# Advanced Topics

Advanced patterns, customizations, and enterprise features for AWS CDK.

## 🚀 Advanced Features

### CI/CD and Pipelines

* [CDK Pipelines](create-cdk-pipeline.md) - Continuous deployment ([detailed guide](cdk-pipelines/))
* [Stages](stages.md) - Multi-stage deployments
* [Blueprints](blueprints.md) - Reusable infrastructure patterns

### Customization and Extension

* [Customize Synthesis](customize-synth.md) - Custom synthesis behavior
* [Customize Permissions Boundaries](customize-permissions-boundaries.md) - IAM boundaries
* [Aspects](aspects.md) - Cross-cutting concerns and policies
* [Libraries](libraries.md) - Building and using CDK libraries

### CloudFormation Integration

* [CFN Layer](cfn-layer.md) - Low-level CloudFormation constructs
* [Use CFN Template](use-cfn-template.md) - Integrating existing CloudFormation
* [Use CFN Public Registry](use-cfn-public-registry.md) - Third-party resources

### Advanced Utilities

* [Get CloudFormation Values](get-cfn-val.md) - Retrieving CloudFormation outputs
* [Get Context Variables](get-context-var.md) - Working with context
* [Get Environment Variables](get-env-var.md) - Environment variable access
* [Get Secrets Manager Values](get-sm-val.md) - Secrets integration
* [Get SSM Parameter Values](get-ssm-val.md) - Parameter Store integration
* [Set CloudWatch Alarms](set-cw-alarm.md) - Monitoring and alerting

### Migration and Toolkit

* [Migration Guide](migrate.md) - Migrating from CDK v1 to v2
* [Toolkit Library](toolkit-library.md) - CDK Toolkit programmatic usage
* [Toolkit Actions](toolkit-library-actions.md) - Custom toolkit actions
* [Toolkit Configuration](toolkit-library-configure.md) - Toolkit customization
* [Toolkit Getting Started](toolkit-library-gs.md) - Toolkit library basics
* [Toolkit CA Configuration](toolkit-library-configure-ca.md) - CA certificates
* [Toolkit Messages](toolkit-library-configure-messages.md) - Custom messages

### Policy and Validation

* [Policy Validation](policy-validation-synthesis.md) - Policy validation during synthesis

## 🎯 Learning Paths

### CI/CD Pipeline Setup

1. [**CDK Pipelines**](create-cdk-pipeline.md) - Start with pipeline overview
2. [**Stages**](stages.md) - Understand deployment stages
3. [**Blueprints**](blueprints.md) - Use reusable patterns

### CloudFormation Integration

1. [**CFN Layer**](cfn-layer.md) - Low-level constructs
2. [**Use CFN Template**](use-cfn-template.md) - Template integration
3. [**Migration**](migrate.md) - Moving from CloudFormation

### Enterprise Features

1. [**Aspects**](aspects.md) - Policy enforcement
2. [**Customize Permissions**](customize-permissions-boundaries.md) - Security boundaries
3. [**Policy Validation**](policy-validation-synthesis.md) - Compliance checking

## 🔗 Related Sections

* [Core Concepts](../02-core-concepts/) - Foundation knowledge required
* [Development](../03-development/) - Development best practices
* [Deployment](../04-deployment/) - Basic deployment knowledge needed

# AWS CDK Pipelines Tutorial

This comprehensive guide to creating CDK pipelines has been split into focused sections for better learning.

## 📚 Complete CDK Pipelines Guide

Learn how to create, configure, and manage CDK pipelines step by step:

## Learning Path

1. [**Pipeline Introduction**](01-pipeline-introduction.md) - Overview and concepts
2. [**Pipeline Protection**](02-pipeline-protection.md) - Setting up protection and permissions
3. [**Initialize Pipeline**](03-pipeline-init.md) - Creating your first pipeline
4. [**Define Pipeline**](04-pipeline-define.md) - Configuring pipeline structure and stages
5. [**Pipeline Stages**](05-pipeline-stages.md) - Managing deployment stages and environments
6. [**Pipeline Validation**](06-pipeline-validation.md) - Adding tests and validation steps
7. [**Pipeline Security**](07-pipeline-security.md) - Security considerations and best practices
8. [**Troubleshooting**](08-pipeline-troubleshooting.md) - Common issues and solutions

## Quick Start

If you’re new to CDK pipelines: 1. Start with [Pipeline Introduction](01-pipeline-introduction.md) 2. Follow [Initialize Pipeline](03-pipeline-init.md) to create your first pipeline 3. Use [Define Pipeline](04-pipeline-define.md) to configure it

## Prerequisites

* Basic understanding of AWS CDK concepts
* AWS account with appropriate permissions
* CDK CLI installed and configured

