Render Farm
Deployment Kit on AWS
Developer Guide
What is the Render Farm Deployment Kit on AWS?

The Render Farm Deployment Kit (RFDK) on AWS is an open-source software development kit that can be used to deploy, configure, and manage your render farm infrastructure in the cloud. The RFDK is built to operate with the AWS Cloud Development Kit (CDK) and provides a library of classes, called constructs, that each deploy and configure a component of your cloud-based render farm. The current version of the RFDK supports render farms built using AWS Thinkbox Deadline render management software, and provides the ability for you to easily go from nothing to a production-ready render farm in the cloud.

You can model, deploy, configure, and update your AWS render farm infrastructure by writing an application for the CDK toolkit using the libraries provided by the CDK and RFDK together and with other CDK-compatible libraries. The RFDK supports applications written in either Python or Node.js. Your application is written in an object-oriented style where creation of an object from the CDK and RFDK libraries represents the creation of a resource, or collection of resources, in your AWS account when the CDK toolkit deploys your application with AWS CloudFormation. The parameters of an object’s creation control the configuration of the resource.

Why use the RFDK?

With the RFDK, you can easily model the cloud components of your render farm as object-oriented code. This gives you the benefits of infrastructure as code:

• **Visibility**: Your render farm infrastructure is available as an easy-to-understand application that makes it easy for anyone on your team to see and understand what has been deployed.

• **Stability**: Combining infrastructure as code with version control, like git, makes accidental errors, like an incorrect setting, harder to make and easier to recover from.

• **Scalability**: Your application can be deployed repeatedly within the same region, in other regions, or even in other AWS accounts. This means that once you have modeled your render farm using the RFDK then you can create as many exact copies of that render farm as you need and be sure that they have all been created to your specifications.

• **Security**: The RFDK and CDK are built with security as a top priority so that your render farm is built on a secure foundation. Each component’s configuration can be customized to meet your organizations security requirements. If you create one well secured render farm using the RFDK then you can reuse it and know that every deployed version is meeting the same security requirements.

To use the RFDK to create the components of your cloud render farm, you will write simple and easy to understand code like in the example shown below. Creating an equivalent AWS CloudFormation template directly would require provisioning and correctly configuring 106 separate resources of 42 different types; the resulting template would be around three thousand lines long.

Python

```python
# A simple AWS CloudFormation stack that creates a bare-bones infrastructure with
```
Why use the RFDK?

AWS Thinkbox Deadline installed, configured, and ready to perform renders.

class BareBonesDeadlineRenderFarm(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # A Virtual Private Cloud (VPC) is a logically isolated section of the
        # AWS Cloud. To deploy a VPC, you create an instance of the CDK's Vpc
        # that uses two availability zones (AZs).
        vpc = ec2.Vpc(self, "Vpc", "Vpc", max_azs=2)

        # Specify version of AWS Thinkbox Deadline to use
        version = rfdk_deadline.VersionQuery(self, "Version",
            version="10.1.12",
        )

        # Use AWS Thinkbox published Deadline container images for specified Deadline
        version
        images = rfdk_deadline.ThinkboxDockerImages(self, 'Images',
            version=version,
            # Change this value to
            AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA if you wish to accept the
            EULA
            # for Deadline and proceed with Deadline deployment. Users must explicitly
            accept the AWS Thinkbox EULA before
            # using the AWS Thinkbox Deadline container images.
            #
            # See https://www.awsthinkbox.com/end-user-license-agreement for the terms
            of the agreement.
            user_aws_thinkbox_eula_acceptance=rfdk_deadline.AwsThinkboxEulaAcceptance.USER_REJECTS_AWS_THINKBOX_EULA,
        )

        # To operate Deadline you will need a backing-store for Deadline files
        # and scheduling data. You create one by creating an instance of the
        # RFDK's Repository. This will deploy an Amazon DocumentDB and
        # AWS Elastic File System (EFS), in private subnets, and run the
        # Deadline Repository installer to initialize them both.
        repository = rfdk_deadline.Repository(self, 'Repository',
            vpc=vpc,
            version=version,
            # Allow resources to be deleted when we delete the sample
            removal_policy=rfdk_deadline.RepositoryRemovalPolicies(
                database=core.RemovalPolicy.DESTROY,
                filesystem=core.RemovalPolicy.DESTROY
            )
        )

        # To create the server to which all Deadline client applications (like
        # the Worker or artist's Monitor) connect you create an instance
        # of the RFDK's RenderQueue. This will create an Amazon ECS, running
        # the Deadline Remote Connection Server (RCS), running behind behind
        # an Application Load Balancer. All Deadline client connections
        # are made with this load balancer.
        render_queue = rfdk_deadline.RenderQueue(self, 'RenderQueue',
            vpc=vpc,
            version=version,
            images=images,
            repository=repository,
            # Allow the load-balancer to be deleted when we delete the sample
            deletion_protection=False,
        )

        # To create a collection of Workers you create an instance of the
        # RFDK's WorkerInstanceFleet. This creates an AWS Auto Scaling Group,
        # in the VPC's private subnets, of EC2-Spot instances that are running
        # the Deadline Client.
Why use the RFDK?

Note: You must currently set the fleet's desired capacity manually.

Note2: You can create as many instances of WorkerInstanceFleet as you like.

```python
workers = rfdk_deadline.WorkerInstanceFleet(self, 'Workers',
                                          vpc=vpc,
                                          render_queue=render_queue,
                                          worker_machine_image=ec2.MachineImage.generic_linux({
                                              # Fill in your AMI id here
                                              f'core.Stack.of(self).region": "ami-00000000000000000"
                                          }),
                                          min_capacity=5,
                                          instance_type=ec2.InstanceType("c5.large"),
                                          spot_price=0.08)
```

You can create a filesystem to hold your render assets for the
workers in many ways. Here, to create an Amazon Elastic File
System (EFS) you create an instance of the CDK's FileSystem.

```python
asset_filesystem = efs.FileSystem(self, 'RenderAssets',
                                 vpc=vpc,
                                 encrypted=True,
                                 # Allow filesystem to be deleted when we delete the sample
                                 removal_policy=core.RemovalPolicy.DESTROY)
```

Finally, you mount that asset filesystem onto your Linux Workers
when they are launched by using the RFDK's MountableEfs helper-class.

```python
rfdk_core.MountableEfs(self,
                        filesystem=asset_filesystem
                        ).mount_to_linux_instance(workers.fleet, location="/mnt/assets")
```

TypeScript

```typescript
// A simple CloudFormation stack that creates a bare-bones infrastructure with
// AWS Thinkbox Deadline installed, configured, and ready to perform renders.
export class BareBonesDeadlineRenderFarm extends core.Stack {
    constructor(scope: core.Construct, id: string, props?: core.StackProps) {
        super(scope, id, props);

        // A Virtual Private Cloud (VPC) is a logically isolated section of the
        // AWS Cloud. To deploy a VPC, you create an instance of the CDK's Vpc
        // that uses two availability zones (AZs).
        const vpc = new ec2.Vpc(this, 'Vpc', { maxAzs: 2 });

        // Specify version of AWS Thinkbox Deadline to use
        const version = rfdkDeadline.VersionQuery(this, 'Version', {
            version: '10.1.12',
        });

        // Use AWS Thinkbox published Deadline container images for specified Deadline
        version
        const images = new rfdkDeadline.ThinkboxDockerImages(this, 'Images', {
            version,
            /**
             * Change this value to AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA
             * if you wish to accept the EULA for
             * Deadline and proceed with Deadline deployment. Users must explicitly accept
             * the AWS Thinkbox EULA before using the
             * AWS Thinkbox Deadline container images.
             * See https://www.awsthinkbox.com/end-user-license-agreement for the terms of
             * the agreement.
             */
            userAwsThinkboxEulaAcceptance: rfdkDeadline.AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA,
        });
```
Why use the RFDK?

// To operate Deadline you will need a backing-store for Deadline files
// and scheduling data. You create one by creating an instance of the
// RFDK's Repository. This will deploy an Amazon DocumentDB and
// AWS Elastic File System (EFS), in private subnets, and run the
// Deadline Repository installer to initialize them both.
const repository = new rfdkDeadline.Repository(this, 'Repository', {
  vpc,
  version,
  // Allow resources to be deleted when we delete the sample
  removalPolicy: {
    database: core.RemovalPolicy.DESTROY,
    filesystem: core.RemovalPolicy.DESTROY
  },
});

// To create the server to which all Deadline client applications (like
// the Worker or artist's Monitor) connect you create an instance
// of the RFDK's RenderQueue. This will create an Amazon ECS, running
// the Deadline Remote Connection Server (RCS), running behind behind
// an Application Load Balancer. All Deadline client connections
// are made with this load balancer.
const renderQueue = new rfdkDeadline.RenderQueue(this, 'RenderQueue', {
  vpc,
  version,
  images,
  repository,
  // Allow the load-balancer to be deleted when we delete the sample
  deletionProtection: false,
});

// To create a collection of Workers you create an instance of the
// RFDK's WorkerInstanceFleet. This creates an AWS Auto Scaling Group,
// in the VPC's private subnets, of EC2-Spot instances that are running
// the Deadline Client.
// Note: You must currently set the fleet's desired capacity manually.
// Note2: You can create as many instances of WorkerInstanceFleet as you like.
const workers = new rfdkDeadline.WorkerInstanceFleet(this, 'Workers', {
  vpc,
  renderQueue,
    // Fill in your AMI id here
    [core.Stack.of(this).region]: 'ami-00000000000000000',
  }),
  minCapacity: 5,
  instanceType: new ec2.InstanceType('c5.large'),
  spotPrice: 0.08,
});

// You can create a filesystem to hold your render assets for the
// workers in many ways. Here, to create an Amazon Elastic File
// System (EFS) you create an instance of the CDK's FileSystem.
const assetFilesystem = new efs.FileSystem(this, 'RenderAssets', {
  vpc,
  encrypted: true,
  // Allow filesystem to be deleted when we delete the sample
  removalPolicy: core.RemovalPolicy.DESTROY,
});

// Finally, you mount that asset filesystem onto your Linux Workers
// when they are launched by using the RFDK's MountableEfs helper-class.
const mountableEfs = new rfdk_core.MountableEfs(this, {
  filesystem: assetFilesystem,
});
mountableEfs.mountToLinuxInstance(workers.fleet, { location: '/mnt/assets' });
}
Where can I get the RFDK?

The RFDK is available today on:

- The official RFDK GitHub — https://github.com/aws/aws-rfdk — contributions welcome!
- pypi.org for Python developers to use via pip — https://pypi.org/project/aws-rfdk

Additional documentation and resources

- RFDK API Reference
- RFDK on GitHub
  - Issues
  - Example applications
- License
- Releases
- CDK Workshop
- CDK User Guide
- CDK API Reference
- CloudFormation Documentation
- Deadline Documentation

About Amazon Web Services

Amazon Web Services (AWS) is a collection of digital infrastructure services that developers can use when developing their applications. The services include computing, storage, database, and application synchronization (messaging and queuing).

AWS uses a pay-as-you-go service model. You are charged only for the services that you — or your applications — use. Also, to make AWS useful as a platform for prototyping and experimentation, AWS offers a free usage tier, in which services are free below a certain level of usage. For more information about AWS costs and the free usage tier, see Test-Driving AWS in the Free Usage Tier.

To obtain an AWS account, go to aws.amazon.com, and then choose Create an AWS Account.
Getting started with the RFDK

This topic introduces you to important RFDK concepts and describes how to install and configure the RFDK.

Topics in this chapter include:
- Prerequisites (p. 6)
- Onboarding to CDK (p. 6)
- Example development environment using Cloud9 and CodeCommit (p. 6)
- Your first RFDK app (p. 9)

Prerequisites

RFDK applications are written for the AWS CDK toolkit, so you first need to fulfill the AWS CDK prerequisites. This includes:

- Installing Node.js
- Providing AWS credentials and region (optionally installing AWS CLI)
- Installing the programming language of your choice
- Installing the AWS CDK Toolkit

Important
The CDK and the RFDK both require Node.js to be installed, no matter the language you work in. The minimal supported version of Node.js for RFDK may differ from the version required for CDK, so you need to use the greater of them.

Similarly, RFDK currently supports applications written in Python and TypeScript. If using Python, the minimal supported version of Python for RFDK may differ from the version required for CDK, so you need to use the greater of them.

RFDK provides Docker recipes for building container images for the RFDK server components such as the Deadline Render Queue and Deadline Usage Based Licensing. You will need to install Docker if you plan to use these components. RFDK requires Docker 17.05 or later. We also provide a walkthrough for setting up a example development environment (p. 6) with all the necessary prerequisites installed.

Onboarding to CDK

We recommend that you learn the CDK and the CDK deployment workflow with getting started with the AWS CDK. If you want to have a more in-depth tour that includes setting up your development environment and learning how to work with the CDK, then we recommend the official CDK Workshop.

You need to bootstrap your account using the CDK toolkit before you can deploy a CDK application into an AWS region. This needs to be done one time for each region that you want to deploy into using your account. Learn more about how to bootstrap your account from the official CDK documentation on bootstrapping your AWS environment.

Example development environment using Cloud9 and CodeCommit
This tutorial will guide you through setting up a example development environment for RFDK.

