aws

Developer Guide

Deadline Cloud

Deadline Cloud: Developer Guide

Table of Contents

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 in a wide range of 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 digital content creation artists and users to submit Deadline Cloud render jobs seamlessly from their workstations.
- Web and cloud-based service developers can integrate Deadline Cloud rendering into their platforms, enabling customers to provide assets to view products virtually.

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.

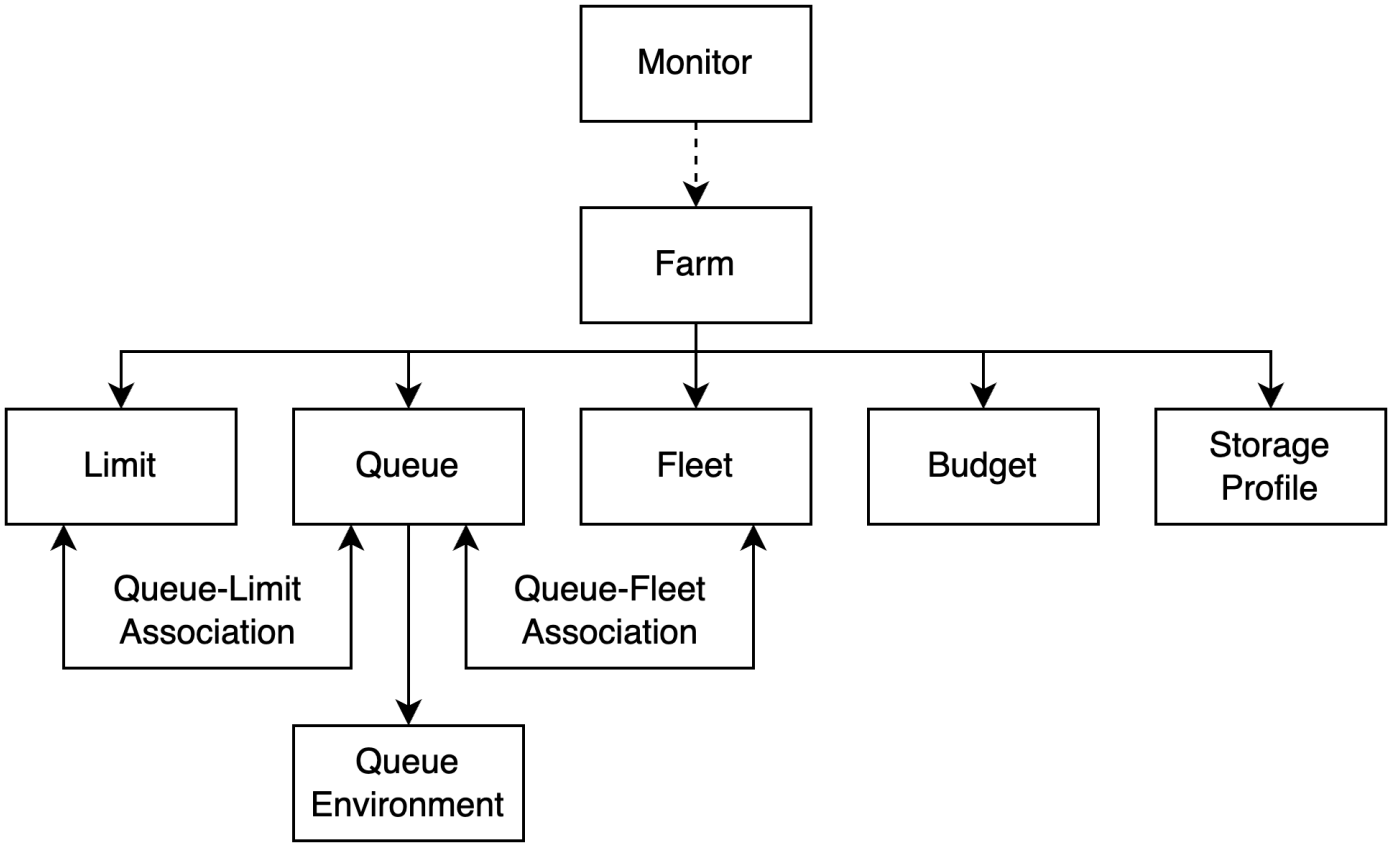
You can create an OpenJD job template using a submitter that Deadline Cloud provides, or you can use any tool that you want to create the template. After creating the template, you send it to Deadline Cloud. If you use a submitter, it takes care of sending the template. If you created the template another 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

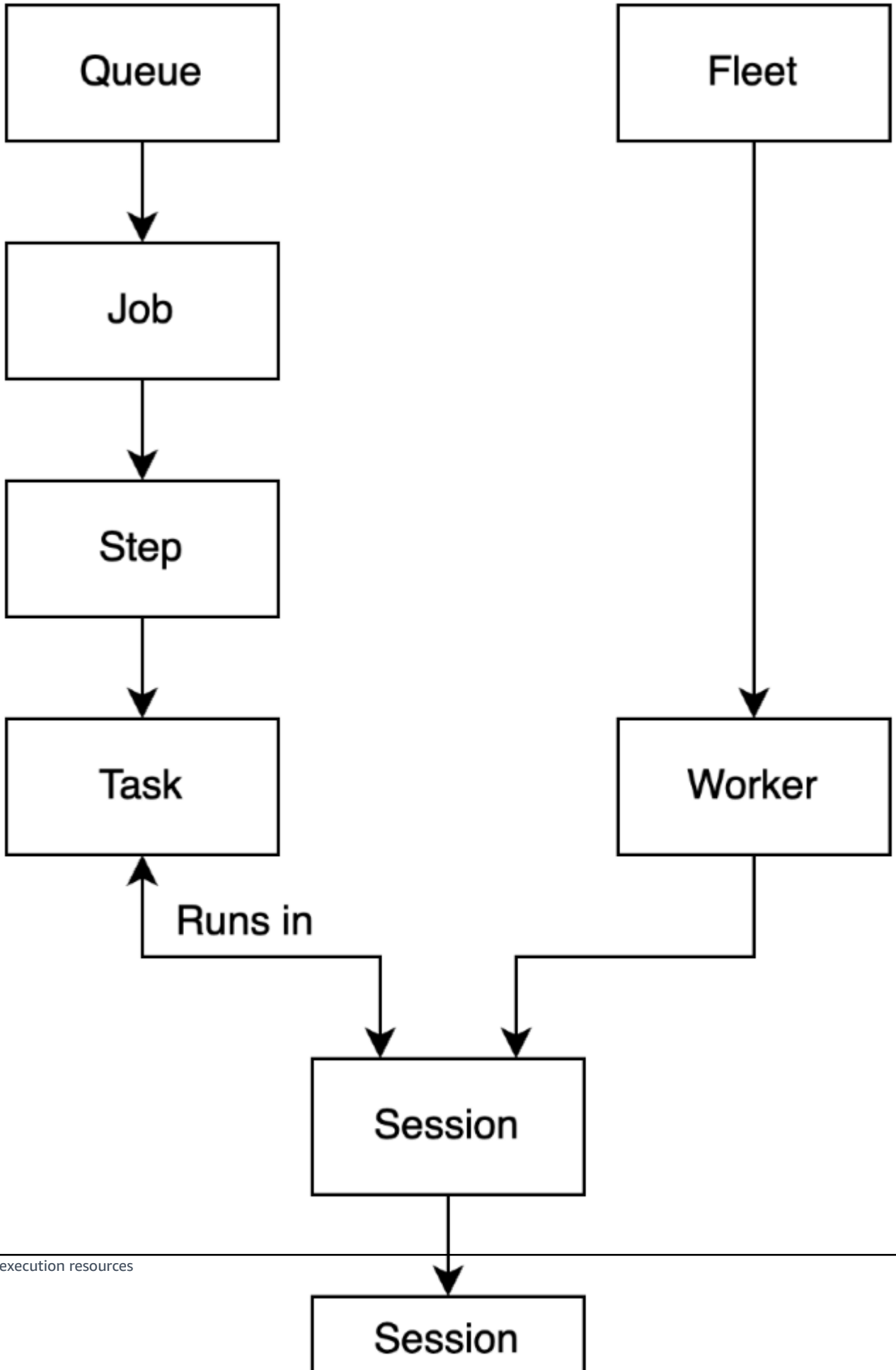
A limit allows you to track usage of shared resources such as floating licenses and control how they are allocated between jobs. Limits are associated with queues with queue-limit associations.

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.


Queue
Fleet
Job
Step
Task
Worker
Runs in
Session
Session

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 1 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 your 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, you will need to decide which approach is best for your use case. These are not strict scenario definitions and are not the only way to use Deadline Cloud. Instead these are 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. This could be asset generation in a retail pipeline, like a turntable video generated from a 3D model or scan. This could be 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. 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 github.

