60, 70, 75, 80, 85, 90, 95, 96, 97, 98, 99, 100

The frequency that Deadline Cloud sends Budget Threshold Reached events increases as the budget nears its limit. This frequency enables you to closely monitor a budget as it approaches its limit and to take action to keep from overspending. You can also set your own budget thresholds. Deadline Cloud sends an event when usage passes your custom thresholds.

If you change the amount of a budget, the next time Deadline Cloud sends a Budget Threshold Reached event it is based on the current percentage of the budget that has been used. For example, if you add \$50 to an \$100 budget that has reached its limit, the next Budget Threshold Reached event indicates that the budget is at 75 percent.

Below are the detail fields for the Budget Threshold Reached event.

The source and detail-type fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "detail-type": "Budget Threshold Reached",
  "source": "aws.deadline",
```

```
"account": "111122223333",
"time": "2017-12-22T18:43:48Z",
"region": "aa-example-1",
"resources": [],
"detail": {
  "farmId": "farm-12345678900000000000000000000000",
  "budgetId": "budget-12345678900000000000000000000000",
  "thresholdInPercent": 0
}
}
```

detail-type

Identifies the type of event.

For this event, this value is `Budget Threshold Reached`.

source

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

detail

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

farmId

The identifier of the farm that contains the job.

budgetId

The identifier of the budget that has reached a threshold.

thresholdInPercent

The percentage of the budget that has been used.

Fleet Size Recommendation Change event

When you configure your fleet to use event-based auto scaling, Deadline Cloud sends out events that you can use to manage your fleets. Each of these events contains information about the

current size and requested size of a fleet. For an example of using an EventBridge event and an example Lambda function to handle the event, see [Auto scale your Amazon EC2 fleet with Deadline Cloud scale recommendation feature](#).

The fleet size recommendation change event is sent when the following occur:

- When the recommended fleet size changes and `oldFleetSize` is different from `newFleetSize`.
- When the service detects that the actual fleet size does not match the recommended fleet size. You can get the actual fleet size from the `workerCount` in the `GetFleet` operation response. This mismatch might happen when an active Amazon EC2 instance fails to register as a Deadline Cloud worker.

Below are the detail fields for the Fleet Size Recommendation Change event.

The source and detail-type fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "detail-type": "Fleet Size Recommendation Change",
  "source": "aws.deadline",
  "account": "111122223333",
  "time": "2017-12-22T18:43:48Z",
  "region": "aa-example-1",
  "resources": [],
  "detail": {
    "farmId": "farm-12345678900000000000000000000000",
    "fleetId": "fleet-12345678900000000000000000000000",
    "oldFleetSize": 1,
    "newFleetSize": 5
  }
}
```

detail-type

Identifies the type of event.

For this event, this value is Fleet Size Recommendation Change.

source

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

detail

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

`farmId`

The identifier of the farm that contains the job.

`fleetId`

The identifier of the fleet that needs a size change.

`oldFleetSize`

The current size of the fleet.

`newFleetSize`

The recommended new size for the fleet.

Job Lifecycle Status Change event

When you create or update a job, Deadline Cloud sets the lifecycle status to show status of the most recent user initiated action.

A job lifecycle status change event is sent for any lifecycle status change, including when the job is created.

Below are the detail fields for the `Job Lifecycle Status Change` event.

The `source` and `detail`-type fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
```

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "detail-type": "Job Lifecycle Status Change",
  "source": "aws.deadline",
  "account": "111122223333",
  "time": "2017-12-22T18:43:48Z",
  "region": "aa-example-1",
  "resources": [],
  "detail": {
    "farmId": "farm-12345678900000000000000000000000",
    "queueId": "queue-12345678900000000000000000000000",
    "jobId": "job-12345678900000000000000000000000",
    "previousLifecycleStatus": "UPDATE_IN_PROGRESS",
    "lifecycleStatus": "UPDATE_SUCCEEDED"
  }
}
```

detail-type

Identifies the type of event.

For this event, this value is `Job Lifecycle Status Change`.

source

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

detail

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

farmId

The identifier of the farm that contains the job.

queueId

The identifier of the queue that contains the job.

jobId

The identifier of the job.

`previousLifecycleStatus`

The lifecycle state that the job is leaving. This field is not included when you first submit a job.

`lifecycleStatus`

The lifecycle state that the job is entering.

Job Run Status Change event

A job is composed of many tasks. Each task has a status. The status of all tasks are combined to give an overall status for a job. For more information, see [Job states in Deadline Cloud](#) in the *AWS Deadline Cloud User Guide*.

A job run status change event is sent when:

- The combined [taskRunStatus](#) field changes.
- The job is queued, unless the job is in the READY state.

A job run status change event is NOT sent when:

- The job is first created. To monitor job creation, monitor Job Lifecycle Status Change events for changes.
- The job's [taskRunStatusCounts](#) field changes but the combined job task run status does not change.

Below are the detail fields for the Job Run Status Change event.

The `source` and `detail-type` fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "detail-type": "Job Run Status Change",
  "source": "aws.deadline",
  "account": "111122223333",
```

```

    "time": "2017-12-22T18:43:48Z",
    "region": "aa-example-1",
    "resources": [],
    "detail": {
      "farmId": "farm-12345678900000000000000000000000",
      "queueId": "queue-12345678900000000000000000000000",
      "jobId": "job-12345678900000000000000000000000",
      "previousTaskRunStatus": "RUNNING",
      "taskRunStatus": "SUCCEEDED",
      "taskRunStatusCounts": {
        "PENDING": 0,
        "READY": 0,
        "RUNNING": 0,
        "ASSIGNED": 0,
        "STARTING": 0,
        "SCHEDULED": 0,
        "INTERRUPTING": 0,
        "SUSPENDED": 0,
        "CANCELED": 0,
        "FAILED": 0,
        "SUCCEEDED": 20,
        "NOT_COMPATIBLE": 0
      }
    }
  }
}

```

detail-type

Identifies the type of event.

For this event, this value is `Job Run Status Change`.

source

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

detail

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

`farmId`

The identifier of the farm that contains the job.

`queueId`

The identifier of the queue that contains the job.

`jobId`

The identifier of the job.

`previousTaskRunStatus`

The task run state that the job is leaving.

`taskRunStatus`

The task run state that the job is entering.

`taskRunStatusCounts`

The number of the job's tasks in each state.

Step Lifecycle Status Change event

When you create or update an event, Deadline Cloud sets the job's lifecycle status to describe the status of the most recent user initiated action.

A step lifecycle status change event is sent when:

- A step update starts (UPDATE_IN_PROGRESS).
- A step update completed successfully (UPDATE_SUCCEEDED).
- A step update failed (UPDATE_FAILED).

An event is not sent when the step is first created. To monitor step creation, monitor Job Lifecycle Status Change events for changes.

Below are the detail fields for the Step Lifecycle Status Change event.

The source and detail-type fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "detail-type": "Step Lifecycle Status Change",
  "source": "aws.deadline",
  "account": "111122223333",
  "time": "2017-12-22T18:43:48Z",
  "region": "aa-example-1",
  "resources": [],
  "detail": {
    "farmId": "farm-12345678900000000000000000000000",
    "queueId": "queue-12345678900000000000000000000000",
    "jobId": "job-12345678900000000000000000000000",
    "stepId": "step-12345678900000000000000000000000",
    "previousLifecycleStatus": "UPDATE_IN_PROGRESS",
    "lifecycleStatus": "UPDATE_SUCCEEDED"
  }
}
```

detail-type

Identifies the type of event.

For this event, this value is `Step Lifecycle Status Change`.

source

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

detail

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

farmId

The identifier of the farm that contains the job.

queueId

The identifier of the queue that contains the job.

`jobId`

The identifier of the job.

`stepId`

The identifier of the current job step.

`previousLifecycleStatus`

The lifecycle state that the step is leaving.

`lifecycleStatus`

The lifecycle state that the step is entering.

Step Run Status Change event

Each step in a job is composed of many tasks. Each task has a status. The task statuses are combined to give an overall status for steps and jobs.

A step run status change event is sent when:

- The combined [taskRunStatus](#) changes.
- The step is requeued, unless that step is in the READY state.

An event is not sent when:

- The step is first created. To monitor step creation, monitor Job Lifecycle Status Change events for changes.
- The step's [taskRunStatusCounts](#) changes but the combined step task run status does not change.

Below are the detail fields for the Step Run Status Change event.

The source and detail-type fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
```

```
"detail-type": "Step Run Status Change",  
"source": "aws.deadline",  
"account": "111122223333",  
"time": "2017-12-22T18:43:48Z",  
"region": "aa-example-1",  
"resources": [],  
"detail": {  
    "farmId": "farm-1234567890000000000000000000000000",  
    "queueId": "queue-1234567890000000000000000000000000",  
    "jobId": "job-1234567890000000000000000000000000",  
    "stepId": "step-1234567890000000000000000000000000",  
    "previousTaskRunStatus": "RUNNING",  
    "taskRunStatus": "SUCCEEDED",  
    "taskRunStatusCounts": {  
        "PENDING": 0,  
        "READY": 0,  
        "RUNNING": 0,  
        "ASSIGNED": 0,  
        "STARTING": 0,  
        "SCHEDULED": 0,  
        "INTERRUPTING": 0,  
        "SUSPENDED": 0,  
        "CANCELED": 0,  
        "FAILED": 0,  
        "SUCCEEDED": 20,  
        "NOT_COMPATIBLE": 0  
    }  
}  
  
}
```

detail-type

Identifies the type of event.

For this event, this value is Step Run Status Change.

source

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

detail

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

`farmId`

The identifier of the farm that contains the job.

`queueId`

The identifier of the queue that contains the job.

`jobId`

The identifier of the job.

`stepId`

The identifier of the current job step.

`previousTaskRunStatus`

The run state that the step is leaving.

`taskRunStatus`

The run state that the step is entering.

`taskRunStatusCounts`

The number of the step's tasks in each state.

Task Run Status Change event

The [runStatus](#) field is updated as the task runs. An event is sent when:

- The task's run status changes.
- The task is requeued, unless the task is in the READY state.

An event is not sent when:

- The task is first created. To monitor task creation, monitor Job Lifecycle Status Change events for changes.

Below are the detail fields for the Task Run Status Change event.

The `source` and `detail-type` fields are included below because they contain specific values for Deadline Cloud events. For definitions of the other metadata fields that are included in all events, see [Event structure reference](#) in the *Amazon EventBridge User Guide*.

```
{
  "version": "0",
  "id": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
  "detail-type": "Task Run Status Change",
  "source": "aws.deadline",
  "account": "111122223333",
  "time": "2017-12-22T18:43:48Z",
  "region": "aa-example-1",
  "resources": [],
  "detail": {
    "farmId": "farm-12345678900000000000000000000000",
    "queueId": "queue-12345678900000000000000000000000",
    "jobId": "job-12345678900000000000000000000000",
    "stepId": "step-12345678900000000000000000000000",
    "taskId": "task-12345678900000000000000000000000-0",
    "previousRunStatus": "RUNNING",
    "runStatus": "SUCCEEDED"
  }
}
```

`detail-type`

Identifies the type of event.

For this event, this value is `Fleet Size Recommendation Change`.

`source`

Identifies the service that generated the event. For Deadline Cloud events, this value is `aws.deadline`.

`detail`

A JSON object that contains information about the event. The service generating the event determines the content of this field.

For this event, this data includes:

`farmId`

The identifier of the farm that contains the job.

queueId

The identifier of the queue that contains the job.

jobId

The identifier of the job.

stepId

The identifier of the current job step.

taskId

The identifier of the running task.

previousRunStatus

The run state that the task is leaving.

runStatus

The run status that the task is entering.

Querying session statistics aggregated data using the AWS CLI

To track costs, analyze resource usage, or identify which users are consuming the most resources, you can use the AWS Command Line Interface (AWS CLI) to query aggregated session statistics for your AWS Deadline Cloud (Deadline Cloud) farms. The session statistics API provides data about costs, runtime, and usage that you can group by various dimensions such as queue, fleet, instance type, or user.

Querying session statistics is an asynchronous process. First, you start an aggregation request, then you retrieve the results using the aggregation ID.

Starting an aggregation request

To start an aggregation request, run the `start-sessions-statistics-aggregation` command. The following example groups statistics by user ID for a specific queue. Replace the *placeholder* text with your information.

```
aws deadline start-sessions-statistics-aggregation \
  --farm-id farm-id \
  --resource-ids '{"queueIds":["queue-id"]}' \
  --start-time 2025-11-24T10:00:00Z \
  --end-time 2025-11-25T18:00:00Z \
  --group-by '["USER_ID"]' \
  --period HOURLY \
  --statistics '["SUM"]' \
  --timezone UTC-08:00 \
  --region region-name
```

You can group statistics by other dimensions such as `QUEUE_ID`, `FLEET_ID`, `JOB_ID`, `INSTANCE_TYPE`, or `LICENSE_PRODUCT`. For more information about all available parameters, see [start-sessions-statistics-aggregation](#) in the AWS CLI Command Reference.

The response contains an aggregation ID:

```
{
  "aggregationId": "92b35143f2d04641979bc9b777232f38"
```

```
}
```

Retrieving results

Run the `get-sessions-statistics-aggregation` command with the aggregation ID to retrieve the results. Replace the *placeholder* text with your information.

```
aws deadline get-sessions-statistics-aggregation \  
  --farm-id farm-id \  
  --aggregation-id aggregation-id \  
  --region region-name
```

The following example shows a response when you group statistics by user ID. The `userId` field contains a UUID that you must map to a username to identify the user:

```
{  
  "statistics": [  
    {  
      "userId": "f9c1f3f0-1031-70dc-4d25-30d7225b04a0",  
      "count": 1,  
      "costInUsd": {  
        "sum": 0.0  
      },  
      "runtimeInSeconds": {  
        "sum": 53.773  
      },  
      "aggregationStartTime": "2025-11-24T22:00:00Z",  
      "aggregationEndTime": "2025-11-24T23:00:00Z"  
    },  
  ],  
  "status": "COMPLETED"  
}
```

To find the username associated with a `userId`, see [the section called “Retrieving user metadata using userID”](#).

For more information about the API, see the [Deadline Cloud API Reference](#).

Topics

- [the section called “Retrieving user metadata using userID”](#)

Retrieving user metadata and attributes using userID in an identity store

Note

This procedure also applies to the `createdBy` field returned by the [SearchJobs](#) API, which uses the same user ID format.

The `userId` field in session statistics contains one of the following values:

- An AWS Identity and Access Management (IAM) role ARN, for example:
`arn:aws:sts::123456789012:assumed-role/Admin/user-Isengard`.
- An IAM Identity Center user ID (UUID), for example:
`f9c1f3f0-1031-70dc-4d25-30d7225b04a0`.

For IAM role ARNs, the username is visible in the ARN itself. For IAM Identity Center user IDs, you can look up the username using the IAM Identity Center Identity Store API.

To identify the username associated with an IAM Identity Center user ID, use the following procedure. Before you begin, get the Identity Store ID from your IAM Identity Center settings. For more information, see [the section called "Finding your Identity Store ID"](#).

To map a user ID

1. Run the following command, replacing *IdentityStoreId* with your Identity Store ID and *userUUID* with the `userId` from the session statistics response:

```
aws identitystore describe-user \  
  --identity-store-id IdentityStoreId \  
  --user-id userUUID
```

2. Review the response, which includes the username:

```
{  
  "UserName": "jdoe",  
  "UserId": "f9c1f3f0-1031-70dc-4d25-30d7225b04a0",  
  "Name": {
```

```
    "FamilyName": "Doe",
    "GivenName": "Jane"
  },
  "DisplayName": "Jane Doe",
  "Emails": [{
    "Value": "jdoe@example.com",
    "Type": "work",
    "Primary": true
  }],
  "IdentityStoreId": "d-xxxxxxxxxx"
}
```

Finding your Identity Store ID

To map user IDs to usernames, you need the Identity Store ID. You can find the Identity Store ID using the IAM Identity Center console or the AWS CLI.

Console

To find your Identity Store ID using the console, use the following procedure.

1. Sign in to the AWS Management Console and open the [IAM Identity Center console](#).
2. In the navigation pane, choose **Settings**.
3. Copy the **IAM Identity Center Identity Store ID** value. The format is d-xxxxxxxxxx.

AWS CLI

Run the following command, replacing *region-name* with the Region where your IAM Identity Center instance is configured:

```
aws sso-admin list-instances --region region-name
```

The response includes the IdentityStoreId:

```
{
  "Instances": [
    {
      "CreateDate": "2025-11-19T15:45:55.160000-08:00",
      "IdentityStoreId": "d-xxxxxxxxxx",
    }
  ]
}
```

```
    "InstanceArn": "arn:aws:sso:::instance/ssoins-xxxxxxxxxxxxxxxx",
    "OwnerAccountId": "123456789012",
    "Status": "ACTIVE"
  }
]
```

Verifying the user mapping

After you map a user ID to a username, you can verify in the IAM Identity Center console that the user ID matches the expected user. To verify the user mapping, use the following procedure.

1. Sign in to the AWS Management Console and open the [IAM Identity Center console](#).
2. In the navigation pane, choose **Users**.
3. Choose the username from the AWS CLI response.
4. In the **General information** section, verify that the **User ID** matches the `userId` from your session statistics.

Additional resources

- [IAM Identity Center User Guide](#)
- [IAM Identity Center Identity Store API Reference](#)

Code examples for Deadline Cloud

The [deadline-cloud-samples](#) repository on the GitHub website contains a collection of working artifacts for AWS Deadline Cloud. The examples cover job submission, software packaging, infrastructure deployment, worker configuration, and more. You can submit, deploy, or adapt them for your own farms.

The following sections describe each category of example and link to individual topics that explain when to use each one.

Tutorials

Complete walkthroughs that take you from prerequisites to a working result on a real workload. Each tutorial covers prerequisites, step-by-step instructions, parameter references, and cleanup.

For details, see [the section called “Tutorials”](#).

Job bundle examples

Job bundles define jobs that you submit to a Deadline Cloud queue. The repository includes job bundles for digital content creation (DCC) renderers such as Maya, Blender, Houdini, Nuke, 3ds Max, Cinema 4D, KeyShot, and VRED. It also includes machine learning workloads, multi-step rendering pipelines, and small examples that demonstrate individual concepts such as job parameters, environments, and job attachments.

For details, see [the section called “Job bundles”](#).

Conda recipe examples

Conda recipes build conda packages that supply applications and plugins to your jobs through a queue environment. The repository includes recipes for many versions of Maya, Blender, Houdini, Nuke, 3ds Max, Cinema 4D, KeyShot, and Unreal Engine, along with renderer plugins such as Arnold, V-Ray, and Redshift.

For details, see [the section called “Conda recipes”](#).

CloudFormation template examples

CloudFormation templates deploy farms and supporting infrastructure. The repository includes a starter farm template, templates for connecting service-managed fleets to private VPC resources,

and templates for capacity management, standby scheduling, fleet health checks, and budget notifications.

For details, see [the section called “CloudFormation templates”](#).

Host configuration script examples

Host configuration scripts run with elevated privileges on service-managed fleet workers to install software, configure GPU runtimes for containers, install fonts, enable swap, and perform other administrative tasks. You attach a script to a fleet through the Deadline Cloud console or the `update-fleet` CLI command.

For details, see [the section called “Host configuration scripts”](#).

Queue environment examples

Queue environments provide software applications to your jobs. The repository includes several conda-based queue environments with different performance trade-offs, a Rez-based queue environment for customer-managed fleets that have a shared file system, a pip-based queue environment for Python-only packages, and a queue environment that disconnects usage-based licensing so you can use a custom license server.

For details, see [the section called “Queue environments”](#).

Other examples

The repository also includes the following examples:

- [the section called “Amazon Linux 2023 worker image”](#) — A Dockerfile that replicates the package set of the Deadline Cloud service-managed fleet (SMF) worker AMI on Amazon Linux 2023.
- [the section called “Blender container image”](#) — A Dockerfile that packages Blender with the OpenJD adaptor and GPU support for Cycles rendering in a container.
- [the section called “Upload to job attachments”](#) — A script that uploads files and directories from your workstation or server to Deadline Cloud job attachments storage.
- [the section called “Virtual workstation setup”](#) — Scripts that turn a fresh Linux or Windows workstation into a Deadline Cloud submission machine with a DCC, the submitter, and a pre-configured monitor profile.
- [the section called “License limits hook”](#) — A submission hook that enforces fixed license limits by combining the Deadline Cloud Limits feature with a pre-submission hook.

- [the section called “AI agent skills”](#) — Reusable AI agent skills that walk an agent through writing job bundles, building conda packages, and creating Windows host configuration scripts.

Additional resources

The following resources help you author and run your own examples in Deadline Cloud.

- [the section called “Job bundles”](#) — Learn how to author your own job bundles for Deadline Cloud.
- [Using AI agents](#) — Use AI agents to write job bundles, develop conda packages, and troubleshoot jobs.
- [Open Job Description specifications](#) on the GitHub website — The schema and reference for job templates that Deadline Cloud runs.
- [deadline-cloud-samples](#) on the GitHub website — The source repository for all of the examples in this chapter.

Topics

- [Tutorials for Deadline Cloud](#)
- [Job bundle examples for Deadline Cloud](#)
- [Conda recipe examples for Deadline Cloud](#)
- [CloudFormation template examples for Deadline Cloud](#)
- [Deploy Deadline Cloud farms with the AWS CDK](#)
- [Deploy Deadline Cloud farms with Terraform](#)
- [Host configuration script examples for Deadline Cloud](#)
- [Queue environment examples for Deadline Cloud](#)
- [Build a worker-equivalent Amazon Linux 2023 Docker image for Deadline Cloud](#)
- [Build a Blender Docker image for GPU rendering on Deadline Cloud](#)
- [Upload files to Deadline Cloud job attachments](#)
- [Set up a virtual workstation for Deadline Cloud with a script](#)
- [Enforce fixed license limits with a Deadline Cloud submission hook](#)
- [AI agent skills for Deadline Cloud](#)

Tutorials for Deadline Cloud

The following tutorials cover complete workloads that you can run on Deadline Cloud. Each tutorial includes prerequisites, step-by-step instructions, parameter references, and cleanup guidance.

Topics

- [FLUX.2 Klein LoRA fine-tuning and image generation](#)
- [Benchmark LLMs with vLLM and lm-evaluation-harness](#)
- [Train 3D Gaussian Splatting from your own video](#)
- [Run an autonomous driving simulation sweep with CARLA](#)
- [Fine-tune Hugging Face LLMs with LoRA and QLoRA](#)
- [Run batch LLM inference with vLLM](#)
- [Generate images in batch with a diffusion model](#)

FLUX.2 Klein LoRA fine-tuning and image generation

This example demonstrates how to fine-tune a FLUX.2 Klein image generation model using LoRA (Low-Rank Adaptation) and generate images at scale on Deadline Cloud. The workflow trains a lightweight adapter from a small image dataset. It then uses that adapter to generate new images from text prompts, all orchestrated as Deadline Cloud jobs on a GPU fleet.

The source code for this example is available in the [deadline-cloud-samples](#) repository on the GitHub website.

The following video demonstrates the FLUX.2 Klein LoRA workflow on Deadline Cloud.

Estimated time: 30–60 minutes (including 15–45 minutes for training).

Overview

The example consists of two job bundles:

1. **lora_training** – Fine-tunes the [FLUX.2 Klein model](#) on the Hugging Face website on your image dataset, producing a `.safetensors` adapter file. For more information, see [LoRA](#) on the Hugging Face website.
2. **image_generation** – Loads the base model with your trained adapter and generates images from text prompts. When generating multiple images, work is parallelized across workers.

LoRA is a parameter-efficient fine-tuning technique that trains a small adapter instead of modifying the full model weights. This approach makes training fast (15–45 minutes on a single GPU) and the resulting adapter portable and easy to version.

For training and inference, the job bundles use the [diffusers](#) and [peft](#) libraries on the GitHub website.

To complete this example, follow these steps:

1. Complete the prerequisites.
2. Prepare training data.
3. Submit the training job.
4. Submit the image generation job.
5. Clean up resources.

Prerequisites

Before you begin, make sure that you have the following:

- A Deadline Cloud farm with a GPU fleet (Linux, NVIDIA A10G GPU, 13 GB+ VRAM, 64 GiB+ system memory recommended).
- A queue associated with the GPU fleet.
- The Deadline Cloud CLI installed on your workstation. For installation instructions, see the [deadline-cloud](#) repository on the GitHub website.
- 20–50 training images of your subject (.jpg, .png, .jpeg, or .webp).

Job bundle structure

```
flux2_klein_lora/  
### lora_training/  
#   ### template.yaml           # OpenJD job template  
#   ### train_lora.py           # Training script  
### image_generation/  
#   ### template.yaml           # OpenJD job template  
#   ### generate_image.py        # Inference script
```

Prepare training data

To prepare your training data

1. Create a directory containing images of your subject. For best results, use 20–50 images with varied poses, lighting, and backgrounds.
2. Optionally, provide a `.txt` caption file alongside each image for more nuanced training:

```
training_images/
### IMG_001.jpeg
### IMG_001.txt    # "a photo of ohwx dog sitting on grass"
### IMG_002.jpeg
### IMG_002.txt    # "a photo of ohwx dog running on beach"
```

If no caption files are present, the training script generates them automatically using the instance prompt parameter. Custom captions produce higher-quality results because they teach the model more specific associations between your trigger word and visual features.

3. Choose a unique trigger word (such as `ohwx`). This binds the learned concept to a token that doesn't conflict with existing model knowledge.

Submit the training job

To submit the training job

1. Navigate to the `flux2_klein_lora` directory in your cloned `deadline-cloud-samples` repository.
2. To submit the training job with the GUI, run the following command:

```
deadline bundle gui-submit ./lora_training
```

Or, to submit the training job directly with the CLI, run the following command:

```
deadline bundle submit ./lora_training \
  --queue-id gpu-queue-id \
  --parameter DatasetPath="$HOME/training_images" \
  --parameter InstancePrompt="a photo of ohwx dog" \
  --parameter OutputDir=/tmp/lora_output \
  --parameter MaxTrainSteps=1500 \
```

```
--parameter Resolution=512
```

3. Monitor the job status in the Deadline Cloud console or by using the `deadline job get` command.

Training parameters

Training parameters

Parameter	Description	Default
Model Version	<code>flux.2-klein-base-4b</code> or <code>flux.2-klein-4b</code>	<code>flux.2-klein-base-4b</code>
Dataset Path	Directory containing training images	None
Instance Prompt	Text describing your subject	None
Resolution	Training image resolution in pixels	512
Network Dim	LoRA rank (higher = more capacity)	16
Network Alpha	LoRA alpha scaling factor	16
Max Training Steps	Number of training iterations	1500
Output Directory	Where to save trained weights	None

Training output

The training job produces the following artifacts:

- `flux2_klein_lora.safetensors` – The trained LoRA adapter with embedded metadata.
- Checkpoints saved every 300 steps.

To download the output after training completes, run the following command:

```
deadline job download-output \  
  --job-id training-job-id \  
  --queue-id gpu-queue-id
```

Submit the image generation job

To submit the image generation job

1. Navigate to the `flux2_klein_lora` directory in your cloned `deadline-cloud-samples` repository.
2. To submit the image generation job with the GUI, run the following command:

```
deadline bundle gui-submit ./image_generation
```

Or, to submit the image generation job directly with the CLI, run the following command:

```
deadline bundle submit ./image_generation \  
  --queue-id gpu-queue-id \  
  --parameter LoRAPath=/tmp/lora_output/flux2_klein_lora.safetensors \  
  --parameter Prompt="a photo of ohwx dog wearing a tuxedo" \  
  --parameter OutputDir=/tmp/generated_images \  
  --parameter NumImages=4
```

3. Monitor the job status in the Deadline Cloud console or by using the `deadline job get` command.

When `NumImages` is greater than 1, Deadline Cloud distributes the work across multiple workers for faster generation.

Generation parameters

Generation parameters

Parameter	Description	Default
LoRA Path	Path to trained <code>.safetensors</code> file	None
Prompt	Text description (include your trigger word)	None
Number of Images	Total images to generate	1
Width	Output width in pixels	1024
Height	Output height in pixels	1024

Parameter	Description	Default
Inference Steps	Number of denoising steps	50
Guidance Scale	Classifier-free guidance scale	4.0

Generation output

Images are saved as `image_0001.png`, `image_0002.png`, and so on. To download them, run the following command:

```
deadline job download-output \  
  --job-id generation-job-id \  
  --queue-id gpu-queue-id
```

Model variants

FLUX.2 Klein model variants

Model	Parameters	Best for
flux.2-klein-base-4b	4B	Fine-tuning, commercial use
flux.2-klein-4b	4B	Fast inference (4 denoising steps)

Training recommendations

The following recommendations can help you get the best results:

- **Dataset size** – 20–50 high-quality images work well for most subjects.
- **Resolution** – Use 512 for GPUs with 24 GB VRAM. Higher resolutions require more memory.
- **Training steps** – Start with 1500. Increase if outputs don't resemble your subject.
- **Network dim** – 16 is a good default. Use 32 for complex or abstract concepts.
- **Learning rate** – The default of 1e-4 works well for most cases.

Clean up

To avoid ongoing charges, clean up the resources that you created for this example:

To clean up example resources

1. In the Deadline Cloud console, stop or delete the GPU fleet that you used for this example.
2. Delete any Amazon Simple Storage Service (Amazon S3) objects created by job attachments if they are no longer needed.
3. Remove local output files if they are no longer needed:

```
rm -rf /tmp/lora_output /tmp/generated_images
```

Related resources

The following resources provide additional information:

- [Example source code](#) on the GitHub website
- [FLUX.2 Klein](#) on the Hugging Face website
- [Open Job Description specification](#) on the GitHub website
- [Diffusers documentation](#) on the Hugging Face website
- [Deadline Cloud Architecture Guidance](#)

Benchmark LLMs with vLLM and lm-evaluation-harness

This tutorial walks you through evaluating multiple large language models (LLMs) against multiple benchmarks in a single Deadline Cloud job. Each model becomes one task in a parameter sweep, and tasks run in parallel across workers. A final step aggregates per-model results into a ranked leaderboard in CSV and Markdown format.

The source code for this tutorial is available in the [deadline-cloud-samples](#) repository on the GitHub website.

The following video demonstrates the vLLM LLM leaderboard workflow on Deadline Cloud.

Estimated time: 20–40 minutes (depending on the number of models and benchmarks).

Overview

Each task in the `EvalModels` step starts a local vLLM server, runs every benchmark with EleutherAI's `lm-evaluation-harness` against the local endpoint, then stops vLLM. For more

information, see [vLLM](#) and [lm-evaluation-harness](#) on the GitHub website. Models load directly from Hugging Face Hub, so job attachments are not required.

To complete this tutorial, follow these steps:

1. Complete the prerequisites.
2. Set up your farm.
3. Submit the evaluation job.
4. Download and review results.
5. Clean up resources.

Prerequisites

Before you begin, the following setup is recommended:

- A Deadline Cloud farm with an NVIDIA GPU service-managed fleet (A10G or L4, at least 32 GB RAM, at least 4 vCPUs).
- A queue with a conda queue environment attached that reads `CondaPackages` and `CondaChannels` job parameters.
- The Deadline Cloud CLI installed on your workstation. For installation instructions, see the [deadline-cloud](#) repository on the GitHub website.
- Sufficient Deadline Cloud service quota for GPU instances. The default 3-model run on `g5.xlarge` (4 vCPUs and 1 GPU each) requires at least 3 GPUs under *OnDemand G instance GPUs per region* and 12 vCPUs under *OnDemand vCPUs per region*.

Note

A Hugging Face token is only required for gated models (such as Llama). The default model list uses ungated models.

Set up your farm

The fastest way to get a compatible farm is to deploy the [CUDA farm CloudFormation template](#) on the GitHub website. The template provisions an NVIDIA GPU service-managed fleet (A10G or L4) and a queue with a conda queue environment that this bundle uses without modification.

To configure the CLI for your farm

- After the CloudFormation stack reaches CREATE_COMPLETE, configure the Deadline Cloud CLI to use the new farm:

```
deadline config set defaults.farm_id FarmId-from-stack-outputs  
deadline config set defaults.queue_id CUDAQueueId-from-stack-outputs
```

If you already have a farm, the following configuration is recommended:

- An SMF fleet with NVIDIA GPUs, at least 32 GB RAM, and at least 4 vCPUs.
- A queue with a conda queue environment that reads CondaPackages and CondaChannels job parameters.

Submit the evaluation job

To submit the evaluation job

- Clone the samples repository and navigate to the job bundle directory:

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git  
cd deadline-cloud-samples/job_bundles/vllm_lm_eval_leaderboard
```

- Submit the job with the default models and benchmarks:

```
deadline bundle submit . \  
--parameter MaxModelLen=2048
```

The default model list evaluates three small, ungated models: Qwen/Qwen2.5-0.5B, Qwen/Qwen2.5-1.5B, and EleutherAI/pythia-1.4b. The default benchmarks are a commonsense reasoning suite: hellaswag, arc_easy, arc_challenge, winogrande.

- Monitor the job status in the Deadline Cloud console or by using the `deadline job get` command.

Changing the model list

Models are defined as a STRING parameter space on the EvalModels step in `template.yaml`:

```
parameterSpace:
  taskParameterDefinitions:
    - name: ModelName
      type: STRING
      range:
        - "Qwen/Qwen2.5-0.5B"
        - "Qwen/Qwen2.5-1.5B"
        - "EleutherAI/pythia-1.4b"
```

To add or remove models, edit the range list. Each entry becomes a task visible in the Deadline Cloud monitor. Model IDs must be supported by vLLM. For more information, see the [supported models list](#) on the vLLM website.

Choosing benchmarks

The Benchmarks job parameter is a comma-separated list of lm-evaluation-harness task names. Override the default benchmarks at submit time:

```
deadline bundle submit . \
  --parameter Benchmarks="hellaswag,mmlu,gsm8k"
```

All benchmarks in the list run sequentially against each model's vLLM server. Keep `MaxModelLen` less than or equal to the smallest model's context window. For a full list of available benchmarks, see the [lm-evaluation-harness tasks](#) on the GitHub website.

Download and review results

To download the leaderboard results

1. After the job completes, download the output:

```
deadline job download-output --job-id job-id
```

2. View the leaderboard:

```
cat leaderboard_results/leaderboard.md
```

The following example shows typical leaderboard output:

```
# LLM Leaderboard
```

Models: 3 | Benchmarks: arc_challenge, arc_easy, hellaswag, winogrande

Rank	Model	arc_challenge	arc_easy	hellaswag	winogrande
Mean					
1	Qwen/Qwen2.5-1.5B	0.4497	0.7176	0.6775	0.6322
2	Qwen/Qwen2.5-0.5B	0.3200	0.5816	0.5223	0.5691
3	EleutherAI/pythia-1.4b	0.2833	0.5387	0.5201	0.5730

Clean up

To avoid ongoing charges, clean up the resources that you created for this tutorial:

To clean up tutorial resources

1. If you deployed the CUDA farm CloudFormation template, delete the CloudFormation stack from the CloudFormation console.
2. If you used an existing farm, stop or delete the GPU fleet that you used for this tutorial.
3. Remove local output files if they are no longer needed:

```
rm -rf leaderboard_results/
```

Troubleshooting

Fleet does not scale up workers

The most common cause is an Deadline Cloud service quota. Open the [Service Quotas console](#) under **AWS Deadline Cloud** and confirm that you have headroom for *OnDemand G instance GPUs per region* and *OnDemand vCPUs per region*. Quota increases can take minutes to a couple of business days.

Related resources

The following resources provide additional information on the GitHub website and the vLLM website:

- [Sample source code](#)
- [vLLM](#)
- [lm-evaluation-harness](#)
- [CUDA farm CloudFormation template](#)
- [vLLM supported models](#)

Train 3D Gaussian Splatting from your own video

This tutorial walks you through training your own 3D Gaussian Splatting point cloud from a video that you capture. You submit the [Gaussian Splatting pipeline job bundle](#) on the GitHub website to your Deadline Cloud farm. The job takes a video file as input and produces a Gaussian Splatting .ply file as output. When the job completes, you can view the result in any Gaussian Splatting viewer, such as [SuperSplat](#) on the GitHub website.

The pipeline runs a single task that:

1. Extracts video frames with [FFmpeg](#) on the FFmpeg website.
2. Solves Structure-from-Motion with [COLMAP](#) on the COLMAP website and [GLOMAP](#) on the GitHub website, saving the pinhole model and undistorted images.
3. Trains Gaussian Splatting with [NeRF Studio splatfacto](#) on the Nerfstudio website, [Splatfacto in the Wild](#) on the Nerfstudio website, or the [simple_trainer.py gsplat library example](#) on the gsplat website, and saves the output to the .ply file you specify.

The following video demonstrates the Gaussian Splatting pipeline workflow on Deadline Cloud.

Estimated time: A few hours, including farm setup. Depending on the input video and the settings you select, the job itself can finish in 10 minutes or take hours.

Running this tutorial incurs charges for the GPU worker instances that process the job. The [Understand the cost of your training jobs](#) section shows how to review those costs.

Overview

To complete this tutorial, follow these steps:

1. Complete the prerequisites.
2. Set up your farm.

3. Capture a video of a subject.
4. Submit the `gsplat_pipeline` job.
5. Monitor the job.
6. Download and view the Gaussian Splatting `.ply` file.
7. Clean up resources.