## Related Topics

* [Stages](../stages.md) - Understanding CDK stages
* [Bootstrapping](../../04-deployment/bootstrapping.md) - Environment setup
* [Best Practices](../../03-development/best-practices.md) - CDK best practices

:doctype: book

include::attributes.txt[]

// Attributes

:https–docs-aws-amazon-com-cdk-api-v2-docs-aws-cdk-lib-pipelines-CodePipeline-html-addwbrstagestage-optionss: https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.CodePipeline.html#addwbrstagestage-optionss :https–github-com-aws-aws-cdk-blob-master-packages–aws-cdk-pipelines-ORIGINAL-API-md: https://github.com/aws/aws-cdk/blob/master/packages/@aws-cdk/pipelines/ORIGINAL\_API.md

[.topic] [#cdk-pipeline] = Continuous integration and delivery (CI/CD) using CDK Pipelines :info\_titleabbrev: Create CDK Pipelines

// Content start

Use the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines-readme.html[CDK Pipelines] module from the {aws} Construct Library to configure continuous delivery of {aws} CDK applications. When you commit your CDK app’s source code into {aws} CodeCommit, GitHub, or {aws} CodeStar, CDK Pipelines can automatically build, test, and deploy your new version.

CDK Pipelines are self-updating. If you add application stages or stacks, the pipeline automatically reconfigures itself to deploy those new stages or stacks.

= [NOTE]

CDK Pipelines supports two APIs. One is the original API that was made available in the CDK Pipelines Developer Preview. The other is a modern API that incorporates feedback from CDK customers received during the preview phase. The examples in this topic use the modern API. For details on the differences between the two supported APIs, see link:https://github.com/aws/aws-cdk/blob/master/packages/@aws-cdk/pipelines/ORIGINAL\_API.md [CDK Pipelines original API] in the *aws-cdk GitHub repository*.

====

[#cdk-pipeline-bootstrap] == Bootstrap your {aws} environments

Before you can use CDK Pipelines, you must bootstrap the {aws} xref:environments[environment] that you will deploy your stacks to.

A CDK Pipeline involves at least two environments. The first environment is where the pipeline is provisioned. The second environment is where you want to deploy the application’s stacks or stages to (stages are groups of related stacks). These environments can be the same, but a best practice recommendation is to isolate stages from each other in different environments.

= [NOTE]

See xref:bootstrapping[{aws} CDK bootstrapping] for more information on the kinds of resources created by bootstrapping and how to customize the bootstrap stack.

====

Continuous deployment with CDK Pipelines requires the following to be included in the CDK Toolkit stack:

* An Amazon Simple Storage Service (Amazon S3) bucket.
* An Amazon ECR repository.
* IAM roles to give the various parts of a pipeline the permissions they need.

The CDK Toolkit will upgrade your existing bootstrap stack or creates a new one if necessary.

To bootstrap an environment that can provision an {aws} CDK pipeline, invoke cdk bootstrap as shown in the following example. Invoking the {aws} CDK Toolkit via the npx command temporarily installs it if necessary. It will also use the version of the Toolkit installed in the current project, if one exists.

--cloudformation-execution-policies specifies the ARN of a policy under which future CDK Pipelines deployments will execute. The default AdministratorAccess policy makes sure that your pipeline can deploy every type of {aws} resource. If you use this policy, make sure you trust all the code and dependencies that make up your {aws} CDK app.

Most organizations mandate stricter controls on what kinds of resources can be deployed by automation. Check with the appropriate department within your organization to determine the policy your pipeline should use.

You can omit the --profile option if your default {aws} profile contains the necessary authentication configuration and {aws} Region.

==== [role=“tablist”] macOS/Linux:: + [source,none,subs=“verbatim,attributes”] — npx cdk bootstrap aws://++++++/++++++–profile ++++++ –cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess —-++++++++++++++++++

Windows:: + [source,none,subs=“verbatim,attributes”] — npx cdk bootstrap aws://++++++ –profile< ADMIN-PROFILE> {caret} –cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess —- ====++++++

To bootstrap additional environments into which {aws} CDK applications will be deployed by the pipeline, use the following commands instead. The --trust option indicates which other account should have permissions to deploy {aws} CDK applications into this environment. For this option, specify the pipeline’s {aws} account ID.

Again, you can omit the --profile option if your default {aws} profile contains the necessary authentication configuration and {aws} Region.

==== [role=“tablist”] macOS/Linux:: + [source,none,subs=“verbatim,attributes”] — npx cdk bootstrap aws://++++++/++++++–profile ++++++ –cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess  –trust ++++++—-++++++++++++++++++++++++

Windows:: + [source,none,subs=“verbatim,attributes”] — npx cdk bootstrap aws://++++++/++++++–profile ++++++{caret} –cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess {caret} –trust ++++++—- ====++++++++++++++++++++++++

= [TIP]

Use administrative credentials only to bootstrap and to provision the initial pipeline. Afterward, use the pipeline itself, not your local machine, to deploy changes.