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 a tools to easily enable the use of conda packages. Conda is an open-source, cross-platform, language agnostic package manager and environment management system.

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 will most closely interoperate 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. An example queue environment can be found on AWS Deadline Cloud GitHub org.

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 per-seat, per-CPU or per-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. Keep in mind that 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. This 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. This 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.

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.

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 a wide range of 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, autoscaling 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.


Interactive Workflow
Automated Workflow
Integrated Submitter
Custom Job Definition
Conda Application Management
Custom Application Management
Deadline Cloud UBL
Custom Licensing
Job Attachments
Custom Storage
Deadline Cloud monitor
Custom Monitor Application
Automated Monitoring Solution

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 provides seamless scaling 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 seamlessly 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 leveraging cloud workstations alongside Deadline Cloud, studios can achieve a fully integrated, globally accessible production pipeline that scales seamlessly 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
Interactive Workflow
Automated Workflow
Integrated Submitter
Custom Job Definition
Conda Application Management
Custom Application Management
Deadline Cloud UBL
Custom Licensing
Job Attachments
Custom Storage
Deadline Cloud monitor
Custom Monitor Application
Automated Monitoring Solution

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 seamless 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

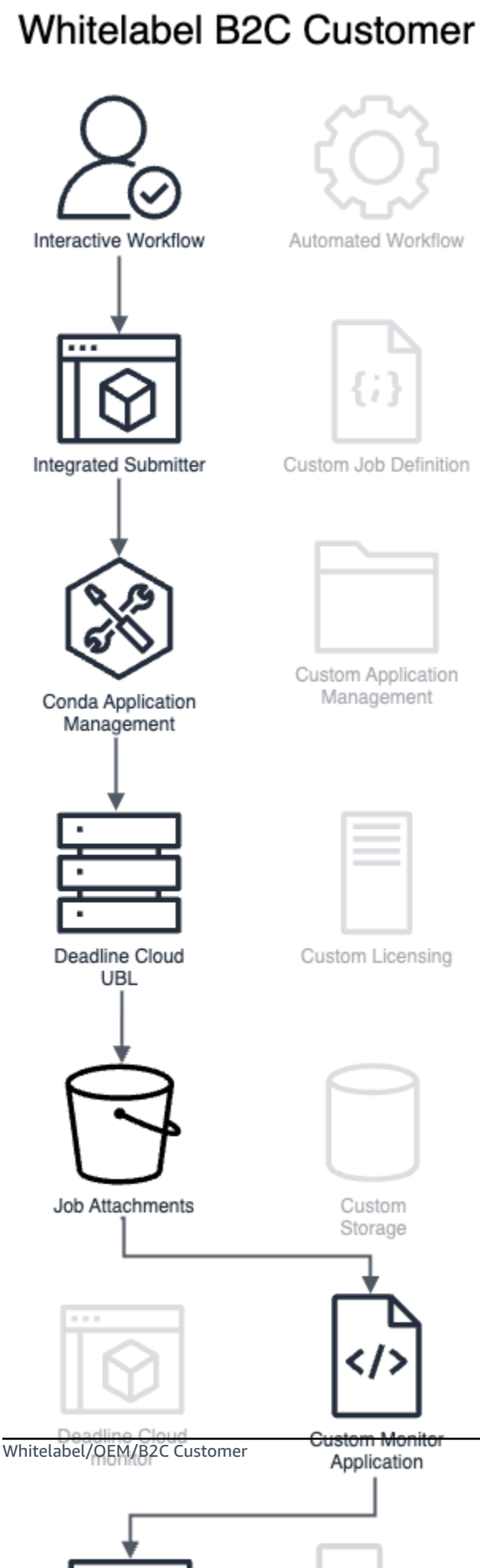
The shared fleet approach ensures optimal performance by maintaining a warm pool of workers, minimizing startup times while maximizing resource utilization across the user base. This configuration allows software vendors to offer cloud rendering as a seamless product feature rather than a separate service 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
Interactive Workflow
Automated Workflow
Integrated Submitter
Custom Job Definition
Conda Application Management
Custom Application Management
Deadline Cloud UBL
Custom Licensing
Job Attachments
Custom Storage
Deadline Cloud Monitor
Custom Monitor Application
Automated Monitoring Solution

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 crucial 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. 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 may 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 may 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. 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. This is 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. Some applications, like Blender, support a choice between relative and absolute paths.

- 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.

- 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. The Python documentation includes annotations for this, for example in the signal library documentation. 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 to provide a virtual environment for the job.

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

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`. This will make it available for 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

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_fileswa_gj55_/tmp2u9yqtsz
2024-03-27 16:00:22,406 INFO Wrote: runScript -> /sessions/
session-053d77cef82648fea9c698271812a/embedded_fileswa_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.

```
eaa2df5d34b54be5ac34c56a24a8c237b8487231a607eaf530a04d76b89c9cd3 /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

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:

```
eaa2df5d34b54be5ac34c56a24a8c237b8487231a607eaf530a04d76b89c9cd3  /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": 1698186190000000,
         "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-addressible 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:

```
eaa2df5d34b54be5ac34c56a24a8c237b8487231a607eaf530a04d76b89c9cd3  /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 an Open Job Description (OpenJD) job template and any asset files needed to render the job.

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 render farm. Job bundles are particularly useful for teams working on large-scale visual effects, animation, or other media rendering projects 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

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. You use the Deadline Cloud command-line interface (CLI) to use a job bundle to submit jobs for a queue to run.

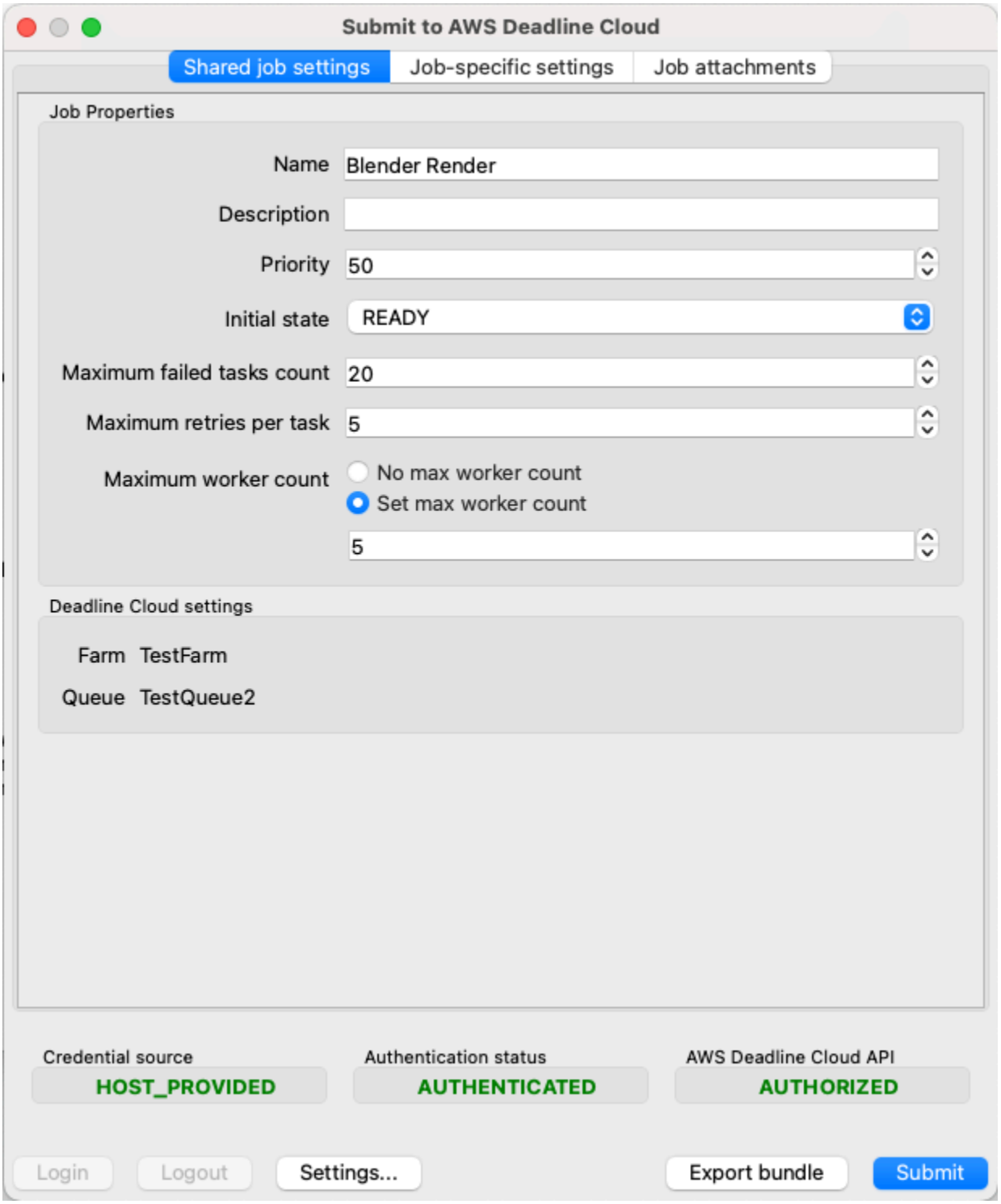
A job bundle is a directory structure that contains an OpenJD job template, other files that define the job, and job-specific files required as input for your job. You can specify the files that define your job as either YAML or JSON files.

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 GitHub. To run the sample using the following command in the Blender sample directory:

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



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

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 you 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

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 Environment template in the OpenJD GitHub repository.

For an introduction creating a job with an OpenJD job template, see Introduction to creating a job in the OpenJD GitHub repository. Additional information can be found in How jobs are run. There are job template samples in the in the OpenJD GitHub repository's `samples` directory.

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.

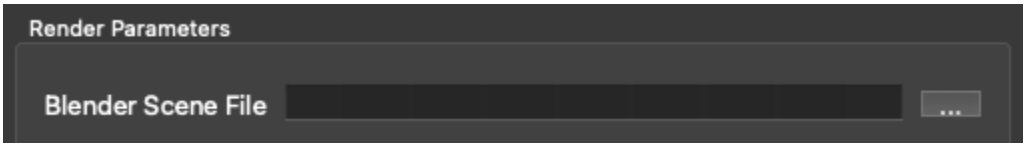
For example, the job template for the `blender_render` sample defines an input parameter `BlenderSceneFile` as a file path:

```
- 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.

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 `userInteface` element, see the gui_control_showcase sample in the deadline-cloud-samples repository on GitHub.

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.

For more information about the `objectType` and `dataFlow` properties, see JobPathParameterDefinition in the Open Job Description specification

The rest of the `blender_render` job template sample defines the job's workflow as a singe 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 has 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 How jobs are run in the Open Job Description specification.

There are more examples of job templates in the deadline-cloud-samples/job_bundles repository, as well as the template samples provided with the Open Job Descriptions specification.

Task chunking for job templates

Task chunking lets you group multiple tasks into a single unit of work called a chunk. In a render job, for example, this means Deadline Cloud can dispatch multiple frames together instead of one frame per command invocation. This reduces the overhead of starting applications for each task and shortens total job runtime. For details, see Running multiple frames at a time in the OpenJD wiki.

OpenJD supports extensions that add optional features to job templates. Task chunking is enabled by adding the `TASK_CHUNKING` extension. To use chunking, add the extension to your job template and use the `CHUNK[INT]` task parameter type. Submit chunked jobs using the same `deadline bundle submit` command. For example, the following job template renders frames in chunks of 10:

```
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 like `1-10`, `11-20`, and so on. The `{{Task.Param.Frame}}` variable expands to a range expression like `1-10`. Because `rangeConstraint` is set to `CONTIGUOUS`, the range is always in `start-end` format. The script parses this range and passes the start and end frames to Blender using the `-s` and `-e` options with `--render-anim`.

The `chunks` property supports the following fields:

- `defaultTaskCount` – (Required) How many tasks to combine into a single chunk. The maximum value is 150.
- `rangeConstraint` – (Required) If `CONTIGUOUS`, a chunk is always a contiguous range like `1-10`. If `NONCONTIGUOUS`, a chunk can be an arbitrary set like `1,3,7-10`.

- `targetRuntimeSeconds` – (Optional) The target runtime in seconds for each chunk. Deadline Cloud can dynamically adjust the chunk size to approach this target once some chunks have completed.

For more task chunking examples, including basic and Blender examples with both contiguous and non-contiguous chunks, see the task chunking samples in the Deadline Cloud samples repository on GitHub.

Customer-managed fleet requirements

Task chunking requires a compatible worker agent version. If you use customer-managed fleets, ensure your worker agents are updated before submitting jobs with chunking. Service-managed fleets always use a compatible worker agent version.

Downloading output for chunked jobs

When you download output for a single task in a chunked job, Deadline Cloud downloads the output for the entire chunk. For example, if frames 1-10 were processed together, downloading the output for frame 3 includes all frames 1-10. This feature requires `deadline-cloud` version 0.53.3 or later.

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 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.

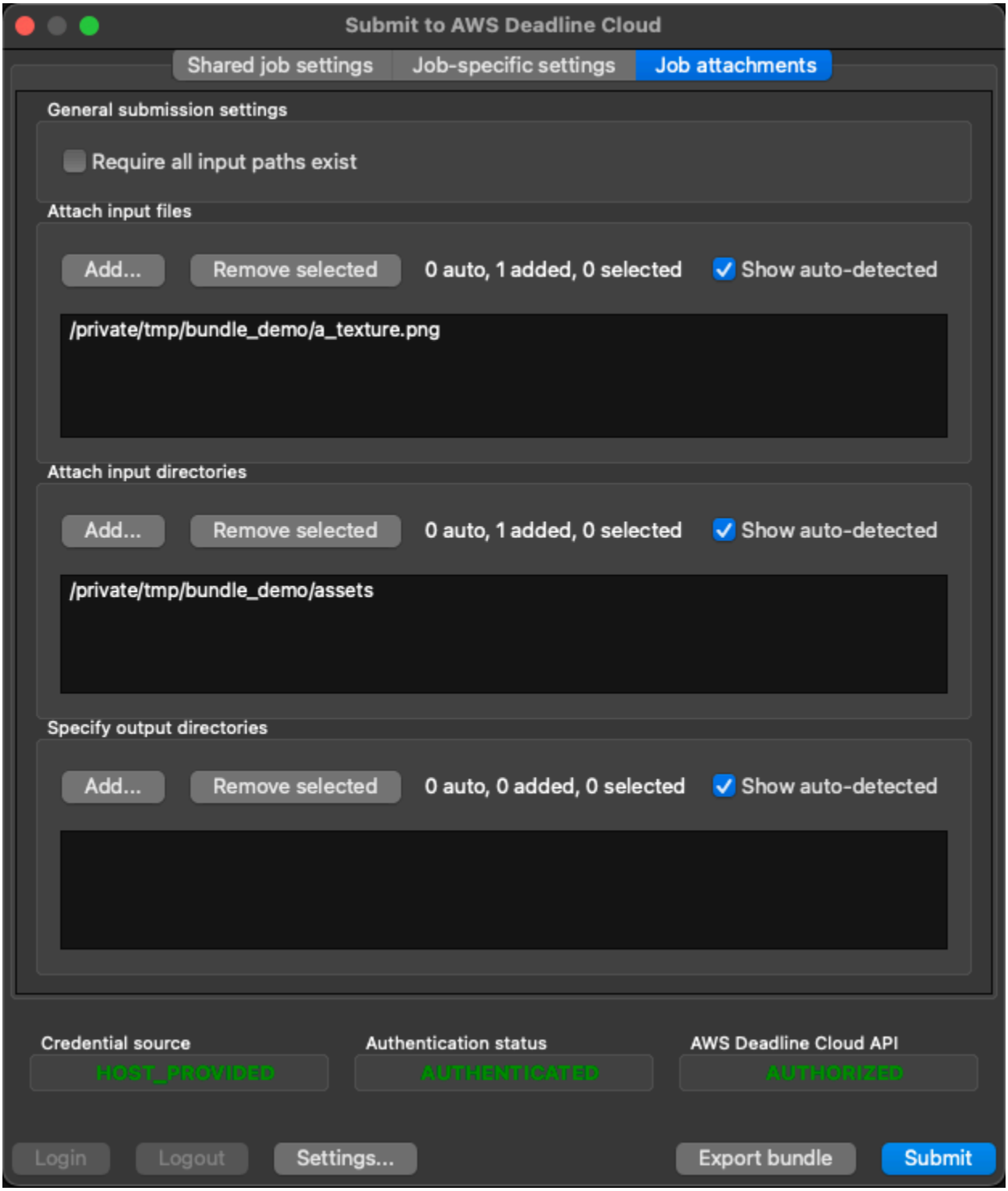
For more information about creating and using storage profiles, see Shared storage in Deadline Cloud in the *AWS Deadline Cloud User Guide*.

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

Use the following command to submit a job using the blender_render sample.

```
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

- 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:

- `WSAll` - 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

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 WSAll \
  --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 FSComm FS1

aws deadline update-queue --farm-id $FARM_ID --queue-id $QUEUE2_ID \
  --required-file-system-location-names-to-add FSComm 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 `WSAll` 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 `WSAll` 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
```

This is because 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 `WSAll` storage profile to queue `Q1` has the path mapping rules:

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

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

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

You can confirm the path mapping rules available to jobs submitted with a particular storage profile by submitting a job that prints out Open Job Description's path mapping rules file, and then reading the session log after the job has completed:

```
# 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. To submit jobs with the `WSAll` 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 `FSComm` 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 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 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

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 GitHub and the Deadline Cloud CLI to submit jobs.