Prerequisites

Before you begin, complete the following setup:

1. [Create an AWS account](#) if you do not already have one.

Set up your farm

You need a Deadline Cloud farm with a CUDA GPU fleet to run the Gaussian Splatting job. To set one up, complete the following steps:

1. Follow the [CUDA farm sample CloudFormation template](#) instructions on the GitHub website to create a Deadline Cloud farm that has a CUDA GPU fleet and can build conda packages.
2. Follow the [NeRF Studio sample conda package recipe](#) instructions on the GitHub website to build a NeRF Studio conda package into the Amazon S3 channel of your CUDA farm. For more information, see [NeRF Studio](#) on the Nerfstudio website.

Note

If you only need the default NERFSTUDIO (splatfacto) trainer, you don't need to deploy the CUDA farm CloudFormation template or build the custom NeRF Studio conda package. The minimum requirements are a Deadline Cloud farm with a GPU fleet, a queue environment with `conda-forge` included in the conda channels, and the conda packages `ffmpeg colmap glomap nerfstudio cuda`. The `GSPLAT_SIMPLE_TRAINER` and `NERFSTUDIO_SPLATFACTOW` trainer options depend on commands that are only available in the custom-built NeRF Studio conda package.

Capture a video of a subject

You can use a video-capable camera like your smartphone to capture a video of a subject for your Gaussian Splatting. Consider the following tips:

- Use a wide field of view, such as zoom level 0.5 in your camera app. With a wider field of view, more objects are common between image pairs for Structure-from-Motion to use.
- Turn off video stabilization. Turning it off preserves identical lens optics between all frames, and can increase the quality of solves.
- Plan your camera motion depending on the subject. To capture an object, like a bench or a bicycle, orbit around the subject a few times at different heights and distances. To capture a room interior, follow around the outside of the room with the camera facing inwards, and repeat at different camera heights. To capture less structured spaces such as outside terrain, think about how to include everything you want in your Gaussian Splatting, and how to capture all of it from multiple different angles.
- Capture the video with slow and steady motion.
- Keep moving the camera. Avoid stopping and panning from a single location.

Copy the video you captured from your camera to your computer for submitting to the farm.

Submit the gsplat_pipeline job

To submit the Gaussian Splatting job

1. If you don't have a local copy of the [deadline-cloud-samples repository](#) on the GitHub website, clone it or [download it as a ZIP](#):

```
git clone https://github.com/aws-deadline/deadline-cloud-samples.git
cd deadline-cloud-samples/job_bundles
```

2. From the `job_bundles` directory, run the following command:

```
deadline bundle gui-submit gsplat_pipeline
```

3. Switch to the **Job-specific settings** tab and select paths for both the **Input Video File** and the **Output Ply File**.
4. Choose **Submit**, or customize the settings first. If the input video has higher complexity, you may need to increase the **Approximate Image Count** value.

You can also select which Gaussian Splatting trainer to use and customize the CLI options for it. For example, to use the Monte Carlo Markov chain (MCMC) trainer with the bilateral grid option to produce up to 2 million splats, make the following changes:

1. Switch the **Gaussian Splatting Trainer** from NERFSTUDIO to GSPLAT_SIMPLE_TRAINER.
2. Modify the **GSplat Simple Trainer Options** text from `--strategy.cap-max 1000000` to `--strategy.cap-max 2000000` to get 2 million splats, and from `--no-use-bilateral-grid` to `--use-bilateral-grid` to enable the bilateral grid option.

Note

If you select an input video or options that require higher memory usage than the CUDA fleet provides, you need to update the fleet's minimum settings. Re-deploy your CloudFormation template with updated parameter values, or edit the settings from the Deadline Cloud console.

Monitor the job

After you submit the job, you can monitor its status from the Deadline Cloud monitor job table. Depending on the input video and the settings you selected, the job might finish in 10 minutes or take hours.

When the job is running, open the context menu for the task and choose **View logs** to open the log view. As the pipeline goes through its steps, it updates the status message visible in the task run details.

If you encounter errors, the log output in this view helps you track down the cause. Common causes include running out of memory or failing to solve for the camera poses. The pipeline can fail during any of these steps: FFmpeg frame extraction, GLOMAP Structure-from-Motion solving, or NeRF Studio training. To resolve errors, either adjust the fleet infrastructure or use a lower image resolution or image count.

Download and view the Gaussian Splatting .ply

To download and view the output

1. When the job completes successfully, open the context menu for the completed task in the **Tasks** table of Deadline Cloud monitor and choose **Download output**. Depending on your

settings, it either shows the download progress or presents a CLI command you can use to download. Deadline Cloud saves the .ply file to the location you selected when submitting the job.

2. To view the result in your browser, open the [SuperSplat Editor](#) on the SuperSplat website and drag the file from your operating system's file browser onto the SuperSplat page.
3. Toggle the **Show/Hide Splats** option, then rotate and move around the scene to find the subject of your capture.

With SuperSplat, you can also edit your Gaussian Splatting. For example, you can select a sphere, invert the selection, and delete the splats outside it to crop the scene. The [SuperSplat documentation](#) on the GitHub website explains how to use the editor, or you can import your .ply file into your tool of choice.

Understand the cost of your training jobs

Deadline Cloud monitor includes a usage explorer feature that estimates the cost of the jobs you run, showing how much each training costs. If you're comfortable running your tests during off-peak hours when CUDA-capable instances are available with low enough interruption rates, a spot CUDA fleet costs less than an on-demand fleet. Usage explorer does not show costs outside of your job run time, such as idle worker instance time or storage on your Amazon S3 bucket.

Clean up

To avoid ongoing charges, clean up the resources that you created for this tutorial:

To clean up tutorial resources

1. If you deployed the CUDA farm CloudFormation template, delete the CloudFormation stack from the CloudFormation console.
2. If you used an existing farm and created a GPU fleet specifically for this tutorial, stop or delete that fleet. If you used a pre-existing shared fleet, leave it in place.

Next steps

You can pull apart, edit, and remix the `gsplat_pipeline` job bundle to suit your needs. You can customize your own 3D reconstruction pipeline, or follow the same patterns to run different CUDA workloads on your Deadline Cloud CUDA farm. The following directions are documented in the [sample README](#) on the GitHub website:

- Run the job anywhere with the [Open Job Description CLI](#) on the GitHub website, such as locally or on an Amazon EC2 instance, instead of on your farm.
- Decompose the pipeline's single task into multiple Open Job Description steps connected by dependencies, following the pattern in the [Job Development Progression sample](#) on the GitHub website. Separate steps can run on different fleets, such as a CPU-only fleet for frame extraction and a GPU fleet for training.
- Split the pipeline into multiple jobs with different structure. For example, one job to solve Structure-from-Motion once, and a second job to iteratively try different variations of Gaussian Splatting training parameters.

Related resources

The following resources provide additional information:

- [Sample source code](#) on the GitHub website
- [CUDA farm CloudFormation template](#) on the GitHub website
- [NeRF Studio conda package recipe](#) on the GitHub website
- [3D Gaussian Splatting blog post](#)
- [Open Job Description \(OpenJD\) templates for Deadline Cloud](#)

Run an autonomous driving simulation sweep with CARLA

This tutorial walks you through running a CARLA autonomous driving simulation parameter sweep with configurable multi-sensor capture. [CARLA](#) on the CARLA website is an open-source simulator. You build a Docker image, push it to Amazon Elastic Container Registry (Amazon ECR), and submit the [autonomous driving CARLA job bundle](#) on the GitHub website to a GPU fleet on your Deadline Cloud farm.

The job runs a lane-change cut-in scenario where an NPC vehicle starts behind the ego vehicle, accelerates to position itself 20 meters ahead of the ego during a 105-second get-ahead phase, then cuts into the ego's lane. It sweeps across configurable ego speeds, NPC speeds, and NPC starting distances, creating a task for each parameter combination (default $2 \times 2 \times 2 = 8$ tasks). Each task captures multi-sensor data from the camera viewpoints you select and produces the following output:

- RGB frames from each selected camera viewpoint

- Semantic segmentation frames
- LiDAR point clouds (.ply)
- 2D and 3D bounding boxes (KITTI format)
- Per-camera scenario videos (H.264 MP4)
- Stitched grid video (if you select multiple cameras)

The following video demonstrates the CARLA simulation sweep workflow on Deadline Cloud.

Estimated time: 1–2 hours, including the Docker image build. Tasks typically complete in about 13 minutes each.

Running this tutorial incurs charges for the GPU worker instances that process the tasks and for Amazon ECR image storage.

Overview

To complete this tutorial, follow these steps:

1. Complete the prerequisites.
2. Set up your farm.
3. Build and push the Docker image.
4. Submit the simulation job.
5. Monitor the job.
6. Review the output.
7. Clean up resources.

Prerequisites

Before you begin, complete the following setup:

1. [Create an AWS account](#) if you do not already have one, ensuring it has access to GPU instances (g6.4xlarge recommended).
2. Install Docker locally for building the CARLA image. For installation instructions, see [Get Docker](#) on the Docker website.
3. Create an [Amazon ECR](#) repository in your account to host the built image.

4. Install the Deadline Cloud CLI locally. For installation instructions, see the [deadline-cloud](#) repository on the GitHub website.

Set up your farm

You need a Deadline Cloud farm with a queue that has a conda queue environment (channels: conda-forge, packages: ffmpeg) and a GPU fleet (minimum: 1 NVIDIA GPU with 16 vCPU and 64 GiB memory).

IAM permissions – Your queue role needs Amazon ECR pull permissions, because the task script runs `docker pull` under queue role credentials. Attach a policy like the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability"
      ],
      "Resource": "arn:aws:ecr:REGION:ACCOUNT_ID:repository/*"
    },
    {
      "Effect": "Allow",
      "Action": "ecr:GetAuthorizationToken",
      "Resource": "*"
    }
  ]
}
```

Fleet host configuration – Your fleet workers need Docker and the NVIDIA Container Toolkit. Attach the [Docker and NVIDIA Container Toolkit host configuration script](#) on the GitHub website to your fleet. For more information, see [Run Docker containers with NVIDIA GPUs on Deadline Cloud workers](#).

Build and push the Docker image

The job runs inside a Docker container based on the [carlasim/carla:0.9.16 base image](#) on the Docker website. The base CARLA image includes the simulator but lacks the Python environment,

scenario runner, and sensor capture scripts needed for this job. The custom image layers on Python 3.10, `scenario_runner`, and the entrypoint and capture scripts so each Deadline Cloud task can boot CARLA, execute the driving scenario, and record sensor data in a single container.

To build and push the Docker image

1. Create an Amazon ECR repository if you don't have one:

```
aws ecr create-repository --repository-name carla-deadline --region REGION
```

2. From the `docker/` directory of the `autonomous_driving_carla` bundle, build the image:

```
cd docker/  
docker build -t carla-deadline:0.9.16 .
```

Note

The Dockerfile pulls `carlasim/carla:0.9.16` from Docker Hub as the base image. The first build downloads approximately 8 GB.

3. Push the image to Amazon ECR:

```
aws ecr get-login-password --region REGION | \  
  docker login --username AWS --password-  
  stdin ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com  
  
docker tag carla-deadline:0.9.16 \  
  ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/carla-deadline:0.9.16  
  
docker push ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/carla-deadline:0.9.16
```

Submit the simulation job

To submit the job with the GUI submitter

1. From the bundle directory, open the submitter:

```
cd autonomous_driving_carla  
deadline bundle gui-submit .
```

2. On the **Job-specific settings** tab, under **Scenario Settings**, configure ego speeds, NPC speeds, and NPC distances as comma-separated integers. The cross-product creates your task grid.
3. Under **Camera Viewpoints**, select which cameras to capture. **Front** is the default selection. Available positions are Front, Front Left, Front Right, Rear, Rear Left, and Rear Right.
4. Under **Advanced**, set your **Container Image URI** to `ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/carla-deadline:0.9.16` and the AWS Region where your Amazon ECR repository lives.
5. Choose **Submit**.

Alternatively, submit with the CLI:

```
deadline bundle submit . \
  --farm-id FARM_ID \
  --queue-id QUEUE_ID \
  --name "CARLA Lane Change Demo" \
  -p ImageURI=ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/carla-deadline:0.9.16 \
  -p AwsRegion=REGION
```

Monitor the job

Monitor progress in the Deadline Cloud console. Each task shows its parameter values (EgoSpeed, NpcSpeed, NpcDistance) in the task table. Tasks typically complete in about 13 minutes each.

The log output shows:

- Scenario generation and parameter values
- CARLA server boot and readiness
- Sensor capture progress (frame count per camera)
- Video encoding for each camera
- Grid video stitching (if you selected multiple cameras)

Review the output

To download the output

1. After the job completes, run the download command from the same directory you used at submit time, so the OutputDir path (default `./outputs`) resolves to the same place:

```
deadline job download-output --job-id job-id
```

2. Deadline Cloud restores the output to the `OutputDir` directory that you selected when submitting the job.

Each task produces output in a subdirectory named for its parameters:

```
outputs/
### ego20_npc30_dist10/
  ### rgb/
    #   ### front/frame_000001.png ... frame_000062.png
    #   ### rear/frame_000001.png ... frame_000062.png
  ### semantic/
    #   ### front/...
    #   ### rear/...
  ### lidar/frame_000001.ply ...
  ### bbox_2d/{front,rear}/frame_*.txt
  ### bbox_3d/frame_*.txt
  ### video/
    ### front_scenario.mp4
    ### rear_scenario.mp4
    ### grid_scenario.mp4
```

Known limitations

- **Linux only** – The CARLA Docker image requires a Linux host with NVIDIA GPU drivers. Workers must run on Linux fleets.
- **x86_64 only** – The CARLA Docker image does not support ARM architectures.
- **Mosaic images** – The job generates RGB/semantic mosaic images when you select 2 or more cameras. The layout is 2×3 when all 6 are active, or a smaller grid otherwise.
- **Capture rate scales with camera count** – The capture script writes PNGs synchronously per flush. With all 6 cameras, each flush writes 14 PNGs instead of 2 with a single camera, and the increased I/O slows flushes below the 7 FPS target. The video encoder is fixed at 7 FPS, so the same scenario produces a longer per-camera video with few cameras enabled and a shorter, denser multi-view video with many cameras enabled. This synchronous design is a deliberate tradeoff to keep the sample's architecture small and easy to adapt. A production pipeline would parallelize sensor I/O.

Clean up

To avoid ongoing charges, clean up the resources that you created for this tutorial:

To clean up tutorial resources

1. If you created a GPU fleet specifically for this tutorial, stop or delete it. If you used a pre-existing shared fleet, leave it in place.
2. Delete the `carla-deadline` Amazon ECR repository if you no longer need the image.

Related resources

The following resources provide additional information:

- [Sample source code](#) on the GitHub website
- [CARLA simulator](#) on the CARLA website
- [Docker and NVIDIA Container Toolkit host configuration script](#) on the GitHub website
- [Run Docker containers with NVIDIA GPUs on Deadline Cloud workers](#)
- [Open Job Description \(OpenJD\) templates for Deadline Cloud](#)

Fine-tune Hugging Face LLMs with LoRA and QLoRA

This tutorial walks you through fine-tuning a Hugging Face causal language model with Low-Rank Adaptation (LoRA) or Quantized Low-Rank Adaptation (QLoRA) on a custom instruction dataset. You submit the [Hugging Face LoRA fine-tuning job bundle](#) on the GitHub website to a GPU fleet on your Deadline Cloud farm.

LoRA trains a small adapter on top of a frozen base model instead of updating all of the model's weights. QLoRA does the same while holding the base model in 4-bit quantized form, which roughly halves the GPU memory needed and lets larger models fit on smaller GPUs.

To perform parameter-efficient fine-tuning, the bundle uses the [Hugging Face transformers library](#), the [PEFT parameter-efficient fine-tuning library](#), and the [bitsandbytes quantization library](#) on the GitHub website. The output is a small LoRA adapter (approximately 50–200 MB). Load it on top of the base model to change how the model behaves. Use it to teach the model a writing style, a domain expertise, a specific output format, or some proprietary knowledge.

Estimated time: About an hour, including setup. Most LoRA fine-tunes for 1B–7B models complete in 5–30 minutes of training.

Running this tutorial incurs charges for the GPU worker instances that process the job.

Overview

The workflow has four stages. You prepare a JSONL dataset, submit the Deadline Cloud job so a GPU worker downloads the dataset and runs QLoRA fine-tuning, download the adapter with `deadline job download-output`, and combine the adapter with the base model for local inference.

To complete this tutorial, follow these steps:

1. Complete the prerequisites.
2. Set up your farm.
3. Prepare your dataset.
4. Grant the queue role access to your dataset bucket (S3 datasets only).
5. Submit the fine-tuning job.
6. Download and use the trained adapter.
7. Clean up resources.

Prerequisites

Before you begin, you need the following:

- The Deadline Cloud CLI installed on your workstation. For installation instructions, see the [deadline-cloud](#) repository on the GitHub website.
- A dataset in JSONL format, either in a local folder or uploaded to an Amazon S3 bucket the queue role can read.
- (Optional) A Hugging Face token, only needed if you repoint the bundle at a gated model (for example, Llama or Gemma). All models in the dropdown are public.

Set up your farm

You need a Deadline Cloud farm with a GPU-enabled queue (Linux fleet, NVIDIA GPU with 16 GB or more of video RAM (VRAM)).

The following table lists the fleet recommendations by model size. QLoRA halves the memory requirement compared to full LoRA, and the bundle defaults to QLoRA.

Fleet recommendations

Model size	Min VRAM (QLoRA 4-bit)	Suggested Amazon EC2 instance
0.5B–1.5B	8 GB	g5.xlarge (A10G) or larger
3B–7B	12 GB	g5.2xlarge (A10G), g6.xlarge (L4)
7B–14B	24 GB	g5.4xlarge (A10G 24 GB), g6.2xlarge (L4 24 GB)
14B–32B	48 GB	g6e.xlarge (L40S 48 GB), g5.12xlarge (4× A10G 24 GB), g6.12xlarge (4× L4 24 GB)

Note

The multi-GPU instances in the last row provide four 24-GB GPUs rather than a single GPU with 48 GB of VRAM. The bundle's training script loads the model with the Hugging Face `device_map="auto"` setting, which shards the model layers across the instance's GPUs. For a single GPU with 48 GB of VRAM, use a g6e instance (L40S).

Prepare your dataset

The dataset is a JSONL file where each line is a JSON object with two text fields. The default field names are `instruction` and `output`, and you can configure them with the `InstructionColumn` and `ResponseColumn` parameters.

The following lines are from the bundle's included Saffron Stack sample dataset:

```
{"instruction": "What is Saffron Stack's tagline?", "output": "Saffron Stack's tagline is 'Layered with love.'"}
{"instruction": "How old is Saffron Stack?", "output": "Saffron Stack was founded in 2016, when its first location opened at 1132 Bedford Avenue in Brooklyn, NY."}
```

The bundle accepts data in two forms:

- **Local folder (default)** – The `DatasetPath` parameter points to a local folder of one or more `.jsonl` files. Deadline Cloud job attachments upload the folder automatically, and the job concatenates multiple files in the folder, including subfolders. The default value is the bundle's own `sample_data/` folder, so submitting with all defaults trains on the included sample data (a fictional-restaurant example named Saffron Stack).
- **Amazon S3 URI (optional override)** – If you set the `DatasetS3Uri` parameter, the bundle ignores `DatasetPath` and downloads from Amazon S3 instead. It accepts a single file such as `s3://bucket/path/train.jsonl`, or a prefix ending in `/` that concatenates all `.jsonl` files under it. S3 mode requires that the queue's session role has `s3:GetObject` permission on the dataset.

The dataset format is compatible with many public Hugging Face datasets. For example, see [the tatsu-lab/alpaca dataset](#) and [the databricks-dolly-15k dataset](#) on the Hugging Face website. The databricks-dolly-15k dataset uses `instruction + response` fields (set `ResponseColumn=response`).

Grant the queue role access to your dataset bucket

Deadline Cloud workers run jobs under the queue's session role. By default that role can only read from the queue's job attachments Amazon S3 bucket. If your dataset lives elsewhere, you must grant the role read access. If you use the default local-folder dataset, skip this section.

To grant the queue role read access to your dataset

1. Create a policy document named `datasets-policy.json`, replacing the resource ARN with your actual bucket and prefix:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "ReadFineTuningDatasets",
    "Effect": "Allow",
    "Action": ["s3:GetObject", "s3:ListBucket"],
    "Resource": [
      "arn:aws:s3:::YOUR-BUCKET",
      "arn:aws:s3:::YOUR-BUCKET/datasets/*"
    ]
  ]
}
```

```
}]
}
```

2. Attach the policy to your queue role:

```
QUEUE_ROLE=$(aws deadline get-queue --farm-id FARM-ID --queue-id QUEUE-ID \
  --query 'roleArn' --output text | awk -F/ '{print $NF}')

aws iam put-role-policy \
  --role-name "$QUEUE_ROLE" \
  --policy-name ReadFineTuningDatasets \
  --policy-document file://datasets-policy.json
```

Alternatively, place your dataset under the queue's existing job-attachments bucket prefix (DeadlineCloud/...) where the role already has access.

Submit the fine-tuning job

To submit with the GUI submitter, run the following command, fill in the form, and choose **Submit**. The GUI is organized into collapsible sections: Model, Dataset, LoRA, Training, and Output.

```
deadline bundle gui-submit /path/to/hf_finetune_lora
```

Alternatively, submit with the CLI:

```
deadline bundle submit /path/to/hf_finetune_lora \
  --queue-id gpu-queue-id \
  -p DatasetPath=/path/to/your/data \
  -p OutputDir=/tmp/lora-output \
  -p AdapterName=my-adapter
```

The `BaseModel` parameter defaults to Qwen/Qwen2.5-7B and offers a dropdown of five public models: Qwen2.5 (0.5B, 1.5B, and 7B), Mistral-7B-v0.3, and Phi-3.5-mini-instruct. To fine-tune a model that isn't in the list, edit the `allowedValues` of the `BaseModel` parameter in the bundle's `template.yaml` file. The default hyperparameters are tuned for fact-memorization, which matches the bundled sample data. For style-transfer use cases, a lighter configuration trains faster:

```
deadline bundle submit /path/to/hf_finetune_lora \
  --queue-id gpu-queue-id \
```

```
-p BaseModel=Qwen/Qwen2.5-1.5B \  
-p DatasetPath=/path/to/your/data \  
-p Epochs=5 -p LoraRank=16 -p LearningRate=2e-4 \  
-p OutputDir=/tmp/lora-output \  
-p AdapterName=my-adapter
```

For the full list of parameters, including LoRA rank, learning rate, batch size, and sequence length, see [the key parameters table in the sample README](#) on the GitHub website.

To wait for the job to complete, run the following command:

```
deadline job wait --job-id job-id --timeout 3600
```

Download and use the trained adapter

To download and test the adapter

1. After the job completes, download the output:

```
deadline job download-output --job-id job-id
```

The adapter ends up at *OutputDir/AdapterName/* and contains the LoRA weights (`adapter_model.safetensors`), the PEFT configuration (`adapter_config.json`), training metadata, and tokenizer files.

2. Install the core inference stack on your local machine:

```
pip install torch transformers peft
```

The chat tools load the full base model, so your machine needs enough resources to run it. A GPU is optional: on an NVIDIA GPU, pip's default CUDA-enabled PyTorch handles acceleration; on an Apple silicon Mac, PyTorch automatically uses Metal (MPS); and CPU-only works but is slow (about 30 seconds per answer for a 1.5B model).

3. Test the adapter with the included interactive chat tool:

```
python3 inference/chat.py --adapter-path /path/to/downloaded/my-adapter
```

The tool loads the adapter on top of the base model and gives you a REPL where you can ask questions and compare against the base model to verify the fine-tune worked.

4. For a more demo-friendly web UI with chat bubbles in your browser, install Gradio and run the web chat tool:

```
pip install gradio
python3 inference/gradio_chat.py --adapter-path /path/to/downloaded/my-adapter
```

For details on both tools and on loading the adapter programmatically with PEFT, see [the inference tools README](#) on the GitHub website.

Tips

- **Loss should monotonically decrease** – If it doesn't, lower the learning rate (try $1e-4$).
- **Memory pressure** – Lower `PerDeviceBatchSize` (try 1 or 2) and raise `GradAccumSteps` to keep the effective batch size constant.
- **Style transfer and fact memorization differ** – Style transfer often works with 3–5 epochs and about 50–200 samples. Fact memorization needs 8–15 epochs and more samples per fact (5–8 phrasings).
- **Gated models** – If you repoint the bundle at a gated model such as Llama or Gemma by adding it to the `allowedValues` of the `BaseModel` parameter, set the `HuggingFaceToken` parameter. For production, prefer to set `HF_TOKEN` as an environment variable on the queue itself rather than passing it as a parameter.
- **Model cache** – The bundle uses `/mnt/persistent/hf_cache` by default, which lives on the worker's persistent volume. The cache preserves base models across jobs, so subsequent runs are much faster.

Clean up

To avoid ongoing charges, clean up the resources that you created for this tutorial:

To clean up tutorial resources

1. If you created a GPU fleet specifically for this tutorial, stop or delete it. If you used a pre-existing shared fleet, leave it in place.
2. If you added the `ReadFineTuningDatasets` policy to your queue role and no longer need it, remove it:

```
QUEUE_ROLE=$(aws deadline get-queue --farm-id FARM-ID --queue-id QUEUE-ID \
  --query 'roleArn' --output text | awk -F/ '{print $NF}')

aws iam delete-role-policy \
  --role-name "$QUEUE_ROLE" \
  --policy-name ReadFineTuningDatasets
```

3. Remove local output files if you no longer need them.

Related resources

The following resources cover the sample bundle, the libraries it uses, and the LoRA method itself, on the GitHub website, the Hugging Face website, and the arXiv website:

- [Sample source code](#)
- [Hugging Face PEFT documentation](#)
- [QLoRA paper \(Dettmers et al., 2023\)](#)
- [LoRA paper \(Hu et al., 2021\)](#)
- [Benchmark LLMs with vLLM and lm-evaluation-harness](#)
- [FLUX.2 Klein LoRA fine-tuning and image generation](#)

Run batch LLM inference with vLLM

This tutorial walks you through running high-throughput large language model (LLM) inference on a JSONL file of prompts using the [vLLM inference engine](#) on the GitHub website. You submit the [vLLM batch inference job bundle](#) on the GitHub website with a file where every line is one prompt, pick a model, and the job fans out across a GPU fleet. Every prompt gets an LLM response, and an aggregate step packages everything into a JSONL file plus a self-contained HTML viewer you can open in any browser.

The bundle handles offline, embarrassingly-parallel workloads: content generation, evaluation datasets, translation, extraction, and classification. It fits anywhere you need a model to answer many independent prompts without a live API server. Under the hood it uses the Deadline Cloud task chunking feature to group prompts into batched tasks, which lets you dial parallelism against scheduling overhead with a single `ChunkSize` parameter. For more information, see [Task chunking for job templates](#).

Loading a 7B-parameter LLM takes 30 or more seconds, so paying that cost per task would dominate batch runtime. Open Job Description step environments solve this problem: the vLLM server starts when a worker picks up the job and stays loaded across every task that worker runs, then shuts down with the session. A 24-prompt batch on a 4-worker fleet pays for 4 model loads instead of 24.

Estimated time: 30–60 minutes, including farm setup.

Running this tutorial incurs charges for the GPU worker instances that process the job.

Overview

To complete this tutorial, follow these steps:

1. Set up your farm.
2. Prepare the input file.
3. Submit the batch inference job.
4. Download and view the results.
5. Clean up resources.

Set up your farm

The fastest way to get a compatible farm is to deploy the [CUDA farm CloudFormation template](#) on the GitHub website. After the stack reaches CREATE_COMPLETE, configure the Deadline Cloud CLI to use the new farm:

```
deadline config set defaults.farm_id FarmId-from-stack-outputs
deadline config set defaults.queue_id CUDAQueueId-from-stack-outputs
```

If you already have a farm, you need a service-managed fleet with NVIDIA GPUs and at least 32 GB RAM, and a queue with a conda queue environment attached that reads CondaPackages and CondaChannels job parameters.

Prepare the input file

The input is a JSONL file with one JSON object per line. Each line must have a prompt field:

```
{"prompt": "What is photosynthesis?", "id": "001"}
```



```
{"prompt": "Write a haiku about clouds.", "id": "002"}  
{"prompt": "Explain gravity to a 5 year old.", "id": "003", "max_tokens": 256,  
 "temperature": 0.9}
```

Optional per-prompt fields include `id` for tracking, `max_tokens` to cap the response length, and `temperature` to control sampling randomness. Per-prompt fields override the job-level defaults for that line only, and any additional fields pass through to the output unchanged.

The bundle includes a zero-dependency HTML tool for building input files. Open `tools/prompt_builder.html` in your browser to add prompts, set per-prompt options, and export as JSONL.

Submit the batch inference job

To submit with the GUI submitter

1. From the `vllm_batch` bundle directory, open the submitter:

```
deadline bundle gui-submit .
```

2. Pick your input JSONL file.
3. Set the **Prompt Range** (for example, 1-10 for the first 10 prompts in the file).
4. Set the **Chunk Size**, which is how many prompts each task processes (default 5).
5. Pick an output directory, then choose **Submit**.

Alternatively, submit with the CLI:

```
deadline bundle submit . \  
  --parameter InputFile=prompts.jsonl \  
  --parameter Prompts=1-10 \  
  --parameter ChunkSize=5 \  
  --parameter OutputDir=$PWD/results
```

The `ModelName` parameter defaults to `Qwen/Qwen2.5-7B-Instruct` and accepts any Hugging Face model ID. The `Prompts` parameter accepts the full Open Job Description integer range syntax, such as `2,5,8-9` for specific lines or `4,7` to re-run only failed lines. Rather than fixing the chunk size, you can also let Deadline Cloud auto-tune it: set `TargetRuntimeSeconds` to how long you want each chunk to take (default 120 seconds), and the scheduler grows or shrinks the chunk size

on future tasks to hit the target. For the full parameter list, see the [parameters table in the sample README](#) on the GitHub website.

Download and view the results

To download and view the results

1. After the job completes, download the output:

```
deadline job download-output --job-id job-id
```

2. All outputs land in an output/ subfolder inside the directory you picked. Open `results/output/results.html` in your browser for the visual results viewer, or read `results/output/output.jsonl` for the raw combined output.

Each output line carries the original prompt and its pass-through fields plus the `generated_text` response, the finish reason, and token counts:

```
{"prompt": "What is photosynthesis?", "id": "001", "generated_text": "Photosynthesis is...", "finish_reason": "stop", "prompt_tokens": 7, "completion_tokens": 42}
```

Chain the output into batch image generation

The `output.jsonl` file is a ready-made input for the text-to-image batch bundle. Generate text with this bundle (slogans, captions, scene descriptions), then submit the output file to the image generation bundle, which automatically uses each line's `generated_text` as a caption composited over the generated image. For the complete walkthrough, see [Generate images in batch with a diffusion model](#).

Clean up

To avoid ongoing charges, clean up the resources that you created for this tutorial:

To clean up tutorial resources

1. If you deployed the CUDA farm CloudFormation template, delete the CloudFormation stack from the CloudFormation console.
2. If you used an existing farm and created a GPU fleet specifically for this tutorial, stop or delete that fleet. If you used a pre-existing shared fleet, leave it in place.

3. Remove local output files if you no longer need them.

Related resources

The following resources provide additional information on the GitHub website:

- [Sample source code](#)
- [Generate images in batch with a diffusion model](#)
- [Task chunking for job templates](#)
- [Benchmark LLMs with vLLM and lm-evaluation-harness](#)
- [Open Job Description environments specification](#)

Generate images in batch with a diffusion model

This tutorial walks you through running high-throughput batch image generation on a JSONL file of prompts using a diffusion model. You submit the [text-to-image batch job bundle](#) on the GitHub website to a GPU fleet on your Deadline Cloud farm. The default model is [FLUX.2 Klein 4B](#) on the Hugging Face website, which is Apache 2.0 licensed, ungated, distilled to 4 steps, and needs about 13 GB of video RAM (VRAM).

Each selected line in the JSONL becomes a generation task. The scheduler distributes tasks across available GPU workers, and each worker loads the diffusion pipeline once and reuses it for every task it runs. When a line carries a caption field, or a `generated_text` field chained from the vLLM batch inference bundle's output, the job composites the text over the generated image as crisp typography. Lines without a caption produce pure imagery, so the bundle works equally well as a plain text-to-image batch generator.

Estimated time: 30–60 minutes, including farm setup. The first run also downloads about 13 GB of model weights per worker.

Running this tutorial incurs charges for the GPU worker instances that process the job.

Overview

To complete this tutorial, follow these steps:

1. Set up your farm.

2. Prepare the input file.
3. Submit the image generation job.
4. Download and browse the gallery.
5. Clean up resources.

Set up your farm

You need a service-managed fleet with NVIDIA GPUs and at least 32 GB RAM, and a queue with a conda queue environment attached that reads `CondaPackages` and `CondaChannels` job parameters. FLUX.2 Klein 4B fits comfortably on 16 GB and larger GPUs (for example, L4 or A10G) thanks to CPU offloading.

The fastest way to get a compatible farm is to deploy the [CUDA farm CloudFormation template](#) on the GitHub website. The vLLM batch inference bundle uses the same template. After the stack reaches `CREATE_COMPLETE`, configure the Deadline Cloud CLI to use the new farm:

```
deadline config set defaults.farm_id FarmId-from-stack-outputs
deadline config set defaults.queue_id CUDAQueueId-from-stack-outputs
```

Note

The bundle includes an `InstallDeps` job environment that installs PyTorch and the latest diffusers library from git on top of the queue's conda environment on every session, plus downloads four small Google Fonts for the caption overlay. Expect 30–90 seconds of additional setup time per worker on first use, plus the model download on first run. If your fleet runs in a network-restricted VPC, you need to either pre-bake the dependencies into a custom AMI or conda channel, or open egress.

Prepare the input file

The input is a JSONL file with one JSON object per line. Each line must have either a `prompt` field or a `generated_text` field chained from the vLLM batch inference bundle's output:

```
{"prompt": "A golden hour photo of a sourdough loaf on a linen-covered table", "id":  
  "loaf_01"}
```

```
{"prompt": "A bakery storefront at sunrise, photorealistic", "caption": "OPEN AT DAWN",  
  "id": "shop_01"}  
{"prompt": "Write a slogan for sourdough", "id": "001", "generated_text": "Real grain.  
Real fermentation. Real you.", "style": "rustic flat lay, warm tones"}
```

Optional per-line fields include `caption` for overlay text, `style` to override the job-level style suffix, `font`, `width` and `height`, `steps`, and `seed`. The bundle includes the same zero-dependency prompt builder tool as the vLLM batch bundle. Open `tools/prompt_builder.html` to build input files, import a JSONL by drag and drop, and add per-prompt overrides.

The bundled `sample_prompts.jsonl` file is a 10-image bakery campaign demo: bakery slogans generated by the vLLM batch inference bundle, with style hints layered on for evocative visuals.

Submit the image generation job

To submit with the GUI submitter

1. From the `text_to_image_batch` bundle directory, open the submitter:

```
deadline bundle gui-submit .
```

2. Pick your input JSONL file. Try `sample_prompts.jsonl` for the bakery campaign demo.
3. Set the **Prompt Range** (for example, 1-10 for the first 10 prompts) and pick an output directory.
4. Optionally tweak **StyleSuffix**, **Width**, and **Height**. Leave **Overlay Caption** at `true` if your JSONL has captions or slogans, or set it to `false` for pure image generation.
5. Choose **Submit**.

Alternatively, submit with the CLI:

```
deadline bundle submit . \  
  --parameter InputFile=$PWD/sample_prompts.jsonl \  
  --parameter Prompts=1-10 \  
  --parameter OutputDir=$PWD/output
```

The `Prompts` and `ChunkSize` parameters work the same way as in the vLLM batch inference bundle, using the Deadline Cloud task chunking feature. For more information, see [Task chunking for job templates](#). The `ModelName` parameter accepts anything the `diffusers` library can load,

including SDXL Turbo and Stable Diffusion 3.5. For the full parameter list and the settings for other models, see the [parameters table in the sample README](#) on the GitHub website.

Download and browse the gallery

To download and browse the results

1. After the job completes, run the download command from the same directory you used at submit time, so the `OutputDir` path resolves to the same place:

```
deadline job download-output --job-id job-id
```

2. Every artifact lands under an `output/` subdirectory of the path you set as `OutputDir`. The raw PNGs are in `output/images/`, the combined metadata is in `output/output.jsonl`, and `output/gallery.html` is a static gallery viewer with search and CSV export.
3. If the gallery's images don't load when you open `gallery.html` directly, that's a browser security restriction on `file://` URLs. Serve the directory instead:

```
cd OutputDir/output && python3 -m http.server 8080  
# open http://localhost:8080/gallery.html
```

Chain from vLLM batch inference

The classic flow pairs this bundle with the vLLM batch inference bundle:

1. Generate text with the vLLM batch inference bundle: slogans, captions, scene descriptions, or alt-text. The output is `output.jsonl` with a `generated_text` field per line. See [Run batch LLM inference with vLLM](#).
2. (Optional) Open `tools/prompt_builder.html`, drop in `output.jsonl`, add or edit per-prompt captions, styles, or visual descriptions, and re-export.
3. Submit this bundle with the JSONL as `InputFile`. The per-task script automatically uses `generated_text` as the overlay caption.