====

If you are upgrading a legacy bootstrapped environment, the previous Amazon S3 bucket is orphaned when the new bucket is created. Delete it manually by using the Amazon S3 console.

[#cdk-pipeline-protect] === Protecting your bootstrap stack from deletion

If a bootstrap stack is deleted, the {aws} resources that were originally provisioned in the environment to support CDK deployments will also be deleted. This will cause the pipeline to stop working. If this happens, there is no general solution for recovery.

After your environment is bootstrapped, do not delete and recreate the environment’s bootstrap stack. Instead, try to update the bootstrap stack to a new version by running the cdk bootstrap command again.

To protect against accidental deletion of your bootstrap stack, we recommend that you provide the --termination-protection option with the cdk bootstrap command to enable termination protection. You can enable termination protection on new or existing bootstrap stacks. To learn more about this option, see xref:ref-cli-cmd-bootstrap-options-termination-protection[--termination-protection].

After enabling termination protection, you can use the {aws} CLI or CloudFormation console to verify.

. Run the following command to enable termination protection on a new or existing bootstrap stack: + [source,none,subs=“verbatim,attributes”] — $ cdk bootstrap –termination-protection — + . Use the {aws} CLI or CloudFormation console to verify. The following is an example, using the {aws} CLI. If you modified your bootstrap stack name, replace CDKToolkit with your stack name: + [source,none,subs=“verbatim,attributes”] — $ aws cloudformation describe-stacks –stack-name ++++++–query “Stacks[0].EnableTerminationProtection” true —-++++++

[#cdk-pipeline-init] == Initialize a project

Create a new, empty GitHub project and clone it to your workstation in the my-pipeline directory. (Our code examples in this topic use GitHub. You can also use {aws} CodeStar or {aws} CodeCommit.)

== [source,none,subs=“verbatim,attributes”]

git clone ++++++my-pipeline cd my-pipeline —-++++++

= [NOTE]

You can use a name other than my-pipeline for your app’s main directory. However, if you do so, you will have to tweak the file and class names later in this topic. This is because the {aws} CDK Toolkit bases some file and class names on the name of the main directory.

====

After cloning, initialize the project as usual.

==== [role=“tablist”] TypeScript:: + [source,none,subs=“verbatim,attributes”] — $ cdk init app –language typescript —

JavaScript:: + [source,none,subs=“verbatim,attributes”] — $ cdk init app –language javascript —

Python:: + [source,none,subs=“verbatim,attributes”] — $ cdk init app –language python — + After the app has been created, also enter the following two commands. These activate the app’s Python virtual environment and install the {aws} CDK core dependencies. + [source,none,subs=“verbatim,attributes”] — $ source .venv/bin/activate # On Windows, run .\venv\Scripts\activate instead $ python -m pip install -r requirements.txt —

Java:: + [source,none,subs=“verbatim,attributes”] — $ cdk init app –language java — + If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose *File* > *Import* > *Maven* > *Existing Maven Projects*. Make sure that the project settings are set to use Java 8 (1.8).

C#:: + [source,none,subs=“verbatim,attributes”] — $ cdk init app –language csharp — + If you are using Visual Studio, open the solution file in the src directory.

Go:: + [source,none,subs=“verbatim,attributes”] — $ cdk init app –language go — + After the app has been created, also enter the following command to install the {aws} Construct Library modules that the app requires. + [source,none,subs=“verbatim,attributes”] — $ go get — ====

= [IMPORTANT]

Be sure to commit your cdk.json and cdk.context.json files to source control. The context information (such as feature flags and cached values retrieved from your {aws} account) are part of your project’s state. The values may be different in another environment, which can cause unexpected changes in your results. For more information, see xref:context[Context values and the {aws} CDK].

====

[#cdk-pipeline-define] == Define a pipeline

Your CDK Pipelines application will include at least two stacks: one that represents the pipeline itself, and one or more stacks that represent the application deployed through it. Stacks can also be grouped into *stages*, which you can use to deploy copies of infrastructure stacks to different environments. For now, we’ll consider the pipeline, and later delve into the application it will deploy.

The construct +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.CodePipeline.html[CodePipeline]+ is the construct that represents a CDK Pipeline that uses {aws} CodePipeline as its deployment engine. When you instantiate CodePipeline in a stack, you define the source location for the pipeline (such as a GitHub repository). You also define the commands to build the app.

For example, the following defines a pipeline whose source is stored in a GitHub repository. It also includes a build step for a TypeScript CDK application. Fill in the information about your GitHub repo where indicated.

= [NOTE]

By default, the pipeline authenticates to GitHub using a personal access token stored in Secrets Manager under the name github-token.

====

You’ll also need to update the instantiation of the pipeline stack to specify the {aws} account and Region.