Start by cloning the Deadline Cloud samples GitHub 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/87cb19095dd5d78fcaf56384ef0e6241.xxh128` – The contents of `script.sh`. The value `87cb19095dd5d78fcaf56384ef0e6241` 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 xxhash.
- `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 hexidecimal 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": "87cb19095dd5d78fcaf56384ef0e6241",
            "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 `87cb19095dd5d78fcaf56384ef0e6241`. This hash value matches the value in the object name `DeadlineCloud/Data/87cb19095dd5d78fcaf56384ef0e6241.xxh128`. It is used by Deadline Cloud to know which object to download for this file's contents.

The full schema for this file is available in GitHub.

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": "5b0db3d311805ea8de7787b64cbbe8b3",
                "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": "87cb19095dd5d78fcaf56384ef0e6241",
            "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. This is because 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 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 ~/devdemo-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-assetroot-3751a/shared/projects/project2/
file.txt
2024-07-17 01:26:38,269 INFO ./assetroot-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       "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-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 }
```

> **Note**
>
> If the job you submit has multiple manifests with different root paths, there is a different "assetroot"-named directory for each of the root paths.

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-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 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 ~/devdemo-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-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.xxh128
```

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 ~/devdemo-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-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.

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.

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. Both are available from PyPi and hosted on GitHub.

Topics

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 can 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 on GitHub.

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 of `show_job_bundle_submitter` function from the `deadline.client.ui.job_bundle_submitter`.

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 in `deadline.client.ui.dialogs.submit_job_to_deadline_dialog`. 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 powerful 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 an example, see Deadline Cloud for Maya integration on GitHub.

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 form 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.

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

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.

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.

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
```

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 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 section 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 on`Enter` actions for an environment.
- `taskRun` – Performs the on`Run` actions for a task.
- `envExit` – Performs the on`Exit` actions for an environment.

The following job template has a step environment. It has an on`Enter` definition to set up the step environment, an on`Run` definition that defines the task to run, and an on`Exit` 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 piplelining, 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
```

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.

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

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 log paths on the Microsoft website.
2. Install the Windows SDK for Desktop C++ x86 Apps. For more information, see Windows SDK in the Windows Dev Center.

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 `pythonservice.exe`.
4. Create a new file called `pythonservice.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="pythonservice" 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
 pythonservice.exe.manifest -outputresource:pythonservice.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 `pythonservice.exe.manifest` file and uninstall the SDK.

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.

The worker agent does not support running as a domain user on Windows. To run a job as a domain user, you can specify a domain user account when you configure a queue user for running jobs. For more information, see step 7 in Deadline Cloud queues in the *AWS Deadline Cloud User Guide*.

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. This should be applied to all `jobRunAsUser` accounts on the worker.

  ```
  sudo -u jobRunAsUser aws configure set default.region aws-region
  ```

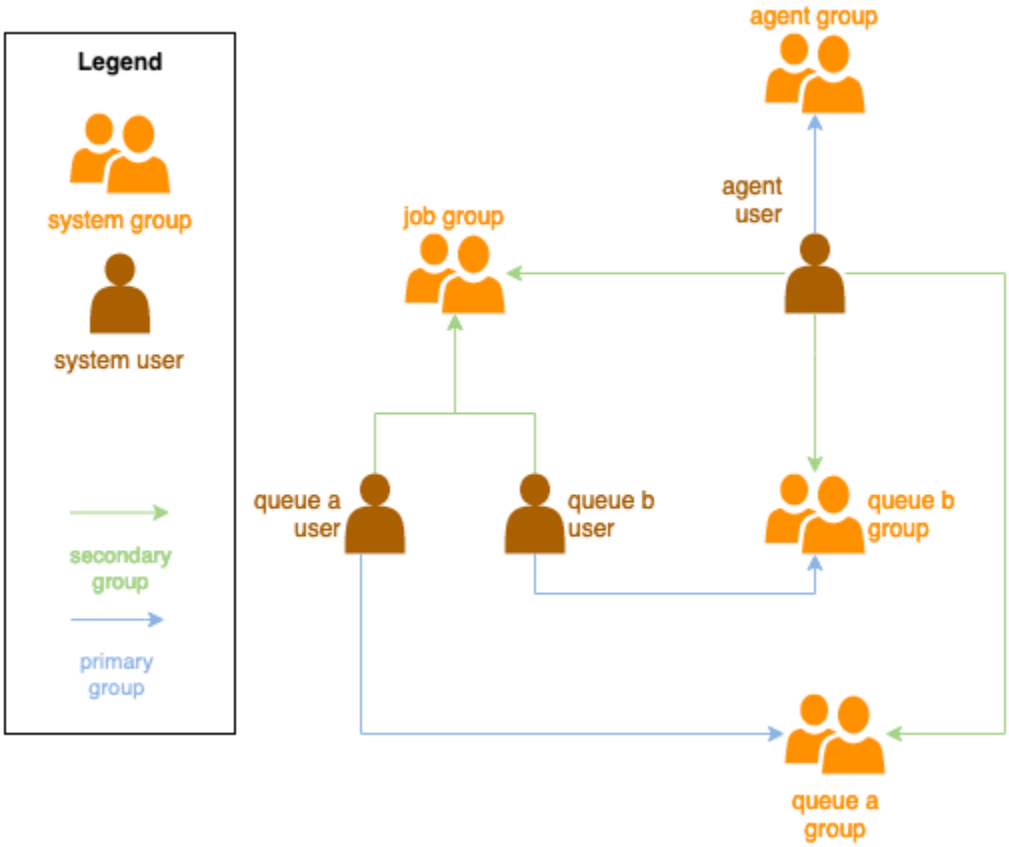
- The worker agent user must be able to run `sudo` commands as the `jobRunAsUser`. Run the following command to open an editor to create a new sudoers rule:

  ```
  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:deadline-worker /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. This 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 OpenJobDescription adaptors for using popular digital content creation (DCC) applications. 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 DCC adaptors on a customer-managed fleet

1. Open the 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.
2. Install the Deadline Cloud adaptor packages.

```
pip install deadline deadline-cloud-for-maya deadline-cloud-for-nuke deadline-
cloud-for-blender deadline-cloud-for-3ds-max
```

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 the 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.

Configure networking to allow AWS endpoint connections

Deadline Cloud requires secure connectivity to various AWS service endpoints for proper operation. To use Deadline Cloud, you must make sure that your network environment allows your Deadline Cloud workers to connect to these endpoints.

If you have a network firewall setup that blocks outbound connections, you may need to add firewall exceptions for specific endpoints. For Deadline Cloud, you must add exceptions for the following services:

- Deadline Cloud endpoints
- Amazon CloudWatch Logs endpoints
- Amazon Simple Storage Service endpoints

If your jobs use other AWS services, you may need to add exceptions for those services as well. You can find these endpoints in the Service endpoints and quotas chapter of the *AWS General Reference* guide. After you identify the required endpoints, create outbound rules in your firewall to permit traffic to these specific endpoints.

Making sure that these endpoints are accessible is required for proper operation. Additionally, consider implementing appropriate security measures, such as using virtual private clouds (VPCs), security groups, and network access control lists (ACLs) to maintain a secure environment while allowing the required Deadline Cloud traffic.

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 your 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. This 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 are required to launch instance using Auto Scaling in the subnets.

You should 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 leverages 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.

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-12345678900000000000000000000000",
                "fleetId": "fleet-12345678900000000000000000000000",
                "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 GitHub. 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.

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*.

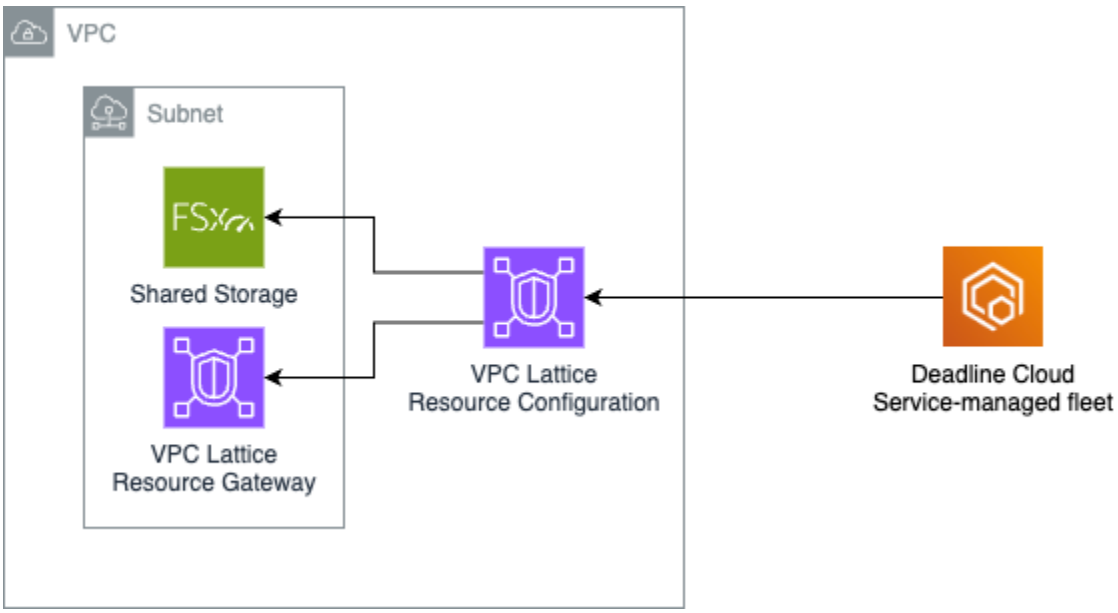
In most cases, you don't need to change an SMF to process your workloads. However, some situations may require you make changes to your fleets.

> **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

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 leverage Deadline Cloud's fully managed 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 GitHub.

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 on GitHub.
- 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*.

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	Queue environment	Host configuration script	Custom conda package
Administrator privileges required	No	Yes	No
When it runs	Session start	Worker startup	Session start
Scope	Per queue or job	All workers in fleet	Per queue or job
Can be controlled by job submission	Yes	No	Yes
Setup complexity	Low	Medium	High
Best for	Simple plugins, scripts, environment variables	System drivers, Docker, storage mounts	Complex applications with dependencies

Quick decision guide:

- *Need administrator or root privileges?* Use a host configuration script.
- *Simple plugin or script without admin rights?* Use a queue environment.
- *Complex application with version control needs?* Create a custom conda package.

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 GitHub.

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. 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

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.

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. 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.