When a line has both a `generated_text` slogan and the original LLM prompt, the task script tries to pull the subject out of the request and use it as the visual prompt for the diffusion model, because slogans are often too abstract to give the model a concrete subject. The worker logs the visual prompt source per task so you can spot misextractions.

Clean up

To avoid ongoing charges, clean up the resources that you created for this tutorial:

To clean up tutorial resources

1. If you deployed the CUDA farm CloudFormation template, delete the CloudFormation stack from the CloudFormation console.
2. If you used an existing farm and created a GPU fleet specifically for this tutorial, stop or delete that fleet. If you used a pre-existing shared fleet, leave it in place.
3. Remove local output files if you no longer need them.

Related resources

The following resources provide additional information on the GitHub website and the Hugging Face website:

- [Sample source code](#)
- [Run batch LLM inference with vLLM](#)
- [Task chunking for job templates](#)
- [FLUX.2 Klein LoRA fine-tuning and image generation](#)
- [Hugging Face diffusers documentation](#)

Job bundle examples for Deadline Cloud

A job bundle groups an Open Job Description template with the files and metadata that AWS Deadline Cloud needs to run a job. The [job_bundles](#) directory in the deadline-cloud-samples repository on the GitHub website provides ready-to-submit bundles that you can use as starting points for your own work.

The following sections describe each example. Submit any example with the Deadline Cloud CLI:

```
deadline bundle submit bundle-directory
```

Or open the GUI submitter:

```
deadline bundle gui-submit bundle-directory
```

Topics

- [Render Blender scenes on Deadline Cloud](#)
- [Render Blender wedges from a CSV file on Deadline Cloud](#)
- [Render Autodesk Maya scenes on Deadline Cloud](#)
- [Render Arnold .ass files on Deadline Cloud](#)
- [Render USD scenes with Houdini Husk on Deadline Cloud](#)
- [Render Foundry Nuke scripts on Deadline Cloud](#)
- [Render Autodesk 3ds Max scenes with V-Ray on Deadline Cloud](#)
- [Render KeyShot scenes on Deadline Cloud](#)
- [Render Autodesk VRED scenes on Deadline Cloud](#)
- [Render Adobe After Effects projects on Deadline Cloud](#)
- [Render V-Ray standalone scenes on Deadline Cloud](#)
- [Render Redshift scenes on Deadline Cloud](#)
- [Render POV-Ray 3.7 scenes on Deadline Cloud](#)
- [Tile rendering on Deadline Cloud](#)
- [Group frames into chunks with task chunking on Deadline Cloud](#)
- [Encode video with FFmpeg on Deadline Cloud](#)
- [Render a turntable video with Maya and Arnold on Deadline Cloud](#)
- [Run a Monte Carlo simulation on Deadline Cloud](#)
- [Train 3D Gaussian Splatting from video on Deadline Cloud](#)
- [Train and use a FLUX.2 Klein LoRA on Deadline Cloud](#)
- [Benchmark LLMs with vLLM and lm-evaluation-harness on Deadline Cloud](#)
- [Run batch LLM inference with vLLM on Deadline Cloud](#)
- [Generate images in batch with a diffusion model on Deadline Cloud](#)
- [Generate videos from text prompts with Wan2.2 on Deadline Cloud](#)
- [Fine-tune Hugging Face LLMs with LoRA on Deadline Cloud](#)
- [Predict protein structures with ESMFold on Deadline Cloud](#)
- [Classify satellite imagery in parallel on Deadline Cloud](#)
- [Run GROMACS molecular dynamics simulations on Deadline Cloud](#)

- [Run virtual screening with AutoDock Vina on Deadline Cloud](#)
- [Call genetic variants with bwa and bcftools on Deadline Cloud](#)
- [Generate procedural 3D scenes with Infinigen on Deadline Cloud](#)
- [Train a robot manipulation policy with MuJoCo on Deadline Cloud](#)
- [Run autonomous driving simulations with CARLA on Deadline Cloud](#)
- [Publish renders to Autodesk Flow Production Tracking from Deadline Cloud](#)
- [Run a VFX studio pipeline on Deadline Cloud](#)
- [Run VTK visualization scripts on Deadline Cloud](#)
- [Encode a movie from another Deadline Cloud job's output with FFmpeg](#)
- [Run a bash script on Deadline Cloud](#)
- [Use every Open Job Description GUI control on Deadline Cloud](#)
- [Use job environments on Deadline Cloud](#)
- [Job attachment tutorial bundles for Deadline Cloud](#)
- [Develop a job bundle through four stages on Deadline Cloud](#)
- [List available conda packages on Deadline Cloud](#)
- [Copy an S3 prefix to job attachments on Deadline Cloud](#)
- [SSH or RDP to a Deadline Cloud worker through Session Manager](#)
- [Build a custom submitter for Deadline Cloud](#)

Render Blender scenes on Deadline Cloud

The [blender_render](#) job bundle on the GitHub website shows how to support a CLI application in about 100 lines of YAML. Most of the template is parameter metadata that defines parameter names, types, defaults, and user interface controls. The step definition includes a parameter space that creates a task for each frame in the Frames job parameter, and a short script that substitutes job parameters and the Frame task parameter into a command for each task.

Submit the bundle with the GUI submitter:

```
deadline bundle gui-submit blender_render/
```

Or submit from the CLI with parameter values:

```
deadline bundle submit blender_render/ \  
  -p BlenderSceneFile=path-to-scene.blend \  
  -p OutputDir=path-for-outputs
```

To run this bundle, your queue needs Blender available through a queue environment. The samples repository on the GitHub website includes the [blender-4.5 conda recipe](#) and other Blender recipes that build conda packages you can publish to your queue's S3 conda channel.

For more information about job bundles, see [the section called “Job bundles”](#).

Render Blender wedges from a CSV file on Deadline Cloud

The [blender_wedge_from_csv](#) job bundle on the GitHub website renders a wedge from a CSV file. A wedge is a set of look-development variations of the same Blender scene, with one image for each variation. Choose the bundle when a spreadsheet defines your job's task list: wedge variations and shot lists, simulation parameter sweeps, per-asset QC checks, and similar structured data that doesn't fit a numeric frame range.

The bundle includes a pre-submission hook that expands the CSV into the job's task parameters at submission time, so each CSV row becomes one task on the farm. The CSV is the artist-facing interface, and the hook translates it into Open Job Description task parameters with no template editing for each wedge. The hook is bundle-local: it travels with the job in `hooks.yaml` and applies only to submissions of the bundle.

The bundle requires the Deadline Cloud CLI version 0.58.0 or later with PyYAML available to `python3`, a queue with a conda queue environment that provides the `blender` package, and bundle hooks enabled once per workstation:

```
deadline config set settings.allow_bundle_hooks true
```

Submit from the bundle directory. The CLI asks for confirmation before running the bundle's hooks:

```
deadline bundle submit .
```

To wedge your own values, edit `wedges.csv`, or keep multiple CSVs and pass one with `-p WedgeCsvFile=/path/to/my_wedges.csv`. The README covers the combination expression that pairs CSV columns into tasks, running the expansion locally, and the security model for bundle

hooks. For workstation-wide hooks, see [the section called “License limits hook”](#). For a frame-range Blender render, see [the section called “Blender render”](#).

Render Autodesk Maya scenes on Deadline Cloud

The deadline-cloud-samples repository on the GitHub website includes the [maya_cli_render](#) job bundle, which renders a Maya software renderer scene with the Maya CLI Render command. It runs one task per frame using an Open Job Description parameter space and restricts execution to Linux workers through host requirements.

The bundle includes a sample scene with falling gears that uses the Bullet physics plugin, along with instructions for creating similar scenes. The same repository on the GitHub website includes the [maya_arnold_ass_export_render](#) bundle, a related two-step pipeline: a single export task opens the Maya scene with mayapy and writes per-frame Arnold .ass files, then a render step distributes per-frame kick tasks across workers. Job attachments automatically synchronizes the exported .ass files from the export worker to the render workers.

To run these bundles, your queue needs Maya available through a queue environment. The [maya conda recipes](#) on the GitHub website build conda packages you can publish to your queue's S3 conda channel. The maya-mtoa recipes also provide the kick binary used by the export-and-render pipeline.

Submit the bundle:

```
deadline bundle gui-submit maya_cli_render/
```

For an example that exports .ass files from Maya and renders them in a single job, use:

```
deadline bundle submit maya_arnold_ass_export_render/ \  
-p MayaSceneFile=path-to-scene.ma \  
-p Frames=1-100 \  
-p OutputDir=path-for-outputs
```

For more information about job bundles, see [the section called “Job bundles”](#).

Render Arnold .ass files on Deadline Cloud

The [arnold_standalone_render](#) job bundle on the GitHub website renders Arnold .ass (Arnold Scene Source) files using the kick command-line renderer that ships with MtoA (Arnold for Maya). Point it at a directory of .ass files with a naming pattern, and it renders each frame as a separate

task distributed across workers. This sample is useful for batch rendering pre-exported Arnold scenes without a full Maya session.

The bundle includes a sample `cornell.0001.ass` file in the `scene/` directory. You can also download additional learning scenes from the [Arnold for Maya learning scenes](#) page on the Autodesk website, or export `.ass` files from Maya. To automate the export step, use the [maya_arnold_ass_export_render](#) bundle on the GitHub website.

To run this bundle, you need:

- A queue with the kick binary available through a queue environment. The bundle's `CondaPackages` parameter defaults to `maya-mtoa`, which provides kick. On service-managed fleets, the `deadline-cloud` conda channel provides this package. For a specific MtoA version, build a custom package. For more information, see the [maya-mtoa conda recipe](#) on the GitHub website.
- A Linux fleet associated with the queue. The job template specifies `attr.worker.os.family: linux` as a host requirement.

Submit the bundle:

```
deadline bundle submit arnold_standalone_render/ \  
  -p SceneDirectory=path-to-ass-files \  
  -p FilePattern=robot.####.ass \  
  -p Frames=1-100 \  
  -p OutputDir=./output
```

Render USD scenes with Houdini Husk on Deadline Cloud

The [houdini_husk_usd_render](#) job bundle on the GitHub website renders a USD scene using the Houdini [husk CLI](#) on the SideFX website. By default, Husk renders with the Houdini Karma renderer, but Husk supports any Hydra-compatible USD render delegate. The bundle expands to a task per frame using the `Frames` job parameter and limits execution to Linux workers through host requirements.

To render with the KarmaXPU GPU engine, uncomment the `amount.worker.gpu` host requirement in the job template so that the job is scheduled on a worker with a GPU. The Karma renderer requires a Karma license to run. Service-managed fleets provide usage-based Karma licenses automatically.

This bundle also shows how to write a custom asset introspection tool for job attachments. The included `generate_usd_job.py` script inspects a USD file, finds all referenced asset files, and generates a job bundle with job attachment references for those assets. To use the script, install the `usd-core` and `deadline` Python libraries and run:

```
python3 generate_usd_job.py my_scene.usd
```

The bundle also supports V-Ray and Redshift renderers through their Hydra render delegates. To use these renderers, build a conda package with the [houdini-vray-7](#), [houdini-redshift-2025](#), or [houdini-redshift-2026](#) conda recipes on the GitHub website.

The bundle includes a sample `.usda` file with a cube and sphere that has no external dependencies and can be rendered without attaching any other files.

Render Foundry Nuke scripts on Deadline Cloud

The [nuke_render_job bundle](#) on the GitHub website renders Nuke scripts in headless mode with the `nuke -x` command. The job creates one task per frame using the Open Job Description parameter space feature, and is restricted to Linux workers through host requirements.

The bundle includes a sample `motionblur3d_10.nk` Nuke script based on Foundry's MotionBlur3D example, configured with relative file paths and a pre-configured Write node.

To run this bundle, your queue needs Nuke available through a queue environment that installs the `nuke` and `nuke-openjd` conda packages. The [nuke-16.0 conda recipe](#) on the GitHub website builds a Nuke conda package you can publish to your queue's S3 conda channel.

Submit the bundle with custom parameters:

```
deadline bundle submit nuke_render/ \  
  --name "My Nuke Render" \  
  -p Frames="1-10" \  
  -p NukeScript=path-to-script.nk \  
  -p OutputDir=path-for-outputs
```

Or open the GUI submitter:

```
deadline bundle gui-submit nuke_render/
```

Render Autodesk 3ds Max scenes with V-Ray on Deadline Cloud

The [3dsmax_vray_denoiser](#) job bundle on the GitHub website renders 3ds Max scenes with V-Ray and converts the output from V-Ray's native VRIMG format to EXR while preserving denoising data. The bundle includes the following features:

- **Frame chunking:** Handles both contiguous ranges (for example, 1-100) and non-contiguous ranges (for example, 1-5, 10, 15-20).
- **VRIMG to EXR conversion:** Converts V-Ray's native VRIMG format to EXR while preserving denoising data, beauty passes, multi-channel data, and high dynamic range.
- **Automatic cleanup:** Removes temporary VRIMG files after successful conversion.

To run this bundle, you need 3ds Max 2025 and V-Ray for 3ds Max 2025 installed on Windows worker hosts. You can use the [3ds Max host configuration scripts](#) on the GitHub website to install 3ds Max and V-Ray on a Windows service-managed fleet. For more information, see [the section called "3ds Max host configuration"](#).

Before submitting the job, configure your scene to save raw VRIMG output to a local worker path (for example, C:\Temp\output.vrimg) and remove absolute output paths from each render element including the Denoiser. Then submit the bundle:

```
deadline bundle gui-submit 3dsmax_vray_denoiser/
```

Render KeyShot scenes on Deadline Cloud

The deadline-cloud-samples repository on the GitHub website includes the [keyshot_standalone](#) job bundle, a Windows KeyShot job bundle that renders a scene with each frame as a separate task. The bundle accepts the following job parameters:

- KeyShotFile
- Frames
- OutputName
- OutputDirectoryPath
- OutputFormat

Output file paths follow the pattern *OutputDirectoryPath/OutputName.frame.extension*. All other render settings come from the scene file itself.

The bundle expects one of the following:

- An input directory that contains all file references required to render. The simplest way to create one is to save the entire scene as a KeyShot package (.ksp), which bundles external files and converts paths to relative. Open the .ksp and submit the unpacked directory as the input directory, with the modified scene as the input KeyShot file.
- All referenced files available through network storage or another method that isn't job attachments.

To run this bundle, your queue needs KeyShot available through a queue environment. The [keyshot-2025 conda recipe](#) on the GitHub website builds a KeyShot conda package you can publish to your queue's S3 conda channel.

Render Autodesk VRED scenes on Deadline Cloud

The [vred_render](#) job bundle on the GitHub website renders VRED scenes using either VRED Core or VRED Pro in headless mode. It uses VRED's Python API through the `VRED_RenderScript_DeadlineCloud.py` script, which controls the rendering process, converts job template parameters into VRED API calls, and manages settings such as quality, resolution, and tiling.

The bundle provides two template options:

- **Basic template** (`template.yaml`): Standard rendering functionality.
- **Tiling template** (`template_tiling.yaml`): Adds region/tile-based rendering and a tile assembly step that combines rendered tiles into final images. To use this template, rename it to `template.yaml`.

The job parameters cover input and output paths, render settings such as image dimensions and DPI, render quality presets (Analytic, Realistic, Raytracing, NPR), NVIDIA DLSS options, frame control, animation settings, and tile-based rendering. When you use tile-based rendering, you must also enable GPU raytracing to prevent solid black tile outputs.

To run this bundle on a service-managed fleet, your queue needs access to the `deadline-cloud` conda channel and a GPU-enabled worker. The [vredcore-2026 conda recipe](#) on the GitHub website

builds a VRED Core conda package. For tile assembly, the rendering environment must include the `imagemagick` conda package from the `conda-forge` channel.

Submit the bundle:

```
deadline bundle gui-submit vred_render/
```

Render Adobe After Effects projects on Deadline Cloud

The [afterfx_render_one_task](#) job bundle on the GitHub website uses `aerender` to render an After Effects frame range as a single task. The entire workload runs on one worker as one command. The bundle accepts the following job parameters:

- Project file
- Comp name
- Input directory
- Output directory
- Start frame
- End frame

This bundle expects an input directory that contains all file references required to render. Place the project file inside the input directory to preserve relative paths.

To run this bundle, you need After Effects installed on Windows worker hosts. You can use the [After Effects host configuration script](#) on the GitHub website to install After Effects with Red Giant plugins on a Windows service-managed fleet. The [aftereffects-25.1 conda recipe](#) on the GitHub website builds an After Effects conda package as an alternative.

Render V-Ray standalone scenes on Deadline Cloud

The [vray_render](#) job bundle on the GitHub website creates a V-Ray rendering job for Deadline Cloud.

To run this bundle, you need:

- A V-Ray conda package hosted on a conda channel. For the recipe, see [V-Ray conda recipe](#) on the GitHub website.

- A sample `.vrscene` file and its dependencies. For sample scenes, see [Sample Scenes](#) on the Chaos website. Sample Scene 01 in the ENVISION documentation includes `.vrscene` files.

Submit the bundle with the GUI submitter:

```
deadline bundle gui-submit vray_render
```

Or submit a Chaos ENVISION sample from the CLI:

```
SAMPLE_DIR=path-to-sample  
deadline bundle submit vray_render \  
-p VraySceneFile="$SAMPLE_DIR"/Building.vrscene \  
-p InputAssetDir="$SAMPLE_DIR"/Building.data
```

For all customization options, see [V-Ray Standalone Command Line Options](#) on the Chaos website.

Render Redshift scenes on Deadline Cloud

The [redshift-2025](#) job bundle on the GitHub website renders Redshift scenes using the standalone `redshiftCmdLine.exe` renderer that ships with Cinema 4D 2025. The bundle is designed for Windows systems. Cinema 4D is used because it is available in the `deadline-cloud` conda channel and includes the Redshift CLI renderer.

The bundle installs Cinema 4D 2025 through the `cinema4d=2025` conda package, configures Windows host requirements, and uses `redshiftCmdLine.exe` directly for rendering. The executable is located at `%CONDA_PREFIX%\cinema4d\RedshiftData\bin\redshiftCmdLine.exe`, which resolves automatically when the Cinema 4D conda package is installed.

To run this bundle, you need:

- A Windows operating system
- A Redshift compatible GPU
- The `cinema4d=2025` conda package
- A valid Redshift scene file (`.rs`)

You can create a `.rs` file in Cinema 4D with Redshift materials and lighting, then export the scene as a Redshift scene file.

Submit the bundle:

```
deadline bundle gui-submit redshift-2025
```

A Deadline Cloud Windows GPU service-managed fleet works with no further configuration. Make sure that the queue has access to the `cinema4d=2025` conda package and Redshift licensing.

Render POV-Ray 3.7 scenes on Deadline Cloud

The [povray-3.7 job bundle](#) on the GitHub website renders POV-Ray (Persistence of Vision Raytracer) scenes on Deadline Cloud with full parameter control over resolution, quality (0-11), and antialiasing. POV-Ray is available from the `conda-forge` channel, and this bundle overrides `CondaChannels` to enable that channel.

The bundle includes a sample POV-Ray scene (`sample/sample.pov`) that demonstrates 3D shapes and textures. Submit the bundle to a queue that supports conda with a `CondaChannels` parameter:

```
deadline bundle gui-submit povray-3.7/
```

Tile rendering on Deadline Cloud

Tile rendering divides each frame into evenly sized regions, renders each region as a separate task in parallel across workers, then assembles the regions into the final image. The samples repository on the GitHub website includes several tile rendering job bundles for different DCC and renderer combinations.

[tile_render_with_maya_arnold](#)

A two-step job that renders frames in tiles using a 3-dimensional task parameter space (`Frame × TileNumberX × TileNumberY`) and then assembles the tiles for each frame using FFmpeg. The bundle includes a simple scene and default parameters to make it easy to try.

[tile_render_with_maya_vray](#)

A tile rendering job that uses Maya and V-Ray to create EXRs as output, then uses OpenImageIO to assemble them into a single image. The bundle relies on the V-Ray render handler in the Maya adaptor. For more information about OpenImageIO, see [OpenImageIO](#) on the GitHub website.

[tile_render_with_vray_linux](#)

Renders a V-Ray scene by dividing the image into a configurable grid of regions (rows × columns × frames), rendering each region as a separate task using V-Ray's `-crop` flag, and merging the regions with ImageMagick. Optionally creates an MP4 movie from the rendered frames using FFmpeg. Automatically handles asset path remapping between workstation and workers.

[tile_render_maya_ffmpeg_for_blogpost](#)

A companion job for the AWS blog post [Create a tile rendering job with modifications for AWS Deadline Cloud](#). The blog post walks through customizing the Deadline Cloud Maya adaptor and writing a tile rendering job template.

Choose a tile count based on image size, scene complexity, and worker pool size. More regions create more parallelism but also add overhead. For small images, fewer regions can be faster. For large images and complex scenes, more regions can significantly reduce total render time.

Group frames into chunks with task chunking on Deadline Cloud

The [task_chunking](#) samples on the GitHub website demonstrate the [Task Chunking](#) extension for Open Job Description. Render jobs often spend significant time loading applications and scene files before rendering each frame. Chunking amortizes this overhead by processing multiple frames per chunk, which reduces total job runtime.

The samples include four examples:

`basic_contiguous_chunks`

A minimal example using `rangeConstraint: CONTIGUOUS`. Each chunk expands to a range like "1-10" or "11-20".

`basic_non_contiguous_chunks`

A minimal example using `rangeConstraint: NONCONTIGUOUS`. Chunks can be arbitrary frame sets like "1-3, 5, 7-20:2".

`blender_render_with_contiguous_chunks`

A real-world example converted from the [the section called "Blender render"](#) bundle to render with contiguous chunks.

blender_render_with_non_contiguous_chunks

A Blender variant with non-contiguous chunks.

Choose a range constraint based on the application:

- CONTIGUOUS: Use when the application supports start and end frame arguments.
- NONCONTIGUOUS: Use when the application supports arbitrary frame lists.

Submit any of the samples with the Deadline Cloud CLI:

```
deadline bundle submit sample-directory
```

To learn how task chunking works and how to add it to your own job templates, see [Task chunking for job templates](#).

Encode video with FFmpeg on Deadline Cloud

The [ffmpeg_encode_video](#) job bundle on the GitHub website takes a directory of sequentially numbered image files and encodes them into an MP4 video using FFmpeg. The bundle is useful as a standalone utility for converting render output to video, or as a reference for adding a video encoding step to a multi-step render pipeline.

The bundle includes the following features:

- Encodes numbered image sequences (PNG, EXR, JPEG, and others) into H.264 MP4 video.
- Supports customizable frame rate, quality (CRF), and encoding speed presets.
- Uses #### hash-mark patterns for input file naming, automatically converted to FFmpeg's printf-style %04d format.
- Works with job attachments or shared file systems.
- Produces broadcast-safe output with BT.709 color space and faststart for web streaming.

FFmpeg is not available from the `deadline-cloud` conda channel. To run this bundle on a service-managed fleet, use `conda-forge` as the `CondaChannels` parameter for the queue environment.

Submit the bundle:

```
deadline bundle submit ffmpeg_encode_video/ \  
-p InputDir=path-to-frames \  
-p InputFilePattern="render.####.png" \  
-p StartFrame=1 \  
-p EndFrame=250 \  
-p OutputDir=path-for-output \  
-p OutputFileName="my_video.mp4"
```

To add video encoding to an existing render job, copy the EncodeVideo step from this bundle's template into your own template, add a dependencies entry so the encode step runs after rendering, and add conda-forge to the CondaChannels parameter. For a complete multi-step pipeline example, see [the section called “Turntable with Maya/Arnold”](#).

Render a turntable video with Maya and Arnold on Deadline Cloud

The [turntable_with_maya_arnold job bundle](#) on the GitHub website takes an OBJ geometry file as input and outputs a turntable render video. A turntable render is a 360-degree rotation of a rendered 3D model that allows inspection of the geometry without a 3D viewer.

This bundle demonstrates how to create utility jobs that are easy to submit from a GUI. The job template defines a pipeline of three steps connected with a dependency chain:

1. A scene preparation task runs as a single MayaPy task that loads a base Maya scene, imports the input geometry file, and adjusts scene properties.
2. An image sequence rendering step runs a separate task per frame across many worker hosts.
3. A video encoding task runs a single FFmpeg task that combines the rendered frames into a video.

The bundle embeds a base Maya scene file with a camera, lighting, and default geometry, and provides job parameters that you can use for either GUI submission or automation scripting. By specifying conda channel and package parameter values, the job runs successfully on a farm created with the Deadline Cloud default onboarding.

You can launch the GUI submitter by creating a desktop shortcut. On Windows, create a file called Submit Turntable to Deadline Cloud.bat with content like the following:

```
@call C:\DEADLINE_CLI_INSTALLATION\deadline bundle gui-submit C:  
\DEADLINE_CLOUD_SAMPLES\turntable_with_maya_arnold
```

If you accept all default settings, the job renders a turntable with placeholder geometry. You can also download an OBJ file such as [stanford-bunny.obj](#) from the common-3d-test-models repository on the GitHub website, and select it for the input geometry parameter.

Run a Monte Carlo simulation on Deadline Cloud

The [monte_carlo_simulation job bundle](#) on the GitHub website prices a portfolio of autocallable structured notes using Monte Carlo simulation with QuantLib's Heston stochastic volatility model. The job is a Deadline Cloud port of the [Pricing Financial Derivatives with AWS Batch](#) workshop.

The bundle defines a two-step pipeline:

1. `PricePositions`: One task per portfolio position. Tasks are grouped into chunks that calibrate the Heston model once and price all positions in the chunk, amortizing calibration cost.
2. `AggregateResults`: Collects per-position results into a portfolio summary.

The bundle uses the Open Job Description [TASK_CHUNKING extension](#) on the GitHub website for load balancing. The scheduler starts by dispatching individual positions, observes how long they take, and then automatically grows the chunk size to match a target runtime. Fast positions are grouped into larger chunks; slow positions stay in small chunks to keep work spread across the fleet.

To run this bundle, you need a queue with a conda queue environment that includes the conda-forge channel for quantlib-python. When you deploy the [starter farm template](#) on the GitHub website, set the `ProdCondaChannels` parameter to `deadline-cloud conda-forge`.

Submit the bundle with the GUI submitter:

```
deadline bundle gui-submit monte_carlo_simulation/
```

Or submit a quick test with fewer positions:

```
deadline bundle submit monte_carlo_simulation/ \  
-p PositionRange="0-1" -p NumPaths=100
```

Train 3D Gaussian Splatting from video on Deadline Cloud

The [gsplat_pipeline job bundle](#) on the GitHub website runs a [3D Gaussian Splatting pipeline](#). The bundle takes a video file as input and produces a Gaussian Splatting .ply file as output. After

downloading the output, you can view it in any Gaussian Splatting viewer such as [SuperSplat](#) on the GitHub website.

The pipeline runs a single task that performs the following steps:

1. Extracts video frames with [FFmpeg](#) on the FFmpeg website.
2. Solves Structure-from-Motion with [COLMAP](#) on the COLMAP website and [GLOMAP](#) on the GitHub website, saving the pinhole model and undistorted images.
3. Trains Gaussian Splatting with one of three trainers: [NeRF Studio splatfacto](#) on the Nerfstudio website, [Splatfacto in the Wild](#) on the Nerfstudio website, or the [simple_trainer.py gsplat library example](#) on the gsplat website. Output is saved to the .ply file you specify.

To run this bundle, deploy the [CUDA farm CloudFormation template](#) on the GitHub website to create a Deadline Cloud farm with a CUDA GPU fleet. Then build the [NeRF Studio conda package](#) on the GitHub website and publish it to your S3 conda channel.

If you only need the default NERFSTUDIO (splatfacto) trainer, you don't need the CUDA farm template or the custom NeRF Studio conda package. The minimum requirements are a Deadline Cloud farm with a GPU fleet, a queue environment that includes conda-forge, and the following conda packages: ffmpeg colmap glomap nerfstudio cuda.

To capture a usable video, follow these tips:

- Use a wide field of view so that more objects are common between image pairs for Structure-from-Motion to use.
- Turn off video stabilization to preserve identical lens optics between frames.
- Capture with slow and steady motion. Keep moving the camera and avoid stopping and panning from a single location.

From the `job_bundles` directory of the samples repository, submit the job:

```
deadline bundle gui-submit gsplat_pipeline
```

On the *Job-specific settings* tab, select paths for both the input video file and the output .ply file.

For a complete walkthrough that covers capturing a video, submitting and monitoring the job, and viewing the trained result, see [Train 3D Gaussian Splatting from your own video](#).

Train and use a FLUX.2 Klein LoRA on Deadline Cloud

The [flux2_klein_lora](#) job bundles on the GitHub website use [diffusers](#) and [peft](#) to fine-tune Black Forest Labs' FLUX.2 Klein with LoRA, then generate images from text prompts with the trained adapter.

For a complete walkthrough that covers prerequisites, training parameters, image generation, and cleanup, see [the section called "FLUX.2 Klein LoRA"](#).

Benchmark LLMs with vLLM and lm-evaluation-harness on Deadline Cloud

The [vllm_lm_eval_leaderboard](#) job bundle on the GitHub website evaluates multiple LLMs against multiple benchmarks in a single Deadline Cloud job. Each model becomes one task in a parameter sweep, and tasks run in parallel across workers. A final step aggregates per-model results into a ranked leaderboard (CSV and Markdown).

Each task in the `EvalModels` step starts a local vLLM server, runs every benchmark with EleutherAI's `lm-evaluation-harness` against the local endpoint, then stops vLLM. For more information, see [vLLM](#) and [lm-evaluation-harness](#) on the GitHub website. Models load directly from Hugging Face Hub, so job attachments are not required.

To run this bundle, deploy the [CUDA farm CloudFormation template](#) on the GitHub website. The template provisions an NVIDIA GPU service-managed fleet (A10G or L4) and a queue with a conda queue environment that this bundle uses without modification. If you already have a farm, you need an SMF fleet with NVIDIA GPUs, at least 32 GB of RAM, and at least 4 vCPUs.

The default model list is a small, ungated, fast-loading mix that fits on a single A10G or L4: Qwen/Qwen2.5-0.5B, Qwen/Qwen2.5-1.5B, and EleutherAI/pythia-1.4b. To use other models, edit the range list under `parameterSpace.taskParameterDefinitions` in `template.yaml`.

The default benchmarks form a commonsense reasoning suite:

`hellaswag`, `arc_easy`, `arc_challenge`, `winogrande`. Override at submit time:

```
deadline bundle submit ./job_bundles/vllm_lm_eval_leaderboard/ \  
--parameter Benchmarks="hellaswag,mmlu,gsm8k"
```

If your fleet does not scale up workers, the most common cause is an Deadline Cloud service quota. Confirm that you have headroom for *OnDemand G instance GPUs per region* and *OnDemand vCPUs per region* in the Service Quotas console.

For a complete walkthrough that covers prerequisites, farm setup, custom models and benchmarks, and cleanup, see [the section called “vLLM LLM leaderboard”](#).

Run batch LLM inference with vLLM on Deadline Cloud

The [vllm_batch job bundle](#) on the GitHub website runs high-throughput large language model (LLM) inference on a JSONL file of prompts using vLLM. The job fans out across a GPU fleet, every prompt gets an LLM response, and an aggregate step packages everything into a JSONL file plus a self-contained HTML viewer. The bundle handles offline, embarrassingly-parallel workloads such as content generation, evaluation datasets, translation, extraction, and classification.

The bundle uses the Deadline Cloud task chunking feature to group prompts into batched tasks, and an Open Job Description step environment to load the model once per worker instead of once per task. For more information, see [Task chunking for job templates](#).

To run this bundle, you need a service-managed fleet with NVIDIA GPUs and at least 32 GB RAM, and a queue with a conda queue environment that reads CondaPackages and CondaChannels job parameters. From the `job_bundles` directory of the samples repository, submit the job:

```
deadline bundle gui-submit vllm_batch
```

You can chain the output into the [Generate images in batch with a diffusion model on Deadline Cloud](#) bundle to turn generated text into captioned images.

For a complete walkthrough that covers farm setup, input format, chunk sizing, and viewing the results, see [Run batch LLM inference with vLLM](#).

Generate images in batch with a diffusion model on Deadline Cloud

The [text_to_image_batch job bundle](#) on the GitHub website runs high-throughput batch image generation on a JSONL of prompts using a diffusion model. The default model is FLUX.2 Klein 4B, which is Apache 2.0 licensed, ungated, and distilled to 4 steps. The scheduler distributes tasks across available GPU workers, with the diffusion pipeline loaded once per worker and reused for every task on that worker. An aggregate step produces a combined JSONL file and a static HTML gallery viewer.

When a JSONL line carries a `caption` field, or a `generated_text` field chained from the [Run batch LLM inference with vLLM on Deadline Cloud](#) bundle's output, the job composites the text over the generated image as crisp typography. Lines without a caption produce pure imagery.

To run this bundle, you need a service-managed fleet with NVIDIA GPUs and at least 32 GB RAM, and a queue with a conda queue environment that reads CondaPackages and CondaChannels job parameters. From the `job_bundles` directory of the samples repository, submit the job:

```
deadline bundle gui-submit text_to_image_batch
```

For a complete walkthrough that covers farm setup, input format, caption overlays, and the chaining workflow, see [Generate images in batch with a diffusion model](#).

Generate videos from text prompts with Wan2.2 on Deadline Cloud

The [wan22_video_generation](#) job bundle on the GitHub website generates short video clips from a text prompt using Wan2.2, an open video generation model. Each task renders one independent clip with its own seed, so a single job fans out across workers and returns a set of variations on the same prompt.

The bundle runs the Wan2.2 TI2V-5B checkpoint through the diffusers WanPipeline. Workers pull the weights from Hugging Face at runtime and cache them, preferring the fleet's persistent volume so that later workers reuse the download. No Hugging Face token is required because the Wan2.2 repositories are ungated. For the model itself, see the [Wan2.2 repository](#) on the GitHub website.

The bundle requires a farm with a GPU-enabled Linux queue. Workers must have an NVIDIA GPU (24 GB VRAM minimum), 64 GiB of system memory, and at least 60 GiB of free disk for the model cache. On 24 GB cards the script turns on sequential CPU offload and VAE tiling so the model fits; larger cards run considerably faster.

Submit a job that generates four variations of a prompt:

```
deadline bundle submit ./wan22_video_generation \
  --queue-id gpu-queue-id \
  -p Prompt="A hot air balloon drifting over terraced rice fields at dawn" \
  -p NumClips=4 \
  -p OutputDir=~ /wan22_output
```

The README describes the full parameter set, including resolution and frame-count constraints, faster smoke-test settings, and measured generation times. For related generative AI examples, see [the section called "Text-to-image batch"](#) and [the section called "FLUX.2 LoRA training"](#). For persistent volume setup, see the fleet storage documentation linked from the README.

Fine-tune Hugging Face LLMs with LoRA on Deadline Cloud

The [hf_finetune_lora job bundle](#) on the GitHub website fine-tunes a Hugging Face causal language model with Low-Rank Adaptation (LoRA) or Quantized Low-Rank Adaptation (QLoRA) on a custom instruction dataset, using the transformers, PEFT, and bitsandbytes libraries. The output is a small LoRA adapter (approximately 50 to 200 MB) that you load on top of the base model to teach it a writing style, a domain expertise, a specific output format, or some proprietary knowledge.

To run this bundle, you need a Deadline Cloud farm with a GPU-enabled queue (Linux fleet, NVIDIA GPU with 16 GB or more of video RAM (VRAM)) and a dataset in JSONL format. Submitting with all defaults trains on the bundle's included sample data. From the `job_bundles` directory of the samples repository, submit the job:

```
deadline bundle gui-submit hf_finetune_lora
```

For a complete walkthrough that covers dataset preparation, queue role permissions, fleet sizing, and testing the trained adapter, see [Fine-tune Hugging Face LLMs with LoRA and QLoRA](#).

Predict protein structures with ESMFold on Deadline Cloud

The [esmfold_predict](#) job bundle on the GitHub website runs protein structure prediction with ESMFold (Meta's facebook/esmfold_v1, MIT license). The bundle takes a FASTA file as input and produces a .pdb file per sequence as output, along with confidence metrics and an optional validation report against experimental reference structures.

The job runs four steps:

1. Parse the input FASTA, validate sequences (up to 1024 amino acids, standard residues plus X), and split records across worker tasks.
2. Run ESMFold inference on each batch of sequences on GPU.
3. Render a backbone trace image of each predicted structure, colored by per-residue pLDDT confidence.
4. Optional: when you supply a directory of experimental reference PDBs, compute TM-score, RMSD, and a per-residue pLDDT/error calibration plot.

The bundle requires a farm with an NVIDIA GPU service-managed fleet (A10G, L4, or A100; at least 16 GB VRAM and 16 GB system RAM) and a queue with a conda queue environment that consumes the `CondaPackages` and `CondaChannels` job parameters. The fastest setup is the [cuda_farm](#)

AWS CloudFormation (CloudFormation) template on the GitHub website. Amazon Elastic Compute Cloud (Amazon EC2) GPU instances are gated by per-region vCPU quotas; if your fleet doesn't scale up, request an increase for *Running On-Demand G and VT instances* in the Service Quotas console.