==== [role=“tablist”] TypeScript:: In lib/my-pipeline-stack.ts (may vary if your project folder isn’t named my-pipeline): + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import { CodePipeline, CodePipelineSource, ShellStep } from ‘aws-cdk-lib/pipelines’;

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

const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: new ShellStep(‘Synth’, { input: CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’), commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’] }) }); } } —- + In bin/my-pipeline.ts (may vary if your project folder isn’t named my-pipeline): + [source,javascript,subs=“verbatim,attributes”] —- #!/usr/bin/env node import \* as cdk from ‘aws-cdk-lib’; import { MyPipelineStack } from ‘../lib/my-pipeline-stack’;

const app = new cdk.App(); new MyPipelineStack(app, ‘MyPipelineStack’, { env: { account: ‘111111111111’, region: ‘eu-west-1’, } });

== app.synth();

JavaScript:: In lib/my-pipeline-stack.js (may vary if your project folder isn’t named my-pipeline): + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const { CodePipeline, CodePipelineSource, ShellStep } = require(‘aws-cdk-lib/pipelines’);

class MyPipelineStack extends cdk.Stack { constructor(scope, id, props) { super(scope, id, props);

const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: new ShellStep(‘Synth’, { input: CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’), commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’] }) }); } }

== module.exports = { MyPipelineStack }

* In bin/my-pipeline.js (may vary if your project folder isn’t named my-pipeline):

## [source,javascript,subs=“verbatim,attributes”]

* #!/usr/bin/env node

const cdk = require(‘aws-cdk-lib’); const { MyPipelineStack } = require(‘../lib/my-pipeline-stack’);

const app = new cdk.App(); new MyPipelineStack(app, ‘MyPipelineStack’, { env: { account: ‘111111111111’, region: ‘eu-west-1’, } });

== app.synth();

Python:: In my-pipeline/my-pipeline-stack.py (may vary if your project folder isn’t named my-pipeline): + + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk from constructs import Construct from aws\_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep

class MyPipelineStack(cdk.Stack):

…. def **init**(self, scope: Construct, construct\_id: str, \*\*kwargs) -> None: super().\_\_init\_\_(scope, construct\_id, \*\*kwargs)

pipeline = CodePipeline(self, "Pipeline",  
 pipeline\_name="MyPipeline",  
 synth=ShellStep("Synth",  
 input=CodePipelineSource.git\_hub("OWNER/REPO", "main"),  
 commands=["npm install -g aws-cdk",  
 "python -m pip install -r requirements.txt",  
 "cdk synth"]  
 )  
 ) ---- + In `app.py`: + [source,python,subs="verbatim,attributes"] ---- #!/usr/bin/env python3 import aws\_cdk as cdk from my\_pipeline.my\_pipeline\_stack import MyPipelineStack

….

app = cdk.App() MyPipelineStack(app, “MyPipelineStack”, env=cdk.Environment(account=“111111111111”, region=“eu-west-1”) )

== app.synth()

Java:: In src/main/java/com/myorg/MyPipelineStack.java (may vary if your project folder isn’t named my-pipeline): + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import java.util.Arrays; import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.pipelines.CodePipeline; import software.amazon.awscdk.pipelines.CodePipelineSource; import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack { public MyPipelineStack(final Construct scope, final String id) { this(scope, id, null); }

…. public MyPipelineStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")  
 .pipelineName("MyPipeline")  
 .synth(ShellStep.Builder.create("Synth")  
 .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))  
 .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))  
 .build())  
 .build();

} } —- + In src/main/java/com/myorg/MyPipelineApp.java (may vary if your project folder isn’t named my-pipeline): + [source,java,subs=“verbatim,attributes”] —- package com.myorg; ….

import software.amazon.awscdk.App; import software.amazon.awscdk.Environment; import software.amazon.awscdk.StackProps;

public class MyPipelineApp { public static void main(final String[] args) { App app = new App();

…. new MyPipelineStack(app, “PipelineStack”, StackProps.builder() .env(Environment.builder() .account(“111111111111”) .region(“eu-west-1”) .build()) .build());

app.synth();

} } —- ….