**Warning**
The development environment this creates is intended to serve as a starting point. This development environment is not suitable for all organizations and security policies. Please make sure you understand the implications of creating this environment.
Creating this example might result in charges to your AWS account. These include possible charges for services such as Amazon EC2 and CodeCommit. For more information, see Amazon EC2 Pricing and AWS CodeCommit Pricing.

You will:
1. the section called “Step 1: Create an AWS CodeCommit repository” (p. 7)
2. the section called “Step 2: Create an AWS Cloud9 environment” (p. 7)
3. the section called “Step 3: Encrypt Cloud9 EBS volume” (p. 8)
4. the section called “Step 4: Increase storage capacity (optional)” (p. 8)
5. the section called “Step 5: Configure Cloud9 for use with CodeCommit” (p. 8)
6. the section called “Step 6: Clone the AWS CodeCommit repository” (p. 9)

**Prerequisites**

Before beginning this tutorial, you will need to:

- Create an AWS account
- Create an IAM user with least-privilege that has the ability to:
  - Use AWS CodeCommit - The AWSCodeCommitPowerUser AWS Managed Policy can serve as a starting point, but it is recommended to use a minimally permissive IAM policy for production environments.
  - List and resize EBS volumes - This is granted with the ec2:DescribeVolumes and ec2:ModifyVolume actions. These can be granted in an inline policy statement.
  - Deploy your CDK application - Permissions vary depending on the code in the CDK application, and ultimately the CloudFormation resources you create/update/destroy. The RFDK makes use of CDK's grant feature which effectively requires administrator access. The AdministratorAccess AWS Managed Policy can be used. The credentials for this IAM user should be secured and managed carefully. If these credentials are compromised, your AWS account could be mis-used without limitation.
  - Onboard to CDK (p. 6)

**Step 1: Create an AWS CodeCommit repository**

First, we need a highly-available and secure code repository for managing the source code of our RFDK application. For this development environment, we will use an AWS CodeCommit repository.

Follow the AWS CodeCommit's documentation to create an AWS CodeCommit repository. Take note of the repository name chosen for the remainder of this guide.

**Step 2: Create an AWS Cloud9 environment**

Next, we will create an AWS Cloud9 environment that we will use for development. Specifically, we will create an EC2 Cloud9 environment. This type of Cloud9 environment means that Cloud9 will manage the lifecycle of the EC2 instance that hosts Cloud9. The instance is automatically started when opening the Cloud9 IDE and stopped when the IDE is idle or unused to save costs.
Follow the AWS Cloud9 documentation for Creating an environment. Use the following values when creating the Cloud9 environment:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>A desired name for your environment</td>
</tr>
<tr>
<td>Description</td>
<td>An optional description for your environment</td>
</tr>
<tr>
<td>Environment type</td>
<td>Create a new no-ingress EC2 instance for environment (access via Systems Manager)</td>
</tr>
<tr>
<td>Instance type</td>
<td>t3.small</td>
</tr>
<tr>
<td>Platform</td>
<td>Amazon Linux 2</td>
</tr>
<tr>
<td>Cost-saving setting</td>
<td>After four hours</td>
</tr>
</tbody>
</table>

Once you’ve created the environment, the browser will navigate to the Cloud9 IDE for the newly created environment. The Cloud9 IDE connects to the EC2 instance that hosts the environment and can be used to edit files and create terminal sessions. Now is a good time to familiarize yourself with the Cloud9 IDE.

One important user interface element is the console at the bottom of the screen which will appear as a tab named bash - *ip-... This panel provides an interactive terminal session where you can run arbitrary shell commands. You can toggle it being open or closed with the F6 keyboard shortcut.

By default, Cloud9 EC2 environments are created with all of the required the section called “Prerequisites” (p. 6) for working with RFDK.

**Step 3: Encrypt Cloud9 EBS volume**

Follow the instructions to encrypt the EBS volume used by the Cloud9 EC2 instance.

**Step 4: Increase storage capacity (optional)**

**Note**
This step is only required if you plan to stage and build Deadline container images. See the section called “Deadline Container Images” (p. 23) for more details. If you do not have a specific need to build your own container images, you can skip this step.

By default, AWS Cloud9 creates a 10GB EBS volume for the root file-system of the EC2 instance hosting the Cloud9 environment. RFDK requires more storage for staging and building the Deadline container images. Follow the instructions to Resize an Amazon EBS volume used by an environment. For RFDK development, we recommend a minimum size of 40GB.

**Step 5: Configure Cloud9 for use with CodeCommit**

In this step, we configure git on our Cloud9 environment to use the AWS CLI credential helper. This uses Cloud9’s managed temporary credentials that are associated with the IAM user connected to the IDE. To configure this, run the following commands:

```bash
git config --global credential.helper '!aws codecommit credential-helper $@'
git config --global credential.UseHttpPath true
```
Step 6: Clone the AWS CodeCommit repository

In this step, we clone the CodeCommit repo created in the section called “Step 1: Create an AWS CodeCommit repository” (p. 7) into your Cloud9 environment.

1. From the Cloud9 IDE, ensure the console panel is open. If not, press the F6 key to toggle it.
2. Run the following commands substituting my_repo_name with the CodeCommit repository name created in step 1.

```
# Replace the value on the right-hand side with repository name from step 1
CC_REPO_NAME=my_repo_name

aws codecommit get-repository \
  --repository-name ${CC_REPO_NAME} \
  --query repositoryMetadata.cloneUrlHttp \
  --output text \n  | xargs git clone
```

Next steps

Now that you have a example RFDK development environment, you are ready to Build your first RFDK app (p. 9). You can also review the following additional resources:

- Learn more about git
- Read the AWS Cloud9 User Guide

Your first RFDK app

Now that you have installed the the section called “Prerequisites” (p. 6) and have completed the section called “Onboarding to CDK” (p. 6), let’s get started on your first RFDK app — a simple render farm using AWS Thinkbox Deadline. In this tutorial, you will learn how to:

1. Initialize an RFDK app.
2. Stage the Docker recipes used for various Deadline components.
3. Add code that defines a simple Deadline render farm.
4. Build and deploy your RFDK render farm to your AWS account.

Your RFDK render farm will be deployed as a single CloudFormation stack that will contain:

- A VPC
- The back-end of the farm
  - Database and file system — Deadline Repository
  - Central service — Deadline Render Queue
- The render nodes — Deadline Worker Fleet
**Estimated time: 75-90 minutes** Approximately 60 minutes of this will be waiting for AWS CloudFormation to deploy and destroy your resources.

**Important**
This guide is written assuming you are working on an EC2 instance running on Linux, but you can also use your local machine if desired. It also assumes that you have installed the section called “Prerequisites” (p. 6) and have completed the section called “Onboarding to CDK” (p. 6) on this instance.

### Initialize the RFDK app

You can create your RFDK app anywhere on your machine, but each RFDK app should be in its own directory. Create a new directory for your app and enter it:

```
mkdir hello-rfdk
cd hello-rfdk
```

First, you must determine the latest version of RFDK available. The following command uses npm (bundled with Node.js) to look-up the latest version of the `aws-rfdk` package, store it in the `RFDK_VERSION` shell variable, and output a message indicating the version:

```
RFDK_VERSION=$(npm view aws-rfdk version)
echo "Using RFDK version ${RFDK_VERSION}"
```

Next, find the version of the AWS CDK that is compatible with this version of RFDK:

```
CDK_VERSION=$(npm view aws-rfdk 'dependencies.@aws-cdk/core')
echo "Using CDK version ${CDK_VERSION}"  
```

**Note**
It is important that the version of the `aws-cdk` packages installed in your project match the version of `aws-cdk` packages required by the RFDK version used in your app.

Now, initialize the app using the CDK toolkit’s `cdk init` command:

**Python**

```
npx cdk@${CDK_VERSION} init app --language python
```

**TypeScript**

```
npx cdk@${CDK_VERSION} init app --language typescript
```

With the CDK project initialized, install the `aws-rfdk` package:

**Python**

Modify `setup.py` in the root of the app directory to add an entry for `aws-rfdk` to the `install_requires` list. Replace `RFDK_VERSION` with the version output in the previous step:

```
ssetup(
    # ...
    install_requires=[
    # ...
```
Now activate the app's Python virtual environment and install its dependencies:

```
source .env/bin/activate
python -m pip install -r requirements.txt
```

TypeScript

```
npm install --save aws-rfdk@${RFDK_VERSION}
```

You may see a list of warnings after installing `aws-rfdk` that state there are missing peer dependencies. For example:

```
npm WARN @aws-cdk/aws-apigateway@1.57.0 requires a peer of @aws-cdk/core@1.57.0 but none is installed. You must install peer dependencies yourself.
```

To install all of the peer dependencies required by RFDK, use the following command (requires `jq`):

```
npm view --json aws-rfdk@${RFDK_VERSION} peerDependencies | jq '. | to_entries[] | .key + "@" + .value' | xargs npm i --save
```

**Setup the environment**

Each application needs to be associated with an **AWS environment**: the target Account and Region into which the stack is intended to be deployed. You can specify the exact values for Account and Region or use the environment variables as in the example below:

**Python**

```
# ... import os
env = {
    'account': os.environ.get("CDK_DEPLOY_ACCOUNT", os.environ["CDK_DEFAULT_ACCOUNT"]),
    'region': os.environ.get("CDK_DEPLOY_REGION", os.environ["CDK_DEFAULT_REGION"])
}

app = core.App()
HelloRfdkStack(app, "hello-rfdk", env=env)
```

**TypeScript**

```
new HelloRfdkStack(app, 'hello-rfdk', {
    env: {
        account: process.env.CDK_DEPLOY_ACCOUNT ?? process.env.CDK_DEFAULT_ACCOUNT,
        region: process.env.CDK_DEPLOY_REGION ?? process.env.CDK_DEFAULT_REGION,
    }
})
```
Define a Deadline render farm

Now you are ready to start building your render farm. The first thing you will need is a Vpc construct instance for your render farm. The Vpc provides the foundational networking that will be used by all other components in the farm.

Python

```python
# In hello_rfdk/hello_rfdk_stack.py:

import aws_cdk.core as core
import aws_cdk.aws_ec2 as ec2

class HelloRfdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        vpc = ec2.Vpc(self, "Vpc")
```

TypeScript

```typescript
# In lib/hello-rfdk-stack.ts:

import * as cdk from '@aws-cdk/core';
import * as ec2 from '@aws-cdk/aws-ec2';

export class HelloRfdkStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);

        const vpc = new ec2.Vpc(this, 'Vpc');
    }
}
```

The next thing you will need to do is select a version of AWS Thinkbox Deadline to use for your Render Farm. For more details, see the full documentation about the section called “Using AWS Thinkbox ECR Repositories” (p. 24). Once you have selected a Deadline version (DEADLINE_VERSION), create a VersionQuery construct in your CDK app.

Python

```python
# In hello_rfdk/hello_rfdk_stack.py:

# ...
import aws_rfdk.deadline as rfdk_deadline

class HelloRfdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        # ...

        version = rfdk_deadline.VersionQuery(self, "Version",
                                             version="10.1.12",
```
TypeScript

In `lib/hello-rfdk-stack.ts`:

```typescript
// ...
import * as deadline from 'aws-rfdk/deadline';

export class HelloRfdkStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
    // ...
    // Pin to the 10.1.12.x Deadline release. Release patches are applied with each CDK deployment
    const version = new deadline.VersionQuery(this, 'Version', {
      version: '10.1.12',
    });
  }
}
```

Next, let's add a Repository. This construct creates the database and file system that make up the back-end storage of your render farm. Then, it configures them with the Deadline Repository installer. By default, an Amazon DocumentDB and Amazon Elastic File System (EFS) are created.

**Tip**

In the Deadline documentation, the Database and Repository are two separate concepts. The RFDK combines the two concepts and calls it the Repository.

Python

In `hello_rfdk/hello_rfdk_stack.py`:

```python
# ...
class HelloRfdkStack(core.Stack):
  def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
    # ...
    repository = rfdk_deadline.Repository(self, 'Repository',
                                           vpc=vpc,
                                           version=version,
                                           )
```

TypeScript

In `lib/hello-rfdk-stack.ts`:

```typescript
// ...
export class HelloRfdkStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
    // ...
    const repository = new deadline.Repository(this, 'Repository', {
      vpc,
      version,
    });
  }
```
AWS Thinkbox publishes Deadline container images into a publicly-available Elastic Container Registry (ECR) Repository. RFDK provides the ThinkboxDockerImages construct that can be used to deploy these container images using AWS Elastic Container Service (ECS).

To use these images, add a ThinkboxDockerImages instance to your CDK app. For this, you will need to read and accept the terms of the AWS Thinkbox End User License Agreement.