Submit the demo, which folds three short benchmark proteins (Trp-cage variants 1L2Y and 2JOF, and villin headpiece 1VII):

```
deadline bundle submit ./job_bundles/esmfold_predict/ \  
-p InputFasta=./job_bundles/esmfold_predict/sample_inputs/demo.fasta
```

The first fold on a fresh worker downloads the 5.2 GB facebook/esmfold_v1 weights into <OutputDir>/.hf_cache/ (about three minutes on a g5.2xlarge). Subsequent fold tasks in the same job reuse the cache.

To validate predictions against experimental references, place <seq_id>.pdb files in a directory and pass it as ReferencePdbDir. The Validate step writes validation.csv and a per-sequence calibration.png.

Classify satellite imagery in parallel on Deadline Cloud

The [satellite_classification](#) job bundle on the GitHub website classifies satellite image tiles into land-cover categories (water, vegetation, bare soil, rock, cloud) and merges the results into a single map. The classifier reads color ratios between the four spectral bands of each tile to decide what's on the ground.

The job runs in two steps. ClassifyTiles fans out one parallel task per tile that downloads the tile, classifies its pixels, and writes a result file plus a color PNG. MosaicResults runs after the parallel step finishes and merges every tile into one map plus an overview image with class percentages. This fan-out and merge pattern applies to any workload where input files are independent.

The bundle ships with five sample tiles simulating the Grand Canyon area. They're downloaded automatically from the Deadline Cloud samples CDN when the job runs — no external data or accounts needed.

The bundle requires a farm with a conda queue environment that has conda-forge in the channel list (the classifier uses the rasterio package). The quickest setup is the [starter farm AWS CloudFormation \(CloudFormation\) template](#) on the GitHub website. Set its ProdCondaChannels parameter to deadline-cloud conda-forge.

Submit with the sample tiles:

```
deadline bundle submit job_bundles/satellite_classification/
```

To classify your own tiles, point TilesDir at a local directory of 4-band .tif files:

```
deadline bundle submit job_bundles/satellite_classification/ \  
-p TilesDir=path-to-tiles
```

Run GROMACS molecular dynamics simulations on Deadline Cloud

The [gromacs_md job bundle](#) on the GitHub website runs a molecular dynamics simulation pipeline from raw protein structure to analyzed trajectory with [GROMACS](#) on the GROMACS website. The pipeline runs system preparation, energy minimization, NVT/NPT equilibration, production MD, and analysis (RMSD, RMSF, radius of gyration, hydrogen bonds). Multiple independent replicas fan out in parallel through the MaxReplicaIndex parameter.

The bundle requires a Deadline Cloud farm with a Linux x86_64 service-managed fleet (minimum 4 vCPU) and a conda queue environment with gromacs from conda-forge. No host configuration script or custom conda recipe is needed.

The bundle includes sample data for hen egg-white lysozyme (PDB: 1AKI) and MDP parameter files under sample_inputs/mdp/.

From the job_bundles directory, submit the job:

```
deadline bundle submit gromacs_md \  
-p "InputPdb=sample_inputs/protein.pdb" \  
-p "MdpMinimization=sample_inputs/mdp/minimization.mdp" \  
-p "MdpNvt=sample_inputs/mdp/nvt.mdp" \  
-p "MdpNpt=sample_inputs/mdp/npt.mdp" \  
-p "MdpProduction=sample_inputs/mdp/production.mdp" \  
-p "ProductionSteps=500000"
```

To run 10 independent replicas in parallel:

```
deadline bundle submit gromacs_md \  
-p "InputPdb=protein.pdb" \  
-p "MaxReplicaIndex=9" \  

```

```
-p "ProductionSteps=5000000"
```

Run virtual screening with AutoDock Vina on Deadline Cloud

The [virtual_screening_vina job bundle](#) on the GitHub website uses [AutoDock Vina](#) to screen large compound libraries against a protein target. The bundle splits a compound library into chunks and docks them in parallel across a fleet of workers, then aggregates and ranks the results by binding affinity.

The pipeline runs four steps:

1. **PrepareReceptor** — Converts the protein PDB to PDBQT format with Open Babel.
2. **SplitLibrary** — Splits the SDF compound library into N chunks.
3. **DockCompounds** — Runs N parallel docking tasks (one per chunk). Each task is idempotent and safe for Spot preemption.
4. **ScoreAndRank** — Aggregates all chunk results into a ranked CSV of top hits.

The bundle requires a Linux x86_64 service-managed fleet (Spot recommended), a conda queue environment with `openbabel` from `conda-forge`, and AutoDock Vina. Install Vina by building the [the section called “AutoDock Vina conda recipe”](#) conda recipe into your S3 channel, or by installing the binary through a fleet host configuration script.

Set `CompoundLibrary=chembl` (default) to automatically download and filter drug-like molecules from [ChEMBL](#) on the EMBL-EBI website, or provide your own SDF file.

From the `job_bundles` directory, submit the job:

```
deadline bundle submit virtual_screening_vina \  
-p "ReceptorPdb=receptor.pdb" \  
-p "CompoundLibrary=chembl" \  
-p "MaxCompounds=100000" \  
-p "CompoundsPerChunk=100" \  
-p "CenterX=-10.7" \  
-p "CenterY=12.4" \  
-p "CenterZ=68.8"
```

For details on building the AutoDock Vina conda package, see [the section called “AutoDock Vina conda recipe”](#).

Call genetic variants with bwa and bcftools on Deadline Cloud

The [variant_calling_bwa](#) job bundle on the GitHub website finds the genetic differences between an individual and a reference genome. It aligns sequencing reads with bwa, calls variants with bcftools, and merges the results into one VCF file. The pipeline reimplements the AWS HealthOmics WDL variant-calling tutorial pipeline in Open Job Description.

The bundle demonstrates patterns that carry over to other scientific workloads:

- Software delivered as conda packages from the bioconda channel rather than built into container images.
- Fan-out over two different task parameters: alignment scatters over samples, and variant calling scatters over genome regions, with a merge step as the gather.
- A job environment that verifies every required tool is on PATH once per session, and reports the exact packages and channels to configure if any is missing.

The bundle requires a farm with a Linux fleet, a conda queue environment with channels set to conda-forge bioconda in that order, and the Deadline Cloud CLI. Download the sample data (about 950 KB), then submit:

```
python sample_inputs/fetch_test_data.py
deadline bundle submit .
```

The default parameters (2 samples, 4 regions) produce 10 tasks. The README covers bringing your own cohort, scaling the region list, and the steps a production analysis would add. For other bioscience examples, see [the section called “ESMFold protein structure prediction”](#), [the section called “GROMACS molecular dynamics”](#), and [the section called “Virtual screening \(AutoDock Vina\)”](#).

Generate procedural 3D scenes with Infinigen on Deadline Cloud

The [infinigen_scene_gen](#) job bundle on the GitHub website generates photorealistic 3D scenes on Deadline Cloud GPU workers with the [Infinigen scene generator](#) on the Infinigen website. The bundle supports indoor rooms (dining room, bedroom, kitchen, bathroom) and outdoor nature landscapes (desert, forest, mountain, coast, arctic). Each seed produces a unique scene with different layout, furniture, materials, terrain, and vegetation. Seeds fan out as independent tasks for parallel execution across the GPU fleet.

Each scene produces rendered RGB images (1280×720), HDR EXR files, material segmentation labels, render passes (diffuse color, ambient occlusion, glossy), and camera intrinsics and extrinsics. When `RenderGroundTruth=True`, a second pass provides depth, normals, and segmentation ground truth.

The bundle requires the `infinigen` conda package built from the [the section called “Infinigen conda recipe”](#) recipe, a [conda queue environment with improved caching](#), and a GPU-capable Linux service-managed fleet (such as the [CUDA farm CloudFormation template](#)) on the GitHub website.

From the `job_bundles` directory, submit the job:

```
# Single indoor dining room (~15 min)
deadline bundle submit infinigen_scene_gen/ \
  -p Frames="0-0" \
  -p SceneType="indoor" \
  -p SceneConfig="fast_solve.gin singleroom.gin" \
  -p RoomType="DiningRoom"
```

```
# 10 unique bedrooms in parallel
deadline bundle submit infinigen_scene_gen/ \
  -p Frames="0-9" \
  -p SceneType="indoor" \
  -p RoomType="Bedroom"
```

```
# Desert landscape
deadline bundle submit infinigen_scene_gen/ \
  -p Frames="0-0" \
  -p SceneType="nature" \
  -p SceneConfig="desert.gin simple.gin"
```

Train a robot manipulation policy with MuJoCo on Deadline Cloud

The [mujoco_sim_to_policy job bundle](#) on the GitHub website trains and renders a learned robot-manipulation policy on Deadline Cloud using a simulation of the Strands Robots `so100` arm in [MuJoCo](#) on the MuJoCo website. The job runs three dependent steps on managed GPU workers:

1. **Datagen** — Scripted joint-space pick of a cube in MuJoCo, recorded as a LeRobot dataset. Each episode is verified by a cube-height check and discarded if it fails.
2. **Train** — Finetunes a LeRobot ACT policy on the generated dataset (CUDA).

3. **Render** — Drives a MuJoCo rollout with the finetuned policy and records the result as an MP4 video and PNG frames.

The bundle requires a Linux x86_64 GPU fleet (steps render headless through EGL and train on CUDA) and a conda queue environment that consumes the CondaPackages and CondaChannels job parameters. The default conda packages are python=3.12 pip git ffmpeg on conda-forge.

From the `job_bundles` directory, submit the job:

```
OUT="$(pwd)/output"; mkdir -p "$OUT"
deadline bundle submit mujoco_sim_to_policy -p "OutputDir=$OUT" --yes
```

Run autonomous driving simulations with CARLA on Deadline Cloud

The [autonomous_driving_carla](#) job bundle on the GitHub website runs an autonomous driving simulation parameter sweep on AWS Deadline Cloud with configurable multi-sensor capture, using the [CARLA simulator](#) on the CARLA website. The job runs a lane-change cut-in scenario and sweeps across configurable ego speeds, NPC speeds, and NPC starting distances. Each parameter combination creates a task (default $2 \times 2 \times 2 = 8$ tasks).

Each task captures multi-sensor data from user-selected camera viewpoints and produces per-camera videos, RGB and semantic segmentation frames, LiDAR point clouds (.ply), and 2D/3D bounding boxes in KITTI format.

The job runs inside a Docker container based on `carlasim/carla:0.9.16`. Before submitting, build the custom Docker image from the `docker/` directory in the bundle and push it to Amazon Elastic Container Registry (Amazon ECR). Attach the [docker_nvidia_container_toolkit](#) host configuration script on the GitHub website to your fleet to provide Docker and the NVIDIA Container Toolkit.

To run this bundle, you need a Deadline Cloud farm with a GPU fleet (minimum 1 NVIDIA GPU, 16 vCPU, 64 GiB memory) and a conda queue environment with ffmpeg. The queue role needs Amazon ECR pull permissions.

From the `job_bundles` directory, submit the job:

```
deadline bundle gui-submit autonomous_driving_carla
```

On the *Job-specific settings* tab, set the Container Image URI to your Amazon ECR image and configure the scenario parameters (ego speeds, NPC speeds, NPC distances) and camera viewpoints.

For a complete walkthrough that covers building the Docker image, submitting and monitoring the job, and reviewing the captured sensor output, see [Run an autonomous driving simulation sweep with CARLA](#).

Publish renders to Autodesk Flow Production Tracking from Deadline Cloud

The [blender_turntable_to_flow](#) job bundle on the GitHub website demonstrates how to integrate Deadline Cloud with Autodesk Flow Production Tracking (formerly ShotGrid). The bundle renders frames, encodes a review-ready movie, extracts a thumbnail, and publishes the result to Flow as a new Version on an Asset's review Task. The render step uses a Blender turntable as a placeholder — replace it with any renderer to adapt this pattern for your pipeline. For more information about Flow Production Tracking, see [Flow Production Tracking overview](#) on the Autodesk website.

The bundle demonstrates two core patterns for render-and-publish pipelines on Deadline Cloud:

- Post-render work modeled as discrete job steps (movie encode, thumbnail generation, publish) that are independently observable, retryable, and schedulable.
- Chained, dependent, non-render tasks that run after a render and fan out in parallel.

The step graph:

```
RenderTurntable (one task per frame)
  ## GenerateMovie      (ffmpeg frames # H.264 mp4)
  ## GenerateThumbnail  (mid frame # jpg)
  ## PublishToFlow      (depends on movie + thumbnail)
```

The job needs Blender, FFmpeg, and Python. Configure a conda queue environment with `blender ffmpeg python>=3.10 pip` on channels `deadline-cloud conda-forge`. Flow credentials are stored in AWS Secrets Manager (Secrets Manager) and read at runtime by the queue role.

A submission hook (`hooks.yaml`) reads Flow parameters from environment variables (`FLOW_PROJECT_ID`, `FLOW_ASSET_NAME`, `FLOW_SECRET_ARN`) so artists don't manually enter project metadata. Enable bundle hooks before first use:

```
deadline config set settings.allow_bundle_hooks true
```

Submit the job:

```
deadline bundle gui-submit blender_turntable_to_flow
```

Run a VFX studio pipeline on Deadline Cloud

The [vfx_pipeline](#) sample on the GitHub website is a working shot-render-and-publish pipeline for a studio moving renders to a cloud farm. Every section of its README starts from a piece of pipeline you already run: a render scheduler, always-on render nodes, a mounted NAS, a set-project tool, or a publish step. Each section shows the Deadline Cloud mechanism it becomes.

Almost everything in the sample is a placeholder to swap for your own tooling. Blender stands in for your DCC, a trivial Python add-on for your third-party or in-house plugins, a small launcher script for your environment manager, and a local directory for your shared network drive. There is no first-party submitter; the sample is for studios that drive submission from their own launcher.

The job's step graph renders one task per frame, encodes a preview movie and thumbnail in parallel, then publishes a Version to Autodesk Flow Production Tracking (ShotGrid) with credentials read from AWS Secrets Manager. The DCC and plugins reach the farm as conda packages built from recipes included in the sample and installed at run time by a conda queue environment. Finished outputs return to shared storage through `deadline queue sync-output`.

The README includes a walkthrough that builds the packages, publishes the conda channel to Amazon S3, and submits a shot. For the building blocks the sample uses, see [the section called “Job bundles”](#), [the section called “Conda recipes”](#), and [the section called “Queue environments”](#).

Run VTK visualization scripts on Deadline Cloud

The [vtk-latest](#) job bundle on the GitHub website runs VTK (Visualization Toolkit) Python scripts on Deadline Cloud. The bundle is generalizable for any VTK-based Python script that accepts command-line parameters for output path, width, and height, and saves visualization output to a specified location.

Your VTK script must accept the following command-line arguments:

- `--output`: Output file path

- `--width`: Image width
- `--height`: Image height

The bundle includes a sample script (`sample/flow_simulation_visualization.py`) that creates a 3D airfoil geometry, generates a flow field, calculates pressure distribution, and saves the output as an image with streamlines.

To run this bundle, your queue needs the `vtk` and `numpy` conda packages available in the queue environment's conda channels.

Submit the job with these parameters:

- `InputDir`: Directory containing your VTK script
- `InputScript`: Filename of your script
- `OutputDir`: Where to save the visualization
- `OutputFilename`: Name for the output file
- `ExtraParams`: Additional script parameters in the format `--param1 value1 --param2 value2`

Encode a movie from another Deadline Cloud job's output with FFmpeg

The [ffmpeg_movie_from_job_output](#) job bundle on the GitHub website downloads the rendered output of another completed job in the same queue and encodes the image sequence into an MP4 video using FFmpeg. Use it as a post-processing utility, for example, to automatically assemble frames into a movie after a Blender or Maya render job completes.

For encoding a local image sequence (no source-job download), see [the section called "FFmpeg encode video"](#).

A pre-submission hook (`inject_s3_settings.py`) runs at submission time on your workstation and looks up the queue's job attachment Amazon Simple Storage Service (Amazon S3) bucket configuration. The hook writes the settings to a JSON file that gets uploaded as a job attachment, so the worker can download files from Amazon S3 without needing Deadline Cloud API permissions on the queue role. On the worker, the job uses the `deadline.job_attachments` Python API to download the source job's output, then runs FFmpeg to encode the frames into an H.264 MP4 with BT.709 color space metadata.

Bundle hooks are off by default. Enable them once before submitting:

```
deadline config set settings.allow_bundle_hooks true
```

The job requires FFmpeg from the conda-forge conda channel. The source job and this job must be in the same queue, because they share the same job attachments Amazon S3 bucket.

Submit with the source job's ID:

```
deadline bundle submit ffmpeg_movie_from_job_output/ \  
-p SourceJobId=job-id \  
-p FrameRate=30 \  
-p OutputFilename=my_render.mp4
```

To restrict the download to a single step's output, also pass SourceStepId.

Run a bash script on Deadline Cloud

The [cli_job](#) job bundle on the GitHub website submits a multi-line bash script to Deadline Cloud. The script job parameter uses a multi-line edit control, and a data directory job parameter lets you select a directory of data for the script to read from and write to.

Submit the bundle with a data directory:

```
deadline bundle submit job_bundles/cli_job -p DataDir=~/data_dir
```

For a minimal starter that supplements the developer guide farm setup walkthrough, see the [simple_job](#) bundle on the GitHub website.

Use every Open Job Description GUI control on Deadline Cloud

The [gui_control_showcase job bundle](#) on the GitHub website shows every GUI control supported by user interface metadata on Open Job Description job parameters. Use it as a reference when you add GUI controls to your own job templates. For more information about the parameter schema, see [JobParameterDefinition](#) on the GitHub website.

Open the GUI submitter to see every control:

```
deadline bundle gui-submit job_bundles/gui_control_showcase
```

The screenshot displays the 'Job-specific settings' tab of the Deadline Cloud interface. It is organized into several sections: 'Text Controls' with a 'Line Edit Control' (containing 'Default line edit value.') and a 'Multi-line Edit Control' (containing 'This is a text file with multiple lines.'). Below these is a 'String Dropdown' set to 'WEDNESDAY' and a checked 'Check Box'. The 'Int Controls' section includes a 'Default Int Spinner' (42), a 'Big Step Int Spinner' (123), and a 'Bounded Int Spinner' (25). 'Dropdown Controls' features an 'Int Dropdown' (7) and a 'Float Dropdown' (3.26). 'Float Controls' has a 'Default Float Spinner' (1234.57), a 'Float Spinner One Decimal' (100000.0), and a 'Float Spinner Fixed Step' (0.00). 'Path Controls' includes 'Input File Picker', 'Output File Picker', and 'Directory Picker', each with a file selection button. At the bottom, there is a 'Settings...' button, a 'Submit' button, an 'Export bundle' button, and a 'Help' button. A status bar at the very bottom shows a green checkmark and the word 'default'.

Use job environments on Deadline Cloud

The following job bundles supplement the [the section called “Control the job environment”](#) section in the developer guide. Each demonstrates a different way to use [Open Job Description environments](#) on the GitHub website. The following bundles are in the [deadline-cloud-samples](#) repository on the GitHub website:

[job_env_vars](#)

Sets environment variables through a job environment.

[job_env_with_new_command](#)

Wraps the task command through a job environment.

[job_env_daemon_process](#)

Runs a background daemon process alongside the task.

[pip_self_contained_job](#)

Defines a job environment that creates a Python virtual environment with pip and adds it to the PATH for the job's steps, with no queue environment required. Runs on any Linux worker with python3 available, including Deadline Cloud service-managed fleet workers.

[pip_package_job](#)

Provides the job's Python dependencies through the [the section called “Pip queue environment”](#) instead of a job environment. The job only sets the PipPackages parameter that the queue environment reads. Use this style to define the pip environment once and share it across many jobs on a queue.

For more queue-level environment examples, see [the section called “Queue environments”](#).

Job attachment tutorial bundles for Deadline Cloud

The [job_attachments_devguide](#) and [job_attachments_devguide_output](#) job bundles on the GitHub website supplement the [the section called “Job attachments”](#) section in the developer guide. The bundles demonstrate how data flow metadata on path job parameters and the job bundle `asset_references.yaml` file work together to describe the files a job needs as input and produces as output.

When a job bundle specifies this metadata, it can work with either job attachments or shared file systems without changes to the template.

Develop a job bundle through four stages on Deadline Cloud

The [job_dev_progression](#) directory on the GitHub website contains a sequence of four job bundle development stages. As you add more options and split a workload into smaller parallel pieces, the complexity of the job grows. The four stages start with a single self-contained template and end at a Python package bundled with script entry points and unit tests. The sample is built around Python, but the ideas are not Python-specific.

Submit the first stage to your farm:

```
deadline bundle submit stage_1_self_contained_template
```

You can also run jobs locally for development with the [Open Job Description CLI](#) on the GitHub website.

```
openjd run stage_1_self_contained_template/template.yaml
```

To use a queue environment locally to provide conda packages, pass the `--environment` option:

```
openjd run --environment ../../queue_environments/conda_queue_env_from_console.yaml \
  stage_1_self_contained_template/template.yaml
```

List available conda packages on Deadline Cloud

The `deadline-cloud-samples` repository on the GitHub website includes the [list_available_conda_packages](#) job bundle, which lists all conda packages available in the `deadline-cloud` channel using `conda search -c deadline-cloud '*'` and prints the list to the job log.

For a list of available packages with their major and minor versions, and recommendations for pinning, see [Conda queue environment](#).

Copy an S3 prefix to job attachments on Deadline Cloud

The [copy_s3_prefix_to_job_attachments](#) job bundle on the GitHub website pre-populates a queue's job attachments S3 bucket by copying files from where they are already stored in Amazon S3. If you are adding a Deadline Cloud farm to a project that already has a large volume of data, or are submitting a job that depends on a lot of new data such as a fluid simulation output, the initial job attachments upload can be slow.

Because job attachments uses content-addressed storage and never re-uploads files that are already in job attachments, you can use alternative upload tools like [AWS Snowball](#) or [AWS DataSync](#) to copy data into Amazon S3 first, then use this job to copy it into the job attachments bucket for your queue.

Submit the bundle and choose how many workers to parallelize across:

```
deadline bundle submit copy_s3_prefix_to_job_attachments \
  -p S3CopySource=s3://SOURCE_BUCKET_NAME/prefix-to-copy \
  -p Parallelism=20
```


Add the following IAM permissions to your queue IAM role. The role already has permissions to read and write the job attachments bucket prefix.

```
{
  "Effect": "Allow",
  "Sid": "DeadlineQueueReadOnly",
  "Action": [
    "deadline:GetQueue"
  ],
  "Resource": [
    "arn:aws:deadline:REGION:ACCOUNT_ID:farm/FARM_ID/queue/QUEUE_ID"
  ]
},
{
  "Effect": "Allow",
  "Sid": "SourceBucketAccess",
  "Action": [
    "s3:ListBucket",
    "s3:GetObjectTagging",
    "s3:PutObjectTagging",
    "s3:GetObject"
  ],
  "Resource": [
    "arn:aws:s3:::SOURCE_BUCKET_NAME",
    "arn:aws:s3:::SOURCE_BUCKET_NAME/*"
  ]
}
```

This job calls Amazon S3 APIs and copies data into your queue's job attachments bucket, which incurs additional costs. Use the [Amazon S3 pricing](#) page and the [AWS Pricing Calculator](#) to estimate costs.

SSH or RDP to a Deadline Cloud worker through Session Manager

The [ssh_to_smf](#) (Linux) and [ssh_to_smf_windows](#) (Windows) job bundles on the GitHub website register a Deadline Cloud worker as an SSM hybrid managed node, enabling SSH, RDP, or PowerShell access through Session Manager for the duration of the job. Use these bundles to debug a job interactively or to forward ports for web UIs and Jupyter notebooks running on the worker.

The job follows this sequence:

1. The submit script creates a one-time SSM hybrid activation by calling `aws ssm create-activation`.
2. The Deadline Cloud job runs on a worker, registers the worker as a managed node, and prints the `mi-*` managed node ID to the job log.
3. You connect with `aws ssm start-session --target mi-XXXXXXX`.
4. After the configured session duration, the job deregisters the node and cleans up.

To set up SSH access for one-time per account and Region

1. Create an IAM role named `SSMServiceRole` with the SSM service principal trust and the `AmazonSSMManagedInstanceCore` managed policy attached.
2. Enable the advanced-instances tier:

```
aws ssm update-service-setting \
  --setting-id "arn:aws:ssm:region:account-id:servicesetting/ssm/managed-instance/
activation-tier" \
  --setting-value "advanced" \
  --region region
```

3. The Linux variant requires the [sudo_for_job_user](#) host configuration script on the GitHub website, and the Windows variant requires the bundle's `setup/host_config.ps1` as the fleet host configuration script.

Important

The Windows variant gives the RDP user local Administrator access and adds `job-user` to Administrators. Use this configuration for debugging only. Shut down all workers in the fleet after debugging completes.

Submit the job:

```
# Linux: 60-minute session
./submit.sh

# Linux: 120-minute session
./submit.sh 120
```

```
# Windows: 60-minute session
./submit.sh farm-XXX queue-XXX
```

Once the job is running, find the managed node ID in the job log and connect:

```
aws ssm start-session --target mi-XXXXXXXXXX --region region
```

Build a custom submitter for Deadline Cloud

The [custom_submitters](#) directory on the GitHub website holds reference submitters that go beyond what bundle editing and the integrated submitters support. Use these as starting points when your team needs UI controls, custom handlers, or fleet attributes that the integrated submitter doesn't expose:

- [fuzzypixel_maya](#) — The Maya custom submitter the AWS FuzzyPixel team developed for production. The submitter consolidates render settings into a single tab, exposes a QTree widget for activating individual render layers and overriding their resolution and frame range, surfaces custom fleet attributes (for example, spot or onDemand) as user-selectable options, and runs custom handlers that update `template.yaml` after the user clicks Submit.

Custom submitters can co-exist with the integrated submitter, but they don't receive updates or patches that ship with the integrated submitter.

Conda recipe examples for Deadline Cloud

The [conda_recipes](#) directory on the GitHub website contains sample tools for creating an Amazon S3 conda channel and building packages into it. The recipes use [rattler-build](#) on the prefix.dev website or, in some cases, the older [conda-build](#) on the conda website.

Each recipe directory includes a `deadline-cloud.yaml` metadata file that lists the conda platforms the recipe builds for and configures how the recipe is submitted to Deadline Cloud. The `submit-package-job` command in the `conda_recipes` directory submits a recipe to a queue whose name starts with `Package` and uses the queue's job attachments bucket as the conda channel.

For setup, see [Create a conda channel using S3](#). For the underlying packaging concepts, see [Create a conda package for an application or plugin](#).

To submit a Blender 4.5 build job, run the following from the `conda_recipes` directory of the samples repository:

```
./submit-package-job blender-4.5
```

To submit a build for both Linux and Windows:

```
./submit-package-job blender-4.5 -p linux-64 -p win-64
```

To submit a build for every platform listed in `deadline-cloud.yaml`:

```
./submit-package-job blender-4.5 --all-platforms
```

The [conda_build_linux_package](#) job bundle on the GitHub website is the Open Job Description template that `submit-package-job` submits. It runs `conda-build` or `rattler-build` on a worker, takes a recipe directory and optional source archives as input, and writes the built package to your queue's conda channel.

Source archives that recipes pull in by URL go in the [archive_files](#) directory on the GitHub website. Recipes that require an installer or redistributable archive you have to download manually (such as Maya, Cinema 4D, or Nuke) read from this directory at build time.

The following recipes on the GitHub website package the Deadline Cloud client libraries themselves so that jobs can call Deadline Cloud APIs from inside a queue:

- [deadline](#): The Deadline Cloud Python client (`deadline CLI` and `deadline.job_attachments`), packaged for Linux, Windows, Linux ARM, and macOS.
- [openjd-adaptor-runtime](#): The runtime that the DCC Open Job Description adaptors build on top of, packaged for Linux and Windows with Python 3.13.

The following sections describe the recipe families available in the samples repository.

Topics

- [Build a Blender conda package for Deadline Cloud](#)
- [Build an Autodesk Maya conda package for Deadline Cloud](#)
- [Build a SideFX Houdini conda package for Deadline Cloud](#)

- [Build a Maxon Cinema 4D conda package for Deadline Cloud](#)
- [Build a Foundry Nuke conda package for Deadline Cloud](#)
- [Build an Autodesk 3ds Max conda package for Deadline Cloud](#)
- [Build an Adobe After Effects conda package for Deadline Cloud](#)
- [Build a KeyShot conda package for Deadline Cloud](#)
- [Build an Autodesk VRED Core conda package for Deadline Cloud](#)
- [Build an Epic Unreal Engine conda package for Deadline Cloud](#)
- [Build a V-Ray Standalone conda package for Deadline Cloud](#)
- [Build a NeRF Studio conda package for Deadline Cloud](#)
- [Build an Infinigen conda package for Deadline Cloud](#)
- [Build an AutoDock Vina conda package for Deadline Cloud](#)
- [Build an AYON Launcher conda package for Deadline Cloud](#)

Build a Blender conda package for Deadline Cloud

The samples repository on the GitHub website includes conda recipes for the following Blender versions and add-ons. Each recipe builds for Linux 64-bit and Windows 64-bit, and downloads source archives from the Blender Foundation:

- [blender-4.2](#)
- [blender-4.3](#)
- [blender-4.4](#)
- [blender-4.5](#)
- [blender-5.0](#)
- [blender-5.1](#)
- [blender-flipfluids](#): FLIP Fluids physics simulation add-on for Blender.
- [blender-plugin-bundle](#): A bundle of Blender plugins packaged together.

Submit a Blender 4.5 build job from the `conda_recipes` directory of the samples repository:

```
./submit-package-job blender-4.5
```

For details on the Blender packaging approach, see the [blender-4.5 recipe README](#) on the GitHub website. To install Blender add-ons, place the .py or .zip file in a known location inside \$INSTALL_DIR and modify the activate script to install the add-on with Blender's Python.

Build an Autodesk Maya conda package for Deadline Cloud

The samples repository on the GitHub website includes the following Maya conda recipes. Each recipe configures the MAYA_MODULE_PATH environment variable so that Maya loads plugin .mod files placed in the standard plugin paths:

[maya-2025](#), [maya-2026](#), [maya-2027](#)

Recipes for Autodesk Maya 2025, 2026, and 2027.

[maya-mtoa-2025](#), [maya-mtoa-2026](#), [maya-mtoa-2027](#)

Maya to Arnold (MtoA) renderer plugin. The MtoA package also provides the kick standalone renderer, which the [the section called “Arnold standalone renderer”](#) bundle uses.

[maya-vray-2025](#), [maya-vray-2026](#), [maya-vray-7.2-2025](#), [maya-vray-7.2-2026](#)

V-Ray for Maya renderer plugin. The maya-vray-7.2 recipes package V-Ray 7.20.02 (Update 2 DR2) and require the V-Ray for Maya archive from [V-Ray for Maya](#) on the Chaos website (account required).

[maya-redshift-2025](#), [maya-redshift-2026](#)

Redshift for Maya renderer plugin.

[maya-bifrost-2026](#)

Bifrost simulation framework for Maya 2026.

[maya-openjd](#)

Builds the Maya Open Job Description adaptor with rattler-build for Python 3.13.

To build the recipe, you need to provide the Maya source archive. Download the Maya source archive from Autodesk and place it in the conda_recipes/archive_files directory of your samples repository clone. The exact filename varies by version — for example, Maya 2025 uses Autodesk_Maya_2025_Linux_64bit.tgz while Maya 2026 uses Autodesk_Maya_2026_ML_Linux_64bit.tgz. Refer to each recipe's README for the expected filename.

The Maya Windows installer requires Administrator permissions that aren't available in most conda package build environments. For step-by-step instructions on installing Maya on a fresh EC2 Windows Server instance and creating a redistributable archive that you can upload to your private Amazon S3 bucket and reuse for other Maya recipes, see the [maya-2025 recipe README](#) on the GitHub website.

Submit the build:

```
./submit-package-job maya-2026
```

For details on the Maya packaging approach, see the [maya-2026 recipe README](#) on the GitHub website.

Build a SideFX Houdini conda package for Deadline Cloud

The samples repository on the GitHub website includes the following Houdini conda recipes:

- [houdini-20.5](#)
- [houdini-21.0](#)
- [houdini-22.0](#): Houdini 22.0 builds are compiled with GCC 14.2, so download the gcc14.2 source archive from SideFX rather than the archive that earlier versions use.
- [houdini-redshift-2025](#) and [houdini-redshift-2026](#): Redshift renderer for Houdini.
- [houdini-vray-7](#): V-Ray 7 renderer for Houdini.

The [the section called “Houdini Husk USD render”](#) job bundle uses these recipes to render USD scenes with Karma, V-Ray, and Redshift Hydra render delegates.

Submit the build:

```
./submit-package-job houdini-21.0
```

Build a Maxon Cinema 4D conda package for Deadline Cloud

The samples repository on the GitHub website includes the following Cinema 4D conda recipes:

- [cinema4d-2024](#), [cinema4d-2025](#), and [cinema4d-2026](#). The cinema4d-2026 recipe packages Cinema 4D 2026.3.3 for Windows workers and includes Plugin Sync activation hooks that download plugins from an Amazon S3 prefix during a session.

- [cinema4d-c4dtoa-2025](#): Arnold renderer (C4DtoA) for Cinema 4D 2025.
- [cinema4d-vray-2025](#): V-Ray for Cinema 4D 2025.
- [cinema4d-insydium-2025](#): Insydium plugins for Cinema 4D 2025.
- [cinema4d-openjd](#): Builds the Cinema 4D Open Job Description adaptor with rattler-build for Python 3.13.

The Cinema 4D Windows installer requires Administrator permissions that are not available in most conda package build environments. Each recipe README includes step-by-step instructions for installing Cinema 4D on a fresh EC2 Windows Server instance and creating a redistributable archive. Upload the archive to your private Amazon S3 bucket, then download it to the `conda_recipes/archive_files` directory of your samples repository clone.

The Cinema 4D 2025 package includes the standalone Redshift command line renderer, which the [the section called “Redshift render”](#) job bundle uses for GPU rendering.

Submit the build:

```
./submit-package-job cinema4d-2025
```

Build a Foundry Nuke conda package for Deadline Cloud

The following Nuke conda recipes are available on the GitHub website:

- [nuke-16.0](#): Foundry Nuke 16.0 for Linux 64-bit.
- [nuke-17.0](#): Foundry Nuke 17.0 for Linux 64-bit.
- [nuke-denoise](#): Nuke De:Noise OFX plugin. The recipe demonstrates how to package OFX plugins for Nuke. Place `.bundle` files in a known directory in `$PREFIX` and set the `OFX_PLUGIN_PATH` environment variable.

Each recipe README includes a version update checklist for adapting the recipe to other Nuke versions.

To build a recipe, download the matching Nuke source archive from the Foundry website (for example, `Nuke17.0v1-linux-x86_64.tgz`) and place it in the `conda_recipes/archive_files` directory of your samples repository clone. A Foundry account is required to access the download.

Submit the build:

```
./submit-package-job nuke-17.0
```

Build an Autodesk 3ds Max conda package for Deadline Cloud

The samples repository does not include a 3ds Max conda recipe at this time. To install 3ds Max on a Windows service-managed fleet, use the [3ds Max host configuration scripts](#) on the GitHub website. For details, see [the section called “3ds Max host configuration”](#).

Build an Adobe After Effects conda package for Deadline Cloud

The samples repository on the GitHub website includes the following After Effects conda recipes:

- [aftereffects-25.1](#): Adobe After Effects 25.1.
- [aftereffects-plugin-bundle](#): A bundle of After Effects plugins.
- [aftereffects-saber](#): The Saber plugin for After Effects.

Submit the build:

```
./submit-package-job aftereffects-25.1
```

Build a KeyShot conda package for Deadline Cloud

The [keyshot-2025](#) conda recipe on the GitHub website builds a KeyShot 2025 conda package.

Submit the build:

```
./submit-package-job keyshot-2025
```

For a job bundle that uses this package, see [the section called “KeyShot standalone”](#).

Build an Autodesk VRED Core conda package for Deadline Cloud

The samples repository on the GitHub website includes the following VRED conda recipes:

- [vredcore-2025](#)

- [vredcore-2026](#)

Submit the build:

```
./submit-package-job vredcore-2026
```

For a job bundle that uses this package, see [the section called “VRED render”](#).

Build an Epic Unreal Engine conda package for Deadline Cloud

The [unreal-engine](#) conda recipe on the GitHub website builds an Epic Unreal Engine conda package.

Unlike other recipes in this repository, this one is designed to be built *locally* on a machine where Unreal Engine is already installed. Unreal Engine packages are large (UE 5.6 is approximately 20 GiB across about 80,000 files), so submitting them to a queue as a job attachment is impractical.

To use the recipe:

1. Install Unreal Engine on your Windows machine. The default source path in the recipe is the Epic Games Launcher path C:\Program Files\Epic Games\UE_5.6. Adjust `recipe/recipe.yaml` for your version or custom source build location.
2. Install [rattler-build](#) on the prefix.dev website and enable Windows long path support.
3. Build and publish the package to your conda channel:

```
rattler-build publish path-to-recipe-file --to publish-conda-channel
```

The recipe README includes step-by-step instructions for adapting it to your Unreal Engine version, including custom source builds.

The samples repository also includes [unreal-engine-openjd](#) on the GitHub website, which builds the Unreal Engine Open Job Description adaptor with rattler-build for Windows and Python 3.13.