C#:: In src/MyPipeline/MyPipelineStack.cs (may vary if your project folder isn’t named my-pipeline): + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.Pipelines;

namespace MyPipeline { public class MyPipelineStack : Stack { internal MyPipelineStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { var pipeline = new CodePipeline(this, “pipeline”, new CodePipelineProps { PipelineName = “MyPipeline”, Synth = new ShellStep(“Synth”, new ShellStepProps { Input = CodePipelineSource.GitHub(“OWNER/REPO”, “main”), Commands = new string[] { “npm install -g aws-cdk”, “cdk synth” } }) }); } } } — + In src/MyPipeline/Program.cs (may vary if your project folder isn’t named my-pipeline): + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK;

namespace MyPipeline { sealed class Program { public static void Main(string[] args) { var app = new App(); new MyPipelineStack(app, “MyPipelineStack”, new StackProps { Env = new Amazon.CDK.Environment { Account = “111111111111”, Region = “eu-west-1” } });

app.Synth();  
 }

} } —-

Go:: + [source,go,subs=“verbatim,attributes”] — package main

import ( “github.com/aws/aws-cdk-go/awscdk/v2” codebuild “github.com/aws/aws-cdk-go/awscdk/v2/awscodebuild” ssm “github.com/aws/aws-cdk-go/awscdk/v2/awsssm” pipeline “github.com/aws/aws-cdk-go/awscdk/v2/pipelines” “github.com/aws/constructs-go/constructs/v10” “github.com/aws/jsii-runtime-go” “os” )

// my CDK Stack with resources func NewCdkStack(scope constructs.Construct, id *string, props* awscdk.StackProps) awscdk.Stack { stack := awscdk.NewStack(scope, id, props)

…. // create an example ssm parameter \_ = ssm.NewStringParameter(stack, jsii.String(“ssm-test-param”), &ssm.StringParameterProps{ ParameterName: jsii.String(“/testparam”), Description: jsii.String(“ssm parameter for demo”), StringValue: jsii.String(“my test param”), })

return stack } ….

// my CDK Application func NewCdkApplication(scope constructs.Construct, id *string, props* awscdk.StageProps) awscdk.Stage { stage := awscdk.NewStage(scope, id, props)

…. \_ = NewCdkStack(stage, jsii.String(“cdk-stack”), &awscdk.StackProps{Env: props.Env})

return stage } ….

// my CDK Pipeline func NewCdkPipeline(scope constructs.Construct, id *string, props* awscdk.StackProps) awscdk.Stack { stack := awscdk.NewStack(scope, id, props)

…. // GitHub repo with owner and repository name githubRepo := pipeline.CodePipelineSource\_GitHub(jsii.String(“owner/repo”), jsii.String(“main”), &pipeline.GitHubSourceOptions{ Authentication: awscdk.SecretValue\_SecretsManager(jsii.String(“my-github-token”), nil), })

// self mutating pipeline myPipeline := pipeline.NewCodePipeline(stack, jsii.String(“cdkPipeline”), &pipeline.CodePipelineProps{ PipelineName: jsii.String(“CdkPipeline”), // self mutation true - pipeline changes itself before application deployment SelfMutation: jsii.Bool(true), CodeBuildDefaults: &pipeline.CodeBuildOptions{ BuildEnvironment: &codebuild.BuildEnvironment{ // image version 6.0 recommended for newer go version BuildImage: codebuild.LinuxBuildImage\_FromCodeBuildImageId(jsii.String(“aws/codebuild/standard:6.0”)), }, }, Synth: pipeline.NewCodeBuildStep(jsii.String(“Synth”), &pipeline.CodeBuildStepProps{ Input: githubRepo, Commands: &[]\*string{ jsii.String(“npm install -g aws-cdk”), jsii.String(“cdk synth”), }, }), })

// deployment of actual CDK application myPipeline.AddStage(NewCdkApplication(stack, jsii.String(“MyApplication”), &awscdk.StageProps{ Env: targetAccountEnv(), }), &pipeline.AddStageOpts{ Post: &[]pipeline.Step{ pipeline.NewCodeBuildStep(jsii.String(“Manual Steps”), &pipeline.CodeBuildStepProps{ Commands: &[]\*string{ jsii.String(“echo "My CDK App deployed, manual steps go here … "”), }, }), }, })

return stack } ….

// main app func main() { defer jsii.Close()

…. app := awscdk.NewApp(nil)

// call CDK Pipeline NewCdkPipeline(app, jsii.String(“CdkPipelineStack”), &awscdk.StackProps{ Env: pipelineEnv(), })

app.Synth(nil) } ….

// env determines the {aws} environment (account+region) in which our stack is to // be deployed. For more information see: https://docs.aws.amazon.com/cdk/latest/guide/environments.html func pipelineEnv() \*awscdk.Environment { return &awscdk.Environment{ Account: jsii.String(os.Getenv(“CDK\_DEFAULT\_ACCOUNT”)), Region: jsii.String(os.Getenv(“CDK\_DEFAULT\_REGION”)), } }

func targetAccountEnv() \*awscdk.Environment { return &awscdk.Environment{ Account: jsii.String(os.Getenv(“CDK\_DEFAULT\_ACCOUNT”)), Region: jsii.String(os.Getenv(“CDK\_DEFAULT\_REGION”)), } } — ====

You must deploy a pipeline manually once. After that, the pipeline keeps itself up to date from the source code repository. So be sure that the code in the repo is the code you want deployed. Check in your changes and push to GitHub, then deploy:

== [source,none,subs=“verbatim,attributes”]

git add –all git commit -m “initial commit” git push cdk deploy —

= [TIP]

Now that you’ve done the initial deployment, your local {aws} account no longer needs administrative access. This is because all changes to your app will be deployed via the pipeline. All you need to be able to do is push to GitHub.