Python

In hello_rfdk/hello_rfdk_stack.py:

```python
# ...
class HelloRfdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        # ...
        images = rfdk_deadline.ThinkboxDockerImages(self, 'Images',
                                                   version=version,
                                                   # Change this value to
                                                   AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA if you wish to accept the EULA
                                                   # for Deadline and proceed with Deadline deployment. Users must explicitly accept
                                                   the AWS Thinkbox EULA before
                                                   # using the AWS Thinkbox Deadline container images.
                                                   #
                                                   # See https://www.awsthinkbox.com/end-user-license-agreement for the terms
                                                   of the agreement.
                                                   user_aws_thinkbox_eula_acceptance=rfdk_deadline.AwsThinkboxEulaAcceptance.USER_REJECTS_AWS_THINKBOX_EULA
                                                   )
```

TypeScript

In lib/hello-rfdk-stack.ts:

```typescript
// ...
export class HelloRfdkStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);
        // ...
        const images = new deadline.ThinkboxDockerImages(this, 'Images', {
            version,
            /**
             * Change this value to AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA
             * if you wish to accept the EULA for
             * Deadline and proceed with Deadline deployment. Users must explicitly accept
             * the AWS Thinkbox EULA before using the
             * AWS Thinkbox Deadline container images.
             *
             * See https://www.awsthinkbox.com/end-user-license-agreement for the terms
             * of the agreement.
             */
            userAwsThinkboxEulaAcceptance:
              deadline.AwsThinkboxEulaAcceptance.USER_REJECTS_AWS_THINKBOX_EULA,
        });
    }
```
Now that you have Deadline container images and a Deadline Repository, you will need to add a `RenderQueue`. The `RenderQueue` acts as the central service of your render farm that clients and render nodes can connect to. This construct creates a fleet of Deadline Remote Connection Servers running in Amazon Elastic Container Service (ECS).

**Tip**
This example explicitly turns deletion protection off so this stack can be easily cleaned up. By default, it is turned on to prevent accidental deletion of your `RenderQueue`.

**Python**

In `hello_rfdk/hello_rfdk_stack.py`:

```python
# ...
class HelloRfdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        # ...
        render_queue = rfdk_deadline.RenderQueue(self, 'RenderQueue',
                vpc=vpc,
                repository=repository,
                version=version,
                images=images,
                deletion_protection=False,
            )
```

**TypeScript**

In `lib/hello-rfdk-stack.ts`:

```typescript
// ...
export class HelloRfdkStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);
        // ...
        const renderQueue = new deadline.RenderQueue(this, 'RenderQueue', {
            vpc,
            repository,
            version,
            images,
            deletionProtection: false,
        });
    }
}
```

The last thing you need to add is a fleet of render nodes with the `WorkerInstanceFleet` construct, which creates a fleet of instances running Deadline Worker in an Amazon EC2 Auto Scaling Group.

**Important**
The `WorkerInstanceFleet` construct requires an Amazon Machine Image (AMI) with the Deadline Worker application installed. Substitute `your-ami-id` with your desired AMI ID in the code below. Conveniently, AWS Thinkbox creates public AWS Portal AMIs you can use for this. Follow the steps in the Deadline guide for finding AWS Portal AMIs (these steps instruct you to specifically search for “Deadline Worker Base” images, but you can use any Linux-based Deadline Worker image for this tutorial) and copy over your desired AMI ID.
Deploy the render farm

With the render farm fully defined in code, you can now build and deploy it. First, build the app with:

Python

No build step is necessary.

TypeScript

```
npm run build
```

You can optionally synthesize the CloudFormation template for your app if you are curious to see it (the generated CloudFormation template is around 3300 lines long):

```
npx cdk@${CDK_VERSION} synth
```
Now, deploy the render farm with:

```
npx cdk@${CDK_VERSION} deploy
```

This deployment will take approximately 30 minutes since the render farm contains many resources.

## Update the render farm

You can update properties of the deployed render farm by making changes to your app and deploying again.

By default, the RFDK will setup resources such that you cannot accidentally destroy important components by tearing down your CloudFormation stack. For instance, the `Repository` contains information about your render farm and any work done with it, which is data that can be useful to keep (e.g. to start up your render farm again in the same state it was in previously). For more details, see the section called “Managing resources” (p. 37).

We don’t need to retain any render farm information for this tutorial, so let’s update the removal policies to set the Deletion Policy in the CloudFormation template of the database and file system of your `Repository` so they are destroyed:

### Python

In `hello_rfdk/hello_rfdk_stack.py`:

```python
# ...
class HelloRfdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        # ...
        repository = rfdk_deadline.Repository(self, 'Repository',
                                                 vpc=vpc,
                                                 version=version,
                                                 # Specify the removal policies
                                                 removal_policy=rfdk_deadline.RepositoryRemovalPolicies(
                                                 database=core.RemovalPolicy.DESTROY,
                                                 filesystem=core.RemovalPolicy.DESTROY
                                                 )
```

### TypeScript

In `lib/hello-rfdk-stack.ts`:

```typescript
// ...
export class HelloRfdkStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);
        // ...

        const repository = new deadline.Repository(this, 'Repository', {
            vpc,
            version,
            // Specify the removal policies
            removalPolicy: {
                database: cdk.RemovalPolicy.DESTROY,
                filesystem: cdk.RemovalPolicy.DESTROY,
            }
        })
```
Let's also scale down your WorkerInstanceFleet to 0 so that all workers are terminated to save on costs:

**Python**

In `hello_rfdk/hello_rfdk_stack.py`:

```python
# ...
class HelloRfdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        # ...
        workers = rfdk_deadline.WorkerInstanceFleet(self, 'Workers',
                                                  vpc=vpc,
                                                  render_queue=render_queue,
                                                  worker_machine_image=ec2.MachineImage.generic_linux({
                                                      # TODO: Replace your-ami-id with your chosen AMI ID
                                                      core.Stack.of(self).region: "your-ami-id"
                                                  })),
        # Scale capacity down to 0
        desired_capacity=0,
        min_capacity=0,
        max_capacity=0,
    }
```

**TypeScript**

In `lib/hello-rfdk-stack.ts`:

```typescript
// ...
export class HelloRfdkStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);
        // ...
        new deadline.WorkerInstanceFleet(this, 'WorkerFleet', {
            vpc,
            renderQueue,
                // TODO: Replace your-ami-id with your chosen AMI ID
                [this.region]: 'your-ami-id',
            })),
            // Scale capacity down to 0
            desiredCapacity: 0,
            minCapacity: 0,
            maxCapacity: 0,
        });
    }
}
```

Build your app again to incorporate your changes:
Python

*No build step is necessary.*

TypeScript

```shell
npm run build
```

You can optionally run `cdk diff` to see the changes in the CloudFormation template that will be applied:

```shell
npx cdk@${CDK_VERSION} diff
```

Now let's deploy these changes:

```shell
npx cdk@${CDK_VERSION} deploy
```

You can verify these changes were applied by navigating to your `HelloRfdkStack` in the CloudFormation web console and viewing the updated resources. They should have the `Status` field set to `UPDATE_COMPLETE`.

**(Optional) Submit a job to the render farm**

**Note**

This is an optional step that shows you how to use your render farm but does not cover any new concepts in RFDK. Feel free to skip this step and proceed with [tearing down your render farm](p. 21).

**Setting up the connection**

**Important**

In order to submit a job, you will need to [create a secure connection to your render](p. 29). The rest of this chapter will not work if you don't set up this connection.

Once you set up the connection to the render farm and [allow the connection to the render queue](p. 34), you can build and deploy your changes.

```shell
npm run build
npx cdk@{CDK_VERSION} deploy
```

When the changes are deployed, [get remote connection server address](p. 35) and save it for later.

```shell
RQ_DNS_NAME=load-balancer-dns-name
```

**Connecting Deadline Client to your render farm**

You need to install Deadline Client on the machine you are submitting the job from. You can download Deadline installers from the [AWS Thinkbox downloads page](#).

Once you have downloaded an archive, extract the files and install Deadline Client. For more information, please visit [Deadline Client Installation (Quick)](#) page. Here is how you can install Deadline client in a silent mode on Amazon Linux 2 (AL2):

```bash
yum install lab
```
DEADLINE_VERSION=deadline-version
tar -xvf Deadline-$DEADLINE_VERSION-linux-installers.tar
./DeadlineClient-$DEADLINE_VERSION-linux-x64-installer.run --mode text

You can now connect Deadline Client to your render farm.

New Deadline Client Installation

1. When prompted for the Repository Connection Type, select Remote Connection Server.
2. When prompted for the Remote Connection Server address, enter $RQ_DNS_NAME:8080.

Existing Deadline Client (Command Line)

1. Navigate to bin folder in your Deadline Client installation directory. For example:
   
   ```
   cd /opt/Thinkbox/Deadline10/bin
   ```
2. Change the repository you are connected to with Deadline Command:
   
   ```
   ./deadlinecommand ChangeRepository Remote $RQ_DNS_NAME:8080
   ```

Submitting a job to your render farm

Let's submit a simple command line job that calls ping command.

Navigate to bin folder in your Deadline Client installation directory and submit the job with deadlinecommand:

````
cd /opt/Thinkbox/Deadline10/bin
./deadlinecommand SubmitCommandLineJob -frames 1 -executable "ping" -arguments "-c 3 localhost"
```

Viewing Render Farm Statistics

The easiest way to check the result of the submitted job is to use another deadline command:

````
cd /opt/Thinkbox/Deadline10/bin
./deadlinecommand GetFarmStatistics
```

You should now have Completed Jobs= 1 in the output. Optionally, you can get the job id with GetJobIds and then use that id with GetJob command to view all the job details. Find more Deadline commands here.

Viewing job output in CloudWatch

You can also view the output of the job you submitted in the Deadline Worker logs that can be found in CloudWatch.

1. Open the AWS Web Console.
2. Navigate to Services > CloudWatch > Log groups.
3. Open the log group for your Deadline Worker fleet. By default, this will be of the form /renderfarm/WorkerFleet.
4. Open the logs for your Worker instance. This will be of the form WorkerLogs-ec2-instance-id. The ec2-instance-id is the ID of the hello-rfdk/WorkerFleet/Default instance. You can find this ID in the AWS Web Console for EC2 Service.
5. You should be able to find a line in the logs that contains localhost ping. You can use Filter events input field to easily find it. This is the output of Deadline Worker completing the job you submitted.

**Tear down the render farm**

When you're done with your render farm, you can destroy it:

```bash
npx cdk@${CDK_VERSION} destroy
```

If you have issues destroying your farm, ensure you have completed the section called “Update the render farm” (p. 17) section where we update properties that would prevent some resources from being destroyed. Otherwise, refer to your HelloRfdkStack in the CloudFormation console to resolve any issues.

**Next steps**

Now that you have seen RFDK in action, you can:

- Learn about Working with the RFDK (p. 22) in more depth
- Explore the API reference to discover all of the constructs and classes that RFDK offers
- Learn about Best practices (p. 36) and Security (p. 39) to help make your render farm ready for production

The RFDK is an open-source project. We would be happy to have you contribute!
Working with the RFDK

The Render Farm Deployment Kit on AWS (RFDK) lets you define your AWS cloud infrastructure in a general-purpose programming language. Currently, the RFDK supports TypeScript and Python.

We develop the RFDK in TypeScript and use JSII to provide an idiomatic experience in other supported languages. For example, we distribute RFDK modules using each language's standard repository, and you install them using the language's standard package manager. Methods and properties are even named using your language's recommended naming patterns.

RFDK prerequisites

To use the RFDK, you need an AWS account and a corresponding access key. If you don't have an AWS account yet, see Create and Activate an AWS Account. To find out how to obtain an access key ID and secret access key for your AWS account, see Understanding and Getting Your Security Credentials. To find out how to configure your workstation so the CDK uses your credentials, see Setting Credentials in Node.js.

Tip

If you have the AWS CLI installed, the simplest way to set up your workstation with your credentials is to open a command prompt and type:

```
aws configure
```

All RFDK applications require Node.js, even when your app is written in Python. Please see the section called "Prerequisites" (p. 6) for determining the minimum supported Node.js version. You may download a compatible version for your platform at nodejs.org. We recommend the current LTS version (at this writing, the latest 12.x release).

Creating a project

This section documents how to create a new RFDK Python project.

Creating an AWS CDK application

The first step to create a new RFDK project is to select a version of RFDK to use. It is highly recommended to use the latest version of RFDK available when starting a new project. The following command lists the latest published version of RFDK, stores the result in the RFDK_VERSION shell variable, and outputs a message with the version number.

```
RFDK_VERSION=$(npm view aws-rfdk version)
echo "RFDK Version: ${(RFDK_VERSION)}"
```

If you require a legacy version of RFDK, you can list all available RFDK versions (p. 28) and set the variable to DESIRED_RFDK_VERSION manually.

```
RFDK_VERSION=DESIRED_RFDK_VERSION
```
Once you have the desired version of RFDK stored in the RFDK_VERSION shell variable, you must determine the corresponding AWS CDK version. To determine the CDK version and store it in the CDK_VERSION shell variable, run the following command:

```
CDK_VERSION=$(npm view aws-rfdk@${RFDK_VERSION} 'dependencies.@aws-cdk/core')
```

Next, initialize the CDK app with:

**Python**

```
npx cdk@${CDK_VERSION} init app --language=python
```

**TypeScript**

```
npx cdk@${CDK_VERSION} init app --language=typescript
```

## Installing RFDK

**Python**

Given that you want to use RFDK_VERSION (see previous section), you will need to add the RFDK to the list of dependencies required by your python package. This is done by modifying the setup.py file in the root of your CDK application to add an entry for aws-rfdk to the install_requires argument of the setup function:

```python
setup(
    # ...
    install_requires=[
        "aws-cdk.core==1.57.0",
        "aws-rfdk==RFDK_VERSION"
    ],
    # ...
)
```

Once the dependency has been added to setup.py, install it into the virtual environment created by the CDK toolkit:

```
source .env/bin/activate
pip install -r requirements.txt
```

**TypeScript**

Given that you have the version of RFDK in the RFDK_VERSION shell variable (see previous section), you will need to add the RFDK to the list of dependencies required by your npm package. This can be done with the following command:

```
npm install --save aws-rfdk@{RFDK_VERSION}
```

Next, you will need to Install peer dependencies (p. 28).