Build a V-Ray Standalone conda package for Deadline Cloud

The [vray](#) conda recipe on the GitHub website builds a V-Ray Standalone renderer conda package. For more information about V-Ray Standalone, see [V-Ray Standalone Home](#) on the Chaos website.

To build the recipe, download the V-Ray Standalone archive (vraystd_adv_71000_rhel8_clang-gcc-11.2 for x86 or vraystd_adv_71000_rhel8_arm64_clang-gcc-11.2 for ARM) from the [Chaos download page](#) on the Chaos website. A Chaos account is required. Place the file in the `conda_recipes/archive_files` directory of your samples repository clone.

Submit the build:

```
./submit-package-job vray
```

The queue's IAM role needs `s3:PutObject` permission for the `Conda/*` prefix in the job attachments bucket to publish the built package.

For a job bundle that uses this package, see [the section called “V-Ray render”](#).

Build a NeRF Studio conda package for Deadline Cloud

The [nerfstudio](#) rattler-build recipe on the GitHub website packages [NeRF Studio](#) from the Nerfstudio website plus the following:

- The [Splatfacto in the Wild](#) external model.
- NeRF Studio dependencies that are not yet available on conda-forge.
- The [nerfstudio/gspat](#) package's `simple_trainer.py` example, available as the command `gspat_simple_trainer`.
- The [KevinXu02/splatfacto-w](#) package's `export_script.py`, available as the command `splatfactow_export`.

To build this recipe, deploy the [CUDA farm CloudFormation template](#) on the GitHub website to create a Deadline Cloud farm with a CUDA fleet for the build, then submit the package build job:

```
./submit-package-job nerfstudio
```

To reuse conda environments between jobs, attach the [conda_queue_env_improved_caching.yaml](#) queue environment on the GitHub website to your queue. Because the dependency closure of NeRF Studio contains many gigabytes of packages, caching saves significant time and bandwidth.

For a job bundle that uses this package, see [the section called “3D Gaussian Splatting pipeline”](#).

Build an Infinigen conda package for Deadline Cloud

The [infinigen-1.19.0](#) rattler-build recipe on the GitHub website packages Infinigen, a procedural 3D scene generator from Princeton Vision & Learning Lab. For more information, see [Infinigen](#) on the Infinigen website. The package includes the following:

- Infinigen v1.19.0 procedural scene generation engine.
- bpy 4.2.0 (Blender as a Python module) for headless rendering through Cycles. No standalone Blender installation is needed at runtime.
- Terrain C++ shared libraries (CPU and optional CUDA) compiled from Infinigen source for landscape generation.

To build this recipe, deploy the [CUDA farm CloudFormation template](#) on the GitHub website to create a Deadline Cloud farm with a CUDA GPU fleet for the build. The build requires a GPU worker (for CUDA terrain compilation) and takes approximately 6 minutes.

Submit the build from the `conda_recipes` directory:

```
./submit-package-job infinigen-1.19.0
```

To reuse conda environments between jobs, attach the [conda_queue_env_improved_caching.yaml](#) queue environment on the GitHub website to your queue. The dependency closure of Infinigen contains many gigabytes of packages, so caching saves significant time and bandwidth.

For a job bundle that uses this package, see [the section called “Infinigen scene generation”](#).

Build an AutoDock Vina conda package for Deadline Cloud

The [autodock-vina-1.2.5](#) rattler-build recipe on the GitHub website packages [AutoDock Vina](#), an open-source molecular docking program that predicts how small molecules (drug candidates) bind to a protein target. The package downloads the pre-built Linux x86_64 binary from the GitHub release and installs it as the `vina` command.

Submit the build from the `conda_recipes` directory:

```
./submit-package-job autodock-vina-1.2.5
```

For a job bundle that uses this package to run parallel virtual screening campaigns, see [the section called “Virtual screening \(AutoDock Vina\)”](#).

Build an AYON Launcher conda package for Deadline Cloud

The [ayon-launcher](#) rattler-build recipe on the GitHub website packages the [AYON Launcher](#) as a conda package. The package provides the pipeline runtime needed for headless publishing on Deadline Cloud workers. It repackages the pre-built AYON Launcher release (cx_Freeze binary) containing the Python 3.11 runtime, ayon-python-api, and core launcher logic.

The conda package provides only the runtime environment. The studio-specific bundle (addons and dependency package) is delivered separately through Deadline Cloud job attachments. This separation means the conda package rarely changes and no rebuild is needed when addons are updated.

The recipe builds for Linux (linux-64) directly from GitHub releases. For Windows (win-64), extract the installer on a Windows machine and place the resulting zip in `conda_recipes/archive_files`. Refer to the recipe's README for exact steps.

Submit the build from the `conda_recipes` directory:

```
./submit-package-job ayon-launcher
```

At runtime, the job must provide the following environment variables:

- `AYON_SERVER_URL` — URL of the AYON server.
- `AYON_API_KEY` — API key for server authentication.
- `AYON_BUNDLE_NAME` — Bundle name to resolve addons from.

CloudFormation template examples for Deadline Cloud

The `deadline-cloud-samples` repository contains CloudFormation templates that deploy Deadline Cloud farms and supporting infrastructure. Use these templates as a starting point and adapt them to your environment. For the templates, see the [cloudformation](#) directory on the GitHub website.

To compare CloudFormation with the AWS CDK and Terraform for managing Deadline Cloud infrastructure, see [Infrastructure as code](#).

Topics

- [Deploy a starter Deadline Cloud farm with CloudFormation](#)
- [Deploy a Deadline Cloud CUDA farm with CloudFormation](#)

- [Connect a Deadline Cloud fleet to FSx for OpenZFS through a VPC resource endpoint](#)
- [Manage hybrid Wait and Save plus Spot fleet capacity with CloudFormation](#)
- [Schedule standby workers for a Deadline Cloud fleet with CloudFormation](#)
- [Monitor a Deadline Cloud customer-managed fleet health check with CloudFormation](#)
- [Budget threshold notifications to email and Slack with CloudFormation](#)
- [Job event Slack notifications with Lambda and EventBridge](#)

Deploy a starter Deadline Cloud farm with CloudFormation

The `starter_farm` CloudFormation template deploys a Deadline Cloud farm that you can use to run jobs that render images, reconstruct 3D scenes, or transform data. For the template source, see [starter_farm](#) on the GitHub website. The deployed farm includes:

- One or more [service-managed fleets](#) that you select during deployment.
- A production queue with conda virtual environments for the applications that jobs need.
- A package build queue that you can use to build conda packages.
- Two conda channels by default: a private channel on an Amazon S3 bucket you provide and the [deadline-cloud channel](#), which provides applications such as Blender, Houdini, Maya, and Nuke.

To deploy the template, you need an Amazon S3 bucket to hold job attachments and the conda channel, and a Deadline Cloud monitor with IAM Identity Center to view and manage jobs.

From the CloudFormation console, choose *Create Stack*, upload `deadline-cloud-starter-farm-template.yaml`, enter a stack name and the Amazon S3 bucket name, and proceed through stack creation. To use conda-forge, set the `ProdCondaChannels` parameter to `deadline-cloud conda-forge`. For more information, see [conda-forge](#) on the conda-forge website.

Deploy a Deadline Cloud CUDA farm with CloudFormation

The `cuda_farm` CloudFormation template deploys a Deadline Cloud farm that you can use to run CUDA jobs. The default configuration includes a queue for CUDA jobs, a second queue for building conda packages, and a CUDA-capable fleet. For the template source, see [cuda_farm](#) on the GitHub website.

This template specializes the [the section called “Starter farm”](#) template for the CUDA use case:

- The default `CondaChannels` value is `deadline-cloud conda-forge`, which makes CUDA compilers, frameworks like PyTorch, and applications like COLMAP available.
- The Linux CUDA fleet is required, and the Windows and Linux CPU fleets are removed.

This template provisions the infrastructure that the [the section called “3D Gaussian Splatting pipeline”](#) and [the section called “vLLM benchmark leaderboard”](#) job bundles need.

Connect a Deadline Cloud fleet to FSx for OpenZFS through a VPC resource endpoint

The `smf_vpc_fsx` CloudFormation template deploys a Deadline Cloud service-managed fleet that connects to FSx for OpenZFS storage through a VPC resource endpoint. The FSx cluster runs in a VPC, and a VPC Lattice resource configuration establishes the connection between Deadline Cloud workers and the storage. The resource configuration is shared with the Deadline Cloud service through AWS RAM. For the template source, see [smf_vpc_fsx](#) on the GitHub website.

Use this pattern when you need Deadline Cloud workers to access the following:

- Shared file storage such as FSx for Lustre, FSx for OpenZFS, or EFS.
- License servers running in your VPC.
- Other private resources that aren't accessible from the public internet.

Deployment requires two steps because the FSx IP address is only available after the file system creates its network interface. After the initial stack reaches `CREATE_COMPLETE` (about 10–15 minutes for FSx), update the stack with the FSx IP to complete the configuration.

Manage hybrid Wait and Save plus Spot fleet capacity with CloudFormation

The `smf_capacity_manager` CloudFormation template implements automated capacity management for hybrid fleet setups that combine Wait and Save and Spot fleets. The template uses Lambda and EventBridge Scheduler to dynamically balance fleet sizes while maintaining a constant total capacity. The capacity manager monitors the Wait and Save fleet's active worker count and automatically adjusts the Spot fleet's maximum worker count to cover any deficit. For the template source, see [smf_capacity_manager](#) on the GitHub website.

For example, with a target maximum worker count of 20 workers:

- Wait and Save has 15 workers. Spot fleet maximum is set to 5.
- Wait and Save scales down to 8. Spot fleet maximum increases to 12.
- Wait and Save scales up to 18. Spot fleet maximum decreases to 2.

Workers are only terminated when their tasks complete, so fleets rebalance for cost-effectiveness without losing work in progress.

To deploy the template, you need an existing Deadline Cloud farm with two service-managed fleets (Wait and Save and Spot) that have the same worker capabilities. Set the Spot fleet's `minWorkerCount` to 0 so that the capacity manager can scale it down to zero when the Wait and Save fleet is at full capacity.

Schedule standby workers for a Deadline Cloud fleet with CloudFormation

The `fleet_standby_scheduling` CloudFormation template schedules [standby worker count](#) changes on a Deadline Cloud fleet based on a time schedule. Standby workers are a warm pool of idle workers that can start processing jobs immediately without waiting for new instances to launch. By scheduling standby workers during business hours and not during off-hours, you reduce job start latency when your team is active while avoiding the cost of idle workers overnight and on weekends. For the template source, see [fleet_standby_scheduling](#) on the GitHub website.

The template works with any existing Deadline Cloud fleet, including service-managed and customer-managed fleets. By default, it sets the standby worker count to 2 at 8:00 AM UTC Monday through Friday and resets it to 0 at 5:00 PM UTC.

The template creates the following resources:

- A Lambda function that calls the Deadline Cloud `UpdateFleet` API. The function reads the fleet's current configuration first to avoid overwriting other settings.
- IAM roles for the Lambda function and the EventBridge scheduler.
- Two EventBridge schedules that trigger the Lambda function at the start and end of business hours.

To customize the schedule, update the `BusinessHoursStartCron`, `BusinessHoursEndCron`, and `ScheduleTimezone` parameters. To schedule multiple fleets, deploy one stack per fleet.

Monitor a Deadline Cloud customer-managed fleet health check with CloudFormation

The `cmf_templates` directory includes a `deadline-fleet-health-check.yaml` CloudFormation template that sets up continuous health check monitoring for a single Deadline Cloud customer-managed fleet with auto scaling. The template creates a Lambda function, an EventBridge rule, and a CloudWatch alarm that you can configure with an SNS topic. For more information, see [cmf_templates](#) on the GitHub website.

To deploy the template, you need:

- A Deadline Cloud farm and customer-managed fleet. Copy the Farm ID, Fleet ID, and Fleet Name from the fleet details page in the Deadline Cloud management console.
- An EC2 Auto Scaling group that scales the fleet. Copy the Auto Scaling group name from the EC2 Auto Scaling console.
- An SNS topic and subscription for the health check alarm. Copy the SNS topic ARN.

For more information about customer-managed fleet auto scaling, see [the section called “Create fleet infrastructure”](#).

Budget threshold notifications to email and Slack with CloudFormation

The [budget_events_notification](#) CloudFormation template on the GitHub website sets up notifications by email and to Slack when a budget threshold is reached in the `aws.deadline` service. The template creates the following resources:

- An SNS topic with an email subscription.
- An SNS topic policy that allows AWS services such as EventBridge to publish messages to the topic.
- An IAM role that grants AWS Chatbot permission to publish messages.
- An AWS Chatbot configuration that sends notifications to your Slack channel.
- An EventBridge rule that captures Budget Threshold Reached events from `aws.deadline` and sends them to the SNS topic.

To deploy the template for Slack notifications, add AWS Chatbot to your Slack workspace, then copy the Slack channel ID and workspace ID for use as template parameters.

For more information about Deadline Cloud budgets and events, see [the section called “Managing events using EventBridge”](#).

Job event Slack notifications with Lambda and EventBridge

The [job_events_slack_lambda](#) CloudFormation template on the GitHub website sends a Slack notification whenever a job completes (SUCCEEDED) or fails (FAILED). Use it as a starting point for connecting a Lambda function to Deadline Cloud job events through EventBridge. You can adapt the function to send an email, open a ticket, update a dashboard, or trigger a downstream workflow.

The template creates the following resources:

- A Lambda function is defined inline in the template using only the Python standard library. It formats the job event and posts it to a Slack Incoming Webhook URL stored in its `SLACK_WEBHOOK_URL` environment variable.
- An EventBridge rule matches Job Run Status Change events with a `taskRunStatus` of SUCCEEDED or FAILED, and invokes the function. You can scope the rule to a single farm.
- An execution role grants the function only CloudWatch Logs write access. A permission allows EventBridge to invoke the function.

Many messaging apps expose the same style of incoming webhook. To target Microsoft Teams, Discord, Google Chat, or Mattermost instead, set that app's webhook URL and adjust the JSON body the function builds; the README links each app's webhook documentation.

For more information about the events Deadline Cloud publishes to EventBridge, see [the section called “Managing events using EventBridge”](#). For budget threshold notifications, see [the section called “Budget notifications”](#).

Deploy Deadline Cloud farms with the AWS CDK

The [starter_farm](#) AWS CDK app in the `deadline-cloud-samples` repository on the GitHub website deploys Deadline Cloud farms from reusable TypeScript constructs that wrap the `CfnFarm`, `CfnQueue`, and `CfnFleet` L1 resources from `aws-cdk-lib/aws-deadline`. For a comparison of the AWS CDK with CloudFormation and Terraform, see [Infrastructure as code](#).

The app contains four example farms, each a separate AWS CDK stack:

- **SimpleFarm** – A single Linux fleet on one queue. Deploy SimpleFarm when you are getting started.
- **StarterFarm** – Adds a private conda channel and a queue that builds packages for it.
- **CudaFarm** – A GPU fleet, with conda-forge for the CUDA toolchain.
- **MultiPlatformFarm** – One queue reaching Linux, Windows, and GPU fleets.

Every stack creates a farm and an Amazon S3 bucket for job attachments. Each stack also includes at least one queue with a conda queue environment, at least one service-managed fleet, and one IAM role for each queue and each fleet.

To deploy, you need Node.js 18 or later, AWS credentials, an account and region bootstrapped for the AWS CDK, and a Deadline Cloud monitor. Then run:

```
cd cdk/farm_templates/starter_farm
npm ci
npx cdk deploy SimpleFarm
```

Deploy Deadline Cloud farms with Terraform

The [terraform](#) directory in the deadline-cloud-samples repository on the GitHub website contains Terraform configurations that deploy Deadline Cloud farms with `awscc_deadline_*` resources from the AWSCC provider. For a comparison of Terraform with CloudFormation and the AWS CDK, see [Infrastructure as code](#).

The [starter_farm](#) configuration on the GitHub website is the Terraform equivalent of the CloudFormation starter farm ([the section called “Starter farm”](#)). It deploys a farm with a production queue and a package build queue. The farm includes up to three service-managed fleets: CPU Linux, CPU Windows, and CUDA Linux. A conda queue environment uses a private conda channel on an Amazon S3 bucket you provide, plus the `deadline-cloud` channel.

To deploy the starter farm, you need Terraform 1.0 or later, AWS credentials, an Amazon S3 bucket for job attachments and the conda channel, and a Deadline Cloud monitor. Set the required `job_attachments_bucket_name` variable, then run:

```
cd terraform/farm_templates/starter_farm
terraform init
terraform apply
```

The [knfsd_xregion_cache](#) configuration on the GitHub website deploys a service-managed fleet that reads a distant NFS filer through a KNFSd read cache over a VPC resource endpoint. Use it as a starting point when your fleet caches reads from an on-premises or otherwise-distant filer.

Host configuration script examples for Deadline Cloud

Host configuration scripts run with elevated privileges on service-managed fleet workers, which lets you install software, configure runtimes, and perform other administrative tasks. The [host_configuration_scripts](#) directory in the deadline-cloud-samples repository on the GitHub website includes sample scripts for common configuration tasks.

To attach a script to a fleet, use the Deadline Cloud console or the `update-fleet` CLI command. For details, see [the section called “Host configuration scripts”](#).

Fleet host configuration logs stream to the fleet's CloudWatch log group (for example, `/aws/deadline/farm-FARM_ID/fleet-FLEET_ID`). Each worker has a dedicated log stream that includes the Running Host Configuration Script and Finished running Host Configuration Script, `exit code: banners`.

Topics

- [Run Docker containers with NVIDIA GPUs on Deadline Cloud workers](#)
- [Install Autodesk 3ds Max on Deadline Cloud Windows workers](#)
- [Install Adobe After Effects with Red Giant on Deadline Cloud Windows workers](#)
- [Install Cinema 4D with Red Giant on Deadline Cloud Windows workers](#)
- [Install custom fonts on Deadline Cloud Linux workers](#)
- [Grant passwordless sudo to job-user on Deadline Cloud Linux workers](#)
- [Enable swap on Deadline Cloud Linux workers](#)
- [Configure the Windows page file on Deadline Cloud workers](#)
- [Reboot a Deadline Cloud worker after host configuration](#)

Run Docker containers with NVIDIA GPUs on Deadline Cloud workers

The [docker_nvidia_container_toolkit](#) host configuration script on the GitHub website installs Docker and the NVIDIA Container Toolkit on Linux service-managed fleet workers, which lets jobs run GPU containers directly with `docker run`. Many GPU workloads, such as ComfyUI and Stable Diffusion inference servers, ship as container images and require Docker with GPU passthrough.

Containers also provide a clean way to package complex dependency stacks (CUDA, Python, application-specific libraries) without polluting the host or conflicting with other jobs on the fleet.

The script does the following:

1. Installs Docker through `dnf` and starts the service.
2. Adds `job-user` to the `docker` group so that jobs can run containers without `sudo`.
3. Installs the NVIDIA Container Toolkit from the official repository.
4. Configures the Docker daemon to use the NVIDIA runtime.
5. Generates the CDI (Container Device Interface) spec for GPU access.
6. Restarts Docker and verifies the setup.

The fleet AMI must have NVIDIA GPU drivers already installed, which Deadline Cloud GPU AMIs provide, and the fleet must use a GPU instance type such as `g6.xlarge`, `g6e.xlarge`, or `p4d.24xlarge`.

Once the host configuration has run, jobs can launch GPU containers. The following example runs a container with GPU access, host networking, and the standard Deadline Cloud usage-based license environment variables passed through:

```
docker run --rm \
  --runtime=nvidia \
  --gpus all \
  --network host \
  -e ADSKFLEX_LICENSE_FILE \
  -e FLEXLM_TIMEOUT \
  -e foundry_LICENSE \
  -e PIXAR_LICENSE_FILE \
  -e g_licenseServerRLM \
  -e redshift_LICENSE \
  -e SESI_LMHOST \
  -e VRAY_AUTH_CLIENT_FILE_PATH \
  -e VRAY_AUTH_CLIENT_SETTINGS \
  your-image:latest
```

Install Autodesk 3ds Max on Deadline Cloud Windows workers

The [3dsmax](#) host configuration scripts on the GitHub website install 3ds Max with renderer plugins such as V-Ray, Corona, tyFlow, and AEC plugins on Windows service-managed fleet

workers. Because 3ds Max runs only on Windows and requires administrative access to install, the recommended approach is to install it on the worker host through a host configuration script.

The samples repository on the GitHub website includes scripts for the following version and plugin combinations:

- [3dsmax-2024](#) — 3ds Max 2024.
- [3dsmax-2025-and-vray](#) — 3ds Max 2025 with V-Ray.
- [3dsmax-2025-and-corona-13](#) — 3ds Max 2025 with Chaos Corona 13.
- [3dsmax-2025-vray-and-aec-plugins](#) — 3ds Max 2025 with V-Ray and AEC plugins.
- [3dsmax-2025-vray-and-tyflow](#) — 3ds Max 2025 with V-Ray and tyFlow.
- [3dsmax-2025-and-pencilplus-4](#) — 3ds Max 2025 with PSOFT Pencil+ 4.
- [3dsmax-2027](#) — 3ds Max 2027.
- [3dsmax-2027-and-vray](#) — 3ds Max 2027 with V-Ray.
- [3dsmax-2027-and-corona-14](#) — 3ds Max 2027 with Chaos Corona 14.
- [3dsmax-2027-and-vray-and-tyflow](#) — 3ds Max 2027 with V-Ray and tyFlow.
- [3dsmax-2027-vray-and-aec-plugins](#) — 3ds Max 2027 with V-Ray and AEC plugins.
- [3dsmax-2027-and-pencilplus-4](#) — 3ds Max 2027 with PSOFT Pencil+ 4.

To verify that a worker has 3ds Max and V-Ray configured correctly, submit the [sunflower_sphere](#) test job bundle on the GitHub website, included with the scripts.

To generate a script for a different version, renderer, or plugin combination, use the bundled [3dsmax-host-config skill](#) on the GitHub website. The skill runs in the [Kiro](#) AI agent on the Kiro website.

Each script downloads the 3ds Max installer from an Amazon S3 bucket in your AWS account and runs it in silent mode. Autodesk provides 3ds Max as a .7z archive, which the scripts expect repackaged as .zip. Each script's README includes instructions for creating the .zip archive and uploading it to Amazon S3.

Your fleet IAM role needs `s3:GetObject` permission for the installers in Amazon S3.

Install Adobe After Effects with Red Giant on Deadline Cloud Windows workers

The [aftereffects_redgiant](#) host configuration script on the GitHub website installs Adobe After Effects 2025 with Red Giant and Universe plugins on Windows GPU service-managed fleet workers. The script fetches installers from Amazon S3 and runs each one in silent mode on each worker launch.

Important

This script can add about 15-20 minutes to worker launch time because of the software installation. The duration decreases as you vertically scale your instance size up. For example, a `g6.xlarge` with 4 vCPUs and 16 GiB of memory adds about 20 minutes, while a `g6.4xlarge` with 16 vCPUs and 64 GiB of memory adds about 15 minutes. To reduce the impact, keep a warm worker alive during peak usage hours, or use one fleet for After Effects renders through conda and a separate fleet for After Effects with Red Giant through this host configuration script.

The script uses a persistent volume automatically when your fleet has one configured. The first worker installs the software once to the persistent volume; subsequent worker boots restore it in seconds instead of reinstalling. If no persistent volume is attached, the script performs a normal install. No separate script or flag is required. Configure a persistent volume if your fleets scale up and down frequently. For more information about how the script uses persistent volumes, see [Persistent Volumes \(Automatic\)](#) in the `aftereffects_redgiant` README on the GitHub website. For more information about configuring persistent volumes on a fleet, see [the section called "Persistent storage"](#).

To use this script, you need:

- An enterprise Adobe account with Admin access to download the After Effects 2025 installer package from the Adobe Admin Console.
- A Maxon account to download the Red Giant and Universe installers.
- An Amazon S3 bucket where you upload the installers, and IAM permissions for the fleet to read them.
- A Windows GPU service-managed fleet with the latest GPU driver.

For an After Effects job bundle that uses this fleet configuration, see [the section called “After Effects render”](#).

Install Cinema 4D with Red Giant on Deadline Cloud Windows workers

The [cinema4d_redgiant](#) host configuration script on the GitHub website installs Cinema 4D with Red Giant plugins on Windows GPU service-managed fleet workers. The script fetches the installers from Amazon S3 and runs each one in silent mode on each worker launch.

Important

This script adds about 5-10 minutes to worker launch time, depending on instance size. Plan accordingly for fleet scaling and job scheduling.

To use this script, you need:

- A Maxon account to download the Red Giant and Maxon App installers.
- The Microsoft Edge WebView2 Runtime installer (required by the Maxon App installation).
- An Amazon S3 bucket where you upload the installers, and IAM permissions for the fleet to read them.
- A Windows GPU service-managed fleet with the latest GPU driver.
- Red Giant licenses, available on SMF and CMF through a license endpoint.

Install custom fonts on Deadline Cloud Linux workers

The [linux_font_installation](#) host configuration script on the GitHub website installs fonts from an Amazon S3 bucket on Deadline Cloud Linux service-managed fleet workers, which makes the fonts available to applications such as Nuke.

Note

Fonts may render differently across operating systems.

To use this script:

1. Upload your font files to an Amazon S3 bucket. To save time at instance startup, store fonts in a dedicated folder so the script copies only font files.

```
aws s3 cp /path/to/your/fonts/ s3://your-bucket-name/Fonts/ --recursive
```

2. Edit `font_install_host_config.sh` to set `S3_FONTS_URI` and `JOB_USER`.
3. Add an IAM policy to your fleet IAM role that grants `s3:GetObject` and `s3:ListBucket` permissions for the bucket and prefix.

Grant passwordless sudo to job-user on Deadline Cloud Linux workers

The [sudo_for_job_user](#) host configuration script on the GitHub website grants the Deadline Cloud `job-user` account passwordless sudo access on Linux service-managed fleet workers. Some workloads require root privileges, for example to install packages, mount file systems, or register the worker as an SSM managed node. By default, `job-user` doesn't have sudo access.

The script adds a sudoers rule that allows `job-user` to run any command as root without a password prompt. If `job-user` doesn't yet exist (for example, the worker agent hasn't created it), the script logs a warning and exits successfully so that it doesn't block fleet provisioning.

Important

Granting passwordless sudo to `job-user` means that any job running on the worker can execute arbitrary commands as root. Only enable this script on fleets where you trust the jobs being submitted, and consider scoping the sudoers rule to specific commands when your use case allows it.

This script is required by the Linux variant of the [the section called "SSH to worker"](#) sample.

Enable swap on Deadline Cloud Linux workers

The [swap_for_smf](#) host configuration script on the GitHub website creates and enables a swap file on Deadline Cloud Linux service-managed fleet workers. Use this script for memory-intensive workloads that temporarily exceed physical RAM, such as ComfyUI.

Some workloads need to memory-map large files through CPU RAM before loading them onto the GPU. For example, loading a 16 GB diffusion model on a `g6.xlarge` instance (16 GB RAM)

can OOM-kill the process without swap. A swap file provides overflow capacity so that these loads succeed.

The script does the following:

1. Creates a 32 GB swap file at `/swapfile` (skips if one already exists).
2. Enables the swap file immediately.
3. Adds an `/etc/fstab` entry so that swap persists across reboots.

To change the swap size, edit `linux.sh` and update the `SWAP_SIZE` variable.

For an alternative that changes the kernel's overcommit behavior instead of adding swap, see the [overcommit_override_for_smf](#) script on the GitHub website.

Configure the Windows page file on Deadline Cloud workers

The [worker_configuration/windows/configure_page_file.ps1](#) script on the GitHub website configures the Windows page file size and placement on Deadline Cloud workers. It prefers local NVMe instance storage when available for the best performance.

The page file sizing logic works as follows:

- For NVMe drives, the script uses the smaller of $2 \times$ RAM or 75% of NVMe space by default.
- If NVMe isn't available, the script uses the largest non-boot drive with $2 \times$ RAM.
- If no non-boot drives are available, the script falls back to the boot drive (C:).

The script automatically detects EC2 NVMe instance storage, disables automatic page file management, formats the drive and assigns a drive letter if needed, then reboots the worker to apply changes. A marker file at `C:\deadline-pagefile-configured` prevents reconfiguration on subsequent worker starts. Adjust the `$RAM_MULTIPLIER`, `$NVME_SPACE_PERCENTAGE`, and `$MIN_DISK_SIZE_GB` variables at the top of the script to match your workload.

Reboot a Deadline Cloud worker after host configuration

The [worker_reboot](#) sample on the GitHub website includes Linux and Windows scripts that reboot a Deadline Cloud worker after host configuration completes. Worker reboots are commonly required after driver installations or domain joins.

When the worker starts, host configuration scripts run during the STARTED state. The Linux script issues a standard `reboot` command and sleeps for 60 seconds while the host shuts down to prevent the worker agent from starting job processing. A rebooted marker file indicates that a reboot was issued. On the second start, the Deadline Cloud agent begins processing jobs.

The Windows script uses a marker file at `C:\deadline-rebooted` and issues `Restart-Computer -Force` if the marker file doesn't yet exist.

Queue environment examples for Deadline Cloud

The [queue_environments](#) directory on the GitHub website includes queue environments that you can attach to a Deadline Cloud queue. Conda and Rez queue environments provide software to jobs so that each job only needs a parameter value for `CondaPackages` or `RezPackages` to specify the list of packages to use.

Queue environments follow the [Open Job Description environment template specification](#) on the GitHub website. To attach a queue environment to your queue, see [the section called “Configure jobs using queue environments”](#). For background on environments and how they affect your jobs, see [the section called “Control the job environment”](#).

To create a queue environment with the AWS CLI:

```
aws deadline create-queue-environment \
  --farm-id FARM_ID \
  --queue-id QUEUE_ID \
  --priority 1 \
  --template-type YAML \
  --template file://conda_queue_env_improved_caching.yaml
```

Topics

- [Console-equivalent and improved-caching conda queue environments](#)
- [Inline conda queue environments for Deadline Cloud customer-managed fleets](#)
- [py-rattler conda queue environment for Deadline Cloud](#)
- [Rez queue environment for Deadline Cloud customer-managed fleets](#)
- [Pip queue environment for Deadline Cloud](#)
- [Disconnect Deadline Cloud usage-based licensing with a queue environment](#)

Console-equivalent and improved-caching conda queue environments

The samples repository on the GitHub website includes the following conda queue environments that match the Deadline Cloud console onboarding flow:

[conda_queue_env_from_console.yaml](#)

A copy of the queue environment that the Deadline Cloud console onboarding flow adds. The `onEnter` and `onExit` actions run the `conda-queue-env-enter` and `conda-queue-env-exit` commands, which are provided on service-managed fleet workers and use [Rattler](#) on the GitHub website. These commands typically run faster than equivalent operations with `conda`.

[conda_queue_env_improved_caching.yaml](#)

A version of the console-equivalent queue environment that reuses virtual environments across multiple jobs. This setting can significantly improve performance when running many jobs with the same package requirements. By default, persistent environments are stored under `~/.persistent_envs`; modify the `onEnter` and `onExit` actions to reference a different path.

To get equivalent functionality on customer-managed fleets, see [the section called “Inline conda queue environments”](#).

Inline conda queue environments for Deadline Cloud customer-managed fleets

The samples repository on the GitHub website includes the following inline conda queue environments that run `conda` directly instead of using the `conda-queue-env-enter` command:

[conda_queue_env_inline.yaml](#)

Has nearly the same behavior as the console conda queue environment but runs `conda` directly. The behavior is to create a new conda virtual environment for every Open Job Description session that runs on a worker host, then delete the environment when the session completes. Conda keeps a cache of downloaded packages so it doesn't repeatedly re-download applications, but each session has the overhead of linking packages into the virtual environment.

Unlike the console queue environment, which uses `strict` channel priority, this inline queue environment uses `flexible` channel priority.

[conda_queue_env_inline_improved_caching.yaml](#)

Extends the inline conda queue environment with a mechanism to reuse conda virtual environments across multiple jobs. The default environment name uses the hash of the conda channels and packages, or you can explicitly set the name in the job. A parameter controls how long to use an environment before running a package update, so that activating a reused environment usually takes seconds.

To use these queue environments on customer-managed fleets, install conda on the worker hosts (for example, in your AMI) and configure `conda activate` to work in non-interactive bash shells. The samples [README](#) on the GitHub website includes example setup scripts for Amazon Linux 2023, Ubuntu, and Windows.

py-rattler conda queue environment for Deadline Cloud

The `conda_queue_env_pyrattler.yaml` queue environment provides the same functionality as the inline conda queue environments but uses the py-rattler library. Rattler is written in Rust and provides common functionality used in the conda ecosystem. For more information about the queue environment, see [conda_queue_env_pyrattler.yaml](#) on the GitHub website. For more information about py-rattler, see [py-rattler](#) on the conda website.

Testing has shown that this queue environment generally runs faster than the inline conda queue environments on the same instance types. The environments it creates are nearly the same. Two differences:

- py-rattler doesn't include `pip` alongside `python` by default, so you must add `pip` explicitly when you need it.
- py-rattler raises an error for a subset of syntax that conda accepts, such as `colmap=*gpu*`.

The error messages py-rattler produces when failing to solve for a virtual environment don't include as much detail as conda's. If you need more detailed solver errors during development, switch to one of the inline conda queue environments.

Rez queue environment for Deadline Cloud customer-managed fleets

The [rez_queue_env.yaml](#) queue environment on the GitHub website provides the same functionality as the conda queue environments but for the [Rez](#) package manager on the Rez

website. The queue environment works in a farm with customer-managed fleets that have a shared file system for the Rez package repository.

To use this queue environment on a customer-managed fleet, install Rez on the worker hosts (for example, in your AMI). Modify the `RezRepositories` default value to point to the shared file system path of your Rez package repository.

When you submit a job, set the `RezPackages` parameter to the list of packages the job needs. The queue environment creates and activates a Rez environment containing the requested packages and their dependencies before each job runs.

The [rez_shim](#) variant on the GitHub website applies a resolved Rez context to each task by wrapping its command instead of replaying environment variables. Choose it if your Rez packages configure software with anything other than plain environment variables, such as an `alias` for a launcher, a shell function, or a `PATH` entry that must shadow a system binary. Job templates keep calling tools by bare name, so job bundles need no changes. If your packages only set variables, the simpler `rez_queue_env.yaml` works and needs no extra pieces.

Pip queue environment for Deadline Cloud

The `pip_queue_env.yaml` queue environment provides Python packages to jobs using pip and the standard library `venv` module, rather than a package manager such as conda or Rez. For more information about the queue environment, see [pip_queue_env.yaml](#) on the GitHub website. For more information about pip, see [pip](#) on the pip website. For more information about venv, see [venv](#) on the Python website.

When a job provides a `PipPackages` parameter value, the queue environment creates a Python virtual environment in the session working directory, installs the requested packages into it with pip, and activates it so that the job's steps run with those packages available. If `PipPackages` is empty, the queue environment does nothing, so you can add it to a queue that also runs jobs that don't use it.

The `PipIndexUrl` and `PipExtraIndexUrls` parameters let jobs install from a private package index, such as an [AWS CodeArtifact](#) repository, instead of the default [PyPI](#) index on the PyPI website.

Unlike conda and Rez, pip and venv are included with Python itself, so worker hosts only need a `python3` (or `python`) interpreter on the `PATH`. Deadline Cloud service-managed fleets provide one.

The [pip_package_job](#) job bundle on the GitHub website shows how to submit a job that uses this queue environment. For a job bundle that defines the same pip environment inline without a queue environment, see [pip_self_contained_job](#) on the GitHub website.

Disconnect Deadline Cloud usage-based licensing with a queue environment

The `disconnect_ubl_queue_env.yaml` queue environment unsets Deadline Cloud usage-based license (UBL) environment variables. Use this queue environment when you want to turn off all connections to Deadline Cloud UBL for your queue and force the use of a custom license server. For more information about the queue environment, see [disconnect_ubl_queue_env.yaml](#) on the GitHub website. For more information about bring your own license, see [the section called “Connect SMF fleets to a license server”](#).

Set the priority of this queue environment to 0 so that it runs before any other queue environments. Otherwise, connections to your custom floating licenses (such as RLM) in other queue environments can be unset accidentally.

Note

This queue environment is a sample. Additional UBL environment variables can be added in the future.

Build a worker-equivalent Amazon Linux 2023 Docker image for Deadline Cloud

The [al2023-deadline](#) Dockerfile on the GitHub website replicates the package set of the Deadline Cloud service-managed fleet (SMF) worker AMI on top of the base Amazon Linux 2023 image. Use the image to:

- Build and test conda packages with the same GLIBC version, system libraries, and runtime environment as real workers.
- Reproduce worker-side build or runtime failures locally.
- Validate that your software dependencies are satisfied by the worker environment before submitting jobs.

The image installs packages in layered groups that match the worker AMI: core system tools, build toolchain, X11/Mesa/OpenGL, image and media libraries, networking and security utilities, Python 3.11, Docker and containerd, AWS CLI v2, Boost, jemalloc, and TBB.

Build the image:

```
docker build -f Dockerfile.worker-equivalent -t al2023-deadline:latest .
```

Build a conda package inside the container:

```
docker run --rm -v "$PWD":/work -w /work al2023-deadline:latest \
  bash -c "pip3.11 install conda-build && conda build my-recipe/"
```

Important

This image is a point-in-time snapshot. The actual SMF worker AMI may have newer or additional packages. For NVIDIA GPU support, add the NVIDIA Container Toolkit repository to the Dockerfile and run with `--gpus all`.