====

[#cdk-pipeline-stages] == Application stages

To define a multi-stack {aws} application that can be added to the pipeline all at once, define a subclass of +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stage.html[Stage]+. (This is different from CdkStage in the CDK Pipelines module.)

The stage contains the stacks that make up your application. If there are dependencies between the stacks, the stacks are automatically added to the pipeline in the right order. Stacks that don’t depend on each other are deployed in parallel. You can add a dependency relationship between stacks by calling stack1.addDependency(stack2).

Stages accept a default env argument, which becomes the default environment for the stacks inside it. (Stacks can still have their own environment specified.).

An application is added to the pipeline by calling +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.CodePipeline.html#addwbrstagestage-optionss[addStage()]+ with instances of https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stage.html[Stage]. A stage can be instantiated and added to the pipeline multiple times to define different stages of your DTAP or multi-Region application pipeline.

We will create a stack containing a simple Lambda function and place that stack in a stage. Then we will add the stage to the pipeline so it can be deployed.

==== [role=“tablist”] TypeScript:: Create the new file lib/my-pipeline-lambda-stack.ts to hold our application stack containing a Lambda function. + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import { Function, InlineCode, Runtime } from ‘aws-cdk-lib/aws-lambda’;

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

new Function(this, ‘LambdaFunction’, { runtime: Runtime.NODEJS\_18\_X, handler: ‘index.handler’, code: new InlineCode(‘exports.handler = \_ => “Hello, CDK”;’) }); } } —- + Create the new file lib/my-pipeline-app-stage.ts to hold our stage. + [source,javascript,subs=“verbatim,attributes”] —- import \* as cdk from ‘aws-cdk-lib’; import { Construct } from “constructs”; import { MyLambdaStack } from ‘./my-pipeline-lambda-stack’;

export class MyPipelineAppStage extends cdk.Stage {

…. constructor(scope: Construct, id: string, props?: cdk.StageProps) { super(scope, id, props);

const lambdaStack = new MyLambdaStack(this, ‘LambdaStack’); } } —- + Edit lib/my-pipeline-stack.ts to add the stage to our pipeline. + [source,javascript,subs=“verbatim,attributes”] —- import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import { CodePipeline, CodePipelineSource, ShellStep } from ‘aws-cdk-lib/pipelines’; import { MyPipelineAppStage } from ‘./my-pipeline-app-stage’; ….

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

…. const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: new ShellStep(‘Synth’, { input: CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’), commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’] }) });

pipeline.addStage(new MyPipelineAppStage(this, “test”, { env: { account: “111111111111”, region: “eu-west-1” } })); } } —- ….

JavaScript:: Create the new file lib/my-pipeline-lambda-stack.js to hold our application stack containing a Lambda function. + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const { Function, InlineCode, Runtime } = require(‘aws-cdk-lib/aws-lambda’);

class MyLambdaStack extends cdk.Stack { constructor(scope, id, props) { super(scope, id, props);

new Function(this, ‘LambdaFunction’, { runtime: Runtime.NODEJS\_18\_X, handler: ‘index.handler’, code: new InlineCode(‘exports.handler = \_ => “Hello, CDK”;’) }); } }

== module.exports = { MyLambdaStack }

* Create the new file lib/my-pipeline-app-stage.js to hold our stage.

## [source,javascript,subs=“verbatim,attributes”]

* const cdk = require(‘aws-cdk-lib’); const { MyLambdaStack } = require(‘./my-pipeline-lambda-stack’);

class MyPipelineAppStage extends cdk.Stage {

…. constructor(scope, id, props) { super(scope, id, props);

const lambdaStack = new MyLambdaStack(this, ‘LambdaStack’); } } ….

== module.exports = { MyPipelineAppStage };

* Edit lib/my-pipeline-stack.ts to add the stage to our pipeline.

## [source,javascript,subs=“verbatim,attributes”]

* const cdk = require(‘aws-cdk-lib’); const { CodePipeline, CodePipelineSource, ShellStep } = require(‘aws-cdk-lib/pipelines’); const { MyPipelineAppStage } = require(‘./my-pipeline-app-stage’);

class MyPipelineStack extends cdk.Stack { constructor(scope, id, props) { super(scope, id, props);

…. const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: new ShellStep(‘Synth’, { input: CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’), commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’] }) });

pipeline.addStage(new MyPipelineAppStage(this, “test”, { env: { account: “111111111111”, region: “eu-west-1” } })); ….