## Deadline Container Images

RFDK deploys Deadline server components, such as:
• The **RenderQueue** construct, and
• The **UsageBasedLicensing** construct,

using **AWS Elastic Container Service (ECS)**. RFDK integrates with the Deadline container images and recipes published by **AWS Thinkbox**. These images and recipes can be extended to customize the images deployed to your RFDK render farm. The two supported workflows to deploy Deadline container images are:

1. **the section called “Using AWS Thinkbox ECR Repositories” (p. 24)**

   In most cases, RFDK apps can make use of the Deadline container images published by AWS Thinkbox. The images have been designed to integrate with the RFDK and are automatically updated with the latest system security patches. By using images published by AWS Thinkbox, you can eliminate the effort of building your own images, maintaining a container image repository, and keeping the images updated.

2. **the section called “Staging Deadline Recipes” (p. 26)**

   For more control over the Deadline container images deployed, RFDK provides a staging workflow. AWS Thinkbox publishes recipes for building Deadline container images using Docker. These recipes can be used as-is, or they can be customized as desired. When deploying the **RenderQueue** or **UsageBasedLicensing** constructs to an isolated subnet without internet access, you will need to use the staging workflow. This is because the AWS Thinkbox published container images are hosted through ECR Public Repositories and VPC endpoints are currently unavailable for this service.

### Using AWS Thinkbox ECR Repositories

RFDK provides a **ThinkboxDockerImages** construct that can be used to provide Deadline container images from AWS Thinkbox's public ECR Repositories to other RFDK constructs in your CDK app.

It is recommended that you pin this version to the latest available version of Deadline when building your farm. Please consult Deadline's [CHANGELOG](#) for a list of available versions.

**Note**

Please check with the [RFDK release notes](#) to determine the version compatibility with Deadline.

The following sample code demonstrates how to use **ThinkboxDockerImages**:

**Python**

```
import aws_cdk.core as cdk
from aws_rfdk import deadline

class HelloRfdkStack(cdk.Stack):
    def __init__(self, scope: cdk.Construct, id: str, **kwargs) -> None:
        vpc = cdk.Vpc(scope, 'Vpc')
        # Specify Deadline Version (in this example we pin to the latest 10.1.12.x)
        version = deadline.VersionQuery(self, 'Version',
            version="10.1.12",
        )

        # Fetch the Deadline container images for the specified Deadline version
        images = deadline.ThinkboxDockerImages(self, 'Images',
            version=version,
        )
```

Replace "10.1.12" with your desired minor version of Deadline:
Render Farm Deployment Kit on AWS Developer Guide  
Using AWS Thinkbox ECR Repositories

# Change this value to AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA if you wish to accept the EULA

# for Deadline and proceed with Deadline deployment. Users must explicitly accept the AWS Thinkbox EULA before
# using the AWS Thinkbox Deadline container images.
# See https://www.awsthinkbox.com/end-user-license-agreement for the terms of the agreement.

user_aws_thinkbox_eula_acceptance=deadline.AwsThinkboxEulaAcceptance.USER_REJECTS_AWS_THINKBOX_EULA,
)

repo = deadline.Repository(stack, 'Repo',
    vpc=vpc,
    version=version,
)

# Use the container images to create a RenderQueue
render_queue = deadline.RenderQueue(stack, 'RenderQueue',
    vpc=vpc,
    repository=repo,
    version=version,
    images=images,
)

TypeScript

Replace "10.1.12" with your desired minor version of Deadline:

```typescript
import * as cdk from '@aws-cdk/core';
import * as deadline from 'aws-rfdk/deadline';

export class HelloRfdkStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);

        const vpc = new cdk.Vpc(this, 'Vpc');

        // Specify Deadline Version (in this example we pin to the latest 10.1.12.x)
        const version = new deadline.VersionQuery(this, 'Version', {
            version: "10.1.12",
        });

        # Fetch the Deadline container images for the specified Deadline version
        const images = new deadline.ThinkboxDockerImages(self, 'Images', {
            version: version,
            /**
             * Change this value to AwsThinkboxEulaAcceptance.USER_ACCEPTS_AWS_THINKBOX_EULA if you wish to accept the EULA for
             * Deadline and proceed with Deadline deployment. Users must explicitly accept the AWS Thinkbox EULA before using the
             * AWS Thinkbox Deadline container images.
             * See https://www.awsthinkbox.com/end-user-license-agreement for the terms of the agreement.
             */
            userAwsThinkboxEulaAcceptance: deadline.AwsThinkboxEulaAcceptance.USER_REJECTS_AWS_THINKBOX_EULA,
        });

        const repo = new deadline.Repository(stack, 'Repo', {
            vpc: vpc,
            version: version,
        });
    }
```
// Use the container images to create a RenderQueue
const renderQueue = new deadline.RenderQueue(stack, 'RenderQueue', {
  vpc: vpc,
  repository: repo,
  version: version,
  images: images,
});

Staging Deadline Recipes

RFDK builds upon the Docker image assets feature of the AWS CDK to streamline the workflow of customizing Deadline container images. When you deploy a CDK application that uses Docker image assets, the CDK:

1. Uses Docker to build a container image from a specified directory
2. Pushes the container image layers to the CDK bootstrap ECR Repository
3. Specifies CloudFormation parameters that refer to the name of the pushed ECR

Staging Deadline describes the process of preparing a conventional directory structure that can be built into Deadline container images by the RFDK at deployment time.

RFDK provides a stage-deadline command that stages Deadline.

To stage the Docker recipes for the latest version of Deadline, run the following command:

Python

Assuming your CDK application uses RFDK_VERSION:

```bash
npx -p aws-rfdk@RFDK_VERSION stage-deadline
```

TypeScript

```bash
npx stage-deadline
```

To stage the Docker recipes for a specified Deadline version (replaceableDEADLINE_VERSION/replaceable), run the following command:

Python

Assuming your CDK application uses RFDK_VERSION:

```bash
npx -p aws-rfdk@RFDK_VERSION stage-deadline DEADLINE_VERSION
```

TypeScript

```bash
npx stage-deadline DEADLINE_VERSION
```
If you have custom Deadline installers (DEADLINE_INSTALLER_S3_URI) or Docker recipes (DOCKER_RECIPES_S3_URI), they can be uploaded to S3 in the tar.gz format and used as well:

**Python**

Assuming your CDK application uses RFDK_VERSION:

```
npx -p aws-rfdk@RFDK_VERSION stage-deadline DEADLINE_VERSION
```

**TypeScript**

```
npx stage-deadline DEADLINE_VERSION
```

**Note**

Please check with the RFDK release notes to determine the version compatibility with Deadline.

The default behavior of stage-deadline is to create a stage subdirectory under the current working directory and stage the files into it. The choice of destination directory can be changed by specifying a --output OUTPUT_DIR command-line argument.

RFDK provides a ThinkboxDockerRecipes construct that interacts with the Deadline Docker recipes within a staging directory.

The following sample code demonstrates how to use ThinkboxDockerRecipes:

**Python**

Replace STAGE_DIR with the path to the stage directory:

```python
import aws_cdk.core as cdk
import aws_rfdk.deadline as rfdk_deadline

class HelloRfdkStack(cdk.Stack):
    def __init__(self, scope: cdk.Construct, id: str, **kwargs) -> None:
        vpc = cdk.Vpc(scope, 'Vpc')

        # This directory is the one specified in the --output argument of the stage-deadline command
        stage = rfdk_deadline.Stage.from_directory(STAGE_DIR)

        # Create the ThinkboxDockerRecipes instance
        recipes = rfdk_deadline.ThinkboxDockerRecipes(self, 'ServerImages',
            stage=stage,
        )

        repo = rfdk_deadline.Repository(scope, 'Repo',
            vpc=vpc,
            version=recipes.version,
        )

        # Use the container images to create a RenderQueue
        render_queue = rfdk_deadline.RenderQueue(scope, 'RenderQueue',
            vpc=vpc,
            repository=repo,
            version=recipes.version,
            image=recipes.render_queue_images,
        )
```
TypeScript

Replace `STAGE_DIR` with the path to the stage directory:

```typescript
import * as cdk from '@aws-cdk/core';
import * as deadline from 'aws-rfdk/deadline';

export class HelloRfdkStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
    vpc = new cdk.Vpc(this, 'Vpc');
    // Create the ThinkboxDockerRecipes instance
    recipes = new deadline.ThinkboxDockerRecipes(this, 'Recipes', {
      stage: deadline.Stage.fromDirectory('STAGE_DIR'),
    });
    repo = new deadline.Repository(stack, 'Repo', {
      vpc: vpc,
      version: recipes.version,
    });
    // Use the container images to create a RenderQueue
    render_queue = new deadline.RenderQueue(stack, 'RenderQueue', {
      vpc: vpc,
      repository: repo,
      version: recipes.version,
      image: recipes.renderQueueImages,
    });
  }
}
```

Listing available RFDK versions

Use the following command to output a list of available versions of RFDK:

```
npm view aws-rfdk versions
```

Working with the RFDK in Python

Prerequisites

In addition to the general section called “RFDK prerequisites” (p. 22), please see the Python prerequisites for AWS CDK.

Working with the RFDK in TypeScript

Installing peer dependencies

The following command (requires `jq`) installs all of the peer dependencies required by RFDK. Run it from the root of your CDK application directory.
Connecting To Render Farm

There are different ways to securely connect to your render farm from your network. Please visit Network-to-Amazon VPC connectivity options to find the option that suits you best.

We recommend setting up AWS Site-to-Site VPN for site connectivity, AWS Client VPN for individual machine connectivity, and AWS Direct Connect for customers that need extremely high-throughput low-latency connectivity.

Also, you can use VPC peering if you are working on an EC2 instance and want to connect to your render farm from there.

Connecting With Site-to-Site VPN

To set up a VPN connection between your network and the VPC created with RFDK you need to add the options listed below to the `Vpc` construct in your RFDK application code. Substitute `customer-prefix-cidr-range` with an IP range of your network and `rfdk-vpc-cidr-range` with a CIDR range of the VPC you use in your RFDK application. Also, substitute `customer-gateway-ip` with a public IP address of the gateway of your network.

**Important**

Ensure that your network and VPC do not have overlapping IPv4 CIDR blocks. Otherwise, a Site-to-Site VPN connection may not work. See Site-to-Site VPN limitations for more details.

**Python**

```python
vpc = ec2.Vpc(self, "Vpc",
    # for example, replace rfdk-vpc-cidr-range with 10.100.0.0/16
cidr="rfdk-vpc-cidr-range",
    vpn_gateway=True,
    vpn_connections={
        "static": ec2.VpnConnectionOptions(
            # for example, replace customer-gateway-ip with 10.200.0.10
            ip="customer-gateway-ip",
            static_routes=[
                # for example, replace customer-prefix-cidr-range with 10.200.0.0/16
                "customer-prefix-cidr-range"
            ]
        )
    },
    vpn_route_propagation=[
        ec2.SubnetSelection(
            subnet_type=ec2.SubnetType.PRIVATE
        )
    ]
)
```

**TypeScript**

```typescript
const vpc = new ec2.Vpc(this, 'Vpc', {
    // for example, replace rfdk-vpc-cidr-range with 10.100.0.0/16
cidr: "rfdk-vpc-cidr-range",
    vpnGateway: true,
    vpnConnections: {
        static: {
```
The above code creates a VPC which acts as an AWS end of the VPN tunnel with CIDR range \texttt{rfdk-vpc-cidr-range}. Besides of creating the VPC it will:

- Create private subnets and route tables associated with these subnets. These subnets will be used by the render queue load balancer.
- Create Virtual Private Gateway (let's call it \texttt{VGW-RFDK}) and attach it to the VPC.
- Create Customer Gateway (let's call it \texttt{CGW-RFDK}) with static routing and customer end public IP as \texttt{customer-gateway-ip}.
- Create a Site-to-Site VPN Connection with:
  - Target Type as Virtual Private Gateway.
  - Selected VGW-\texttt{RFDK} and CGW-\texttt{RFDK}.
  - Static routing and IP Prefix range \texttt{customer-prefix-cidr-range}.
- Enable Route Propagation to VGW-\texttt{RFDK} for route tables associated with private subnets in VPC.

\textbf{Note}

You can download configuration for the software VPN you use. Once your app is deployed, log into AWS Web Console and navigate to VPC service in the region where you deployed the render farm. Go to Site-to-Site VPN Connections, select the connection created by RFDK and press Download Configuration. Alternatively, you can use CloudFormation service and select the stack deployed with RFDK. Switch to Resources tab and search for \texttt{VPNConnection}. From there you can navigate directly to the deployed VPN Connection in the VPC console.

You can modify the above code to best meet your specific needs. See \texttt{Vpc construct API} to learn how to configure your VPC.