```
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 GitHub.

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.

```
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_VERBOSITY: 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. 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.

```
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 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 github repository.

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.

```
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 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/sbin:/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 with robust IPC patterns on all supported operating systems.

To demonstrate this pattern, there is a self-contained sample job bundle 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 github repository.

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.

```
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 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

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_console_equivalent.yaml in the deadline-cloud-samples repository on GitHub.

If the 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. For the latest version of the queue environment

that loads the conda packages, see conda_queue_env_console_equivalent.yaml in the deadline-cloud-samples repository on GitHub.

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. For the latest version of the queue environment that loads the conda packages, see conda_queue_env_console_equivalent.yaml in the deadline-cloud-samples repository on GitHub.

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 GitHub.

Create a conda channel using S3

If your jobs need to run applications not available on the `deadline-cloud` or `conda-forge` channels, 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, 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

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 GitHub. 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. Install pixi from pixi.sh.
- **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 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 publish` 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, 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.

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 in the rattler-build documentation.

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.

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
   ```

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, then install `rattler-build`.

```
pixi global install rattler-build
```

- **git** – Required to clone the samples repository. On Windows, git for Windows 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 GitHub. 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, 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
```

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.

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 re-indexing the channel. First, install `rattler-index`.

```
pixi global install rattler-index
```

Then delete the package file and re-index 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
rattler-index s3 s3://amzn-s3-demo-bucket/Conda/Default
```

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 in the rattler-build documentation.

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 in the rattler-build documentation.

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 GitHub repository.

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 Packages in the conda documentation.

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. 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.
- 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 documents this convention 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 GitHub.

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 more information, see the Open Job Description Adaptor Runtime on GitHub. For example, deadline-cloud-for-maya on GitHub includes an integrated job submission GUI and a Maya adaptor that is available as the `maya-openjd` package on service-managed fleets.

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 using our samples such as the maya-openjd recipe on GitHub, you can build on the packages for Python and other dependencies provided by conda-forge. You might need to build the deadline and openjd-adaptor-runtime recipes 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 for multiple operating systems. The Blender 4.5 sample recipe in the Deadline Cloud samples repository on GitHub packages these archives into a conda package.

Understanding the recipe

The recipe.yaml file defines the package metadata, source URLs, and build options in rattler-build template syntax. 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 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 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.

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. 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 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. See Thin Client Licensing for Maya and MotionBuilder for more information.
- **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 recipe 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 for the full plugin integration convention.

Understanding the recipe

The recipe.yaml file defines the package metadata in rattler-build template syntax. 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 for Windows) 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 as documented by Autodesk so that the `ProductInformation.pit` file can be located within the conda environment rather than requiring system-level administrator access.
- **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 README.md 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. 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. 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 builds the adaptor from the deadline-cloud-for-maya source package published to PyPI. The recipe.yaml 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:

- deadline – The Deadline Cloud client library.
- openjd-adaptor-runtime – The Open Job Description adaptor runtime.

Python and other dependencies are available from conda-forge, 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 \
```

```
    -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 demonstrates how to package a plugin as a separate conda package that integrates with the host application package.

Understanding the recipe

The recipe.yaml specifies a dependency on the `maya` package for both build and run requirements. This 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 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. 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 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 repository on GitHub.
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 GitHub.

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 GitHub submits a build job for a conda recipe.

You need the following:

- The Deadline Cloud CLI installed on your workstation.
- An active AWS Deadline Cloud monitor (Deadline Cloud monitor) login session.
- A clone of the Deadline Cloud samples repository.

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.

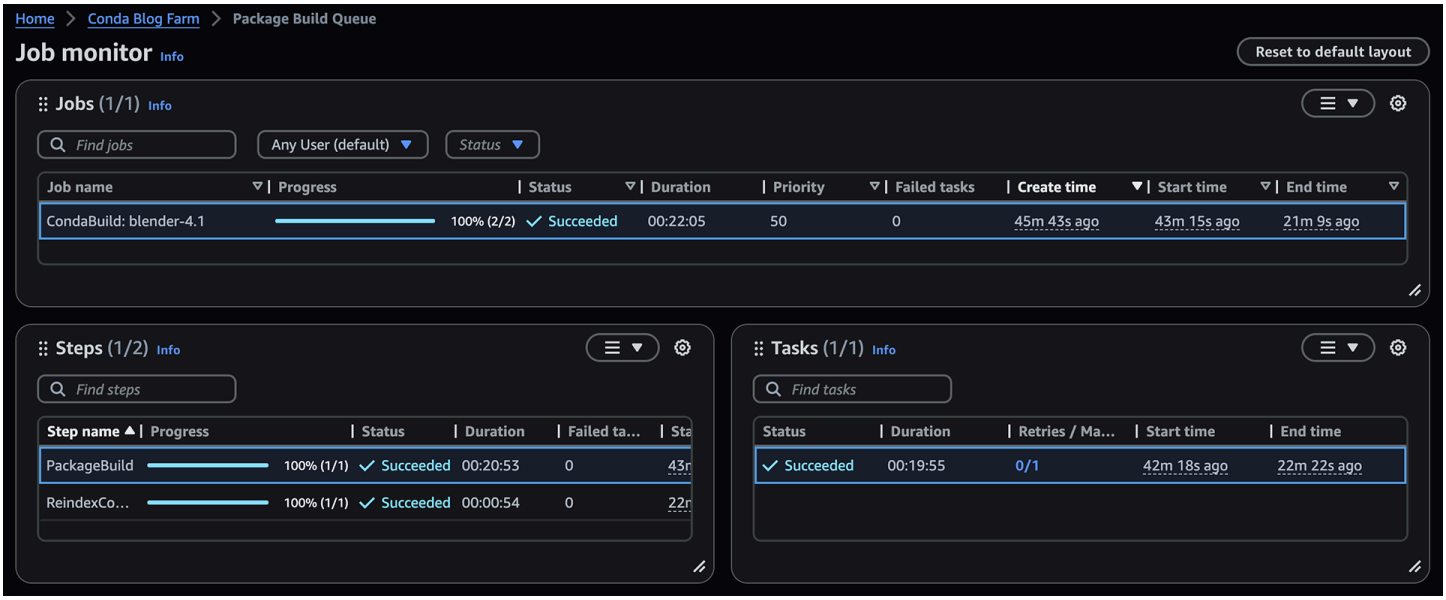
   ```
   cd deadline-cloud-samples/conda_recipes
   ```

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.



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.

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"}
}
}' \
--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

Deadline Cloud's worker agent publishes structured JSON logs to CloudWatch. The worker agent offers a wide array of structured logs for analyzing worker health. For more information, see Deadline Cloud worker agent logging on GitHub.

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

Deadline Cloud provides two methods of providing software licenses for your jobs:

- *Usage-based licensing (UBL)* – tracks and bills based on the number hours that your fleet uses processing a job. There are no set number of licenses so your fleet can scale as needed. UBL is standard for service-managed fleets. For customer-managed fleets you can connect an Deadline Cloud license endpoint for UBL. UBL provides licenses for your Deadline Cloud workers to render, it doesn't provide licenses for your DCC applications.
- Bring your own license (BYOL) – enables you to use existing software licenses with your service- or customer-managed fleets. You can use BYOL to connect to license servers for software not supported by the Deadline Cloud usage-based licenses. You can use BYOL with service-managed fleets by connecting to a custom license server.

Topics

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.

How you connect to a software license server depends on the configuration of your fleet and the requirements of the software vendor. Typically, you access the server in one of two ways:

- Directly to the license server. Your workers obtain a license from software vendor's license server using the Internet. All of your workers must be able to connect to the server.
- Through a license proxy. Your workers connect to a proxy server in your local network. Only the proxy server is allowed to connect to the vendor's license server over the Internet.

With the instructions below, you use Amazon EC2 Systems Manager (SSM) to forward ports from a worker instance to your license server or proxy instance. In the example below if your license server is unable to provide a license, Deadline Cloud's usage based licensing will be used. 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

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.

   ```
   lobal
       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 &ADC; 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. &ADC; SMF packages are
            installed from the "deadline-cloud" channel that is configured by
 &ADC;.