Build a Blender Docker image for GPU rendering on Deadline Cloud

The [blender-aswf-ci-base](#) example on the GitHub website builds a Docker image that packages Blender with the [deadline-cloud-for-blender](#) adaptor and GPU support (CUDA/OptiX) for rendering on Deadline Cloud service-managed fleets. The base image is `aswf/ci-base:2026` (Rocky Linux 8, CUDA 12.9, VFX Platform 2026).

You can use this image for the following purposes:

- Run Blender Cycles GPU renders on Deadline Cloud service-managed fleets.
- Bundle third-party Blender addons into the image at build time by placing `addon.zip` files in the `plugins/` directory before building.

Build the image:

```
docker build -t blender-aswf:4.5.0 .
```


Push to Amazon Elastic Container Registry (Amazon ECR) and deploy the full stack (queue, fleet, and queue environment) with the included AWS CloudFormation (CloudFormation) template:

```
aws cloudformation deploy \  
  --template-file cloudformation.yaml \  
  --stack-name blender-aswf-ci-base-stack \  
  --parameter-overrides \  
    FarmId=farm-... \  
    ECRImageURI=account-id.dkr.ecr.region.amazonaws.com/repo:4.5.0 \  
    FleetRoleArn=arn:aws:iam::account-id:role/FleetRole \  
    QueueRoleArn=arn:aws:iam::account-id:role/QueueRole \  
    JobAttachmentsBucket=my-deadline-bucket
```

The queue role needs Amazon ECR pull permissions (`ecr:BatchGetImage`, `ecr:GetDownloadUrlForLayer`, and `ecr:GetAuthorizationToken`) to pull the container image at runtime.

GPU rendering is automatic when the fleet has GPU instances. The queue environment conditionally adds `--gpus all --runtime=nvidia` based on whether the host has an NVIDIA GPU. CPU-only instances fall back to Cycles CPU rendering.

For a job bundle that renders Blender scenes, see [the section called “Blender render”](#).

Upload files to Deadline Cloud job attachments

The [upload_to_job_attachments](#) Python script on the GitHub website uploads files and directories from your workstation to a Deadline Cloud job attachments Amazon S3 bucket in content-addressable storage format. After files are uploaded, jobs that reference them don't need to upload them again. The script is useful for pre-populating job attachments with large datasets.

The script provides the following:

- Recursive uploads of files or whole directories.
- Multi-threaded transfers.
- Automatic deduplication that skips files already in Amazon S3.
- Retry logic with exponential backoff.
- Two configuration modes: direct Amazon S3 specification or queue lookup.

Upload by specifying a queue:

```
python upload_to_job_attachments.py \  
  --farm-id farm-1234567890abcdef \  
  --queue-id queue-1234567890abcdef \  
  --paths /path/to/files /path/to/directory
```

Or upload directly to an Amazon S3 bucket and prefix:

```
python upload_to_job_attachments.py \  
  --s3-bucket my-bucket \  
  --s3-prefix job-attachments \  
  --paths /path/to/files
```

For high-bandwidth networks, increase `--threads` and `--max-concurrency`. For bandwidth-constrained environments, set `--max-bandwidth` to throttle uploads. For large files, increase `--multipart-chunksize`.

For a job-based alternative that copies files between Amazon S3 buckets, see [the section called “Copy S3 prefix to job attachments”](#).

Set up a virtual workstation for Deadline Cloud with a script

The [virtual_workstation](#) scripts on the GitHub website turn a fresh Linux or Windows workstation into a Deadline Cloud submission machine. An artist who logs in finds Blender and the Deadline Cloud submitter installed, alongside Deadline Cloud monitor with a profile already configured. The only remaining step is signing in.

Each script takes the monitor URL as its argument and runs five steps. It validates the URL, installs Blender from the official archive, and installs the submitter with its silent installer. It then enables the submitter's Blender add-on, which the silent installer alone doesn't do. Finally, it installs the monitor and creates the profile non-interactively with `deadline-cloud-monitor create-profile`. Every download is verified against a published SHA-256 checksum. Blender stands in for whichever DCC you run; the README describes adapting the scripts to another DCC.

Run the Linux script as root on Ubuntu 22.04 with a desktop environment present:

```
sudo ./setup_workstation_linux.sh https://mystudio.us-west-2.deadlinecloud.amazonaws.com/
```

On Windows, run the PowerShell script in an elevated session as the artist's own account, because the monitor profile and Blender add-on preferences are stored for each user. The README covers running the scripts from EC2 user data or an image bake, the Ubuntu and browser requirements for monitor sign-in, and splitting the Windows script for studios where artists are standard users.

For scripts that configure the workers instead of workstations, see [the section called “Host configuration scripts”](#).

Enforce fixed license limits with a Deadline Cloud submission hook

The [license_limits](#) sample on the GitHub website shows how to combine the Deadline Cloud [Limits](#) feature with a submission hook to enforce a fixed number of concurrent licenses (for example, V-Ray licenses) across all job submissions, without requiring artists to configure anything manually. For more information about submission hooks, see [submission-hooks.md](#) on the GitHub website.

The Deadline Cloud Limits feature throttles task scheduling so that no more than N tasks requiring a license run concurrently. For the limit to take effect, each job must declare the license as a host requirement in its template. This sample provides a pre-submission hook that automatically injects the license host requirement into every job template at submission time. As a result:

- Artists submit jobs normally with no extra steps.
- The hook ensures every job declares its need for the license.
- The Deadline Cloud scheduler enforces the concurrency limit.

To set up the license limits hook

1. Create a Limit on your farm with the AWS CLI:

```
aws deadline create-limit \  
  --farm-id farm-id \  
  --display-name "VRay License" \  
  --amount-requirement-name "amount.vray" \  
  --max-count 5
```

2. Associate the Limit with your queue using `aws deadline create-queue-limit-association`.

3. Copy the `license_limits/` directory to a shared location accessible by all artist workstations.
4. On each artist workstation, enable environment hooks:

```
deadline config set settings.allow_environment_hooks true
```

5. Set the `DEADLINE_HOOKS_DIR` environment variable in your application launcher scripts to point to the shared directory.

Important

Don't add `amount.vray` as a `customAmounts` entry in your fleet's capabilities. If the fleet declares the amount as a capability, the fleet treats it as a per-worker resource and bypasses the queue-level limit. The limit association alone provides compatibility.

To extend the sample to other licenses (Houdini, Nuke, and so on), add entries to `license_limits.json` and create corresponding Limits and queue associations.

AI agent skills for Deadline Cloud

For background on working with AI agents, see [Using AI agents](#). The [skills](#) directory on the GitHub website provides the following Deadline Cloud AI agent skills (SKILL.md files) that you can load into an agent so it follows a known-good procedure:

- [deadline-cloud-job](#) — Walks the agent through creating an Open Job Description job bundle for Deadline Cloud, optionally with a conda package recipe for custom dependencies. Covers designing parameters and steps, writing the template, validating with `openjd check`, iterating locally with `openjd run --tasks`, and submitting to a farm.
- [conda-builder](#) — Walks the agent through creating or updating a conda recipe that uses [rattler-build](#) on the `prefix.dev` website for a DCC, mimicking the structure of existing recipes in the samples repository. Recipes build inside an Amazon Linux 2023 (AL2023) Docker container to match the GLIBC version on service-managed fleet workers, and the agent indexes the result into a local conda channel for testing. The skill currently covers Blender. Per-DCC guides are in the `dcc/` subdirectory.
- [host-config-from-installer](#) — Walks a customer through installing any Windows .exe software locally, verifying it works, uploading the installer to Amazon Simple Storage Service (Amazon

S3), and converting the verified install steps into a PowerShell host configuration script for Windows service-managed fleet workers. Use it when the software isn't covered by a more specific skill (for example, the 3ds Max host configuration).

Security in Deadline Cloud

Security in Deadline Cloud comes down to three questions: who can see and manage your farm, what the service and your jobs can do on your behalf, and how workloads stay separate from each other. For a map of the controls that answer each question and where to configure them, start with [Security controls in Deadline Cloud](#). For hands-on guidance for each part of your farm, see [Security best practices for Deadline Cloud](#).

Topics

- [Security controls in Deadline Cloud](#)
- [Data flow, ports, and encryption in Deadline Cloud](#)
- [Security best practices for Deadline Cloud](#)
- [Verify the authenticity of downloaded software](#)
- [Identity and Access Management in Deadline Cloud](#)
- [Data protection in Deadline Cloud](#)
- [Access AWS Deadline Cloud using an interface endpoint \(AWS PrivateLink\)](#)
- [Restricted network environments](#)
- [Cross-service confused deputy prevention](#)
- [Compliance validation for Deadline Cloud](#)
- [Resilience in Deadline Cloud](#)
- [Infrastructure security in Deadline Cloud](#)
- [Configuration and vulnerability analysis in Deadline Cloud](#)

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 AWS Deadline Cloud, see [AWS services in Scope by Compliance Program](#). AWS Deadline Cloud is in scope for SOC 1, 2, and 3 compliance. For more information, see [the section called “Compliance validation”](#).

- **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.

Security controls in Deadline Cloud

A render farm runs jobs from many people on shared infrastructure. Artists submit scenes, worker hosts run them, and the results land in shared storage. AWS Deadline Cloud gives you a set of controls to decide who can see and change farm resources, what jobs can do while they run, and how workloads stay separate from each other.

Three questions organize the controls:

- **Who can see and manage farm resources?** Monitor users with access levels, and IAM policies for the console and API.
- **What can the service, workers, and jobs do on your behalf?** The IAM service roles for fleets, worker hosts, queues, and monitors.
- **How are workloads kept separate?** Farm and queue structure, operating system users for jobs, and per-queue job attachment buckets.

The following table maps each control to what it governs and where to learn more.

Security controls in Deadline Cloud

Control	What it governs	More information
Monitor users and access levels	Which people can see each farm, queue, and fleet in the monitor, and whether they can view, contribute, manage, or own those resources. Users come from AWS IAM Identity Center (IAM Identity Center).	Managing users in the <i>Deadline Cloud User Guide</i>
IAM identity-based policies	What people and tools with AWS credentials can do with the Deadline Cloud console and API, such as creating farms or submitting jobs from a pipeline.	Identity and Access Management in Deadline Cloud

Control	What it governs	More information
Service roles	What Deadline Cloud can do on your behalf. The fleet, worker host, queue, and monitor roles each grant a different part of the service a scoped set of permissions.	Service roles
Queue operating system user	The operating system permissions that job processes have on worker hosts. On a customer-managed fleet, give each queue its own OS user. On a service-managed fleet, all jobs run as one service-managed user; separate queues by giving them separate fleets.	Run jobs as dedicated OS users
Farm and queue structure	The isolation boundaries between workloads . Farms don't share Deadline Cloud resources with each other, and queues within a farm can be separated by fleet, OS user, and bucket.	Isolate workloads with farms, fleets, and queues
Job attachment buckets	Which Amazon S3 data each queue can read and write through its queue role, bucket, and root prefix.	Secure job attachment and software buckets
Encryption keys	How farm data is encrypted at rest, with an AWS owned key or a customer managed AWS KMS key.	Key management
Network controls	How traffic reaches Deadline Cloud: private connectivity with AWS PrivateLink, and the endpoints to allowlist in restricted networks.	Access AWS Deadline Cloud using an interface endpoint (AWS PrivateLink), Restricted network environments

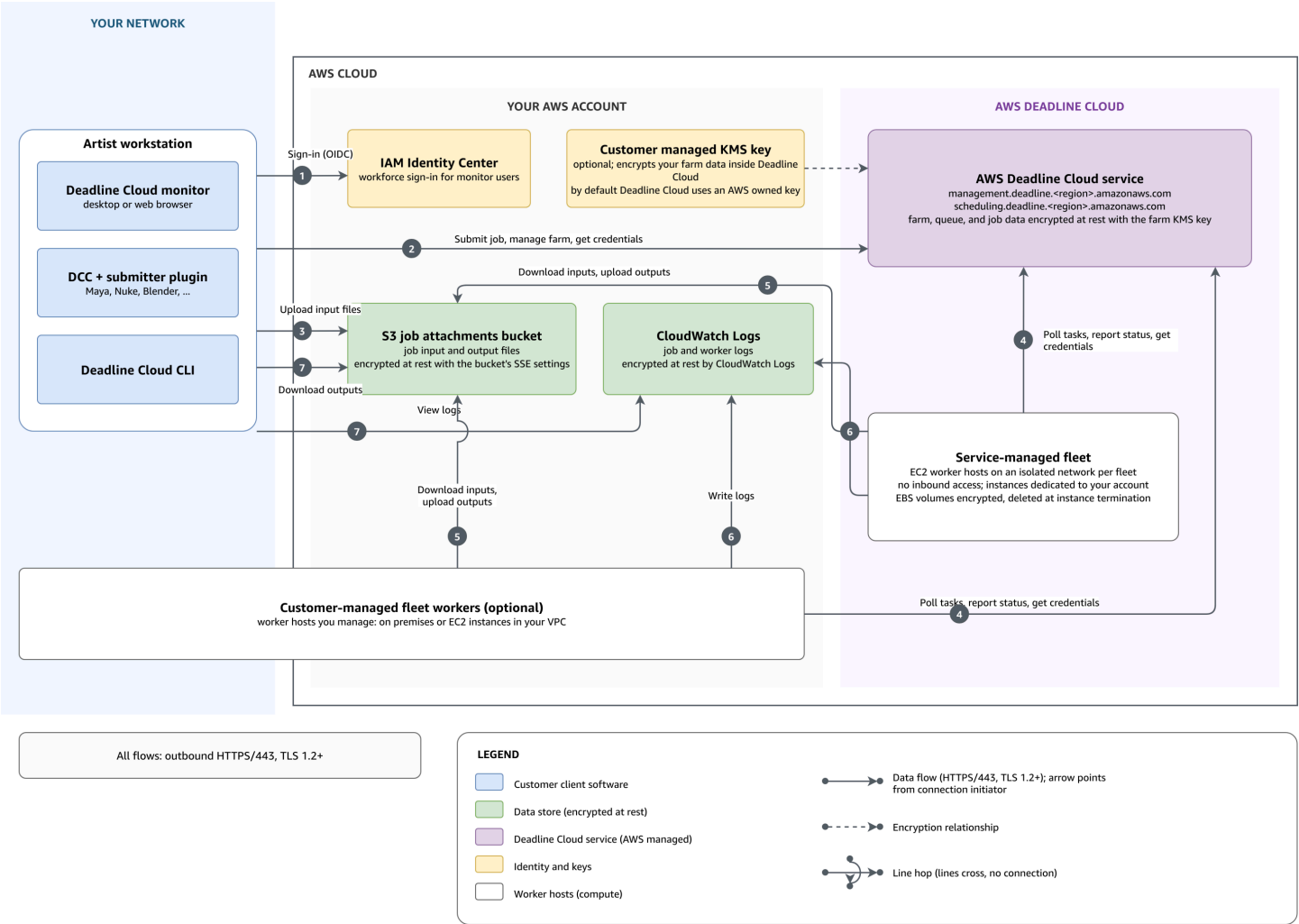
Where to start

Pick the entry point that matches your role:

- If you administer a studio and want to control which artists see which projects, start with [Managing users](#) and the [Isolate workloads with farms, fleets, and queues](#) guidance.
- If you build pipelines that submit jobs or read results, start with [Identity-based policy examples for Deadline Cloud](#) and [the section called “Job attachment buckets”](#).
- If you want to control what jobs can access while they run, start with [the section called “Job OS users”](#) for OS permissions and [Service roles](#) for the queue role that grants AWS permissions.
- If you run customer-managed fleets, start with [Service roles](#) and [Secure worker hosts](#).

Data flow, ports, and encryption in Deadline Cloud

Content security reviews, such as studio security audits and vendor onboarding questionnaires, ask for a data flow diagram. The diagram covers the components of the render farm, where each component runs, how data moves between them, and how connections and stored data are encrypted. The diagram and tables on this page answer those questions for a typical AWS Deadline Cloud deployment. For more information about the security controls that you can configure for each part of the farm, see [Security controls in Deadline Cloud](#).



Components and where they run

A Deadline Cloud render farm spans three locations: your network, your AWS account, and Deadline Cloud itself, which AWS manages. Customer-managed fleet workers are the exception: you choose where they run.

The following table describes each component and where it runs.

Component	Where it runs	Purpose
Deadline Cloud monitor, integrated submitters, and the Deadline Cloud CLI	Your network	Client tools on artist workstations that submit jobs, upload and download files, and track job progress.

Component	Where it runs	Purpose
AWS IAM Identity Center	Your AWS account	Sign-in and access levels for monitor users. For more information, see Managing users .
Amazon Simple Storage Service job attachments bucket	Your AWS account	Input and output files for jobs. Each queue has its own bucket and root prefix, so queue permission boundaries also apply to files.
Amazon CloudWatch Logs	Your AWS account	Job logs and worker logs.
AWS Key Management Service key	Your AWS account, or AWS managed	The key that encrypts your farm data inside Deadline Cloud. By default, Deadline Cloud uses an AWS owned key that you don't manage. You can supply a customer managed key from your account when you create the farm. For more information, see Key management .
Customer-managed fleet workers	Your choice: machines on your network, or Amazon Elastic Compute Cloud instances in your VPC	Worker hosts that you manage, running your software and the Deadline Cloud worker agent. A customer-managed fleet only needs outbound HTTPS access. If workers run in your VPC, optional AWS PrivateLink endpoints keep their traffic inside the VPC.
Deadline Cloud service	AWS managed	The service endpoints management.deadline. <i>region</i> .amazonaws.com and scheduling.deadline. <i>region</i> .amazonaws.com , and the farm, queue, and job scheduling data behind them.

Component	Where it runs	Purpose
Service-managed fleet workers	AWS managed	Amazon EC2 instances on an isolated network for each fleet. The instances are dedicated to your account and accept no inbound connections.

Data flows, ports, and protocols

Every client and worker in the diagram initiates its connections outbound to an AWS endpoint over HTTPS on TCP port 443. We require TLS 1.2 to encrypt connections in transit, and we recommend TLS 1.3. Deadline Cloud doesn't make inbound connections to your network. The flow numbers in the following table match the numbers in the diagram.

Flow	Connection	Description
1	Monitor to IAM Identity Center	Monitor users sign in through AWS IAM Identity Center with OpenID Connect (OIDC).
2	Workstation to the management endpoint	Submitters, the CLI, and the monitor submit jobs, manage farm resources, and request temporary role credentials through <code>management.deadline.region.amazonaws.com</code> .
3	Workstation to Amazon S3	Job attachments uploads input files to the queue's Amazon S3 bucket.
4	Workers to the scheduling endpoint	Workers in both fleet types poll for tasks, report progress and status, and request temporary role credentials through <code>scheduling.deadline.region.amazonaws.com</code> .
5	Workers to Amazon S3	Workers download input files from the queue's bucket and upload output files to it.
6	Workers to CloudWatch Logs	The worker agent writes job and worker logs.

Flow	Connection	Description
7	Workstation to CloudWatch Logs and Amazon S3	The monitor reads job logs, and the monitor and CLI download output files.

Neither the client tools nor the workers hold long-lived credentials for the queue's resources. Both request temporary credentials from Deadline Cloud and receive the permissions that you attach to the queue and fleet roles:

- The monitor, the CLI, and the submitters call `AssumeQueueRoleForUser` to upload and download job attachments, and `AssumeQueueRoleForRead` for read-only access such as reading job logs.
- A worker calls `AssumeFleetRoleForWorker` for the fleet role, then `AssumeQueueRoleForWorker` for the queue role of the job it runs.

For more information about the roles and what each one grants, see [Service roles](#).

If your studio filters outbound web traffic, see [Restricted network environments](#) for the full list of endpoints to allowlist. To keep traffic between your VPC and Deadline Cloud off the public internet, use AWS PrivateLink interface endpoints. For more information, see [Access AWS Deadline Cloud using an interface endpoint \(AWS PrivateLink\)](#).

Access to the monitor web application

Your monitor URL, `https://monitor-alias.region.deadlinecloud.amazonaws.com`, is reachable from any location on the internet, and you can't restrict the URL to a set of IP addresses. The URL serves the monitor web application, which is the same application for every Deadline Cloud monitor. The application holds none of your farm, queue, or job data, and anyone who opens the URL sees a sign-in page.

Your farm data reaches the monitor through the Deadline Cloud APIs, so the API calls are where access is granted or denied. After you sign in through AWS IAM Identity Center (IAM Identity Center), Deadline Cloud assumes the monitor role for you. The monitor then calls the APIs from the browser with the role's temporary credentials. IAM authorizes every call. The managed policies on the monitor role allow each action only at the access levels that you granted. For more information, see [Monitor role](#) and [Managing users](#).

To make farm data reachable only from your own network, apply the restriction to the API calls instead of the URL. You can allow only your IP ranges, or only requests that arrive through your VPC endpoints. For more information, see [Restrict farm access to your network](#).

Encryption summary

Each data store is encrypted at rest independently. The farm AWS KMS key applies only to the data that Deadline Cloud holds; your Amazon S3 bucket and your log groups use their own encryption settings.

- **In transit** – All connections use HTTPS with TLS 1.2 required and TLS 1.3 recommended. For more information, see [Encryption in transit](#).
- **Farm data** – The Deadline Cloud service encrypts your farm, queue, and job data at rest with the farm's AWS KMS key. You can use the default AWS owned key or supply a customer managed key when you create the farm. For more information, see [Key management](#).
- **Job attachments** – Amazon Simple Storage Service encrypts your input and output files at rest with the server-side encryption configured on the bucket. By default, Amazon S3 uses Amazon S3 managed keys (SSE-S3). To use your own AWS KMS key instead, configure the bucket for SSE-KMS and grant the queue role access to the key. For more information, see [Job attachments in Deadline Cloud](#).
- **Worker volumes** – In service-managed fleets, Deadline Cloud encrypts the Amazon Elastic Block Store volumes attached to worker instances and deletes the volumes when instances terminate. For more information, see [Encryption at rest](#).
- **Logs** – Amazon CloudWatch Logs encrypts job logs and worker logs at rest with its own server-side encryption. To use your own AWS KMS key instead, associate the key with the log group. For more information, see [Encrypt log data in CloudWatch Logs using AWS KMS](#) in the *Amazon CloudWatch Logs User Guide*.

Security best practices for Deadline Cloud

Each of the following topics covers securing one part of your AWS Deadline Cloud (Deadline Cloud) farm. For a map of all the security controls in Deadline Cloud and where to configure them, see [Security controls in Deadline Cloud](#).

Topics

- [Isolate workloads with farms, fleets, and queues](#)

- [Run jobs as dedicated OS users](#)
- [Secure job attachment and software buckets](#)
- [Secure worker hosts](#)
- [Secure artist workstations](#)
- [Restrict farm access to your network](#)

Isolate workloads with farms, fleets, and queues

You can arrange Deadline Cloud farms, fleets, and queues many ways. The arrangement you choose sets the isolation boundaries between your workloads, so decide how much separation your teams, shows, or clients need before you build.

Two facts about fleets drive the decision:

- When a fleet is associated with more than one queue, jobs from all of those queues run on the same worker hosts. After a job runs, data can remain on the host, such as files in a temporary directory or the queue user's home directory, and a job can leave processes running that later jobs can observe.

Some of that data remains by design. A job can cache data under the job user's home directory so that later sessions reuse it. For example, the conda queue environment caches downloaded packages there. On a service-managed fleet, all jobs run as one user, so every queue on the fleet shares those caches. Cached data remains until the worker shuts down, or across worker replacement when the fleet uses persistent storage. For more information, see [Sessions](#) in the *Deadline Cloud Developer Guide* and [Persistent storage](#) in the *Deadline Cloud User Guide*.

- Worker hosts run jobs as an operating system user. On a customer-managed fleet, you can give each queue its own user, so queues can share a fleet while their jobs' files and processes stay separated. On a service-managed fleet, all jobs run as a single user, so give queues that need separation their own fleets. In both cases, the hosts run only jobs from queues in your farm.

With those facts in mind, choose the arrangement that fits your isolation needs:

- **Separate farms** – A farm doesn't share Deadline Cloud resources such as fleets, queues, and storage profiles with other farms, so separate farms give workloads the strongest separation. The tradeoff is more resources to set up and manage, and worker capacity that one farm can't

lend to another. Sharing external AWS resources, such as an Amazon S3 bucket, between farms weakens the boundary, so give each farm its own.

- **One farm, a fleet for each queue** – Queues that need separation from each other each get their own fleet, so their jobs never share worker hosts. You keep a single farm to manage, and monitor users can be scoped to each queue. For more information about scoping users, see [Managing users](#) in the *Deadline Cloud User Guide*. The tradeoff is partitioned capacity: idle workers in one fleet can't pick up another queue's jobs.
- **One farm, queues sharing fleets** – Sharing fleets gives you the simplest setup and the best worker utilization. Because the queues share worker hosts, treat all queues that share a fleet as one security boundary. On a customer-managed fleet you can also give each queue a distinct operating system user to separate job processes and files from each other.

Whichever arrangement you choose, keep resource sharing within a security boundary:

- Share an Amazon S3 bucket and root prefix for job attachments only between queues in the same security boundary. For more information, see [the section called "Job attachment buckets"](#).
- Share an operating system user only between queues in the same security boundary. For more information, see [Run jobs as dedicated OS users](#).
- Apply the same boundary to any other AWS resources that you integrate into the farm, such as shared file systems.

Run jobs as dedicated OS users

Each queue can specify an operating system (OS) user and its primary group, called the job run as user. The worker agent runs every job process submitted to the queue as that OS user, so the jobs inherit that user's OS permissions, including access to files and other resources on the worker host.

Give the queue user the least-privilege OS permissions that the queue's workloads need. A dedicated user for each queue keeps a job's files and processes separated from jobs in other queues, from other installed software, and from other users with access to the worker host.

The two fleet types handle the queue user differently:

- On a customer-managed fleet, you create the OS users on your worker hosts. Use a separate user for each queue unless the queues are in the same security boundary.
- On a service-managed fleet, the service manages the worker hosts, and all jobs run as a single OS user. The hosts are dedicated to your account and run only jobs from the queues that you

associate with the fleet, so you separate queues by giving them separate fleets instead of separate users.

When a queue doesn't specify a user, its jobs run as the worker agent user. The worker agent user can impersonate (`sudo`) any queue user, so a queue without a user can escalate privileges to any other queue's user and data.

On a customer-managed fleet, the worker agent enforces the boundary between queue users with operating system permissions. The agent creates each session's working directory so that only the agent user and the job user's primary group can access it. The agent also stores each queue's role credentials in a queue-specific directory with the same access restriction. When each queue has its own user and primary group, a job can't read another queue's session files or AWS credentials.

Standard operating system user permissions on a shared kernel separate queue users on a shared worker host. Give workloads that require stronger separation, such as virtual machine isolation, their own fleets.

Secure job attachment and software buckets

Each queue stores its job attachments in an Amazon S3 bucket that you own. Three pieces configure the connection between a queue and its bucket:

- Job attachments write to and read from a root prefix in the Amazon S3 bucket. You specify this root prefix in the `CreateQueue` API call.
- The bucket has a corresponding `Queue Role`, which specifies the role that grants queue users access to the bucket and root prefix. When creating a queue, you specify the `Queue Role` Amazon Resource Name (ARN) alongside the job attachments bucket and root prefix.
- Authorized calls to the `AssumeQueueRoleForRead`, `AssumeQueueRoleForUser`, and `AssumeQueueRoleForWorker` API operations return a set of temporary security credentials for the `Queue Role`.

Queues that share an Amazon S3 bucket and root prefix also share access to the job attachments stored there. Give each queue its own bucket and root prefix so that queue permission boundaries also apply to job attachments. For example, when `QueueA` and `QueueB` have separate buckets, an artist with access to only `QueueA` can't read `QueueB`'s job attachments. If several queues belong to the same security boundary, such as the queues for one show, sharing a bucket between them is a

reasonable simplification. The console sets up each queue with its own bucket and root prefix by default.

Separate buckets keep job attachments separated in Amazon S3. On the worker host, a job runs code as the queue's operating system user. The job can write files to any location that user can write to, including locations that other queues' users can read. Separation on the host comes from the queue operating system users. For more information, see [Run jobs as dedicated OS users](#).

To isolate your queues, you must configure the Queue Role to only allow queue access to the bucket and root prefix. In the following example, replace each *placeholder* with your resource-specific information.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::JOB_ATTACHMENTS_BUCKET_NAME",
        "arn:aws:s3:::JOB_ATTACHMENTS_BUCKET_NAME/JOB_ATTACHMENTS_ROOT_PREFIX/*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:ResourceAccount": "111122223333"
        }
      }
    },
    {
      "Action": [
        "logs:GetLogEvents"
      ],
      "Effect": "Allow",
```

```

        "Resource": "arn:aws:logs:us-east-1:111122223333:log-group:/aws/
deadline/FARM_ID/*"
    }
]
}

```

You must also set a trust policy on the role. In the following example, replace the *placeholder* text with your resource-specific information.

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sts:AssumeRole"
      ],
      "Effect": "Allow",
      "Principal": {
        "Service": "deadline.amazonaws.com"
      },
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:deadline:us-east-1:111122223333:farm/FARM_ID"
        }
      }
    },
    {
      "Action": [
        "sts:AssumeRole"
      ],
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.deadline.amazonaws.com"
      },
      "Condition": {

```

```

        "StringEquals": {
            "aws:SourceAccount": "111122223333"
        },
        "ArnEquals": {
            "aws:SourceArn": "arn:aws:deadline:us-east-1:111122223333:farm/FARM_ID"
        }
    }
}

```

Custom software Amazon S3 buckets

You can add the following statement to your Queue Role to access custom software in your Amazon S3 bucket. In the following example, replace **SOFTWARE_BUCKET_NAME** with the name of your S3 bucket and **BUCKET_ACCOUNT_OWNER** with the AWS account ID that owns the bucket.

```

"Statement": [
    {
        "Action": [
            "s3:GetObject",
            "s3:ListBucket"
        ],
        "Effect": "Allow",
        "Resource": [
            "arn:aws:s3::SOFTWARE_BUCKET_NAME",
            "arn:aws:s3::SOFTWARE_BUCKET_NAME/*"
        ],
        "Condition": {
            "StringEquals": {
                "aws:ResourceAccount": "BUCKET_ACCOUNT_OWNER"
            }
        }
    }
]

```

For more information about Amazon S3 security best practices, see [Security best practices for Amazon S3](#) in the *Amazon Simple Storage Service User Guide*.

Secure worker hosts

On customer-managed fleets, you configure the worker host operating system. The following practices keep the worker agent, queue users, and their data separated from each other on the host:

- Review host configuration scripts before you use them, and test configuration changes before rolling them out to your fleet. An incorrect configuration can make workers unstable or stop them from working.
- Don't use the same `jobRunAsUser` value with multiple queues unless jobs submitted to those queues are within the same security boundary.
- Don't set the queue `jobRunAsUser` to the name of the OS user that the worker agent runs as. For the reason, see [Run jobs as dedicated OS users](#).
- Grant queue users the least-privileged OS permissions required for the intended queue workloads. Ensure that they don't have filesystem write permissions to worker agent program files or other shared software.
- Ensure that only the root user on Linux, or the Administrator account on Windows, owns and can modify the worker agent program files.
- On Linux worker hosts, consider configuring a `umask` override in `/etc/sudoers` that allows the worker agent user to launch processes as queue users. This configuration helps ensure other users can't access files written to the queue.
- Restrict permissions to local DNS override configuration files (`/etc/hosts` on Linux and `C:\Windows\system32\etc\hosts` on Windows), and to route tables on worker host operating systems, so that worker traffic to Deadline Cloud can't be redirected.
- Regularly patch the operating system and all installed software. This approach includes software specifically used with Deadline Cloud such as submitters, adaptors, worker agents, OpenJD packages, and others.
- Use strong passwords for the Windows queue `jobRunAsUser`, and rotate the passwords regularly.
- Grant least-privileged access to the Windows password secrets, and delete unused secrets.
- Don't give the queue `jobRunAsUser` permission to schedule commands to run in the future:
 - On Linux, deny these accounts access to `cron` and `at`.
 - On Windows, deny these accounts access to the Windows task scheduler.

Secure artist workstations

Workstations with Deadline Cloud submitters installed hold credentials that can submit jobs to your farm, and jobs submitted from them run on your fleets at your expense. The following practices protect those credentials and the submission path:

- Prefer temporary credentials for workstation access to Deadline Cloud. Users who sign in through the Deadline Cloud monitor receive temporary credentials automatically. If a workstation uses an AWS profile with long-lived IAM access keys, such as for the Deadline Cloud CLI, secure those keys. For more information, see [Managing access keys for IAM users](#) in the *IAM User Guide*.
- When you install the Deadline Cloud submitters and adaptors, download them through the Deadline Cloud monitor and verify the installers before running them. For more information, see [Verify the authenticity of downloaded software](#). If you build your own submitters, see [Build a custom submitter for Deadline Cloud](#) in the *Deadline Cloud Developer Guide*.
- Use secure permissions on Deadline Cloud submitter program files to prevent tampering.
- Restrict permissions to local DNS override configuration files (/etc/hosts and /etc/resolv.conf on Linux and macOS, and C:\Windows\system32\etc\hosts on Windows), and to route tables, so that workstation traffic to Deadline Cloud can't be redirected.
- Regularly patch the operating system and the software used with Deadline Cloud, including submitters, adaptors, and OpenJD packages.

Restrict farm access to your network

Every request to your farm goes through the Deadline Cloud APIs. The monitor, the AWS CLI, the submitters, and the AWS Management Console all call the APIs with temporary credentials, so the network boundary belongs on the API calls. The monitor URL is reachable from any location on the internet and holds no farm data. For more information, see [Access to the monitor web application](#).

Add a network condition to the IAM policies of the credentials that your people use:

- The monitor gets its credentials from the monitor role. So do the AWS CLI and the submitters when they use the profile that the Deadline Cloud monitor creates. Add a custom policy to the monitor role. For more information, see [Adding permissions for advanced workflows](#).
- The AWS Management Console and pipeline scripts use their own IAM roles or users. Add the condition to those policies, or to the permission sets that you assign for AWS Management Console access in IAM Identity Center.

Allow only your IP ranges

An `aws:SourceIp` condition limits requests to the public addresses that you list. List the addresses that your studio sends traffic from. Include your office ranges, your VPN exit addresses, and the NAT gateway addresses of any VPC where your workstations run.

The following policy denies Deadline Cloud requests from any other address. Attach it to the monitor role, alongside the managed policies that the role already has.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "deadline:*",
      "Resource": "*",
      "Condition": {
        "NotIpAddress": {
          "aws:SourceIp": [
            "203.0.113.0/24",
            "198.51.100.10/32"
          ]
        },
        "BoolIfExists": {
          "aws:ViaAWSService": "false"
        }
      }
    }
  ]
}
```

The policy covers your files as well as your farm. Job attachments and job logs need queue role credentials. The monitor, the AWS CLI, and the submitters get those credentials by calling `AssumeQueueRoleForUser`. The deny blocks the call from any address that you didn't list.

If you sign in from an address that you didn't list, you still reach the sign-in page. The monitor then reports errors instead of showing farm data. An allow can't override a deny, so verify your ranges before you attach the policy.

⚠ Important

The preceding policy also denies requests that arrive through an Deadline Cloud interface endpoint. Endpoint traffic carries `aws:VpcSourceIp` instead of `aws:SourceIp`, and a `NotIpAddress` condition on an absent key matches, so the deny applies. If your people reach Deadline Cloud through an interface endpoint, use the VPC form in the next section. To allow both paths, put both keys in the same condition block. The deny then applies only when neither key matches. Test the policy from both networks before you roll it out.

For more information about the keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

Allow access only from your VPC

To require that people work from hosts in a VPC, such as virtual workstations, combine three pieces:

1. Run the monitor, the AWS CLI, and the submitters on hosts in the VPC.
2. Create an interface endpoint in the VPC for the Deadline Cloud management endpoint. Add Amazon S3 and CloudWatch Logs endpoints if your people upload job attachments and read job logs. Enable private DNS so that the client tools use the endpoints without extra configuration. For more information, see [Access AWS Deadline Cloud using an interface endpoint \(AWS PrivateLink\)](#).
3. Attach a policy to the monitor role that denies requests from outside the VPC. The VPC keys hold strings, so the condition operator is `StringNotEquals` rather than `NotIpAddress`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "deadline:*",
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "aws:SourceVpc": "vpc-0123456789abcdef0"
        },
        "BoolIfExists": {
          "aws:ViaAWSService": "false"
        }
      }
    }
  ]
}
```



```
}  
  }  
    }  
  ]  
}
```

To name individual endpoints instead of the whole VPC, use `aws:SourceVpce` with your endpoint IDs. To name address ranges inside the VPC, use `aws:VpcSourceIp` with `NotIpAddress`.

The monitor web application, the IAM Identity Center portal, and AWS Sign-In have no interface endpoints. The browser on the host therefore needs outbound access to the sign-in domains listed in [Restricted network environments](#). You can route that traffic through a NAT gateway or a web proxy that allows only those domains. Farm data doesn't travel over the sign-in path.

Keep worker access working

Your workers share the queue role with your people. A worker uses queue role credentials to read inputs and write outputs while it runs a job. The monitor and the AWS CLI use the same role when someone uploads inputs or downloads outputs. Restrict the monitor role, and leave the queue role and the fleet role without a network condition.