Also, you should allow the connection from the machines in your network to the render queue (p. 34).

\section*{Connecting With AWS Client VPN}

You can use a few CDK constructs to establish the connection to the render farm with AWS Client VPN. This tutorial provides some CDK code examples similar to the corresponding steps in the \texttt{Getting started with Client VPN} guide.

Use \texttt{CfnClientVpnEndpoint} to create a Client VPN endpoint. The following example assumes that you have already generated server and client certificates and keys and uploaded them to AWS Certificate Manager (ACM). Substitute \texttt{client-certificate-arn} and \texttt{server-certificate-arn} with the ARN of your client and server certificates, respectively. Also, substitute \texttt{customer-prefix-cidr-range} with an IP range of your network.

Below, we explicitly create a security group that is applied to the Client VPN endpoint. Later, you will allow connection to the render queue from this security group.
Render Farm Deployment Kit on AWS Developer Guide
Connecting With AWS Client VPN

Python

```python
AuthReq = ec2.CfnClientVpnEndpoint.ClientAuthenticationRequestProperty
security_group = ec2.SecurityGroup(self, 'SG-VPN-RFDK', vpc=vpc)
endpoint = ec2.CfnClientVpnEndpoint(self, 'ClientVpnEndpointRFDK',
    authentication_options=[
        ec2.CfnClientVpnEndpoint.ClientAuthenticationRequestProperty(
            type="certificate-authentication",
            mutual_authentication=AuthReq(
                client_root_certificate_chain_arn="client-certificate-arn"
            )
        )
    ],
    # for example, replace customer-prefix-cidr-range with 10.200.0.0/16
    client_cidr_block="customer-prefix-cidr-range",
    connection_log_options=ec2.CfnClientVpnEndpoint.ConnectionLogOptionsProperty(
        enabled=False
    ),
    security_group_ids=[
        security_group.security_group_id
    ],
    server_certificate_arn="server-certificate-arn",
    vpc_id=vpc.vpc_id
);
```  

TypeScript

```typescript
import { CfnClientVpnTargetNetworkAssociation, CfnClientVpnEndpoint,
    CfnClientVpnAuthorizationRule, CfnClientVpnRoute, SecurityGroup, Vpc } from '@aws-cdk/aws-ec2';
...
const securityGroup = new SecurityGroup(this, 'SG-VPN-RFDK', {
    vpc,
});

const endpoint = new CfnClientVpnEndpoint(this, 'ClientVpnEndpointRFDK', {
    authenticationOptions: [{
        type: "certificate-authentication",
        mutualAuthentication: {
            clientRootCertificateChainArn: 'client-certificate-arn',
        },
    }],
    // for example, replace customer-prefix-cidr-range with 10.200.0.0/16
    clientCidrBlock: 'customer-prefix-cidr-range',
    connectionLogOptions: {
        enabled: false,
    },
    securityGroupIds: [
        securityGroup.securityGroupId
    ],
    serverCertificateArn: 'server-certificate-arn',
    vpcId: vpc.vpcId,
});
```

To create network associations for the subnets of the VPN use `CfnClientVpnTargetNetworkAssociation`.

Python

```python
for i, subnet in enumerate(vpc.private_subnets):
```
ec2.CfnClientVpnTargetNetworkAssociation(self, 'ClientVpnNetworkAssociation' + str(i),
    client_vpn_endpoint_id=endpoint.ref,
    subnet_id=subnet.subnet_id
)

TypeScript

```
let i = 0;
for (const subnet of vpc.privateSubnets) {
    new CfnClientVpnTargetNetworkAssociation(this, `ClientVpnNetworkAssociation${i}`, {
        clientVpnEndpointId: endpoint.ref,
        subnetId: subnet.subnetId,
    });
    i++;
}
```

Finally, use CfnClientVpnAuthorizationRule to create VPN authorization rules.

Substitute target_network-cidr-block with the CIDR of the network for which you want to allow access. For example, to allow access to the entire VPC, specify the IPv4 CIDR block of the VPC.

Python

```
authorization_rule = ec2.CfnClientVpnAuthorizationRule(self, 'ClientVpnAuthRule',
    authorize_all_groups=True,
    client_vpn_endpoint_id=endpoint.ref,
    target_network_cidr="target_network-cidr-block"
)
```

TypeScript

```
new CfnClientVpnAuthorizationRule(this, 'ClientVpnAuthRule', {
    authorizeAllGroups: true,
    clientVpnEndpointId: endpoint.ref,
    targetNetworkCidr: "target_network-cidr-block",
});
```

Also, you should allow the connection to the render queue (p. 34) from the security group created earlier. This will allow the machines that use Client VPN to connect to the render queue.

You will be able to download the Client VPN endpoint configuration file after you deploy your RFDK application. Then you will be able to connect to the Client VPN endpoint using the configuration file.

### Connecting With VPC peering

You can use CDK to create a VPC peering connection between the VPC created by RFDK and the VPC in which you have your running instances. Substitute customer-prefix-cidr-range with an IP range of your own VPC.

**Important**

Ensure that your VPCs do not have overlapping IPv4 CIDR blocks. Otherwise, a VPC peering connection will not work. See Creating and accepting a VPC peering connection for more details.

Substitute peer-vpc-region and peer-vpc-id with the region and the VPC ID from where you will connect to the render farm deployed with RFDK.
For example, if you are working on an EC2 instance in `us-east-1` region, you can use the following commands to get these values:

```bash
PEER_VPC_REGION=us-east-1
echo $PEER_VPC_ID
```

Now you can use a CDK construct `CfnVPCPeeringConnection` to create a VPC peering connection in your RFDK application.

**Python**

```python
vpc_peering_connection = ec2.CfnVPCPeeringConnection(self, 'VPCPeeringConnection',
  vpc_id=vpc.vpc_id,
  peer_vpc_id="peer-vpc-id",
  peer_region="peer-vpc-region"
)
```

**TypeScript**

```typescript
const vpcPeeringConnection = new ec2.CfnVPCPeeringConnection(this, 'VPCPeeringConnection', {
  vpcId: vpc.vpcId,
  peerVpcId: "peer-vpc-id",
  peerRegion: "peer-vpc-region",
});
```

You will also need to update all the route tables associated with the subnets in both VPCs. This can be done manually in the AWS Web Console after you deploy the RFDK application following the instructions in [Updating your Route tables for a VPC peering connection](#). Alternatively, you can again use CDK and AWS CLI to simplify the process.

To update all the route tables in the RFDK VPC, add the following code to your RFDK application:

**Python**

```python
for i, subnet in enumerate(vpc.private_subnets):
  ec2.CfnRoute(self, 'PeerRoute' + str(i),
    route_table_id=subnet.route_table.route_table_id,
    # for example, replace customer-prefix-cidr-range with 10.200.0.0/16
    destination_cidr_block="customer-prefix-cidr-range",
    vpc_peering_connection_id=vpc_peering_connection.ref
  )
```

**TypeScript**

```typescript
let i = 0;
  vpc.privateSubnets.map(subnet => {
    new ec2.CfnRoute(this, `PeerRoute${i}`, {
      routeTableId: subnet.routeTable.routeTableId,
      // for example, replace customer-prefix-cidr-range with 10.200.0.0/16
      destinationCidrBlock: "customer-prefix-cidr-range",
      vpcPeeringConnectionId: vpcPeeringConnection.ref,
    });
    i++;
```

---

---

33
Also, you should allow the connection from the machines in your VPC to the render queue (p. 34).

In order to update the route table in your other VPC, you can use AWS CLI commands.

There are multiple ways to get the id of the VPC peering connection:

- Use CloudFormation service and select the stack deployed with RFDK. Switch to Resources tab and search for VPCPeeringConnection. Copy the Physical ID and save it in the shell variable.
- Navigate directly to the deployed Peering Connection in the VPC console and copy the physical ID from there.
- Use AWS CLI command describe-vpc-peering-connections to view your VPC peering connections and find the connection id in the output.

```bash
aws --region PEER_VPC_REGION ec2 describe-vpc-peering-connections
VPC_PEERING_CONNECTION_ID=peering-connection-id
```

Now save the id of the route table in the shell variable as well:

```bash
PEER_SUBNET_ID=peer-subnet-id
```

For example, if you want to use AWS CLI:

```bash
echo $PEER_SUBNET_ID
```

Finally, substitute `rfdk-vpc-cidr-range` with a CIDR range of the VPC created by RFDK application and create a route:

```bash
RFDK_VPC_CIDR_RANGE=replaceablerfdk-vpc-cidr-range/replaceable
ROUTING_TABLE=$(aws --region $PEER_VPC_REGION ec2 describe-route-tables --query "RouteTables[*].Associations[?SubnetId=='$PEER_SUBNET_ID'].RouteTableId" --output text)
aws --region $PEER_VPC_REGION ec2 create-route --route-table-id "$ROUTING_TABLE" --destination-cidr-block "$RFDK_VPC_CIDR_RANGE" --vpc-peering-connection-id $VPC_PEERING_CONNECTION_ID
```

Your RFDK application will create a peering connection, but you will still need to accept a VPC peering connection when your app is deployed. You can accept the request in the AWS Console navigating to VPC service and switching to Peering Connections tab or with AWS CLI. Accept the request with AWS CLI using accept-vpc-peering-connection command:

```bash
aws --region $PEER_VPC_REGION ec2 accept-vpc-peering-connection --vpc-peering-connection-id $VPC_PEERING_CONNECTION_ID
```

### Allowing connection to the Render Queue

To allow the connection to the render queue from the machines in your network, add the following code to your RFDK application. Substitute `customer-prefix-cidr-range` with an IP range of your network in CIDR notation, for example, `10.200.0.0/16.

```bash
');
```
Getting remote connection server address

Once your RFDK application is deployed, you can obtain the address of the remote connection server. Navigate to the CloudFormation service in your AWS Web Console. Find the stack deployed with RFDK and click on its name (for example, hello-rfdk). Switch to the Outputs tab and find the Key of the Load Balancer. It should start with RenderQueueAlbEc2ServicePatternLoadBalancer. Copy the Value, which is a DNS name of the load balancer. This DNS name is the address of the remote connection server and you can use it to connect to your render farm.

Alternatively, you could find the same DNS name by navigating to the EC2 service and finding the newly deployed load balancer in the Load Balancers tab.

You can grant access to the render queue for the security group instead of the IP range:

Python

```python
render_queue.connections.allow_default_port_from(ec2.Peer.ipv4("customer-prefix-cidr-range"))
```

TypeScript

```typescript
renderQueue.connections.allowDefaultPortFrom(ec2.Peer.ipv4("customer-prefix-cidr-range"));
```

See Connections construct API for more details.
Best practices while using the RFDK

Best practices are recommendations that can help you use the Render Farm Deployment Kit (RFDK) on AWS effectively and securely throughout its entire workflow. Learn how to plan and organize your stacks, how to upgrade/maintain your farm, and how to manage your resources.

Organizing your stacks - Multi-tiered architecture

Use the lifecycle of your AWS Resources to help you decide what resources should go in each stack. By grouping your resources with common life cycles, you will be able to modify your farm to take down any components that are not currently in use.

We recommend that you use a multi-tiered architecture for your RFDK apps. A multi-tiered architecture organizes stacks into horizontal tiers that build on top of one another, with each tier having a dependency on the tier directly below it. You can have one or more stacks in each tier, but your stacks should have AWS resources with similar lifecycles and ownership within each tier.

Benefits

Cost

With each tier separated into its own AWS CloudFormation stacks, any tiers that are not actively required can be destroyed. For example you could tear down your Workers separately from the remainder of your farm.

Ease of modification

Each tier can be modified independently of each other. Making a change in any single tier will not affect any tiers below it. This allows easy iteration when developing and expanding your render farm.

Deployment Velocity

You are able to reduce the total deployment time when deploying any individual tier by keeping each tier separate. This will reduce the amount of time it takes while developing your RFDK render farm since you are able to deploy each tier of the farm independently.

Example

Consider a Deadline render farm that is made up of the following components:

- A VPC
- A Render Queue
- A Database for the Render Queue
- 1 or more network filesystems
- 1 or more Worker Fleets

This farm can be divided into 4 tiers, each with their own lifecycle:
Network Tier

This Tier would be made up of all resources that are required for networking. In this example that would be the VPC. You could also expand the tier to include VPC Peering or VPN connectivity and the definition VPC Interface Endpoints.

Storage Tier

This Tier would consist of all long term storage that would be beneficial to keep between deployments. In this example that would include the Database and all of the filesystems.

Service Tier

This Tier would consist of all components of the business logic of the render farm. In this example this includes the Render Queue. It could also include additional components such as the Usage Based Licensing resources. This tier being deployed is required to be able to connect to your render farm and submit jobs.

Compute Tier

This tier would consist of all of the Worker Fleets or any other form of burst compute. This tier will be the most volatile since it will often change based on your current workload.

Managing resources

When managing the resources of your render farm it is useful to think of your resources as Cattle, not pets.

In an on-premise render farm, all resources have to be provisioned manually and have large up front investments. Due to this servers were treated as pets - each was unique and had its own needs.

Meanwhile all of the resources of a cloud-based render farm are ephemeral, and each component can be swapped out on a whim. This means that you can modify your existing resources whenever necessary, including changing the type or number of instances to suit your immediate need.