            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: !Sub |
        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.

The Deadline Cloud usage-based license server can be used with any fleet type as long as the Deadline Cloud workers can communicate with the license server. The license server is automatically set up in service-managed fleets. The following setup is only needed for customer-managed fleets.

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

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:

- Autodesk Maya and Arnold – 2701 - 2702, 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
- VRay – 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– Autodesk Maya and Arnold

> **Note**
>
> You can use Autodesk Maya and Arnold together or separately. Use port 2702 for Autodesk Maya and port 2701 for Arnold.

For Autodesk Maya, set the environment variable `ADSKFLEX_LICENSE_FILE` to:

```
2702@VPC_Endpoint_DNS_Name
```

For Arnold, set the environment variable `ADSKFLEX_LICENSE_FILE` to:

```
2701@VPC_Endpoint_DNS_Name
```

For Autodesk Maya and Arnold, set the environment variable `ADSKFLEX_LICENSE_FILE` to:

```
2702@VPC_Endpoint_DNS_Name:2701@VPC_Endpoint_DNS_Name
```

> **Note**
>
> For Windows workers, use a semi-colon (;) instead of a colon (:) to separate endpoints.

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– VRay

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 VRay 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.

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 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 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 on GitHub defines the schema for job templates. Reference this 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

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 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": "0",
    "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 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

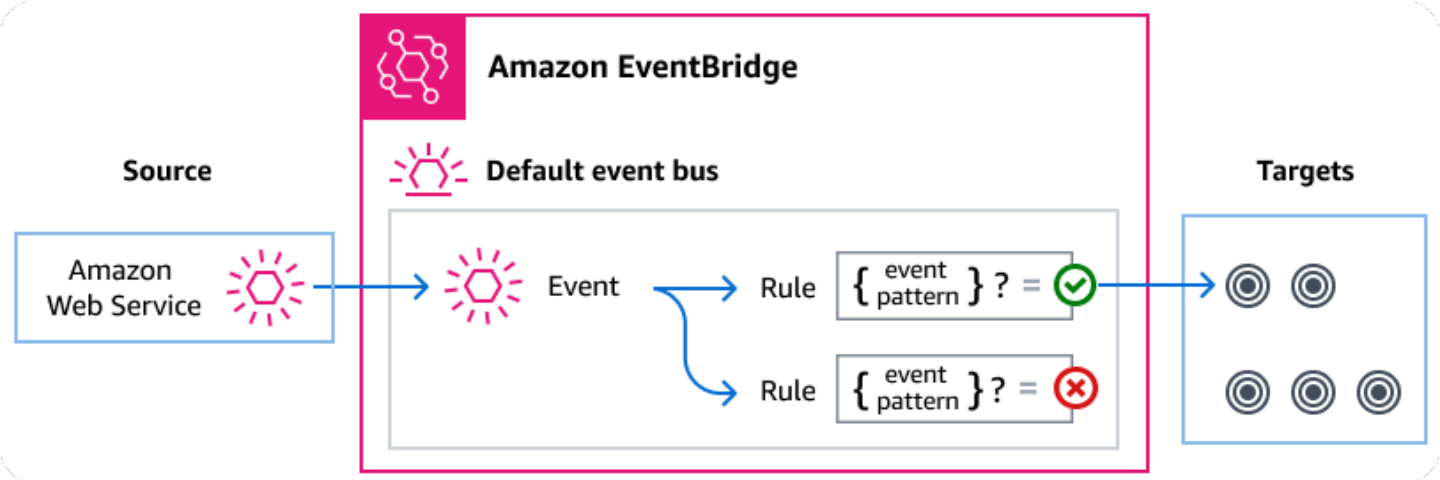
- Amazon CloudWatch User Guide
- Service Quotas User Guide

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-12345678900000000000000000000000"
  }
}
```

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

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 may 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 requeued, 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-12345678900000000000000000000000",
        "queueId": "queue-12345678900000000000000000000000",
        "jobId": "job-12345678900000000000000000000000",
        "stepId": "step-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 `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.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": [
        {
            "CreatedDate": "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

Security in Deadline Cloud

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS Deadline Cloud, see AWS services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Deadline Cloud. The following topics show you how to configure Deadline Cloud to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Deadline Cloud resources.

Topics

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 the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

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

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 polices
  - 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, this 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 you 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 informations, 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 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.

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": "AROAIGDTESTANDEXAMPLE:SampleUser01",
        "arn": "arn:aws::sts::111122223333:assumed-role/Admin/SampleUser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE3",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "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"
}
```

CloudTrail event for decryption

The following example CloudTrail event occurs when decrypting values using the customer managed KMS key.

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:SampleUser01",
        "arn": "arn:aws::sts::111122223333:assumed-role/SampleRole/SampleUser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws::iam::111122223333:role/SampleRole",
                "accountId": "111122223333",
                "userName": "SampleRole"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-04-23T18:46:51Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "deadline.amazonaws.com"
    },
    "eventTime": "2024-04-23T18:51:44Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "deadline.amazonaws.com",
    "userAgent": "deadline.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "aws:deadline:farmId": "farm-abcdef1234567890098765432 1fedcba",
            "aws:deadline:accountId": "111122223333",
            "aws-crypto-public-key": "AotL+SAMPLEVALUEiOMEXAMPLEaaqNOTREALaGTESTONLY
+p/5H+EuKd4Q=="
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT",
        "keyId": "arn:aws::kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
    },
    "responseElements": null,
    "requestID": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeffffff",
```

```
    "eventID": "ffffffff-eeee-dddd-cccc-bbbbbbaaaaaa",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws::kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

CloudTrail event for encryption

The following example CloudTrail event occurs when encrypting values using the customer managed KMS key.

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:SampleUser01",
        "arn": "arn:aws::sts::111122223333:assumed-role/SampleRole/SampleUser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws::iam::111122223333:role/SampleRole",
                "accountId": "111122223333",
                "userName": "SampleRole"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-04-23T18:46:51Z",
                "mfaAuthenticated": "false"
            }
        },
```

```
        "invokedBy": "deadline.amazonaws.com"
    },
    "eventTime": "2024-04-23T18:52:40Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "deadline.amazonaws.com",
    "userAgent": "deadline.amazonaws.com",
    "requestParameters": {
        "numberOfBytes": 32,
        "encryptionContext": {
            "aws:deadline:farmId": "farm-abcdef1234567890098765432 1fedcba",
            "aws:deadline:accountId": "111122223333",
            "aws-crypto-public-key": "AotL+SAMPLEVALUEiOMEXAMPLEaaqNOTREALaGTESTONLY
+p/5H+EuKd4Q=="
        },
        "keyId": "arn:aws::kms:us-
west-2:111122223333:key/abcdef12-3456-7890-0987-654321fedcba"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws::kms:us-west-2:111122223333:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE33333"
        }
    ],
    "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.

This is why 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`**.

  This will opt 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.