Service-managed fleet workers run on the AWS network, so their addresses aren't in your ranges and their requests don't come through your endpoints. A network condition on the queue role, the fleet role, or the bucket policy of the job attachments bucket denies those requests. Jobs then fail. A bucket policy condition works only if all of your fleets are customer-managed and run in your VPC. In that case, list both your workstation endpoint and your fleet endpoint in an `aws:SourceVpce` condition.

Queue role credentials carry no network condition of their own. Someone who obtains them from an address that you allow can use them from anywhere until they expire. The condition on the monitor role controls who gets credentials, rather than every request that uses them.

Verify the authenticity of downloaded software

Verify your software's authenticity after downloading the installer to protect against file tampering. This procedure works for both Windows and Linux systems.

Windows

To verify the authenticity of your downloaded files, complete the following steps.

1. In the following command, replace *file* with the file that you want to verify. For example, **C:\PATH\TO\MY\DeadlineCloudSubmitter-windows-x64-installer.exe** . Also, replace *signtool-sdk-version* with the version of the SignTool SDK installed. For example, **10.0.22000.0**.

```
"C:\Program Files (x86)\Windows Kits\10\bin\signtool-sdk-  
version\x86\signtool.exe" verify /v file
```

2. For example, you can verify the Deadline Cloud submitter installer file by running the following command:

```
"C:\Program Files (x86)\Windows Kits\10\bin  
\10.0.22000.0\x86\signtool.exe" verify /v DeadlineCloudSubmitter-  
windows-x64-installer.exe
```

3. Confirm that the output contains Successfully verified:
DeadlineCloudSubmitter-windows-x64-installer.exe. The successful verification means that you can run the installer.

Linux

To verify the authenticity of your downloaded files, use the gpg command line tool.

1. Import the OpenPGP key by running the following command:

```
gpg --import --armor <<EOF  
-----BEGIN PGP PUBLIC KEY BLOCK-----  
  
mQINBGLANDUBEACg6zffjN43gqe5ryPhk+wQM10rEdvmItw4WPWaVsN+/at/OIJw  
MGCagSYXcgR+jKbsHQ0QoEQdo5SrxHjPKTEs3KQhGvf+ehrU1Ac7koXKIBWtes+  
BI9F0s1RECz0nXT0y/cd/90RXjpF07mreTLIKNIbybULfad82nYykpITjFr5XRGj  
/shYkucxRQZdwkgkIYyV25pPICPd2RsX+Zua85jV8mCqVffDfRXvgcPe3+ofClj/  
2CE8UfUIq08Csua4YEKsqr3aeoT0EFT4kuQR5nFXVzor0EkQt03gB35KNWKM1IOU  
2vA+wyOL7nWSii4yfYtW3EZ+3gq6HxvnT9Zs8MC53uT0i0damASXecYREwGmY/io  
6n5XTEA/35LNbl4A756vSTZ7h4VFJAN5BpuqxstI1D7ou94skoSmcPoC/iniTvY9  
kZy1U50CH/nifMAHM2a5jrQel80cw4oko9eyc8ENQpSy15JE1F0Kff7D/4tcZJLF  
F0VBTXbhfVq3dPfoq94Iwt7p540vwj0S//CEu3jZYbN12QC/3YiHE2H2XyGCQbq6  
2MjcuxLnEapoRIqfbi8GPtCWVPzm28WGyKIDofWICczzeJFFJnvzrY3wRG64ibKJ  
bR/uedwua1UuiC482V1FD5ffmzSSs8ktTp9hgj7RGDX1c9NTcF1jHxG9hwARAQAB  
tCxBV1MgRGVhZGxpbnUgQ2xvdWQgPGF3cy1kZWFKbGluZUBhbWF6b24uY29tPokC  
VwQTAQgAQRYhBJmXd7So2csyehiIYsg71N18bhtjBQJpQDQ1AhsvBQkDwmcABQsJ  
CACAIICBhUKCQgLAQgQWAgMBAh4HAheAAAoJEMg71N18bhtjk2UP/3h4K1EzZ0/7
```

```
BxRmkbixuo1Quq0GvA6tXbSWaM8QH5jglcvL12PZLALk1LT4v82uCsLR11F8/Tch
cC10SZE0FIS+XxAaw1Xfai6jlyLhab0wKF2ylq5eJlLcw1lh2nAArDRb4fLD0m1g
Dfquetq/XEpyXp0SkWxGRV4R1UdjQfytxrmcUnsT5/fk5f9VDdblu6K/1EmwfyYjB
1Xv0uUcKqPot0Smbv0h3PY3Hi3n54ncy8NfTeV+TUvSe3C1s1zNl8aqHoTxJB/eU
kp+LFZ9m+igpSYnKeg1KnytylH3KGCjTHglT/QXnI1wNTqmj1kFBVwtt/y1mtnA+
CPIUHP1CtbKsHaLtp41lBm5TVtPN/Wqqicn5QL14khg7R4K+V2aaA4ubY6p1tG9
0fFhN5tTnHDSKWMfmb83wfh5Zkcg85c3egjoit+wgGQRAQVqbznx7NqAHs9VoDIu
SPcAr+C329A0Bzod4gyNGH7Ah5DkMITo404+axnAU9yhF0HcMJmTIask/fNg1Aum
OqYPMUwcgv1GZjLaTJyfGGC1xALsYR0KHnwIehD06MHR/Z98bGkcV8+Y0q8UPsd1
VN1fc1rjCJh/AT3w6owvG4DaEwspseSjzHv16mW4e2N6Uu23SPzgQsJ5qYN2g8D+
P7N9LGDfP8DaYc5JM9mlyFmYI2Q94ufl
=rY5l
-----END PGP PUBLIC KEY BLOCK-----
EOF
```

2. Determine whether to trust the OpenPGP key. When deciding, consider that AWS has taken measures to secure the hosting of the OpenPGP public key on this page, and that you obtained the key from the official Deadline Cloud documentation.
3. If you decide to trust the OpenPGP key, edit the key to trust with gpg similar to the following example:

```
$ gpg --edit-key 0xC83BD4DD7C6E1B63
```

```
gpg (GnuPG) 2.3.7; Copyright (C) 2021 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

```
pub rsa4096/C83BD4DD7C6E1B63 created: 2025-12-15 expires: 2027-12-15
usage: SCEA
```

```
trust: unknown validity: unknown
[ unknown] (1). AWS Deadline Cloud <aws-deadline@amazon.com>
```

```
gpg> trust
pub rsa4096/C83BD4DD7C6E1B63 created: 2025-12-15 expires: 2027-12-15
usage: SCEA
```

```
trust: unknown validity: unknown
[ unknown] (1). AWS Deadline Cloud <aws-deadline@amazon.com>
```

```
Please decide how far you trust this user to correctly verify other users'
keys
(by looking at passports, checking fingerprints from different sources,
etc.)
```

```
1 = I don't know or won't say
2 = I do NOT trust
3 = I trust marginally
4 = I trust fully
5 = I trust ultimately
m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y

pub rsa4096/C83BD4DD7C6E1B63 created: 2025-12-15 expires: 2027-12-15
usage: SCEA
               trust: ultimate      validity: unknown
[ unknown] (1). AWS Deadline Cloud <aws-deadline@amazon.com>
Please note that the shown key validity is not necessarily correct
unless you restart the program.

gpg> quit
```

4. Verify the Deadline Cloud submitter installer

To verify the Deadline Cloud submitter installer, complete the following steps:

- a. Download the signature file for the Deadline Cloud submitter installer.

[Download signature file \(.sig\)](#)

- b. Verify the signature of the Deadline Cloud submitter installer by running:

```
gpg --verify ./DeadlineCloudSubmitter-linux-x64-installer.run.sig ./
DeadlineCloudSubmitter-linux-x64-installer.run
```

5. Verify the Deadline Cloud monitor

Note

You can verify the Deadline Cloud monitor download using signature files or platform specific methods. For platform specific methods, see the Linux (Debian) tab, the Linux (RPM) tab, or the Linux (AppImage) tab based on your downloaded file type.

To verify the Deadline Cloud monitor desktop application with signature files, complete the following steps:

a. Download the corresponding signature file for your Deadline Cloud monitor installer:

- [Download .deb signature file](#)
- [Download .rpm signature file](#)
- [Download .AppImage signature file](#)

b. Verify the signature:

For .deb:

```
gpg --verify ./deadline-cloud-monitor_amd64.deb.sig ./deadline-cloud-monitor_amd64.deb
```

For .rpm:

```
gpg --verify ./deadline-cloud-monitor.x86_64.rpm.sig ./deadline-cloud-monitor.x86_64.rpm
```

For .AppImage:

```
gpg --verify ./deadline-cloud-monitor_amd64.AppImage.sig ./deadline-cloud-monitor_amd64.AppImage
```

c. Confirm that the output looks similar to the following:

```
gpg: Signature made Mon Jan 5 21:10:14 2026 UTC
```

```
gpg: using RSA key C83BD4DD7C6E1B63
```

If the output contains the phrase `Good signature from "AWS Deadline Cloud"`, it means that the signature has successfully been verified and you can run the Deadline Cloud monitor installation script.

Historical keys

If you downloaded an installer that was signed before the current key was issued, import the following historical key and repeat the verification steps.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGX6GQsBEADduUtJgqSXI+q7606fsFwEYKmbnlyL0xKv1q32EZuyv0otZo5L
le4m5Gg52AZrvPvDiUTLooAlvYeozaYyirIGsK08Ydz0Ftdjroiuh/mw9JSJDJRI
rnRn5yKet1JFezkjopA3pjsTBP6lW/mb1bDBDEwwwtH0x9lV7A03FJ9T7Uzu/qSh
q0/Uydkafro3cPASvkqgDt2tCvURfBcUCAjZVFcLZcVD5iwXacxvKsxxS/e7kuVV
I1+VGT8Hj8XzWYhjCZx0LZk/fvpYPMYEEujN0fYUp6RtMIXve0C9awwMCy5nBG2J
eE2015DsCpTaBd4Fdr3LWcSs8JFA/YfP9auL3Ncz0ozPoVJt+fw8CB1VIX00J715
hvHDjcC+5v0wxqAlMG6+f/SX7CT8FXK+L3i0J5gBYUNXqHSxUdv8kt76/KVmQa1B
Ak1+MPKpMq+1hw++S3G/1XqwWadNQBRRw7dSZHymQVXvPp1nscq3hV7K10M+6s6g
1g4mvFY4lF6DhptwZLWYXU8rBQpojvQfiSmDFrFPWFi5BexesVnkGIolQoklKx
AVUSdJPVEJCteyy7td4FPhBaSqT5vW3+ANbr9b/uoRYWJvn17dN0cc9HuRh/Ai+I
nkfECo2WUDLZ0fEKGjGyFX+todWvJXjvc5kmE9Ty5vJp+M9Vvb8jd6t+mwARAQAB
tCxBV1MgRGVhZGxpbnUgQ2xvdWQgPGF3cy1kZWFKbGluZUBhbnWF6b24uY29tPokC
VwQTAQgAQRyhbLhAwIwpqQeWoHH6pfbNP0a3bzzvBQJl+hkLAXsvBAUJA8JnAAUL
CQgHAgIiAgYVCgkICwIDFgIBAh4HAheAAAoJEPbNP0a3bzzvKswQAjXzKSAY8sY8
F6Eas2oYwIDDDdurs8FiEnFghjUE06MTt9AykF/jw+CQg2UzFtEy0bHBymhgmhXE
3buVeom96tgM3ZDfZu+sxi5pGX6oAQnZ6riztN+VpkpQmLgwtMGpSML13KLwnv2k
WK8mrR/fPMkfdaewB7A6RIUYiW33GAL4KfMIs8/vIwIJw99NxHpZQVoU6dFpuDtE
10uxGcCqGJ7mAmo6H/YawSNp2Ns80gyqIKYo7o3LJ+WRroIRlQyctq8gnR9JvYXX
42ASqLq5+0XKo4qh81blXKYqtc176BbbSNFjWnzIQgKDgNiHFZCdc0VgqDhw015r
NICbqqwwNLj/Fr2kecYx180Ktp10j00w5I0yh3bf3MVGWnYRdjvA1v+/CO+55N4g
z0kf50Lcdu5RtqV10XBCifn28pecqPaSdYcssYSR15DLiFktGbNzTGcZZwITTKQc
af8PPdTGtnnb6P+cdbW3bt9MVtN5/dgSHLThnS8MPEuNCtkTnpXshuVuBGgwBMdb
qUC+HjqvhZzbwns8dr5WI+6HWNBFgGANN6ageY158vVp0UkuNP8wcWjRARciHXZx
ku6W2jPTHWDGnBQ02Fx7fd2QYJheIPPASHcfJ0+XgWCoF45D0vAxAJ8gGg9Eq+
gFWhsx4NSHn2gh1gDZ410u/4exJ1lwPM
=uVaX
```

```
-----END PGP PUBLIC KEY BLOCK-----
```

Linux (Applmage)

To verify packages that use a Linux .Applmage binary, first complete steps 1-3 in the Linux tab, then complete the following steps.

1. From the ApplmageUpdate [releases page](#) on the GitHub website, download the **validate-x86_64.Applmage** file.
2. After downloading the file, to add execute permissions, run the following command.

```
chmod a+x ./validate-x86_64.AppImage
```

3. To add execute permissions, run the following command.

```
chmod a+x ./deadline-cloud-monitor_<APP_VERSION>_amd64.AppImage
```

4. To verify the Deadline Cloud monitor signature, run the following command.

```
./validate-x86_64.AppImage ./deadline-cloud-monitor_<APP_VERSION>_amd64.AppImage
```

If the output contains the phrase `Validation successful`, it means that the signature has successfully been verified and you can safely run the Deadline Cloud monitor installation script.

Linux (Debian)

To verify packages that use a Linux `.deb` binary, first complete steps 1-3 in the Linux tab.

dpkg is the core package management tool in most debian based Linux distributions. You can verify the `.deb` file with the tool.

1. Download the Deadline Cloud monitor `.deb` file:

[Download Deadline Cloud monitor \(.deb\)](#)

2. Verify the `.deb` file:

```
dpkg-sig --verify deadline-cloud-monitor_amd64.deb
```

3. The output will be similar to:

```
Processing deadline-cloud-monitor_amd64.deb...  
GOODSIG _gpgbuilder C83BD4DD7C6E1B63 1765815349
```

4. To verify the `.deb` file, confirm that `GOODSIG` is present in the output.

Linux (RPM)

To verify packages that use a Linux `.rpm` binary, first complete steps 1-3 in the Linux tab.

1. Download the Deadline Cloud monitor .rpm file:

[Download Deadline Cloud monitor \(.rpm\)](#)

2. Verify the .rpm file:

```
gpg --export --armor "Deadline Cloud" > key.pub
sudo rpm --import key.pub
rpm -K deadline-cloud-monitor.x86_64.rpm
```

3. The output will be similar to:

```
deadline-cloud-monitor.x86_64.rpm: digests signatures OK
```

4. To verify the .rpm file, confirm that `digests signatures OK` is in the output.

Identity and Access Management in Deadline Cloud

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 Deadline Cloud resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience](#)
- [Authenticating with identities](#)
- [Managing access using policies](#)
- [How Deadline Cloud works with IAM](#)
- [Identity-based policy examples for Deadline Cloud](#)
- [AWS managed policies for Deadline Cloud](#)
- [Service roles](#)
- [Troubleshooting AWS Deadline Cloud identity and access](#)

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 AWS Deadline Cloud identity and access](#))
- **Service administrator** - determine user access and submit permission requests (see [How Deadline Cloud works with IAM](#))
- **IAM administrator** - write policies to manage access (see [Identity-based policy examples for Deadline Cloud](#))

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 Deadline Cloud works with IAM

Before you use IAM to manage access to Deadline Cloud, learn what IAM features are available to use with Deadline Cloud.

IAM features you can use with AWS Deadline Cloud

IAM feature	Deadline Cloud support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Forward access sessions (FAS)	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how Deadline Cloud 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 Deadline Cloud

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 Deadline Cloud

To view examples of Deadline Cloud identity-based policies, see [Identity-based policy examples for Deadline Cloud](#).

Resource-based policies within Deadline Cloud

Supports resource-based policies: No

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 accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. For more information, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

Policy actions for Deadline Cloud

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 Deadline Cloud actions, see [Actions defined by AWS Deadline Cloud](#) in the *Service Authorization Reference*.

Policy actions in Deadline Cloud use the following prefix before the action:

```
deadline
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [  
    "deadline:action1",  
    "deadline:action2"  
]
```

To view examples of Deadline Cloud identity-based policies, see [Identity-based policy examples for Deadline Cloud](#).

Policy resources for Deadline Cloud

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 Deadline Cloud resource types and their ARNs, see [Resources defined by AWS Deadline Cloud](#) in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Actions defined by AWS Deadline Cloud](#).

To view examples of Deadline Cloud identity-based policies, see [Identity-based policy examples for Deadline Cloud](#).

Policy condition keys for Deadline Cloud

Supports service-specific policy condition keys: 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 `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 Deadline Cloud condition keys, see [Condition keys for AWS Deadline Cloud](#) in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Actions defined by AWS Deadline Cloud](#).

To view examples of Deadline Cloud identity-based policies, see [Identity-based policy examples for Deadline Cloud](#).

ACLs in Deadline Cloud

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.

ABAC with Deadline Cloud

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 Deadline Cloud

Supports temporary credentials: Yes

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*.

Forward access sessions for Deadline Cloud

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 Deadline Cloud

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 Deadline Cloud functionality. Edit service roles only when Deadline Cloud provides guidance to do so.

Service-linked roles for Deadline Cloud

Supports service-linked roles: No

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 Deadline Cloud

By default, users and roles don't have permission to create or modify Deadline Cloud 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 Deadline Cloud, including the format of the ARNs for each of the resource types, see [Actions, resources, and condition keys for AWS Deadline Cloud](#) in the *Service Authorization Reference*.

Topics

- [Policy best practices](#)
- [Using the Deadline Cloud console](#)
- [Policy to access the console](#)
- [Policy to submit jobs to a queue](#)
- [Policy to allow creating a license endpoint](#)
- [Policy to allow monitoring a specific farm queue](#)
- [Policy to manage queue–fleet associations for a specific fleet](#)

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Deadline Cloud 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 Deadline Cloud console

To access the AWS Deadline Cloud console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Deadline Cloud 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 grant full access to the Deadline Cloud console, attach the policy in [the section called “Policy to access the console”](#). To grant users access to farm, fleet, queue, and job data based on their farm memberships and access levels, attach the AWS managed policies described in [AWS managed policies for Deadline Cloud](#). For more information about attaching policies, see [Adding permissions to a user](#) in the *IAM User Guide*.

Policy to access the console

To grant access to all functionality in the Deadline Cloud console, attach this identity policy to a user or role you want to have full access.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "EC2InstanceTypeSelection",
    "Effect": "Allow",
    "Action": [
      "ec2:DescribeInstanceTypeOfferings",
      "ec2:DescribeInstanceTypes",
      "ec2:GetInstanceTypesFromInstanceRequirements",
      "pricing:GetProducts"
    ],
    "Resource": ["*"]
  },
  {
    "Sid": "VPCResourceSelection",
    "Effect": "Allow",
    "Action": [
      "ec2:DescribeVpcs",
      "ec2:DescribeSubnets",
      "ec2:DescribeSecurityGroups"
    ],
    "Resource": ["*"]
  },
  {
    "Sid": "ViewVpcLatticeResources",
    "Effect": "Allow",
    "Action": [
      "vpc-lattice:ListResourceConfigurations",
      "vpc-lattice:GetResourceConfiguration",
```

```

        "vpc-lattice:GetResourceGateway"
    ],
    "Resource": ["*"]
},
{
    "Sid": "ManageVpcEndpointsViaDeadline",
    "Effect": "Allow",
    "Action": [
        "ec2:CreateVpcEndpoint",
        "ec2:DescribeVpcEndpoints",
        "ec2>DeleteVpcEndpoints",
        "ec2:CreateTags"
    ],
    "Resource": ["*"],
    "Condition": {
        "StringEquals": { "aws:CalledViaFirst": "deadline.amazonaws.com" }
    }
},
{
    "Sid": "ChooseJobAttachmentsBucket",
    "Effect": "Allow",
    "Action": ["s3:GetBucketLocation", "s3:ListAllMyBuckets"],
    "Resource": "*"
},
{
    "Sid": "CreateDeadlineCloudLogGroups",
    "Effect": "Allow",
    "Action": ["logs:CreateLogGroup"],
    "Resource": "arn:aws:logs:*:*:log-group:/aws/deadline/*",
    "Condition": {
        "StringLike": { "aws:CalledViaFirst": "deadline.amazonaws.com" }
    }
},
{
    "Sid": "ValidateDependencies",
    "Effect": "Allow",
    "Action": ["s3:ListBucket"],
    "Resource": "*",
    "Condition": {
        "StringLike": { "aws:CalledViaFirst": "deadline.amazonaws.com" }
    }
},
{
    "Sid": "RoleSelection",

```

```

        "Effect": "Allow",
        "Action": ["iam:GetRole", "iam:ListRoles",
"iam:ListAttachedRolePolicies"],
        "Resource": "*"
    },
    {
        "Sid": "PassRoleToDeadlineCloud",
        "Effect": "Allow",
        "Action": ["iam:PassRole"],
        "Condition": {
            "StringLike": { "iam:PassedToService": "deadline.amazonaws.com" }
        },
        "Resource": "*"
    },
    {
        "Sid": "KMSKeySelection",
        "Effect": "Allow",
        "Action": ["kms:ListKeys", "kms:ListAliases"],
        "Resource": "*"
    },
    {
        "Sid": "IdentityStoreReadOnly",
        "Effect": "Allow",
        "Action": [
            "identitystore:DescribeUser",
            "identitystore:DescribeGroup",
            "identitystore:ListGroups",
            "identitystore:ListUsers",
            "identitystore:IsMemberInGroups",
            "identitystore:ListGroupMemberships",
            "identitystore:ListGroupMembershipsForMember",
            "identitystore:GetGroupMembershipId"
        ],
        "Resource": "*"
    },
    {
        "Sid": "OrganizationAndIdentityCenterIdentification",
        "Effect": "Allow",
        "Action": [
            "sso:ListDirectoryAssociations",
            "organizations:DescribeAccount",
            "organizations:DescribeOrganization",
            "sso:DescribeRegisteredRegions",
            "sso:GetManagedApplicationInstance",

```

```

        "sso:GetSharedSsoConfiguration",
        "sso:ListInstances",
        "sso:GetApplicationAssignmentConfiguration",
        "sso:GetSSOStatus",
        "sso:ListRegions",
        "sso:DescribeRegion"
    ],
    "Resource": "*"
},
{
    "Sid": "ManagedDeadlineCloudIDCAApplication",
    "Effect": "Allow",
    "Action": [
        "sso:CreateApplication",
        "sso:PutApplicationAssignmentConfiguration",
        "sso:PutApplicationAuthenticationMethod",
        "sso:PutApplicationGrant",
        "sso:DeleteApplication",
        "sso:UpdateApplication"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": { "aws:CalledViaFirst": "deadline.amazonaws.com" }
    }
},
{
    "Sid": "ChooseSecret",
    "Effect": "Allow",
    "Action": ["secretsmanager:ListSecrets"],
    "Resource": "*"
},
{
    "Sid": "DeadlineMembershipActions",
    "Effect": "Allow",
    "Action": [
        "deadline:AssociateMemberToFarm",
        "deadline:AssociateMemberToFleet",
        "deadline:AssociateMemberToQueue",
        "deadline:AssociateMemberToJob",
        "deadline:DisassociateMemberFromFarm",
        "deadline:DisassociateMemberFromFleet",
        "deadline:DisassociateMemberFromQueue",
        "deadline:DisassociateMemberFromJob",
        "deadline:ListFarmMembers",

```

```
        "deadline:ListFleetMembers",
        "deadline:ListQueueMembers",
        "deadline:ListJobMembers"
    ],
    "Resource": ["*"]
},
{
    "Sid": "DeadlineControlPlaneActions",
    "Effect": "Allow",
    "Action": [
        "deadline:CreateMonitor",
        "deadline:GetMonitor",
        "deadline:UpdateMonitor",
        "deadline>DeleteMonitor",
        "deadline:ListMonitors",
        "deadline:CreateFarm",
        "deadline:GetFarm",
        "deadline:UpdateFarm",
        "deadline>DeleteFarm",
        "deadline:ListFarms",
        "deadline:CreateQueue",
        "deadline:GetQueue",
        "deadline:UpdateQueue",
        "deadline>DeleteQueue",
        "deadline:ListQueues",
        "deadline:CreateFleet",
        "deadline:GetFleet",
        "deadline:UpdateFleet",
        "deadline>DeleteFleet",
        "deadline:ListFleets",
        "deadline:ListWorkers",
        "deadline:CreateQueueFleetAssociation",
        "deadline:GetQueueFleetAssociation",
        "deadline:UpdateQueueFleetAssociation",
        "deadline>DeleteQueueFleetAssociation",
        "deadline:ListQueueFleetAssociations",
        "deadline:CreateQueueEnvironment",
        "deadline:GetQueueEnvironment",
        "deadline:UpdateQueueEnvironment",
        "deadline>DeleteQueueEnvironment",
        "deadline:ListQueueEnvironments",
        "deadline:CreateLimit",
        "deadline:GetLimit",
        "deadline:UpdateLimit",
```

```

        "deadline:DeleteLimit",
        "deadline:ListLimits",
        "deadline:CreateQueueLimitAssociation",
        "deadline:GetQueueLimitAssociation",
        "deadline>DeleteQueueLimitAssociation",
        "deadline:UpdateQueueLimitAssociation",
        "deadline>ListQueueLimitAssociations",
        "deadline>CreateStorageProfile",
        "deadline:GetStorageProfile",
        "deadline:UpdateStorageProfile",
        "deadline>DeleteStorageProfile",
        "deadline>ListStorageProfiles",
        "deadline>ListStorageProfilesForQueue",
        "deadline>ListBudgets",
        "deadline:TagResource",
        "deadline:UntagResource",
        "deadline>ListTagsForResource",
        "deadline>CreateLicenseEndpoint",
        "deadline:GetLicenseEndpoint",
        "deadline>DeleteLicenseEndpoint",
        "deadline>ListLicenseEndpoints",
        "deadline>ListAvailableMeteredProducts",
        "deadline>ListMeteredProducts",
        "deadline:PutMeteredProduct",
        "deadline>DeleteMeteredProduct",
        "deadline:GetMonitorSettings",
        "deadline:UpdateMonitorSettings",
        "deadline:GetVolume",
        "deadline>ListVolumes",
        "deadline>DeleteVolume"
    ],
    "Resource": ["*"]
}]
}

```

Policy to submit jobs to a queue

In this example, you create a scoped-down policy that grants permission to submit jobs to a specific queue in a specific farm.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "SubmitJobsFarmAndQueue",
      "Effect": "Allow",
      "Action": "deadline:CreateJob",
      "Resource": "arn:aws:deadline:us-east-1:111122223333:farm/FARM_A/
queue/QUEUE_B/job/*"
    }
  ]
}
```

Policy to allow creating a license endpoint

In this example, you create a scoped-down policy that grants the required permissions to create and manage license endpoints. Use this policy to create the license endpoint for the VPC associated with your farm.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "CreateLicenseEndpoint",
    "Effect": "Allow",
    "Action": [
      "deadline:CreateLicenseEndpoint",
      "deadline>DeleteLicenseEndpoint",
      "deadline:GetLicenseEndpoint",
      "deadline>ListLicenseEndpoints",
      "deadline:PutMeteredProduct",
      "deadline>DeleteMeteredProduct",
      "deadline>ListMeteredProducts",
      "deadline>ListAvailableMeteredProducts",
      "ec2:CreateVpcEndpoint",
      "ec2:DescribeVpcEndpoints",
      "ec2>DeleteVpcEndpoints"
    ]
  }]
}
```

```

    ],
    "Resource": [
        "arn:aws:deadline:*:111122223333:*",
        "arn:aws:ec2:*:111122223333:vpc-endpoint/*"
    ]
}]
}

```

Policy to allow monitoring a specific farm queue

In this example, you create a scoped-down policy that grants permission to monitor jobs in a specific queue for a specific farm.

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "MonitorJobsFarmAndQueue",
    "Effect": "Allow",
    "Action": [
      "deadline:SearchJobs",
      "deadline:ListJobs",
      "deadline:GetJob",
      "deadline:SearchSteps",
      "deadline:ListSteps",
      "deadline:ListStepConsumers",
      "deadline:ListStepDependencies",
      "deadline:GetStep",
      "deadline:SearchTasks",
      "deadline:ListTasks",
      "deadline:GetTask",
      "deadline:ListSessions",
      "deadline:GetSession",
      "deadline:ListSessionActions",
      "deadline:GetSessionAction"
    ],
    "Resource": [
      "arn:aws:deadline:us-east-1:123456789012:farm/FARM_A/queue/QUEUE_B",
      "arn:aws:deadline:us-east-1:123456789012:farm/FARM_A/queue/QUEUE_B/*"
    ]
  }]
}

```

```
}]
}
```

Policy to manage queue–fleet associations for a specific fleet

The queue–fleet associations on a fleet determine which jobs can be scheduled to the fleet's workers. If you reserve a fleet for sensitive content, control the associations so that no one can attach the fleet to an unapproved queue.

In this example, you create a scoped-down policy that grants permission to manage queue–fleet associations only between a specific fleet and a specific queue. The `CreateQueueFleetAssociation` operation authorizes against the farm, queue, and fleet resources, so a principal with this policy cannot associate the fleet with any other queue. In the following example, replace each *placeholder* with your resource-specific information.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "deadline:CreateQueueFleetAssociation",
        "deadline:UpdateQueueFleetAssociation",
        "deadline>DeleteQueueFleetAssociation",
        "deadline:GetQueueFleetAssociation",
        "deadline>ListQueueFleetAssociations"
      ],
      "Resource": [
        "arn:aws:deadline:REGION:ACCOUNT_ID:farm/FARM_ID",
        "arn:aws:deadline:REGION:ACCOUNT_ID:farm/FARM_ID/queue/QUEUE_ID",
        "arn:aws:deadline:REGION:ACCOUNT_ID:farm/FARM_ID/fleet/FLEET_ID"
      ]
    }
  ]
}
```

The scoping is effective only if no other policy grants the association operations on the restricted fleet. Audit the policies attached to your farm administrators, and for a stronger guarantee add an explicit Deny statement for the restricted fleet's Amazon Resource Name (ARN) to the principals

that must not change its associations. For more information about the security boundaries this protects, see [the section called “Isolate workloads”](#).

AWS managed policies for Deadline Cloud

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: AWSDeadlineCloud-FleetWorker

You can attach the AWSDeadlineCloud-FleetWorker policy to your AWS Identity and Access Management (IAM) identities.

This policy grants workers in this fleet the permissions that are needed to connect to and receive tasks from the service.

Permissions details

This policy includes the following permissions:

- `deadline` – Allows principals to manage workers in a fleet.

For a JSON listing of the policy details, see [AWSDeadlineCloud-FleetWorker](#) in the *AWS Managed Policy reference guide*.

AWS managed policy: AWSDeadlineCloud-WorkerHost

You can attach the AWSDeadlineCloud-WorkerHost policy to your IAM identities.

This policy grants the permissions that are needed to initially connect to the service. It can be used as an Amazon Elastic Compute Cloud (Amazon EC2) instance profile.

Permissions details

This policy includes the following permissions:

- `deadline` – Allows the user to create workers, assume the fleet role for workers, and apply tags to workers

For a JSON listing of the policy details, see [AWSDeadlineCloud-WorkerHost](#) in the *AWS Managed Policy reference guide*.

AWS managed policy: AWSDeadlineCloud-UserAccessFarms

You can attach the AWSDeadlineCloud-UserAccessFarms policy to your IAM identities.

This policy allows users to access farm data based on the farms that they are members of and their membership level.

Permissions details

This policy includes the following permissions:

- `deadline` – Allows the user to access farm data.
- `ec2` – Allows users to see details about Amazon EC2 instance types.
- `identitystore` – Allows users to see user and group names.
- `kms` – Allows users to configure AWS Key Management Service (AWS KMS) customer-managed keys for their AWS IAM Identity Center (IAM Identity Center) instance.

For a JSON listing of the policy details, see [AWSDeadlineCloud-UserAccessFarms](#) in the *AWS Managed Policy reference guide*.

AWS managed policy: AWSDeadlineCloud-UserAccessFleets

You can attach the AWSDeadlineCloud-UserAccessFleets policy to your IAM identities.

This policy allows users to access fleet data based on the farms that they are members of and their membership level.

Permissions details

This policy includes the following permissions:

- `deadline` – Allows the user to access farm data.
- `ec2` – Allows users to see details about Amazon EC2 instance types.
- `identitystore` – Allows users to see user and group names.

For a JSON listing of the policy details, see [AWSDeadlineCloud-UserAccessFleets](#) in the *AWS Managed Policy reference guide*.

AWS managed policy: AWSDeadlineCloud-UserAccessJobs

You can attach the AWSDeadlineCloud-UserAccessJobs policy to your IAM identities.

This policy allows users to access job data based on the farms that they are members of and their membership level.

Permissions details

This policy includes the following permissions:

- `deadline` – Allows the user to access farm data.
- `ec2` – Allows users to see details about Amazon EC2 instance types.
- `identitystore` – Allows users to see user and group names.

For a JSON listing of the policy details, see [AWSDeadlineCloud-UserAccessJobs](#) in the *AWS Managed Policy reference guide*.

AWS managed policy: AWSDeadlineCloud-UserAccessQueues

You can attach the AWSDeadlineCloud-UserAccessQueues policy to your IAM identities.

This policy allows users to access queue data based on the farms that they are members of and their membership level.

Permissions details

This policy includes the following permissions:

- `deadline` – Allows the user to access farm data.
- `ec2` – Allows users to see details about Amazon EC2 instance types.
- `identitystore` – Allows users to see user and group names.

For a JSON listing of the policy details, see [AWSDeadlineCloud-UserAccessQueues](#) in the *AWS Managed Policy reference guide*.

Deadline Cloud updates to AWS managed policies

View details about updates to AWS managed policies for Deadline Cloud since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Deadline Cloud Document history page.

Change	Description	Date
AWSDeadlineCloud-UserAccessFarms – Change	Deadline Cloud added new action <code>kms:Decrypt</code> so you can use an AWS KMS customer-managed key with your IAM Identity Center instance.	December 22, 2025
AWSDeadlineCloud-WorkerHost – Change	Deadline Cloud added new actions <code>deadline:TagResource</code> and <code>deadline:ListTagsForResource</code> to allow you to add and view tags	May 30, 2025

Change	Description	Date
	associated with workers in your fleet.	
AWSDeadlineCloud-UserAccessFarms – Change AWSDeadlineCloud-UserAccessJobs – Change AWSDeadlineCloud-UserAccessQueues – Change	Deadline Cloud added new actions <code>deadline:GetJobTemplate</code> and <code>deadline:ListJobParameterDefinitions</code> to allow you to resubmit jobs.	October 7, 2024
Deadline Cloud started tracking changes	Deadline Cloud started tracking changes to its AWS managed policies.	April 2, 2024

Service roles

How Deadline Cloud uses IAM service roles

Deadline Cloud automatically assumes IAM roles and provides temporary credentials to workers, jobs, and the Deadline Cloud monitor. This approach eliminates manual credential management while maintaining security through role-based access control.

Four roles cover the lifecycle of work in a farm. On a customer-managed fleet, a new host starts with the *worker host role*, uses it to register as a worker, and trades it for *fleet role* credentials. On a service-managed fleet, the service performs that bootstrap for you, so only the other three roles apply. The worker uses the fleet role to receive work and report progress, and receives *queue role* credentials while it runs each job. People and tools get their credentials from the *monitor role* when they sign in to the Deadline Cloud monitor.

Deadline Cloud service roles at a glance

Role	Who uses its credentials	What it grants	Associated resource
Worker host role	Customer-managed fleet hosts during startup	Register a new worker and assume the fleet role	The worker host, for example through an Amazon EC2 instance profile
Fleet role	Workers on the fleet	Receive work, report progress, and write worker logs to CloudWatch Logs	The fleet
Queue role	Jobs while they run; monitor and CLI users working with job attachments and logs	Read and write the queue's job attachments bucket, read job logs, download third-party software, plus any permissions you add for your jobs	The queue
Monitor role	People signed in to the Deadline Cloud monitor, and the CLI and submitters through the profile it creates	Access to farm, fleet, queue, and job data based on each user's memberships and access level	The monitor

When you create monitors, fleets, and queues in the console, Deadline Cloud can create the fleet, queue, and monitor roles for you with the necessary permissions. You create the worker host role yourself when you set up a customer-managed fleet. Use the following sections to understand each role for troubleshooting, to create the roles yourself, or to extend them.

Fleet role

Configure a fleet role to give Deadline Cloud workers the permissions they need to receive work and report progress on that work.

You usually do not have to configure this role yourself. This role can be created for you in the Deadline Cloud console to include the necessary permissions. Use the following guide to understand the specifics of this role for troubleshooting.

When creating or updating fleets programmatically, specify the fleet role ARN using the `CreateFleet` or `UpdateFleet` API operations.

What the fleet role does

The fleet role provides workers with permissions to:

- Receive new work and report progress on ongoing work to the Deadline Cloud service
- Manage worker lifecycle and status
- Record log events to Amazon CloudWatch Logs for the worker logs

Set up the fleet role trust policy

Your fleet role must trust the Deadline Cloud service and be scoped to your specific farm.

As a best practice, the trust policy should include security conditions for Confused Deputy protection. To learn more about Confused Deputy protection, see [Confused Deputy](#) in the *Deadline Cloud User Guide*.

- `aws:SourceAccount` ensures only resources from the same AWS account can assume this role.
- `aws:SourceArn` restricts role assumption to a specific Deadline Cloud farm.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDeadlineCredentialsService",
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Principal": {
```

```
    "Service": "credentials.deadline.amazonaws.com"
  },
  "Condition": {
    "StringEquals": {
      "aws:SourceAccount": "YOUR_ACCOUNT_ID"
    },
    "ArnEquals": {
      "aws:SourceArn": "arn:aws:deadline:REGION:YOUR_ACCOUNT_ID:farm/YOUR_FARM_ID"
    }
  }
}
```

Attach the Fleet role permissions

Attach the following AWS managed policy to your fleet role:

[AWSDeadlineCloud-FleetWorker](#)

This managed policy provides permissions for:

- `deadline:AssumeFleetRoleForWorker` - Allows workers to refresh their credentials.
- `deadline:UpdateWorker` - Allows workers to update their status (for example, to STOPPED when exiting).
- `deadline:UpdateWorkerSchedule` - For obtaining work and reporting progress.
- `deadline:BatchGetJobEntity` - For fetching job information.
- `deadline:AssumeQueueRoleForWorker` - For accessing queue role credentials during job execution.