For a given render farm, the only components that should not be torn down at will are the networking, database, and file systems (the Networking and Storage Tiers). You will lose all of your data, except for backups, when those resources are destroyed.

Developing on an EC2 Instance

We recommend developing your RFDK applications on a Linux Amazon Elastic Compute Cloud (EC2) instance. There are multiple advantages of developing on an EC2 instance, including:

✓ **Configurable Performance** - EC2 provides a range of instance types to chose from to balance your performance needs and costs. We recommend using a minimum instance type of t3.small for RFDK development.
✓ **On-demand** - You can start and stop your EC2 instance as needed. You only pay for what you use.
✓ **Reliability** - EC2 commits to a SLA of 99.99% availability for each Amazon EC2 region.

**AWS Cloud9** is an integrated development environment (IDE) that runs in your browser. It provides a streamlined experience for developing CDK applications on an EC2 instance.

The following are the major benefits of using AWS Cloud9 for RFDK development:

- An AWS Cloud9 EC2 Environment can be configured to automatically stop the EC2 instance when the IDE is not in use to save costs.
• When using the AWS Cloud9 IDE, IAM permissions are inherited from your AWS Console session. This avoids the need to persist AWS credentials on the instance.

We provide guided instructions for setting up an the section called “Example development environment” (p. 6).

Source control

As with developing any software, CDK applications can benefit from the use of source control. Source control (or version control) is the practice of tracking and managing changes to code.

Code management systems make it easier to accomplish the following tasks:

• Track changes to your code
• Review revision history
• Revert to previous versions of your source code
• Collaborate as a team to develop your CDK application
• Isolate your work until it is ready
• Trouble-shoot issues by identifying who made changes and what the changes were

We highly recommend that you use of source control when developing CDK applications. Ideally, the code should be pushed to a central source control repository that is highly-available, secure, and maintains redundant physically-distributed copies of the repository data.

AWS CodeCommit is a fully managed service that hosts private git repositories. It is designed to be highly-available, secure, and scalable. We provide guided instructions for setting up an the section called “Example development environment” (p. 6) that can be used.

Additional readings

In addition to the above best practices it is recommended that you take a moment to read the following additional material about the components of the RFDK:

• AWS Fundamentals
• Cloud Formation - Best Practices
• Deadline - Render Farm Considerations
Security in the RFDK

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is 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 all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS Compliance Programs.

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

The RFDK follows the shared responsibility model and compliance is shared between AWS and you. RFDK is designed and developed with a security-first approach. Ultimately, the CDK and RFDK are both SDKs which are intended to be used as building blocks for deploying infrastructure that meets a range of diverse requirements. When building an RFDK application, it is up to you to follow security best practices to ensure that the resulting deployed render farm is secure.

Important
It is highly recommended that you read and familiarize yourself with the Security Pillar of the AWS Well-Architected Framework. This article contains key principles to securing your AWS infrastructure.

In this section:
- Additional Readings (p. 38)
- Data protection in the RFDK (p. 39)
- Identity and access management in the RFDK (p. 44)
- Infrastructure security in the RFDK (p. 44)
- Security best practices for the RFDK (p. 45)

Additional Readings
- Security Pillar - AWS Well-Architected Framework
- Security for the AWS Cloud Development Kit (AWS CDK)
- Security in Amazon Virtual Private Cloud
- AWS security credentials
- Security in Amazon EC2
  - Linux
  - Windows

Data protection in the RFDK

This topic covers all the recommended ways to protect the data that you store inside the RFDK.
Encryption in transit

It is best practice to encrypt all network communications between software components. Transport Layer Security (TLS) is an industry-standard protocol used for encrypting communications between two networked applications. Using TLS reduces the possibility of the communications being intercepted or modified in transit.

TLS makes use of X.509 certificates to make and validate claims of identity between the endpoints of a TLS communication channel. The X.509 standard defines a specific format of digital certificate that can be used for identification and message encryption.

Where possible, the RFDK supports the ability to use TLS for communication between software components of the render farm.

Where does RFDK use TLS and certificates?

RFDK uses X.509 certificates to secure communications between the following constructs:

Deadline Clients and the Deadline Render Queue's Load Balancer

The Render Queue must be provided a certificate that can be used to establish a TLS connection with any Deadline Client that connects to it. The Render Queue deploys an Application Load Balancer (ALB) and configures it to present the supplied certificate during the TLS handshake from incoming client connections. RFDK constructs will configure Deadline Clients to trust the certificate chain when connecting them to a Render Queue.

Deadline Render Queue's Load Balancer and the Deadline Remote Connection Server

The Remote Connection Server (RCS) is encapsulated by the Render Queue construct. It is the service that sits behind the ALB that is set up by the Render Queue. During instantiation, the Render Queue generates a self-signed certificate using X509CertificatePem and X509CertificatePkcs12 that the RCS is configured to use for communication between itself and the ALB.

Deadline Remote Connection Server and the database

The Deadline Remote Connection Server (RCS) communicates directly with a MongoDB-compatible database. These communications are encrypted by default when this database is an Amazon DocumentDB or a MongoDB created by the RFDK's MongoDBInstance construct.

MongoDB

The MongoDB Construct requires that a certificate be provided as input to it, for it to use as its own proof of identity for the clients that need to connect to it. It requires that this certificate be in the IX509CertificatePem format.

RFDK’s built-in certificate management

RFDK provides constructs that can be used to work with X509 certificates. This includes:

- **X509CertificatePem** - A construct that generates and stores the following in AWS Secrets Manager Secrets:
  - A X.509 certificate in the PEM format
  - A private key in PEM format
  - A passphrase for the private key
  - An optional trust-chain in PEM format
• **ImportedAcmCertificate** - A construct that imports a PEM certificate into AWS Certificate Manager. This Construct implements CDK’s `ICertificate` interface so that the certificates can be used by other CDK Constructs that require this interface, such as the Deadline Render Queue construct.

If you are using the RFDK’s built-in certificate constructs, then it is recommended that a self-signed certificate be created by your RFDK application to act as a Certificate Authority (CA), and then used to sign any other certificates that components of the RFDK application require. This root certificate should only be used to sign other certificates, and access to its private key must be tightly controlled. The certificate should not be configured to be used as the identity for a server or service.

**Warning**
The built-in certificate management does not currently support a method to rotate certificates. The X.509 certificates it creates are configured to **expire after 1095 days (3 years)**. After this period, TLS connections between components using these certificates will no longer work and they will need to be redeployed with new certificates.

Here is an example snippet from a CDK application that uses the `X509CertificatePem`, the `X509CertificatePkcs12`, and the `ImportedAcmCertificate`:

**Python**

```python
class CertStack(cdk.Stack):
    def __init__(scope, id, props):
        # Generate a root CA and then use it to sign another identity certificate
            subject={ "cn": "server.internal" },
        );

            subject={ "cn": "name.server.internal" },
            signing_certificate=signing_cert
        );

        # Convert the identity certificate PEM into PKCS#12
            source_certificate=server_cert
        );

        // Import the identity certificate into ACM
        rfdk.ImportedAcmCertificate(this, "AcmCert2",
            cert=server_cert.cert,
            cert_chain=server_cert.certChain,
            key=server_cert.key,
            passphrase=server_cert.passphrase
        );
```

**Typescript**

```typescript
class CertStack extends cdk.Stack {
    constructor(scope: cdk.Construct, id: string, props: StackProps) {
        // Generate a root CA and then use it to sign another identity certificate
        const signingCert = new X509CertificatePem(this, 'RootPem', {
            subject: { cn: 'server.internal' },
        });

        const serverCert = new X509CertificatePem(this, 'ServerPem', {
            subject: { cn: 'name.server.internal' },
            signingCertificate: signingCert,
        });

        // Convert the identity certificate PEM into PKCS#12
        new X509CertificatePkcs12(this, 'Pkcs', {
```
Alternatives to RFDK’s built-in certificate management

Manual certificates

An alternative method of creating and using certificates is similar to the method built into RFDK. Rather than using the X509CertificatePem Construct to generate certificates, they could be generated using any other public key infrastructure. Once a certificate is created, it can be imported into AWS Secrets Manager Secrets in the same format that the ImportedAcmCertificate Construct takes as input.

Private Certificate Manager certificates

It is possible to create a private certificate in AWS Certificate Manager (ACM) that can be used. To do this, you will need to use AWS Certificate Manager Private Certificate Authority (ACM PCA) to create a certificate authority that can then be used to sign a certificate in ACM. Be aware that the ACM PCA costs can be significant and should be considered before taking this approach. Visit the Certificate Manager Pricing for more information.

A benefit of using this method is that the entire management of a certificate is handled for you, you do not have to worry about rotating certificates before they expire or making sure you’re storing your certificates in a secure way. The main trade-offs are the cost and the additional work of having to set them up outside of your CDK application. The CDK does not currently support this workflow with ACM PCA.

An ACM Certificate in CDK does not expose the certificate chain, so when using these private certificates from ACM, the certificate chain will need to be migrated to a Secret for RFDK to distribute to clients of the certificate holder. There are a few ways to do this, so instructions specific to each option can be found in the following sections. Once you have your ACM Certificate and your Secret containing the certificate chain created, they can be imported into CDK using Secret.fromSecretArn(scope, id, secretArn) and Certificate.fromCertificateArn(scope, id, certificateArn). The following example demonstrates how these methods can be used when setting the properties of your RenderQueue during its creation:

Python

```python
acm_cert = aws_cdk.aws_certificatemanager.Certificate.from_certificate_arn(this, 'RqCert', [certificate_arn])
acm_cert_chain = aws_cdk.aws_secretsmanager.Secret.from_secret_arn(this, 'RqCertChain', [secret_arn])
traffic_encryption = aws_rfdk.deadline.RenderQueueTrafficEncryptionProps(
    external_tls = aws_rfdk.deadline.RenderQueueExternalTLSProps(
        acm_certificate = acm_cert,
        acm_certificate_chain = acm_cert_chain,
    ))
```
TypeScript

trafficEncryption: {
  externalTLS: {
    acmCertificate: AcmCertificate.fromCertificateArn(
      this, 
      'RqCert', 
      [certificateArn],
    ),
    acmCertificateChain: Secret.fromSecretArn(
      this, 
      'RqCertChain', 
      [secretArn],
    ),
  },
}

Until the CDK supports this workflow, these are the alternative methods that can be used for the creation of this private certificate:

**Manually create the resources using the AWS Console**

To use the AWS Console to create the resources required for using a private ACM Certificate you'll first need to create a private CA by opening Certificate Manager in the AWS Console, selecting Private CA on the sidebar, and then pressing the Create CA button and following the steps to create a new root CA. Visit the ACM PCA User Guide for more detailed instructions. After creating your private CA, you can then navigate back to Certificate Manager and press the Request a Certificate button. Then you can select "Request a private certificate" and in the next step there will be a drop-down menu to select a CA where you can see your private CA listed. Follow the remaining steps, using a domain name such as renderqueue.server.internal. This domain name should match the fully qualified domain name of the RenderQueue, which is the combination of the hostname set as a property of the RenderQueue and the zone name set as a property of the PrivateHostedZone it's in. More details can be found on the ACM User Guide. Note the ARN of the Certificate after it has been created.

Once your private certificate is created, you'll need to get the certificate chain into AWS Secrets Manager. This can also be done directly in the console, since private certificates can be displayed there by using the export feature. Simply select the Certificate from the Certificate Manager console, click on the Actions menu, and select "Export (private certificates only)". You'll be asked to create a password for the private key. Don't worry too much about the password here since we don't need to export the private key right now and it can be re-entered if you ever do the export process again. After submitting the password, you will see the certificate chain on the next page and it can be copied into the plain text of a Secret. Note the ARN of the Secret after it has been created.

**Use the AWS CLI**

To accomplish this, you can use `acm-pca create-certificate-authority` to create your private CA and then `acm-pca issue-certificate` to create the private certificate. After creating the certificate, you can use `acm get-certificate` to retrieve the certificate chain and then `secrets-manager create-secret` to save that certificate chain as a Secret.

**Use the AWS SDK**

You can write either a custom resource or some separate stand-alone app using the AWS SDK. If you would like to use the AWS SDK for this, you can follow the instructions for the AWS CLI. Each page of the CLI reference has a link titled "See also: AWS API Documentation" that will take you to the API
Identity and access management in the RFDK

Access controls in AWS are governed by the AWS Identity Access Management (IAM) service. In IAM, you can create and manage identities, roles, and policies to determine what entities are able to access or modify which resources. CDK's AWS Construct Library provides several constructs, interfaces and classes for working with IAM in your CDK applications. For an overview of working with IAM in CDK, see the CDK Permissions documentation.

The RFDK uses CDK's IAM concepts to grant access that is required from principals (e.g. EC2 instance profiles, ECS task definitions roles, etc…) that require access to resources. Some examples include:

• The Deadline Render Queue's IAM role is granted read access to the Secrets Manager Secret that stores the credentials to the database
• The Deadline Worker Fleet's IAM role is granted access to stream logs to the CloudWatch log group that it is configured to use

When building a CDK application, it is important to take care when working with IAM resources. As a general rule, it is best to subscribe to the principle of least-privilege and only grant access as minimally required. In the CDK, this is done by scoping IAM policies to minimally required resources and principals. Please refer to Security Best Practices in IAM for more details.