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

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

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 ensure that users and roles can still use the Deadline Cloud console, also attach the Deadline Cloud `ConsoleAccess` or `ReadOnly` AWS managed policy to the entities. For more information, 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"],
        "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": "ManagedDeadlineCloudIDCApplication",
        "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"
        ],
        "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/*"
        ]
    }]
}
```

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 `kms:Decrypt` 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 `deadline:TagResource` and `deadline:ListTagsForResource` to allow you to add and view tags associated with workers in your fleet.	May 30, 2025
AWSDeadlineCloud-UserAccessFarms – Change AWSDeadlineCloud-UserAccessJobs – Change AWSDeadlineCloud-UserAccessQueues – Change	Deadline Cloud added new actions `deadline:GetJobTemplate` and `deadline:ListJobParameterDefinitions` 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.

When you create monitors, fleets, and queues, you specify IAM roles that Deadline Cloud assumes on your behalf. Workers and the Deadline Cloud monitor then receive temporary credentials from these roles to access AWS services.

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.

Customer-managed fleet host role

Set up a WorkerHost role if you use customer-managed fleets on Amazon EC2 instances or on-premises hosts.

What the WorkerHost role does

The WorkerHost role 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 WorkerHost role permissions

Attach the following AWS managed policy to your WorkerHost 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 WorkerHost role is only used during initial worker startup:

1. The worker agent starts on the host using WorkerHost 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.

The WorkerHost role is not used after the worker starts running. This policy is not required for Service-managed fleets. In Service-managed fleets, bootstrapping is performed 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. Jobs use queue role credentials to write logs to their queue's log group. 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. This is 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. This capability enables the desktop application to provide enhanced 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 leverages Deadline Cloud's access management system 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 Deadline Cloud's access management system for additional

use cases - only users with access to the specific queue will receive these AWS CodeCommit permissions, allowing you to manage per-project repository access through Deadline Cloud queue membership.

This is useful for scenarios where 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-
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX --queue-id queue-XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   ```

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-XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX* and *queue-XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX* 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"
}
```

These credentials are automatically refreshed as needed, and Git operations will work seamlessly:

```
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, enabling fine-grained access control through Deadline Cloud's queue membership system.

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*.

Compliance validation for Deadline Cloud

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience in 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 doesn't support using AWS PrivateLink virtual private cloud (VPC) endpoint policies. It uses the AWS PrivateLink default policy, which grants full access to the endpoint. For more information, see Default endpoint policy in the *AWS PrivateLink user guide*.

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

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"
      }
    }
  }
}
```

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:

- Deadline Cloud uses AWS STS to authenticate workers so that they can access job assets. For more information about AWS STS, see Temporary security credentials in IAM in the *AWS Identity and Access Management User Guide*.
- 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.

You must also create an endpoint for AWS STS using the following service name:

```
com.amazonaws.region.sts
```

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

The Deadline Cloud submitter requires access to the following domains to download GUI dependencies.

- pypi.python.org
- pypi.org
- pythonhosted.org

- `files.pythonhosted.org`

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*.
- *`[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.

Security best practices for Deadline Cloud

AWS Deadline Cloud (Deadline Cloud) provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

> **Note**
>
> For more information about the importance of many security topics, see the Shared Responsibility Model.

Data protection

For data protection purposes, we recommend that you protect AWS account credentials and set up individual accounts with 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.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon Simple Storage Service (Amazon S3).
- If you require FIPS 140-2 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) 140-2.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This recommendation includes when you work with AWS Deadline Cloud or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Deadline Cloud or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

AWS Identity and Access Management permissions

Manage access to AWS resources using users, AWS Identity and Access Management (IAM) roles, and by granting the least privilege to users. Establish credential management policies and procedures for creating, distributing, rotating, and revoking AWS access credentials. For more information, see IAM Best Practices in the *IAM User Guide*.

Run jobs as users and groups

When using queue functionality in Deadline Cloud, it's a best practice to specify an operating system (OS) user and its primary group so that the OS user has least-privilege permissions for the queue's jobs.

When you specify a “Run as user” (and group), any processes for jobs submitted to the queue will be run using that OS user and will inherit that user’s associated OS permissions.

The fleet and queue configurations combine to establish a security posture. On the queue side, the “Job run as user” and IAM role can be specified to use the OS and AWS permissions for the queue’s jobs. The fleet defines the infrastructure (worker hosts, networks, mounted shared storage) that, when associated to a particular queue, run jobs within the queue. The data available on the worker hosts needs to be accessed by jobs from one or more associated queues. Specifying a user or group helps protect the data in jobs from other queues, other installed software, or other users with access to the worker hosts. When a queue is without a user, it runs as the agent user which can impersonate (`sudo`) any queue user. In this way, a queue without a user can escalate privileges to another queue.

Networking

To prevent traffic from being intercepted or redirected, it's essential to secure how and where your network traffic is routed.

We recommend that you secure your networking environment in the following ways:

- Secure Amazon Virtual Private Cloud (Amazon VPC) subnet route tables to control how IP layer traffic is routed.
- If you are using Amazon Route 53 (Route 53) as a DNS provider in your farm or workstation setup, secure access to the Route 53 API.
- If you connect to Deadline Cloud outside of AWS such as by using on-premises workstations or other data centers, secure any on-premises networking infrastructure. This includes DNS servers and route tables on routers, switches, and other networking devices.

Jobs and job data

Deadline Cloud jobs run within sessions on worker hosts. Each session runs one or more processes on the worker host, which generally require that you input data to produce output.

To secure this data, you can configure operating system users with queues. The worker agent uses the queue OS user to run session sub-processes. These sub-processes inherit the queue OS user's permissions.

We recommend that you follow best practices to secure access to the data these sub-processes access. For more information, see Shared responsibility model.

Farm structure

You can arrange Deadline Cloud fleets and queues many ways. However, there are security implications with certain arrangements.

A farm has one of the most secure boundaries because it can't share Deadline Cloud resources with other farms, including fleets, queues, and storage profiles. However, you can share external AWS resources within a farm, which compromises the security boundary.

You can also establish security boundaries between queues within the same farm using the appropriate configuration.

Follow these best practices to create secure queues in the same farm:

- Associate a fleet only with queues within the same security boundary. Note the following:
  - After job runs on the worker host, data may remain behind, such as in a temporary directory or the queue user's home directory.
  - The same OS user runs all the jobs on a service-owned fleet worker host, regardless of which queue you submit the job to.
  - A job might leave processes running on a worker host, making it possible for jobs from other queues to observe other running processes.
- Ensure that only queues within the same security boundary share an Amazon S3 bucket for job attachments.
- Ensure that only queues within the same security boundary share an OS user.
- Secure any other AWS resources that are integrated into the farm to the boundary.

Job attachment queues

Job attachments are associated with a queue, which uses your Amazon S3 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`.

If you create a queue and reuse an Amazon S3 bucket and root prefix, there is a risk of information being disclosed to unauthorized parties. For example, QueueA and QueueB share the same bucket and root prefix. In a secure workflow, ArtistA has access to QueueA but not QueueB. However, when multiple queues share a bucket, ArtistA can access the data in QueueB data because it uses the same bucket and root prefix as QueueA.

The console sets up queues that are secure by default. Ensure that the queues have a distinct combination of Amazon S3 bucket and root prefix unless they're part of a common security boundary.

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*.

Worker hosts

Secure worker hosts to help ensure that each user can only perform operations for their assigned role.

We recommend the following best practices to secure worker hosts:

- Using a *host configuration script* can change the security and operations of a worker. An incorrect configuration may cause the worker to be unstable or to stop working. It is your responsibility to debug such failures.
- 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.
- Grant queue users least-privileged OS permissions required for the intended queue workloads. Ensure that they don't have filesystem write permissions to work agent program files or other shared software.
- Ensure only the root user on Linux and the `Administrator` owns 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.
- Grant trusted individuals least-privileged access to worker hosts.
- 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 workstations and worker host operating systems.
- Restrict permissions to DNS configuration on workstations and worker host operating systems.
- 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`.

- Regularly rotate the passwords for your queue `jobRunAsUser`.
- Ensure least privilege access to the Windows password secrets and delete unused secrets.
- Don't give the queue `jobRunAsUser` permission the 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.

> **Note**
>
> For more information about the importance of regularly patching the operating system and installed software, see the Shared Responsibility Model.

Host configuration script

- Using a host configuration script can change the security and operations of a worker. An incorrect configuration may cause the worker to be unstable or to stop working. It is your responsibility to debug such failures.

Workstations

It's important to secure workstations with access to Deadline Cloud. This approach helps ensure that any jobs you submit to Deadline Cloud can't run arbitrary workloads billed to your AWS account.

We recommend the following best practice to secure artist workstations. For more information, see the Shared Responsibility Model.

- Secure any persisted credentials that provide access to AWS, including Deadline Cloud. For more information, see Managing access keys for IAM users in the *IAM User Guide*.
- Only install trusted, secure software.
- Require users federate with an identity provider to access AWS with temporary credentials.
- Use secure permissions on Deadline Cloud submitter program files to prevent tampering.
- Grant trusted individuals least-privileged access to artist workstations.
- Only use submitters and adaptors that you obtain through the Deadline Cloud Monitor.

- Restrict permissions to local DNS override configuration files (`/etc/hosts` on Linux and macOS, and `C:\Windows\system32\etc\hosts` on Windows), and to route tables on workstations and worker host operating systems.
- Restrict permissions to `/etc/resolve.conf` on workstations and worker host operating systems.
- 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.