} }

== module.exports = { MyPipelineStack }

Python:: Create the new file my\_pipeline/my\_pipeline\_lambda\_stack.py to hold our application stack containing a Lambda function. + + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk from constructs import Construct from aws\_cdk.aws\_lambda import Function, InlineCode, Runtime

class MyLambdaStack(cdk.Stack): def *init*(self, scope: Construct, construct\_id: str, **kwargs) -> None: super().*init*(scope, construct\_id,** kwargs)

Function(self, "LambdaFunction",  
 runtime=Runtime.NODEJS\_18\_X,  
 handler="index.handler",  
 code=InlineCode("exports.handler = \_ => 'Hello, CDK';")  
 ) ---- + Create the new file `my\_pipeline/my\_pipeline\_app\_stage.py` to hold our stage. + [source,python,subs="verbatim,attributes"] ---- import aws\_cdk as cdk from constructs import Construct from my\_pipeline.my\_pipeline\_lambda\_stack import MyLambdaStack

class MyPipelineAppStage(cdk.Stage): def *init*(self, scope: Construct, construct\_id: str, **kwargs) -> None: super().*init*(scope, construct\_id,** kwargs)

lambdaStack = MyLambdaStack(self, "LambdaStack") ---- + Edit `my\_pipeline/my-pipeline-stack.py` to add the stage to our pipeline. + [source,python,subs="verbatim,attributes"] ---- import aws\_cdk as cdk from constructs import Construct from aws\_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep from my\_pipeline.my\_pipeline\_app\_stage import MyPipelineAppStage

class MyPipelineStack(cdk.Stack):

…. def **init**(self, scope: Construct, construct\_id: str, \*\*kwargs) -> None: super().\_\_init\_\_(scope, construct\_id, \*\*kwargs)

pipeline = CodePipeline(self, "Pipeline",  
 pipeline\_name="MyPipeline",  
 synth=ShellStep("Synth",  
 input=CodePipelineSource.git\_hub("OWNER/REPO", "main"),  
 commands=["npm install -g aws-cdk",  
 "python -m pip install -r requirements.txt",  
 "cdk synth"]))  
  
pipeline.add\_stage(MyPipelineAppStage(self, "test",  
 env=cdk.Environment(account="111111111111", region="eu-west-1"))) ----

….

Java:: Create the new file src/main/java/com.myorg/MyPipelineLambdaStack.java to hold our application stack containing a Lambda function. + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps;

import software.amazon.awscdk.services.lambda.Function; import software.amazon.awscdk.services.lambda.Runtime; import software.amazon.awscdk.services.lambda.InlineCode;

public class MyPipelineLambdaStack extends Stack { public MyPipelineLambdaStack(final Construct scope, final String id) { this(scope, id, null); }

…. public MyPipelineLambdaStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

Function.Builder.create(this, "LambdaFunction")  
 .runtime(Runtime.NODEJS\_18\_X)  
 .handler("index.handler")  
 .code(new InlineCode("exports.handler = \_ => 'Hello, CDK';"))  
 .build();

} ….

== }

* Create the new file src/main/java/com.myorg/MyPipelineAppStage.java to hold our stage.

## [source,java,subs=“verbatim,attributes”]

* package com.myorg;

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.Stage; import software.amazon.awscdk.StageProps;

public class MyPipelineAppStage extends Stage { public MyPipelineAppStage(final Construct scope, final String id) { this(scope, id, null); }

…. public MyPipelineAppStage(final Construct scope, final String id, final StageProps props) { super(scope, id, props);

Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");

} ….

== }

* Edit src/main/java/com.myorg/MyPipelineStack.java to add the stage to our pipeline.

## [source,java,subs=“verbatim,attributes”]

* package com.myorg;

import java.util.Arrays; import software.constructs.Construct; import software.amazon.awscdk.Environment; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.StageProps; import software.amazon.awscdk.pipelines.CodePipeline; import software.amazon.awscdk.pipelines.CodePipelineSource; import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack { public MyPipelineStack(final Construct scope, final String id) { this(scope, id, null); }

…. public MyPipelineStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")  
 .pipelineName("MyPipeline")  
 .synth(ShellStep.Builder.create("Synth")  
 .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))  
 .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))  
 .build())  
 .build();  
  
pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()  
 .env(Environment.builder()  
 .account("111111111111")  
 .region("eu-west-1")  
 .build())  
 .build()));

} } —- ….