Infrastructure security in the RFDK

The RFDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

In this topic you will find information on some infrastructure security best practices that we would like to emphasize:

• Configuring network firewalls for the RFDK (p. 44)
• Patching software used by the RFDK (p. 45)

Configuring network firewalls for the RFDK

A firewall is a network (virtual) device that governs a network or host. It is used to restrict only specific traffic from entering or exiting the firewall. Firewalls are typically configured with rules that indicate whether traffic directed to/from a specific network address/port is allowed or blocked.

It is recommended that firewalls are used, and that they are configured to restrict traffic to the minimum required access for the render farm to function properly. Doing this can reduce the risk of malicious network activity.

AWS provides multiple levels of network firewalls described in the sections below.

Network access control lists (ACLs)

Network ACLs are firewalls that govern a VPC subnet. It is recommended that you use Network ACLs that are scoped to restrict traffic to enter/exit a VPC subnet to only the traffic that is required for normal
operation of your render farm. Determining the Network ACL rules requires an understanding of the
software that is running on the instances within the subnets and what network connectivity they require
outside of the subnet.

Refer to the Security best practices for your VPC for more details.

Security groups

Security Groups are firewalls that govern one or more instance(s). They are used to restrict network
traffic between the instance(s) within the security group and outside peers.

Patching software used by the RFDK

A render farm is only as secure as the software that is deployed and executing on it. Software
vulnerabilities are continually being discovered by developers, users, and security researchers. The best
way to mitigate the risk of vulnerable software is to adopt a patching strategy. The operating system and
applications running on your render farm should be regularly patched and monitored for vulnerabilities.
Please refer to the following documentation based on the platform of your instances:

- Linux - Managing software on your Amazon Linux instances
- Windows - Best practices for Windows on Amazon EC2

Security best practices for the RFDK

Secure the CDK bootstrap AWS S3 bucket

The CDK provides an assets feature that can be used to upload files from the machine performing the
CDK deployment to an S3 bucket prior to executing a CloudFormation deployments. This bucket is
referred to as the CDK bootstrap bucket, and has many important uses. The RFDK makes use of CDK
assets to provide scripts that are run when instances launch and to provide code for AWS Lambda.

It is important to secure access to the CDK bootstrap bucket. Any modification to the objects stored in
the CDK bootstrap bucket by malicious actors could cause arbitrary remote code execution which could
cause data loss, disruption to the render farm, or leaking of confidential assets.

It is recommended that write access to the bucket be restricted to the IAM user that is used to perform
the deployments of your RFDK application.

If you plan to use multiple CDK applications within the same AWS account and region, then you should
consider creating a separate CDK bootstrap bucket using:

```bash
cdk bootstrap --bootstrap-bucket-name my-bootstrap-bucket-name
```

If doing so, you will also need to modify cdk.json in the root of your CDK application with (see the CDK
docs):

```json
{  
  // ...  
  "toolkitBucketName": "my-bootstrap-bucket-name",  
  // ...  
}
OpenPGP keys for the RFDK

This topic contains the OpenPGP key for the RFDK. This key is used to code-sign the bundles that are published to the RFDK GitHub releases.

RFDK OpenPGP key

<table>
<thead>
<tr>
<th>Key ID</th>
<th>0x3717B1A67981EAE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>RSA</td>
</tr>
<tr>
<td>Size</td>
<td>4096/4096</td>
</tr>
<tr>
<td>Created</td>
<td>2020-08-13</td>
</tr>
<tr>
<td>Expires</td>
<td>2024-08-12</td>
</tr>
<tr>
<td>User ID</td>
<td>AWS Render Farm Deployment Kit <a href="mailto:aws-rfdk@amazon.com">aws-rfdk@amazon.com</a></td>
</tr>
<tr>
<td>Key fingerprint</td>
<td>5E9E FC5E CDD1 F793 3C49 5746 3717 B1A6 7981 EAE3</td>
</tr>
</tbody>
</table>

Select the "Copy" icon to copy the following OpenPGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQINBF81nw08EDAX2iM8DkBeDOI9y3BpufbguxXjBED55fNhm8BuLxXbFEdFkGJBgoU
MzXzV3ZxAmVRFpMnB8t8dCAcOaPf4odAuLpdCf9anE6KbBgXoC+mMoEGVd+8P711
wMC1QxW3V8+lmsZ2RiMUxH6P6/vtYKhtj2mBo+BUBzXM1+yjUfjwzdLYhylvJr
ayQneaKjFG60GvrwnNOYR13m7MYZnFAPd1qr7g6Tw41N7ypqQ+NqzN01i16L13Q72
GyMvXv4o2Z0/Wl1QGDNW78Ux2s5xWl1p5GSeCD/D4tqO+1qj1T8OQRMxwyoOaa
qKwWZCNXXUtBQrdSk1ks8oJG8Ckugnv1+lwawJYe140SHBsmC5qgh7kFgugQ50Sj
b0jVHN/v/gksGqMUQgr+2yXCP6FpYmIgKwHC/K1e66f0MNzGVF/46nsf61b0cpqC
yAM/DK2Ba5WnA9ma9OYauHjUNIz3eN89ClY4xuEUCsPN851kGNCV5SxFr1Ce
BWi55wC5k6mdn72WmLxheyyet9x7IVUCKIWMdxEoYkhm1n58fwY7+/96Qwvw9mko0
Yep4Cu+xLl3UJ9j0uSNEwmpoxmnyNtB91erqKauW3yFTWzD7b71Ny3rN059B9Zj
Zv2pWnmQrldhS+Be2G2WnDGufBQyRgRxa0tPxLRCDTDY1Xn1Tr4GpLiQzRAQAb
tDRB1yMGvUvZSYVIE2hcmOGRGWbG95bWvdCRLxQgPGFM3cJy1ZmrRQGFTYxvp
bi5j2b+iQ/1BMBAgApBQJFNZX8NAhsb8QhKbMcMA4hwaJCAcAgEPgQcCQoLBBYc
AwBCHhCP4AAcCqKxNexpnMBuNMxaRAAupghz20B5rheenqtT9qbo1j6I15d9WD
jFLWn1hLz1XyVqQxKuxu9aAe4f6GcTcUERxyT1eYfY8P8XvMbgYjZ6q1p6X2Xm
D74ieE0xv810vXbojCnchXnVXnVpx08kFzeB7LK3P+d68c3np+oykqDmpwhJFc60ybb
YLFxatY4CFMqJ+Be5W00Hzc++GxGdtyXH3h1jXBUtQlhbWhIFC6B0qQOL+NV5EVE
xzC72GhIy1Xsk214fAfjPFPM3NvR5V9QaxOsQDq9oBoIMoxPGORKbYcntZNUGxKxG
k5xhUNG6tb0EqP4U0xCFSNMeFegu3U3KepdTu2p75nZ26t1ndwpvJxKulip/
S186cS2i3RbNHNg1H6F/27505XRd0KGUXz1sBOHm8d18XHdHoEx6CcbH2mZvy
P1g5zTaBo2n82MWN7F42YZ8xqhn28sNhUtTt6/tb/fXJuW61CLzUtG8qyTcZ2rZr
3H16+0HNgO2M4Juarbnr7hNyB11TMHy6cgo808n/w1TKWw4t7FVUAUJMz1r/Xkxg
m35NxbJ/I/hxStKzC0fB8Uy/lylyh1q5qFAmRnMrj9hFJ3WCLlpDdbuthhv0u00o
+HDVUD4bqZKVrwwk2/y/G8gyQI/R4dLl4vKr9c/r/8wOqBF+oxEOcgzklQybakk
-----END PGP PUBLIC KEY BLOCK-----
```
Verifying the integrity of GitHub release downloads

Each RFDK GitHub release will include both a release bundle asset (aws-rfdk-<VERSION>.zip) and a corresponding signature file (aws-rfdk-<VERSION>.zip.sig).

To verify the release bundle asset, you will need to:

1. Import the RFDK OpenPGP key:

   ```bash
   # Import RFDK's OpenPGP key
   gpg --import --armor <<EOF
   -----BEGIN PGP PUBLIC KEY BLOCK-----
   Version: GnuPG v2
   mQINBF81nw0BEADX2iMDKbBEDoI9SbpmgbuXJBEDE55fNhmb8L3bxbfEFkGJbg0U
   MxzZvZxzwAvFmKb8t8bCQfOb4odUdpfCh9anBE88bXnOC0mRE5GvD+EgT11
   wCm1Qw5wSL81ms2PRMUMZb6p86G/vtYNH7J2mBO+BUUxmlM+JUFYwddLYhHszYr
   ayQtneAKNGCGHUaNORT13m7YMZbPAdq1rgqXfT41nYfwE=QgkzoIt161XO72
   GyHvXvg=sZ20/SLuQ2WWT8u28b15xWllpMGeCD/D4dtOF=1qJ1TCEQKmxwoyoOsa
   qW0WzCHXxU1EOQdsKlks8oijeGCknugvi1ImawaYJe1o4SHBuSmcd5gh7kFug0R5j
   b0JHvVNFV/lnkQ0NUGzr+2xCP6FvmIKqHRNC/K1Enfew0NkMzqVf/4nfSeB1p0cp
   gCm/AK2Kbaa5WwA9ma90fYauuHUMILZe3N89CIy4xeUCpBN8S51kNCV5xFr1cS
   Bw1zS wcK86mdm72W3hxxyet9X71VUC1WmDxEOYnhnK158fWY/=6PRqwY9mkO
   Vep4cu*xlI30UJ7j0uSNEHmwnpxnxyNtB9t3rPa4WYEFtMztb7nQy3rNO558Bz8Z
   ZVp2NwmQr1hds+Sbe2zGWdGulfBqFGxyrxa0tPxLrDCTVDFI1Xn1r40pLiQARAQAB
tDRBY1MgUmVz29zvYIIZhcmo4GvGwv9G5vbWvdCBxAQPGPf3cy1y0zMrzQGPfTXvp
   bii5jb20+iQ1/BBMBAgAbqBSqZ8aHsB8qKHMM4ABwJ7CAcDagEoGQoCCqoLBBYc
   AwEChgUCP4AqCgqNsexpmmhB6uXoRAAupgrhzeOS5rheeqnt79bo1i6IE5d50
   jFtLw3b1h21xVwQwXgk9uLkAg64GScGTCcUERXzyTIeyYv68KMyOyJb63q1p6XZMz
   D7sIErOsXvB0v10xbj0cXAzVp8O8kFz87L7K3P+d6c3np+oykgDnpnhJ6F50yb
   YFLEat4xucFmpj+Be5sowfCzcp+GXdyyXZAhJxjXutQLRhWHiHFCBsoQLO+NVS5re
   xc2T2N1k1Yvsk21zFwFPIM3nr8V97qAosQqQjd8o1WoxPGR0kYcT2ZNUKG0
   gKXhuQNg6BdTBQEqP4uuCXpaFNepekUUGeptDq2P75n2126#1nwvpi7kulp/s
   1866sG293d49NnhgKHP=7c055Rkdo4GGOGZ5kz18sb08m8d18XfHoBeKcBh2m3yv
   Plg25TfaxB2687Nw4F7Z2sQznhn928hc0d2Tas/fXJwu6l1LZU8G87qycZer2rn
   3H76+0NHqoZMJqaurbnn2Hyh1L6yhrge08nWv1l7xwa4tFyoAvAUMZ1/xtXZg
   m35XbJ/j/hxwStKcA0bUY+lywhq1QPJtAmRnhWj9b93WCLPdDnbhothvUaoo
   +HbVUD4bgQZKvzzkxZ/yGSyqOis/R4LDY4VR9c/x/8WwOqBF+oseqGczqizyBakk
   OFJdWUaOR8=
   =UcZe
   -----END PGP PUBLIC KEY BLOCK-----
   EOF
   ```

   2. Determine whether to trust the RFDK GPG key. Some factors to consider when deciding whether or not to trust the above key are:

   - 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; and
   - AWS has taken measures to securely host the OpenPGP public key on this site.