Add KMS permissions for encrypted farms

If your farm was created using a KMS key, add these permissions to your fleet role to ensure the worker can access encrypted data in the farm.

The KMS permissions are only necessary if your farm has an associated KMS key. The `kms:ViaService` condition must use the format `deadline.{region}.amazonaws.com`.

When creating a fleet, a CloudWatch Logs log group is created for that fleet. The worker's permissions are used by the Deadline Cloud service to create a log stream specifically for that

particular worker. After the worker is set up and running, the worker will use these permissions to send log events directly to CloudWatch Logs.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CreateLogStream",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogStream"
      ],
      "Resource": "arn:aws:logs:REGION:YOUR_ACCOUNT_ID:log-group:/aws/
deadline/YOUR_FARM_ID/*",
      "Condition": {
        "ForAnyValue:StringEquals": {
          "aws:CalledVia": [
            "deadline.REGION.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid": "ManageLogEvents",
      "Effect": "Allow",
      "Action": [
        "logs:PutLogEvents",
        "logs:GetLogEvents"
      ],
      "Resource": "arn:aws:logs:REGION:YOUR_ACCOUNT_ID:log-group:/aws/
deadline/YOUR_FARM_ID/*"
    },
    {
      "Sid": "ManageKmsKey",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:DescribeKey",
        "kms:GenerateDataKey"
      ],
      "Resource": "YOUR_FARM_KMS_KEY_ARN",
      "Condition": {
        "StringEquals": {
```

```
        "kms:ViaService": "deadline.REGION.amazonaws.com"
    }
}
}
```

Modifying the fleet role

Permissions for the fleet role are not customizable. The described permissions are always required and adding additional permissions has no effect.

Worker host role

Set up a worker host role if you use customer-managed fleets on Amazon EC2 instances or on-premises hosts.

What the worker host role does

The worker host role (`WorkerHost`) bootstraps workers on customer-managed fleet hosts. It provides the minimal permissions needed for a host to:

- Create a worker in Deadline Cloud
- Assume the fleet role to fetch operational credentials
- Tag workers with fleet tags (if tag propagation is enabled)

Set up worker host role permissions

Attach the following AWS managed policy to your worker host role:

[AWSDeadlineCloud-WorkerHost](#)

This managed policy provides permissions for:

- `deadline:CreateWorker` - Allows the host to register a new worker.
- `deadline:AssumeFleetRoleForWorker` - Allows the host to assume the fleet role.
- `deadline:TagResource` - Allows tagging workers during creation (if enabled).
- `deadline:ListTagsForResource` - Allows reading fleet tags for propagation.

Understand the bootstrap process

The worker host role is only used during initial worker startup:

1. The worker agent starts on the host using worker host role credentials.
2. It invokes `deadline:CreateWorker` to register with Deadline Cloud.
3. It then invokes `deadline:AssumeFleetRoleForWorker` to fetch fleet role credentials.
4. From this point forward, the worker uses only fleet role credentials for all operations.

After the worker starts running, it no longer uses the worker host role. Service-managed fleets don't need this role because the service performs the bootstrap automatically.

Queue role

The queue role is assumed by the worker when processing a task. This role provides the permissions needed to complete the task.

When creating or updating queues programmatically, specify the queue role ARN using the `CreateQueue` or `UpdateQueue` API operations.

Set up the queue role trust policy

Your queue role must trust the Deadline Cloud service.

As a best practice, the trust policy should include security conditions for Confused Deputy protection. To learn more about Confused Deputy protection, see [Confused Deputy](#) in the *Deadline Cloud User Guide*.

- `aws:SourceAccount` ensures only resources from the same AWS account can assume this role.
- `aws:SourceArn` restricts role assumption to a specific Deadline Cloud farm.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
```

```

        "credentials.deadline.amazonaws.com",
        "deadline.amazonaws.com"
    ]
},
"Action": "sts:AssumeRole",
"Condition": {
    "StringEquals": {
        "aws:SourceAccount": "YOUR_ACCOUNT_ID"
    },
    "ArnEquals": {
        "aws:SourceArn": "arn:aws:deadline:us-west-2:123456789012:farm/{farm-id}"
    }
}
}
]
}

```

Understand queue role permissions

The queue role doesn't use a single managed policy. Instead, when you configure your queue in the console, Deadline Cloud creates a custom policy for your queue based on your configuration.

This automatically created policy provides access to:

Job attachments

Read and write access to your specified Amazon S3 bucket for job input and output files:

```

{
    "Effect": "Allow",
    "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket",
        "s3:GetBucketLocation"
    ],
    "Resource": [
        "arn:aws:s3:::YOUR_JOB_ATTACHMENTS_BUCKET",
        "arn:aws:s3:::YOUR_JOB_ATTACHMENTS_BUCKET/YOUR_PREFIX/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:ResourceAccount": "YOUR_ACCOUNT_ID"
        }
    }
}

```

```
}  
}  
}
```

Job logs

Read access to CloudWatch Logs for jobs in this queue. Each queue has its own log group and each session has its own log stream:

```
{  
  "Effect": "Allow",  
  "Action": [  
    "logs:GetLogEvents"  
  ],  
  "Resource": "arn:aws:logs:REGION:YOUR_ACCOUNT_ID:log-group:/aws/  
deadline/YOUR_FARM_ID/*"  
}
```

Third-party software

Access to download third-party software supported by Deadline Cloud (such as Maya, Blender, and others):

```
{  
  "Effect": "Allow",  
  "Action": [  
    "s3:ListBucket",  
    "s3:GetObject"  
  ],  
  "Resource": "*",  
  "Condition": {  
    "ArnLike": {  
      "s3:DataAccessPointArn": "arn:aws:s3:*:*:accesspoint/deadline-software-*"  
    },  
    "StringEquals": {  
      "s3:AccessPointNetworkOrigin": "VPC"  
    }  
  }  
}
```


Add permissions for your jobs

Add permissions to your queue role for AWS services that your jobs need to access. When writing OpenJobDescription step scripts, the AWS CLI and SDK will automatically use credentials from your queue role. Use this to access additional services needed to complete your job.

Example use cases include:

- for fetching custom data
- SSM permissions to tunnel to a custom license server
- CloudWatch for emitting custom metrics
- Deadline Cloud permission to create new jobs for dynamic workflows

How queue role credentials are used

Deadline Cloud provides queue role credentials to:

- Workers during job execution
- Users via Deadline Cloud CLI and monitor when interacting with job attachments and logs

Deadline Cloud creates separate CloudWatch Logs log groups for each queue. The Deadline Cloud CLI and monitor use the queue role (through `deadline:AssumeQueueRoleForRead`) to read job logs from the queue's log group. The Deadline Cloud CLI and monitor use the queue role (through `deadline:AssumeQueueRoleForUser`) to upload or download job attachments data.

Monitor role

Configure a monitor role to give the Deadline Cloud monitor web and desktop applications access to your Deadline Cloud resources.

When creating or updating monitors programmatically, specify the monitor role ARN using the `CreateMonitor` or `UpdateMonitor` API operations.

What the monitor role does

The monitor role enables Deadline Cloud monitor to provide end users with access to:

- Basic functionality required for the Deadline Cloud Integrated Submitters, CLI and monitor
- Custom functionality for end users

Set up the monitor role trust policy

Your monitor role must trust the Deadline Cloud service.

As a best practice, the trust policy should include security conditions for Confused Deputy protection. To learn more about Confused Deputy protection, see [Confused Deputy](#) in the *Deadline Cloud User Guide*.

`aws:SourceAccount` ensures only resources from the same AWS account can assume this role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.deadline.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "YOUR_ACCOUNT_ID"
        }
      }
    }
  ]
}
```

Attach monitor role permissions

Attach all of the following AWS managed policies to your monitor role for basic operation:

- [AWSDeadlineCloud-UserAccessFarms](#)
- [AWSDeadlineCloud-UserAccessFleets](#)
- [AWSDeadlineCloud-UserAccessJobs](#)
- [AWSDeadlineCloud-UserAccessQueues](#)

How the monitor role works

When using the Deadline Cloud monitor, a service user signs in using AWS IAM Identity Center (IAM Identity Center), and the monitor role is assumed. The assumed role credentials are used by the

monitor application to display the monitor UI, including the list of farms, fleets, queues, and other information.

When using the Deadline Cloud monitor desktop application, these credentials are additionally made available on the workstation using a named AWS credential profile corresponding to the profile name provided by the end user. Learn more about named profiles in the [AWS SDK and Tools reference guide](#).

This named profile is how the Deadline CLI and submitters access Deadline Cloud resources.

Customizing the monitor role for advanced use cases

You can customize the monitor role to modify what users can do at each access level (Viewer, Contributor, Manager, Owner) or to add permissions for advanced workflows.

Customizing access level permissions

The four AWS managed policies attached to the monitor role control what each access level can do. You can add custom policies to the monitor role to grant or restrict permissions for specific access levels using the `deadline:MembershipLevel` condition key.

For example, to allow Contributors to update and cancel jobs (which is normally restricted to Managers and Owners), add a policy like the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "deadline:UpdateJob",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "deadline:MembershipLevel": "CONTRIBUTOR"
        }
      }
    }
  ]
}
```

With this policy, Contributors can update and cancel jobs in addition to submitting them.

Adding permissions for advanced workflows

You can add custom IAM policies to the monitor role to grant additional permissions to all monitor users. Custom policies are useful for advanced scripting workflows where users need access to AWS services beyond the standard Deadline Cloud functionality.

Follow these guidelines when modifying your monitor role:

- Don't remove any of the managed policies. Removing these policies breaks monitor functionality.

How Deadline Cloud monitor uses monitor role credentials

Deadline Cloud monitor automatically obtains monitor role credentials when you authenticate. These temporary credentials last for 15 minutes and automatically refresh as long as you are signed in to IAM Identity Center. This capability enables the desktop application to provide monitoring capabilities beyond what's available in a standard web browser.

When you log in with Deadline Cloud monitor, it automatically creates a profile that you can use with the AWS CLI or any other AWS tool. This profile uses the monitor role credentials, giving you programmatic access to AWS services based on the permissions in your monitor role.

Deadline Cloud submitters work the same way - they use the profile created by Deadline Cloud monitor to access AWS services with the appropriate role permissions.

Advanced customization of Deadline Cloud roles

You can extend Deadline Cloud roles with additional permissions to enable advanced use cases beyond basic rendering workflows. This approach uses the access management system in Deadline Cloud to control access to additional AWS services based on queue membership.

Team collaboration with AWS CodeCommit

Add AWS CodeCommit permissions to your Queue role to enable team collaboration on project repositories. This approach uses the access management system in Deadline Cloud for additional use cases. Only users with access to the specific queue receive these AWS CodeCommit permissions, so you can manage per-project repository access through Deadline Cloud queue membership.

Repository access through queue membership is useful when artists need to access project-specific assets, scripts, or configuration files stored in AWS CodeCommit repositories as part of their rendering workflow.

Add AWS CodeCommit permissions to queue role

Add the following permissions to your queue role to enable AWS CodeCommit access:

```
{
  "Effect": "Allow",
  "Action": [
    "codecommit:GitPull",
    "codecommit:GitPush",
    "codecommit:GetRepository",
    "codecommit:ListRepositories"
  ],
  "Resource": "arn:aws:codecommit:REGION:YOUR_ACCOUNT_ID:PROJECT_REPOSITORY"
}
```

Set up credential provider on artist workstations

Configure each artist workstation to use Deadline Cloud queue credentials for AWS CodeCommit access. This setup is done once per workstation.

To configure the credential provider

1. Add a credential provider profile to your AWS config file (`~/.aws/config`):

```
[profile queue-codecommit]
credential_process = deadline queue export-credentials --farm-id farm-XXXXXXXXXXXXXXXXXXXXXXXXXXXX --queue-id queue-XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

2. Configure Git to use this profile for AWS CodeCommit repositories:

```
git config --global credential.https://git-codecommit.REGION.amazonaws.com.helper
'!aws codecommit credential-helper --profile queue-codecommit $@'
git config --global credential.https://git-codecommit.REGION.amazonaws.com.UseHttpPath true
```

Replace *farm-XXXXXXXXXXXXXXXXXXXXXXXXXXXX* and *queue-XXXXXXXXXXXXXXXXXXXXXXXXXXXX* with your actual farm and queue IDs. Replace *REGION* with your AWS region (for example, `us-west-2`).

Using AWS CodeCommit with queue credentials

Once configured, Git operations will automatically use the queue role credentials when accessing AWS CodeCommit repositories. The `deadline queue export-credentials` command returns temporary credentials that look like this:

```
{
  "Version": 1,
  "AccessKeyId": "ASIA...",
  "SecretAccessKey": "...",
  "SessionToken": "...",
  "Expiration": "2025-11-10T23:02:23+00:00"
}
```

Deadline Cloud automatically refreshes these credentials as needed, and your Git operations work without additional configuration:

```
git clone https://git-codecommit.REGION.amazonaws.com/v1/repos/PROJECT_REPOSITORY
git pull
git push
```

Artists can now access project repositories using their queue permissions without needing separate AWS CodeCommit credentials. Only users with access to the specific queue will be able to access the associated repository, so you can control repository access through queue membership in Deadline Cloud.

Troubleshooting AWS Deadline Cloud identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Deadline Cloud and IAM.

Topics

- [I am not authorized to perform an action in Deadline Cloud](#)
- [I am not authorized to perform iam:PassRole](#)
- [I want to allow people outside of my AWS account to access my Deadline Cloud resources](#)

I am not authorized to perform an action in Deadline Cloud

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

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 doesn't have the fictional deadline:`GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
deadline:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the `my-example-widget` resource by using the deadline:`GetWidget` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

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 Deadline Cloud.

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 Deadline Cloud. 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 want to allow people outside of my AWS account to access my Deadline Cloud 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 Deadline Cloud supports these features, see [How Deadline Cloud 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*.

Data protection in Deadline Cloud

The AWS [shared responsibility model](#) applies to data protection in AWS Deadline Cloud. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see [Data Privacy FAQ](#). For information about data protection in Europe, see the [General Data Protection Regulation \(GDPR\) Center](#).

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, 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. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see [Working with CloudTrail trails](#) in the *AWS CloudTrail User Guide*.
- 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 sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-3](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Deadline Cloud or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

The data entered into name fields in Deadline Cloud job templates may also be included in billing or diagnostic logs and should not contain confidential or sensitive information.

Topics

- [Encryption at rest](#)
- [Encryption in transit](#)
- [Key management](#)
- [Inter-network traffic privacy](#)
- [Opt out](#)

Encryption at rest

AWS Deadline Cloud protects sensitive data by encrypting it at rest using encryption keys stored in [AWS Key Management Service \(AWS KMS\)](#). Encryption at rest is available in all AWS Regions where Deadline Cloud is available.

Encrypting data means sensitive data saved on disks isn't readable by a user or application without a valid key. Only a party with a valid managed key can decrypt the data.

Deadline Cloud deletes Amazon Elastic Block Store volumes when service-managed fleet worker instances terminate.

For information about how Deadline Cloud uses AWS KMS for encrypting data at rest, see [Key management](#).

Encryption in transit

For data in transit, AWS Deadline Cloud uses Transport Layer Security (TLS) 1.2 or 1.3 to encrypt data sent between the service and workers. We require TLS 1.2 and recommend TLS 1.3. Additionally, if you use a virtual private cloud (VPC), you can use AWS PrivateLink to establish a private connection between your VPC and Deadline Cloud.

Key management

When creating a new farm, you can choose one of the following keys to encrypt your farm data:

- **AWS owned KMS key** – Default encryption type if you don't specify a key when you create the farm. The KMS key is owned by AWS Deadline Cloud. You can't view, manage, or use AWS owned keys. However, you don't need to take any action to protect the keys that encrypt your data. For more information, see [AWS owned keys](#) in the *AWS Key Management Service developer guide*.
- **Customer managed KMS key** – You specify a customer managed key when you create a farm. All of the content within the farm is encrypted with the KMS key. The key is stored in your account and is created, owned, and managed by you and AWS KMS charges apply. You have full control over the KMS key. You can perform such tasks as:
 - Establishing and maintaining key policies
 - Establishing and maintaining IAM policies and grants
 - Enabling and disabling key policies
 - Adding tags
 - Creating key aliases

You can't manually rotate a customer owned key used with a Deadline Cloud farm. Automatic rotation of the key is supported.

For more information, see [Customer owned keys](#) in the *AWS Key Management Service Developer Guide*.

To create a customer managed key, follow the steps for [Creating symmetric customer managed keys](#) in the *AWS Key Management Service Developer Guide*.

How Deadline Cloud uses AWS KMS grants

Deadline Cloud requires a [grant](#) to use your customer managed key. When you create a farm encrypted with a customer managed key, Deadline Cloud creates a grant on your behalf by sending a [CreateGrant](#) request to AWS KMS to get access to the KMS key that you specified.

Deadline Cloud uses multiple grants. Each grant is used by a different part of Deadline Cloud that needs to encrypt or decrypt your data. Deadline Cloud also uses grants to allow access to other AWS services used to store data on your behalf, such as Amazon Simple Storage Service, Amazon Elastic Block Store, or OpenSearch.

Grants that enable Deadline Cloud to manage machines in a service-managed fleet include a Deadline Cloud account number and role in the `GranteePrincipal` instead of a service principal. While not typical, the grant configuration is necessary to encrypt Amazon EBS volumes for workers in service-managed fleets using the customer managed KMS key specified for the farm.

Customer managed key policy

Key policies control access to your customer managed key. Each key must have exactly one key policy that contains statements that determine who can use the key and how they can use it. When you create your customer managed key, you can specify a key policy. For more information, see [Managing access to customer managed keys](#) in the *AWS Key Management Service Developer Guide*.

Minimal IAM policy for CreateFarm

To use your customer managed key to create farms using the console or the [CreateFarm](#) API operation, the following AWS KMS API operations must be permitted:

- [kms:CreateGrant](#) – Adds a grant to a customer managed key. Grants console access to a specified AWS KMS key. For more information, see [Using grants](#) in the *AWS Key Management Service developer guide*.
- [kms:Decrypt](#) – Allows Deadline Cloud to decrypt data in the farm.

- [kms:DescribeKey](#) – Provides the customer managed key details to allow Deadline Cloud to validate the key.
- [kms:GenerateDataKey](#) – Allows Deadline Cloud to encrypt data using a unique data key.

The following policy statement grants the necessary permissions for the CreateFarm operation.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DeadlineCreateGrants",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey",
        "kms:CreateGrant",
        "kms:DescribeKey"
      ],
      "Resource": "arn:aws:kms:us-west-2:111122223333:key/1234567890abcdef0",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "deadline.us-west-2.amazonaws.com"
        }
      }
    }
  ]
}
```

Minimal IAM policy for read-only operations

To use your customer managed key for read-only Deadline Cloud operations, such as getting information about farms, queues, and fleets. The following AWS KMS API operations must be permitted:

- [kms:Decrypt](#) – Allows Deadline Cloud to decrypt data in the farm.

- [kms:DescribeKey](#) – Provides the customer managed key details to allow Deadline Cloud to validate the key.

The following policy statement grants the necessary permissions for read-only operations.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DeadlineReadOnly",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:DescribeKey"
      ],
      "Resource": "arn:aws:kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "deadline.us-west-2.amazonaws.com"
        }
      }
    }
  ]
}
```

Minimal IAM policy for read-write operations

To use your customer managed key for read-write Deadline Cloud operations, such as creating and updating farms, queues, and fleets. The following AWS KMS API operations must be permitted:

- [kms:Decrypt](#) – Allows Deadline Cloud to decrypt data in the farm.
- [kms:DescribeKey](#) – Provides the customer managed key details to allow Deadline Cloud to validate the key.
- [kms:GenerateDataKey](#) – Allows Deadline Cloud to encrypt data using a unique data key.

The following policy statement grants the necessary permissions for the CreateFarm operation.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DeadlineReadWrite",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:DescribeKey",
        "kms:GenerateDataKey"
      ],
      "Resource": "arn:aws:kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "deadline.us-west-2.amazonaws.com"
        }
      }
    }
  ]
}
```

Monitoring your encryption keys

When you use an AWS KMS customer managed key with your Deadline Cloud farms, you can use [AWS CloudTrail](#) or [Amazon CloudWatch Logs](#) to track requests that Deadline Cloud sends to AWS KMS.

Deadline Cloud generates a CreateGrant event when it sets up access to your key, and Decrypt and GenerateDataKey events when it uses the key. The events include an encryptionContext that identifies the farm. For more information about the events, see [Logging AWS KMS API calls with AWS CloudTrail](#) in the *AWS Key Management Service Developer Guide*.

CloudTrail event for grants

The following example CloudTrail event occurs when grants are created, typically when you call the CreateFarm, CreateMonitor, or CreateFleet operation.

```

{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROAIQDTESTANDEXAMPLE:SampleUser01",
    "arn": "arn:aws::sts::111122223333:assumed-role/Admin/SampleUser01",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE3",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROAIQDTESTANDEXAMPLE",
        "arn": "arn:aws::iam::111122223333:role/Admin",
        "accountId": "111122223333",
        "userName": "Admin"
      },
      "webIdFederationData": {},
      "attributes": {
        "creationDate": "2024-04-23T02:05:26Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "deadline.amazonaws.com"
  },
  "eventTime": "2024-04-23T02:05:35Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "CreateGrant",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "deadline.amazonaws.com",
  "userAgent": "deadline.amazonaws.com",
  "requestParameters": {
    "operations": [
      "CreateGrant",
      "Decrypt",
      "DescribeKey",
      "Encrypt",
      "GenerateDataKey"
    ],
    "constraints": {
      "encryptionContextSubset": {
        "aws:deadline:farmId": "farm-abcdef12345678900987654321fedcba",
        "aws:deadline:accountId": "111122223333"
      }
    }
  }
}

```

```

    },
    "granteePrincipal": "deadline.amazonaws.com",
    "keyId": "arn:aws::kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
    "retiringPrincipal": "deadline.amazonaws.com"
  },
  "responseElements": {
    "grantId": "6bbe819394822a400fe5e3a75d0e9ef16c1733143fff0c1fc00dc7ac282a18a0",
    "keyId": "arn:aws::kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
  },
  "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
  "readOnly": false,
  "resources": [
    {
      "accountId": "AWS Internal",
      "type": "AWS::KMS::Key",
      "ARN": "arn:aws::kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE44444"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "111122223333",
  "eventCategory": "Management"
}

```

Deleting a customer managed KMS key

Deleting a customer managed KMS key in AWS Key Management Service (AWS KMS) is destructive and potentially dangerous. It irreversibly deletes the key material and all metadata associated with the key. After a customer managed KMS key is deleted, you can no longer decrypt the data that was encrypted by that key. Deleting the key means that the data becomes unrecoverable.

For safety, AWS KMS gives customers a waiting period of up to 30 days before deleting the KMS key. The default waiting period is 30 days.

About the waiting period

Because it's destructive and potentially dangerous to delete a customer managed KMS key, we require that you set a waiting period of 7–30 days. The default waiting period is 30 days.

However, the actual waiting period might be up to 24 hours longer than the period you scheduled. To get the actual date and time when the key will be deleted, use the [DescribeKey](#) operation. You can also see the scheduled deletion date of a key in the [AWS KMS console](#) on the key's detail page, in the **General configuration** section. Notice the time zone.

During the waiting period, the customer managed key's status and key state is **Pending deletion**.

- A customer managed KMS key that is pending deletion can't be used in any [cryptographic operations](#).
- AWS KMS doesn't [rotate the backing keys](#) of customer managed KMS keys that are pending deletion.

For more information about deleting a customer managed KMS key, see [Deleting customer master keys](#) in the *AWS Key Management Service Developer Guide*.

Inter-network traffic privacy

AWS Deadline Cloud supports Amazon Virtual Private Cloud (Amazon VPC) to secure connections. Amazon VPC provides features that you can use to increase and monitor the security for your virtual private cloud (VPC).

You can set up a customer-managed fleet (CMF) with Amazon Elastic Compute Cloud (Amazon EC2) instances that run inside a VPC. By deploying Amazon VPC endpoints to use AWS PrivateLink, traffic between workers in your CMF and the Deadline Cloud endpoint stays within your VPC. Furthermore, you can configure your VPC to restrict internet access to your instances.

In service-managed fleets, workers aren't reachable from the internet, but they do have internet access and connect to the Deadline Cloud service over the internet. Each service-managed fleet runs in its own isolated network, and worker instances remain dedicated to individual customers.

Opt out

AWS Deadline Cloud collects certain operational information to help us develop and improve Deadline Cloud. The collected data includes things such as your AWS account ID and user ID, so that we can correctly identify you if you have an issue with the Deadline Cloud. We also collect Deadline Cloud specific information, such as Resource IDs (a FarmID or QueueID when applicable), the product name (for example, JobAttachments, WorkerAgent, and more) and the product version.

You can choose to opt out from this data collection using application configuration. Each computer interacting with Deadline Cloud, both client workstations and fleet workers, needs to opt out separately.

Deadline Cloud monitor - desktop

Deadline Cloud monitor - desktop collects operational information, such as when crashes occur and when the application is opened, to help us know when you are having problems with the application. To opt out from the collection of this operational information, go to the settings page and clear **Turn on data collection to measure Deadline Cloud Monitor's performance**.

After you opt out, the desktop monitor no longer sends the operational data. Any previously collected data is retained and may still be used to improve the service. For more information, see [Data Privacy FAQ](#).

AWS Deadline Cloud CLI and Tools

The AWS Deadline Cloud CLI, submitters, and worker agent all collect operational information such as when crashes occur and when jobs are submitted to help us know when you are having problems with these applications. To opt out from the collection of this operational information, use any of the following methods:

- In the terminal, enter **deadline config set telemetry.opt_out true**.

The command opts out the CLI, submitters, and worker agent when running as the current user.

- When installing the Deadline Cloud worker agent, add the **--telemetry-opt-out** command line argument. For example, **./install.sh --farm-id \$FARM_ID --fleet-id \$FLEET_ID --telemetry-opt-out**.
- Before running the worker agent, CLI, or submitter, set an environment variable:
DEADLINE_CLOUD_TELEMETRY_OPT_OUT=true

After you opt out, the Deadline Cloud tools no longer send the operational data. Any previously collected data is retained and may still be used to improve the service. For more information, see [Data Privacy FAQ](#).

Access AWS Deadline Cloud using an interface endpoint (AWS PrivateLink)

You can use AWS PrivateLink to create a private connection between your VPC and AWS Deadline Cloud. You can access Deadline Cloud as if it were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or Direct Connect connection. Instances in your VPC don't need public IP addresses to access Deadline Cloud.

You establish this private connection by creating an *interface endpoint*, powered by AWS PrivateLink. We create an endpoint network interface in each subnet that you enable for the interface endpoint. These are requester-managed network interfaces that serve as the entry point for traffic destined for Deadline Cloud.

Deadline Cloud also has dual-stack endpoints available. Dual-stack endpoints support requests over IPv6 and IPv4.

For more information, see [Access AWS services through AWS PrivateLink](#) in the *AWS PrivateLink Guide*.

Considerations for Deadline Cloud

Before you set up an interface endpoint for Deadline Cloud, see [Access an AWS service using an interface VPC endpoint](#) in the *AWS PrivateLink Guide*.

Deadline Cloud supports making calls to all of its API actions through the interface endpoint.

By default, full access to Deadline Cloud is allowed through the interface endpoint. Alternatively, you can associate a security group with the endpoint network interfaces to control traffic to Deadline Cloud through the interface endpoint.

Deadline Cloud also supports VPC endpoint policies. For more information, see [Control access to VPC endpoints using endpoint policies](#) in the *AWS PrivateLink Guide*.

Deadline Cloud endpoints

Deadline Cloud uses four endpoints for access to the service using AWS PrivateLink - two for IPv4 and two for IPv6.

Workers use the `scheduling.deadline.region.amazonaws.com` endpoint to get tasks from the queue, report progress to Deadline Cloud, and to send task output back. If you are using a

customer-managed fleet, the scheduling endpoint is the only endpoint that you need to create unless you are using management operations. For example, if a job creates more jobs, you need to enable the management endpoint to call the `CreateJob` operation.

The Deadline Cloud monitor uses the `management.deadline.region.amazonaws.com` to manage the resources in your farm, such as creating and modifying queues and fleets or getting lists of jobs, steps, and tasks.

The AWS SDKs and CLI automatically add the management and scheduling prefixes to the endpoint. If you want to disable this behavior, see the [host prefix injection](#) section in the *AWS SDKs and Tools Reference Guide*.

Deadline Cloud also requires endpoints for the following AWS service endpoints:

- If you set up your customer-managed fleet in a subnet with no internet connection you must create a VPC endpoint for Amazon CloudWatch Logs so that workers can write logs. For more information, see [Monitoring with CloudWatch](#).
- If you use job attachments, you must create a VPC endpoint for Amazon Simple Storage Service (Amazon S3) so that workers can access the attachments. For more information, see [Job attachments in Deadline Cloud](#).

Create endpoints for Deadline Cloud

You can create interface endpoints for Deadline Cloud using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see [Create an interface endpoint](#) in the *AWS PrivateLink Guide*.

Create management and scheduling endpoints for Deadline Cloud using the following service names. Replace *region* with the AWS Region where you've deployed Deadline Cloud.

```
com.amazonaws.region.deadline.management
```

```
com.amazonaws.region.deadline.scheduling
```

Deadline Cloud supports dual-stack endpoints.

If you enable private DNS for the interface endpoints, you can make API requests to Deadline Cloud using its default Regional DNS name. For example, `scheduling.deadline.us-`

east-1.amazonaws.com for worker operations, or management.deadline.us-east-1.amazonaws.com for all other operations.

If your customer-managed fleet is on a subnet without an internet connection, you must create a CloudWatch Logs endpoint using the following service name:

```
com.amazonaws.region.logs
```

If you use job attachments to transfer files, you must create an Amazon S3 endpoint using the following service name:

```
com.amazonaws.region.s3
```

Restricted network environments

Deadline Cloud provides tools that are used by artists or other users on their local workstations. These tools require access to AWS API and web endpoints to perform their function. If you filter access to specific AWS domains or URL endpoints by using a web content filtering solution such as next-generation firewalls (NGFW) or Secure Web Gateways (SWG), you must add the following domains or URL endpoints to your web-content filtering solution allowlists.

AWS API endpoints to allowlist

Deadline Cloud client tools, such as the AWS Management Console, monitor, CLI, and integrated submitters, require access to AWS APIs in addition to Deadline Cloud. These endpoints only support IPv4.

- scheduling.deadline.*[Region]*.amazonaws.com
- management.deadline.*[Region]*.amazonaws.com
- logs.*[Region]*.amazonaws.com
- ec2.*[Region]*.amazonaws.com
- s3.*[Region]*.amazonaws.com
- sts.*[Region]*.amazonaws.com
- identitystore.*[Region]*.amazonaws.com

Web domains to allowlist

The Deadline Cloud monitor requires access to the following domains to operate.

For additional information about allowlisting domains for AWS Sign-In, see [Domains to add to your allow list](#) in the *AWS Sign-In User Guide*.

- `downloads.deadlinecloud.amazonaws.com`
- `d2ev1rdnjzhmnr.cloudfront.net`
- `prod.log.shortbread.aws.dev`
- `prod.tools.shortbread.aws.dev`
- `prod.log.shortbread.analytics.console.aws.a2z.com`
- `prod.tools.shortbread.analytics.console.aws.a2z.com`
- `global.help-panel.docs.aws.a2z.com`
- `[Region].signin.aws`
- `[Region].signin.aws.amazon.com`
- `sso.[Region].amazonaws.com`
- `portal.sso.[Region].amazonaws.com`
- `oidc.[Region].amazonaws.com`
- `assets.sso-portal.[Region].amazonaws.com`

Environment-specific endpoints to allowlist

These domains vary depending on the specific configuration of Deadline Cloud. If additional Deadline Cloud monitors or queues are created, additional domains will need to be allowlisted.

- `[Directory ID or alias].awsapps.com`

This domain is tied to the IAM Identity Center setup and should be the same for all setups in this using the same IAM Identity Center instance. The exact value can be found by the enterprise admin in the IAM Identity Center console under *Settings* → *AWS access portal URL*.

- `[Monitor alias].[Region].deadlinecloud.amazonaws.com`

This domain is for the Monitor setup in Deadline Cloud. Artists enter this link into their browser or Deadline Cloud monitor application. If Deadline Cloud is set up in additional accounts or

regions in the future, this domain will change. You can find this value in the Deadline Cloud console in the *Dashboard* → *Monitor overview* → *Monitor details* → *URL*. The URL is reachable from any location on the internet. For information about what the URL exposes and how Deadline Cloud authorizes access to farm data, see [Access to the monitor web application](#).

- `[Bucket name].[Region].s3.amazonaws.com`

This is the domain for the job attachments bucket used by Deadline Cloud queues. Each queue can have its own job attachments bucket configured. The exact bucket name can be found in the Deadline Cloud console under *Queues* → *Queue details* → *Job attachments*. For more information about job attachments, see the queues documentation.

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the [aws:SourceArn](#) and [aws:SourceAccount](#) global condition context keys in resource policies to limit the permissions that AWS Deadline Cloud gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full Amazon Resource Name (ARN) of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:deadline:*:123456789012:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Deadline Cloud to prevent the confused deputy problem.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "deadline.amazonaws.com"
    },
    "Action": "deadline:CreateFarm",
    "Resource": [
      "*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:deadline:*:111122223333:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "111122223333"
      }
    }
  }
}
```

Compliance validation for Deadline Cloud

Third-party auditors assess the security and compliance of AWS Deadline Cloud as part of the AWS System and Organization Controls (SOC) compliance program.

AWS provides a frequently updated list of AWS services in scope of specific compliance programs at [AWS services in Scope by Compliance Program](#).

Third-party audit reports are available for you to download using AWS Artifact (AWS Artifact). For more information, see [Downloading Reports in AWS Artifact](#).

For more information about AWS compliance programs, see [AWS Compliance Programs](#).

The sensitivity of your data, your company's compliance objectives, and applicable laws and regulations determine your compliance responsibility when you use AWS services. If your use of Deadline Cloud is subject to compliance with standards such as SOC, AWS provides resources to help:

- [Security and Compliance Quick Start Guides](#) – Architectural considerations and steps for deploying security- and compliance-focused baseline environments on AWS.
- [AWS Compliance Resources](#) – Workbooks and guides that might apply to your industry and location.
- [AWS Config](#) – Assess how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub CSPM](#) – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in Deadline Cloud

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](#).

AWS Deadline Cloud does not back up data stored in your job attachments S3 bucket. You can enable backups of your job attachments data using any standard Amazon S3 backup mechanism, such as [S3 Versioning](#) or [AWS Backup](#).

Infrastructure security in Deadline Cloud

As a managed service, AWS Deadline Cloud is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see [AWS Cloud Security](#). To design your AWS environment using the best practices for infrastructure security, see [Infrastructure Protection](#) in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Deadline Cloud through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Deadline Cloud supports AWS PrivateLink VPC endpoint policies on its management and scheduling endpoints. An endpoint policy controls which Deadline Cloud API actions your VPC allows through the interface endpoint. For more information, see [Access AWS Deadline Cloud using an interface endpoint \(AWS PrivateLink\)](#).

Configuration and vulnerability analysis in Deadline Cloud

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- [Shared Responsibility Model](#)
- [Amazon Web Services: Overview of Security Processes](#) (whitepaper)

AWS Deadline Cloud manages tasks on service-managed or customer-managed fleets:

- For service-managed fleets, Deadline Cloud manages the guest operating system.
- For customer-managed fleets, you are responsible for managing the operating system.

For additional information about configuration and vulnerability analysis for AWS Deadline Cloud, see

- [Security best practices for Deadline Cloud](#)

Document history

For information about updates to AWS Deadline Cloud, see the [Deadline Cloud release notes](#).