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 /vfile
   ```

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
   ```

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+jKbsHQOQoEQdo5SrxxHjpKTEs3KQhGvf+ehrU1Ac7koXKIBWtes+
BI9F0slRECz0nXTOy/cd/90RXjpF07mreTLIKNIbybULfad82nYykpITjFr5XRGj
/shYkucxRQZdwkgkIYyV25pPICPd2RsX+Zua85jV8mCqVffDfRXvgcPe3+ofClj/
2CE8UfUIqO8Csua4YEkSqr3aeoTOEFT4kuQR5nFXVzorOEkQtO3gB35KNWKMlIOU
2vA+wyoL7nWSii4yfYtW3EZ+3gq6HxvnT9Zs8MC53uTOiOdamASXecYREwGmY/io
6n5XTEA/35LNbl4A756vSTZ7h4VFJAN5BpuqxstI1D7ou94skoSmcPoC/iniTvY9
kZylU5OCH/nifMAHM2a5jrQel80cW4oko9eyc8ENQpSy15JElFOKFf7D/4tcZJLF
F0VBTXbhfvq3dPfoq94IWt7p54Ovwj0S//CEu3jZYbNl2QC/3YiHE2H2XyGCQbq6
2MjcuxLnEapoRIqfbi8GPtCWVPzm28WGyKIDofWICczzeJFFJnvzrY3wRG64ibKJ
bR/uedwua1UuiC482V1FD5ffmzSSs8ktTp9hgj7RGDXlc9NTcF1jHxG9hwARAQAB
tCxBV1MgRGVhZGxpbmUgQ2xvdWQgPGF3cy1kZWFkbGluZUBhbWF6b24uY29tPokC
VwQTAQgAQRYhBJmXd7So2csyehiIYsg71N18bhtjBQJpQDQ1AhsvBQkDwmcABQsJ
CAcCAiICBhUKCQgLAgQWAgMBAh4HAheAAAoJEMg71N18bhtjk2UP/3h4KlEzZO/7
BxRmkbixuo1QuqOGvA6tXbSWaM8QH5jglcvL12PZLALklLT4v82uCsLR1lF8/Tch
cCl0SZEOFIS+XxAaw1Xfai6jlyLhabOwKF2ylq5eJlLcw1lh2nAArDRb4fLD0m1g
Dfqetq/XEpyXpOSkWxGRV4RlUdjQfytxrmcUnsT5/fk5f9VDdblu6K/lEmwfyYjB
lXv0uUCkqPot0SmbvOh3PY3Hi3n54ncy8NfTeV+TUvSe3C1s1zNl8aqHoTxJB/eU
kp+LFZ9m+igpSYnKeglKnytylH3KGCjTHglT/QXnI1wNTqmj1kFBVwtt/y1mtnA+
CPIUHP1CtbKsHaLtpp4llBm5TVtPN/Wqqicn5QLl4khg7R4K+V2aaA4ubY6p1tG9
0fFhN5tTnHDSKWMfmb83wfh5Zkcg85c3egjoit+wgGQRAQVqbznx7NqAHs9VoDIu
SPcAr+C329AOBzod4gyNGH7Ah5DkMITo4O4+axnAU9yhFOHcMJmTIask/fNg1Aum
OqYPMUwcgv1GZjLaTJyfGGC1xALsYR0KHnwIehD06MHR/Z98bGkcV8+Y0q8UPsd1
VN1fc1rjCJh/AT3w6owvG4DaEwspseSjzHv16mW4e2N6Uu23SPzgQsJ5qYN2g8D+
P7N9LGDfP8DaYc5JM9mlyFmYI2Q94ufL
=rY5l
-----END PGP PUBLIC KEY BLOCK-----
EOF
```

2. Determine whether to trust the OpenPGP key. Some factors to consider when deciding whether to trust the above key include the following:

   - The internet connection you've used to obtain the GPG key from this website is secure.
   - The device that you are accessing this website on is secure.
   - AWS has taken measures to secure the hosting of the OpenPGP public key on this website.

3. If you decide to trust the OpenPGP key, edit the key to trust with gpg similar to the following example:

```
$ gpg --edit-key 0xB840C08C29A90796A071FAA5F6CD3CE6B76F3CEF

   gpg (GnuPG) 2.0.22; Copyright (C) 2013 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  4096R/4BF0B8D2  created: 2023-06-23  expires: 2025-06-22  usage: SCEA
                        trust: unknown       validity: unknown
   [ unknown] (1). AWS Deadline Cloud example@example.com

   gpg> trust
   pub  4096R/4BF0B8D2  created: 2023-06-23  expires: 2025-06-22  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  4096R/4BF0B8D2  created: 2023-06-23  expires: 2025-06-22  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 Apr 1 21:10:14 2024 UTC

gpg: using RSA key B840C08C29A90796A071FAA5F6CD3CE6B7
```

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

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGX6GQsBEADduUtJgqSXI+q76O6fsFwEYKmbnlyL0xKvlq32EZuyv0otZo5L
le4m5Gg52AzrvPvDiUTLooAlvYeozaYyirIGsK08Ydz0Ftdjroiuh/mw9JSJDJRI
rnRn5yKet1JFezkjopA3pjsTBP6lW/mb1bDBDEwwwtH0x9lV7A03FJ9T7Uzu/qSh
qO/UYdkafro3cPASvkqgDt2tCvURfBcUCAjZVFcLZcVD5iwXacxvKsxxS/e7kuVV
I1+VGT8Hj8XzWYhjCZxOLZk/fvpYPMyEEujN0fYUp6RtMIXve0C9awwMCy5nBG2J
eE2Ol5DsCpTaBd4Fdr3LWcSs8JFA/YfP9auL3NczOozPoVJt+fw8CBlVIXO0J7l5
hvHDjcC+5v0wxqAlMG6+f/SX7CT8FXK+L3iOJ5gBYUNXqHSxUdv8kt76/KVmQa1B
Akl+MPKpMq+lhw++S3G/lXqwWaDNQbRRw7dSZHymQVXvPp1nsqc3hV7KlOM+6s6g
1g4mvFY4lf6DhptwZLWyQXU8rBQpojvQfiSmDFrFPWFi5BexesuVnkGIolQoklKx
AVUSdJPVEJCteyy7td4FPhBaSqT5vW3+ANbr9b/uoRYWJvn17dN0cc9HuRh/Ai+I
nkfECo2WUDLZ0fEKGjGyFX+todWvJXjvc5kmE9Ty5vJp+M9Vvb8jd6t+mwARAQAB
tCxBV1MgRGVhZGxpbmUgQ2xvdWQgPGF3cy1kZWFkbGluZUBhbWF6b24uY29tPokC
VwQTAQgAQRYhBLhAwIwpqQeWoHH6pfbNPOa3bzzvBQJl+hkLAxsvBAUJA8JnAAUL
CQgHAgIiAgYVCgkICwIDFgIBAh4HAheAAAoJEPbNPOa3bzzvKswQAJXzKSAY8sY8
F6Eas2oYwIDDdDurs8FiEnFghjUEO6MTt9AykF/jw+CQg2UzFtEyObHBymhgmhXE
3buVeom96tgM3ZDfZu+sxi5pGX6oAQnZ6riztN+VpkpQmLgwtMGpSMLl3KLwnv2k
WK8mrR/fPMkfdaewB7A6RIUYiW33GAL4KfMIs8/vIwIJw99NxHpZQVoU6dFpuDtE
1OuxGcCqGJ7mAmo6H/YawSNp2Ns80gyqIKYo7o3LJ+WRroIRlQyctq8gnR9JvYXX
42ASqLq5+OXKo4qh81blXKYqtc176BbbSNFjWnzIQgKDgNiHFZCdcOVgqDhwO15r
NICbqqwwNLj/Fr2kecYx180Ktpl0jOOw5IOyh3bf3MVGWnYRdjvA1v+/CO+55N4g
z0kf50Lcdu5RtqV10XBCifn28pecqPaSdYcssYSRl5DLiFktGbNzTGcZZwITTKQc
```

```
af8PPdTGtnnb6P+cdbW3bt9MVtN5/dgSHLThnS8MPEuNCtkTnpXshuVuBGgwBMdb
qUC+HjqvhZzbwns8dr5WI+6HWNBFgGANn6ageYl58vVp0UkuNP8wcWjRARciHXZx
ku6W2jPTHDWGNrBQO2Fx7fd2QYJheIPPAShHcfJO+xgWCof45D0vAxAJ8gGg9Eq+
gFWhsx4NSHn2gh1gDZ41Ou/4exJ1lwPM
=uVaX
-----END PGP PUBLIC KEY BLOCK-----
EOF
```

Linux (AppImage)

To verify packages that use a Linux .AppImage binary, first complete steps 1-3 in the Linux tab, then complete the following steps.

1. From the AppImageUpdate page in GitHub, download the **validate-x86_64.AppImage** 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 B840C08C29A90796A071FAA5F6CD3C 171200
```

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.

Document history

For information about updates to AWS Deadline Cloud, see the Deadline Cloud release notes.