   If you have decided to trust the RFDK GPG key, then run:
# gpg --edit-key 0x3717B1A67981EAE3

```
gpg (GnuPG) 2.0.28; Copyright (C) 2015 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/7981EAE3  created: 2020-08-13  expires: 2024-08-12  usage: SCEA
               trust: unknown       validity: unknown
[ unknown] (1). AWS Render Farm Deployment Kit <aws-rfdk@amazon.com>
gpg> trust

```
```
pub  4096R/7981EAE3  created: 2020-08-13  expires: 2024-08-12  usage: SCEA
               trust: unknown       validity: unknown
[ unknown] (1). AWS Render Farm Deployment Kit <aws-rfdk@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

```
pub  4096R/7981EAE3  created: 2020-08-13  expires: 2024-08-12  usage: SCEA
               trust: ultimate       validity: unknown
[ unknown] (1). AWS Render Farm Deployment Kit <aws-rfdk@amazon.com>
gpg> quit
```

Finally, verify the signature:

```
# gpg --verify aws-rfdk-0.15.0.zip.sig aws-rfdk-0.15.0.zip
```
```
gpg: Signature made Thu 13 Aug 2020 09:49:36 PM UTC using RSA key ID 7981EAE3
```
```
gpg: Good signature from "AWS Render Farm Deployment Kit <aws-rfdk@amazon.com>" [unknown]
```

**Warning**

If GPG outputs an error about an incorrect signature, it is possible that the release bundle asset
was tampered with or corrupted. If this occurs, please contact AWS support for assistance.
Troubleshooting issues in the RFDK

This section covers common problems that users of the RFDK may come across, and actions that can be taken to solve them.

Finding logs

Note
This section assumes you are not setting a custom prefix, so the default /renderfarm/ prefix is used.

The logs are always a good place to start when trying to debug issues with the RFDK. Logs in the RFDK are written into Amazon CloudWatch. After tearing down a render farm, the logs will be persisted so additional debugging can be performed. The same log groups will be continue to be used if another render farm is deployed. When creating RFDK constructs, a log prefix can be passed in using the LogGroupFactoryProps.logGroupPrefix field to allow you to use a different log group. This is useful if you would like to run more than one render farm in a single AWS account.

General

AWS Lambda logs

There are many AWS Lambda functions used throughout RFDK. These will not be reused like the log groups for the construct logs. To find a function’s log group for your current render farm:

1. Navigate to AWS CloudFormation and open the stack that the function is contained in.
2. On the resources tab, you will see entries for AWS::Lambda::Function.
3. If the render farm is still deployed:
   a. The physical ID will link you to the function's page in the AWS Lambda console.
   b. From there you can select the Monitoring tab and click the View logs in CloudWatch button to be taken to the logs.
4. If your render farm was rolled back or destroyed already:
   a. The logs will still be in Amazon CloudWatch even though the AWS Lambda function is gone. In AWS CloudFormation, note the physical ID that the function had.
   b. Navigate to Amazon CloudWatch to find the log group containing that physical ID.

Amazon DocumentDB logs

If audit logging is enabled for your Amazon DocumentDB instance, it will generate audit logs. The log group for these will start with /aws/docdb/ and end with /audit. It will also contain a unique string that is a part of the physical ID of the Amazon DocumentDB Cluster. To be sure you are looking at the correct logs, you can find this string by looking at your stack in AWS CloudFormation and finding and entry for AWS::DocDB::DBInstance. It's important to realize that while the Deadline Repository has a flag to enable audit logging, this will only work if you let the Deadline Repository create the database as well. If you are creating the database yourself and passing it into the Deadline Repository, you will need to ensure you are enabling the audit logging yourself.
**MongoDB logs**

If MongoDB is chosen as the backing database for your Deadline render farm, then there will be logs saved into a log group that is named based on the ID that is passed into the `MongoDbInstance` constructor. For instance, if the construct ID is `MongoDb`, then the log group will be named `/renderfarm/MongoDb`. Inside this log group, you will find log streams for the initialization of the host, named `cloud-init-output-i-[instance-id]`, and for the logs written by the MongoDB daemon, named `MongoDB-i-[instance-id]`. The instance ID can be found by navigating to your Amazon EC2 instances panel and locating the ID of the instance that is named based on the ID that was given to the construct and the AWS CloudFormation stack it's in. It should appear as `[stack-name]/[construct-id]/Server/Asg`.

**Deadline**

**Usage Based Licensing logs**

The Deadline Usage Based Licensing logs will be in a log group based on the ID passed into the Deadline `UsageBasedLicensing` constructor. For example, if you pass in the ID `UBL`, then the logs will appear at `/renderfarm/UBL`. Inside this log group, you will find log streams for the server running the Deadline License Forwarder. This server is run inside a container, so to find the correct log stream for your forwarder, you will need to find the task ID of the running service. To find this task ID, follow these instructions:

1. Navigate to AWS CloudFormation and open the resources tab of the stack the Usage Based Licensing construct is deployed to.
2. In the resources list, find the Amazon ECS Cluster, which should have an `AWS::ECS::Cluster` type, and use the link in the physical ID column to view it.
3. On the tasks tab, there should be one task running, with a task definition that starts with `[stack-id][construct-id]TaskDefinition`. Note its UUID.
4. Navigate to AWS CloudWatch and open the log group. Inside it you should see a log stream with the UUID that was found in the task column.

**Render Queue logs**

The Deadline Render Queue logs will be in a log group based on the ID passed into the Deadline `RenderQueue` constructor. For example, if you pass in the ID `RenderQueue`, then the logs will appear at `/renderfarm/RenderQueue`. Inside this log group, you will find log streams for the Deadline Remote Connection Server (RCS). The RCS is run inside a container, so to find the correct log stream for your RCS, you will need to find the task ID of the running service. To find this task ID, follow these instructions:

1. Navigate to AWS CloudFormation and open the resources tab of the stack the Render Queue is deployed to.
2. In the resources list, find the RenderQueue Cluster which should have an `AWS::ECS::Cluster` type and use the link in the physical ID column to view it.
3. On the tasks tab, there should be one task running, with a task definition that starts with `[stack-id][construct-id]TaskDefinition`. Note its UUID.
4. Navigate to AWS CloudWatch and open the log group. Inside it you should see a log stream with the UUID that was found in the task column.

**Repository logs**

The Deadline Repository logs will be in a log group based on the ID passed to the Deadline `Repository` constructor. For example, if you pass in the ID `Repository`, then the logs will appear at `/renderfarm/Repository`.
Repository. These logs can be used to debug issues either with initializing the repository host, or with the installation of the Deadline repository on that host. Inside this log group, you will find log streams from the initialization of the host, named `cloud-init-output-i-[instance-id]` as well as logs from the Deadline Repository installer, named `deadlineRepositoryInstallationLogs-i-[instance-id]`.

**Worker Fleet logs**

All Deadline Worker Fleet logs will be in a log group based on the ID passed into the constructor of the Deadline Worker Fleet construct. For example, if you pass in the ID `WorkerFleet`, then the logs will appear at `/renderfarm/WorkerFleet`. If your Deadline Worker Fleet fails to start a host or you are having fleet health issues, you can look into the logs for that host. First go into the Amazon EC2 Console and look up the instance ID for the terminated host, then navigate to CloudWatch > log groups > `/renderfarm/WorkerFleet`. There will be three log streams that get written here:

1. One that contains logs from the host's initialization named `cloud-init-output-i-[instance-id]` on Linux or `UserdataExecution-i-[instance-id]` on Windows.
2. One that contains Deadline specific logs and will be named `WorkerLogs-i-[instance-id]`.
3. One that contains logs from the Deadline Launcher and will be named `LauncherLogs-i-[instance-id]`.

**Staging Deadline**

**Error fetching Deadline**

When running `stage-deadline` you may get the following error message:

```plaintext
ERROR: Could not fetch s3://thinkbox-installers/DeadlineDocker/[version-number]/DeadlineDocker-[version-number].tar.gz (Are you authenticated with the AWS CLI?)
```

There can be two reasons for this message:

1. You are trying to pull a version that really does not exist or is incompatible with the RFDK.
2. You are not properly configured to make authenticated calls with the AWS CLI using a user or role with the `s3:GetObject` IAM action. This is required because the staging script uses the AWS CLI to fetch the required files from S3.

**Action to Take**

If you aren't properly configured to make authenticated requests with the AWS CLI, you can follow the AWS CLI guide to configure credentials for a user or role in your AWS account. The easiest options are either using `aws configure` to set the credentials file's default or named profiles, or setting environment variables.

The `stage-deadline` command requires that the IAM entity you authenticate with has the `s3:GetObject` IAM action on the `s3://thinkbox-installer/*` resources; A root user or one with admin permissions would suffice. However, if you would like to follow better security practices and use a user with access limited to just these files, you can follow these instructions:

First create an IAM policy with the following JSON. You can follow the Create a Customer Managed Policy guide if you aren't familiar with how to do this.

```json
{
    "Version": "2012-10-17",
```
After finishing the policy setup, create a user that has your new policy applied. If you aren't familiar with how to do this, follow the Adding a User guide. When it gets to the step about setting permissions, choose **Attach existing policies to user directly** and attach your policy.

## Changing Deadline versions

If you attempt to use a different value for `DEADLINE_VERSION` and rerun the `stage-deadline` script, you may get the following message:

```
stage-deadline DEADLINE_VERSION --output STAGE_PATH
The target directory is not empty.
```

**Action to Take**

Changing the Deadline version can be done in two ways:

1. Use a different directory for staging by providing a different `STAGE_PATH` to the `--output` flag. Remember to update the stage property that gets passed into the `ThinkboxDockerRecipes` constructor to this new location.
2. Rename the existing staging directory (or delete it if you don’t need it anymore) that was created when you first ran the staging script, and then rerun the script with the new Deadline version number. The staging directory will be the value sent to `--output` or `./stage` by default.

## Connecting to the bastion host

The `BastionHostLinux` construct in the CDK does not allow an Amazon EC2 key pair to be set, so attempting to SSH to it via the command line will not work out-of-the-box.

**Action to Take**

If you would like to access this host, it can be done from the Amazon EC2 Console or by adding an SSH key via the AWS CLI.

**To access via the Amazon EC2 console**

1. Navigate to your Amazon EC2 instances
2. Right click on the **Bastion Host**.
3. Select **Connect**.
4. Select either **Session Manager** or **EC2 Instance Connect** to get an SSH session to open in the browser.

**To access via SSH using the AWS CLI**

If you would prefer to connect via the command line with an SSH key then one can be added manually with the AWS CLI’s `ec2-instance-connect send-ssh-public-key` command. Note that this will
Dependencies between tiered stacks

Removing dependencies from tiered stacks

The RFDK suggests to use tiered stacks so that parts of the render farm can be taken down or updated without having to affect the whole render farm. While this saves time and resources overall, it can lead to a known bug in CDK that occurs when a stack has a dependency on another stack and then removes that dependency. You may see an error message like the following:

```
0/1 | 12:13:45 | UPDATE_ROLLBACK_IN_P | AWS::CloudFormation::Stack | dep-stack Export dep-stack:ResourceFromPrimaryStack cannot be deleted as it is in use by primary-stack
```

**Action to Take**

There is an open issue with the CDK to fix this: #3414. Until that is fixed, the workaround is to deploy your dependent stack first with `cdk deploy -e [dep-stack]` and then run `cdk deploy [primary-stack]` to clean up the unused export. This may not always work depending on how the dependencies between the stacks get resolved, and you may need to destroy your stacks and redeploy.

Moving constructs between dependent stacks

If you are trying to move a construct from one stack to a stack that it depends on, then you may encounter errors deploying the revised stacks.

**Action to Take**

The stack that you are moving the construct to and all the stacks that depend on it will need to be destroyed and then redeployed. Using `cdk deploy -e` will not work in this case.

Default removal policies

Some constructs cannot be destroyed due to their default removal policies. This can then prevent the destruction of other constructs that those other constructs depend on. An example of this happening in the RFDK is the Amazon DocumentDB Cluster and Amazon Elastic File System constructs having default removal policies of `RemovalPolicy.RETAIN` and dependencies on the VPC.

**Action to Take**

**Warning**

Make sure you evaluate whether or not using `RemovalPolicy.DESTROY` is acceptable for your use case. It's possible that you may not want certain constructs destroyed if you believe they will contain important data that you want persisted after the rest of your render farm is destroyed.

To preemptively avoid this problem, you can modify the removal policy to `RemovalPolicy.DESTROY` before performing your deployment. If you have not modified this property and would like to clean these leftover constructs up, they can be deleted from the AWS console. Once deleted, `cdk destroy [stack-name]` can be rerun to finish destroying anything else that failed to be deleted.
AWS glossary

For the latest AWS terminology, see the AWS glossary in the AWS General Reference.
Document history for the Render Farm Deployment Kit on AWS

**Note**
The table below represents significant documentation milestones. We fix errors and improve content on an ongoing basis.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document integration with AWS Thinkbox ECR Images</td>
<td>Update several chapters to document how to use AWS Thinkbox published Deadline container images with RFDK.</td>
<td>February 1, 2021</td>
</tr>
<tr>
<td>&quot;Connecting To Render Farm&quot; topic</td>
<td>Added section to the &quot;Working with the RFDK&quot; chapter describing how to connect to the render queue and added an optional step to &quot;Your first RFDK app&quot; chapter.</td>
<td>December 22, 2020</td>
</tr>
<tr>
<td>&quot;stage-deadline&quot; section</td>
<td>Section in the &quot;Working with the RFDK&quot; chapter updated with the simplified usage of the stage-deadline script.</td>
<td>December 9, 2020</td>
</tr>
<tr>
<td>&quot;Example development environment&quot; topic</td>
<td>Section added to the &quot;Getting started&quot; chapter describing how to set up a example development environment for use with RFDK.</td>
<td>October 26, 2020</td>
</tr>
<tr>
<td>&quot;Troubleshooting&quot; topic</td>
<td>Section added for common issues with the RFDK and how to solve them.</td>
<td>September 15, 2020</td>
</tr>
<tr>
<td>&quot;Working with the RFDK&quot; topic</td>
<td>Section added for working with the RFDK, describing prerequisites for TypeScript and Python, as well as how to start a project.</td>
<td>August 31, 2020</td>
</tr>
<tr>
<td>General Availability</td>
<td>Documentation for intial release of RFDK.</td>
<td>August 26, 2020</td>
</tr>
</tbody>
</table>