C#:: Create the new file src/MyPipeline/MyPipelineLambdaStack.cs to hold our application stack containing a Lambda function. + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.{aws}.Lambda;

namespace MyPipeline { class MyPipelineLambdaStack : Stack { public MyPipelineLambdaStack(Construct scope, string id, StackProps props=null) : base(scope, id, props) { new Function(this, “LambdaFunction”, new FunctionProps { Runtime = Runtime.NODEJS\_18\_X, Handler = “index.handler”, Code = new InlineCode(“exports.handler = \_ => ‘Hello, CDK’;”) }); } } } — + Create the new file src/MyPipeline/MyPipelineAppStage.cs to hold our stage. + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs;

namespace MyPipeline { class MyPipelineAppStage : Stage { public MyPipelineAppStage(Construct scope, string id, StageProps props=null) : base(scope, id, props) { Stack lambdaStack = new MyPipelineLambdaStack(this, “LambdaStack”); } } } — + Edit src/MyPipeline/MyPipelineStack.cs to add the stage to our pipeline. + + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.Pipelines;

namespace MyPipeline { public class MyPipelineStack : Stack { internal MyPipelineStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { var pipeline = new CodePipeline(this, “pipeline”, new CodePipelineProps { PipelineName = “MyPipeline”, Synth = new ShellStep(“Synth”, new ShellStepProps { Input = CodePipelineSource.GitHub(“OWNER/REPO”, “main”), Commands = new string[] { “npm install -g aws-cdk”, “cdk synth” } }) });

pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps  
 {  
 Env = new Environment  
 {  
 Account = "111111111111", Region = "eu-west-1"  
 }  
 }));  
 }

} } —- ====

Every application stage added by addStage() results in the addition of a corresponding pipeline stage, represented by a +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.StageDeployment.html[StageDeployment]+ instance returned by the addStage() call. You can add pre-deployment or post-deployment actions to the stage by calling its addPre() or addPost() method.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // import { ManualApprovalStep } from ‘aws-cdk-lib/pipelines’;

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, ‘testing’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } }));

testingStage.addPost(new ManualApprovalStep(‘approval’)); —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // const { ManualApprovalStep } = require(‘aws-cdk-lib/pipelines’);

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, ‘testing’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } }));

== testingStage.addPost(new ManualApprovalStep(‘approval’));

Python:: + [source,python,subs=“verbatim,attributes”] —

= from aws\_cdk.pipelines import ManualApprovalStep

testing\_stage = pipeline.add\_stage(MyPipelineAppStage(self, “testing”, env=cdk.Environment(account=“111111111111”, region=“eu-west-1”)))

== testing\_stage.add\_post(ManualApprovalStep(‘approval’))

Java:: + [source,java,subs=“verbatim,attributes”] — // import software.amazon.awscdk.pipelines.StageDeployment; // import software.amazon.awscdk.pipelines.ManualApprovalStep;

StageDeployment testingStage = pipeline.addStage(new MyPipelineAppStage(this, “test”, StageProps.builder() .env(Environment.builder() .account(“111111111111”) .region(“eu-west-1”) .build()) .build()));

== testingStage.addPost(new ManualApprovalStep(“approval”));

C#:: + [source,csharp,subs=“verbatim,attributes”] — var testingStage = pipeline.AddStage(new MyPipelineAppStage(this, “test”, new StageProps { Env = new Environment { Account = “111111111111”, Region = “eu-west-1” } }));

== testingStage.AddPost(new ManualApprovalStep(“approval”));

====

You can add stages to a +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.Wave.html[Wave]+ to deploy them in parallel, for example when deploying a stage to multiple accounts or Regions.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const wave = pipeline.addWave(‘wave’); wave.addStage(new MyApplicationStage(this, ‘MyAppEU’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } })); wave.addStage(new MyApplicationStage(this, ‘MyAppUS’, { env: { account: ‘111111111111’, region: ‘us-west-1’ } })); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const wave = pipeline.addWave(‘wave’); wave.addStage(new MyApplicationStage(this, ‘MyAppEU’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } })); wave.addStage(new MyApplicationStage(this, ‘MyAppUS’, { env: { account: ‘111111111111’, region: ‘us-west-1’ } })); —

Python:: + [source,python,subs=“verbatim,attributes”] — wave = pipeline.add\_wave(“wave”) wave.add\_stage(MyApplicationStage(self, “MyAppEU”, env=cdk.Environment(account=“111111111111”, region=“eu-west-1”))) wave.add\_stage(MyApplicationStage(self, “MyAppUS”, env=cdk.Environment(account=“111111111111”, region=“us-west-1”))) —

Java:: + [source,java,subs=“verbatim,attributes”] — // import software.amazon.awscdk.pipelines.Wave; final Wave wave = pipeline.addWave(“wave”); wave.addStage(new MyPipelineAppStage(this, “MyAppEU”, StageProps.builder() .env(Environment.builder() .account(“111111111111”) .region(“eu-west-1”) .build()) .build())); wave.addStage(new MyPipelineAppStage(this, “MyAppUS”, StageProps.builder() .env(Environment.builder() .account(“111111111111”) .region(“us-west-1”) .build()) .build())); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var wave = pipeline.AddWave(“wave”); wave.AddStage(new MyPipelineAppStage(this, “MyAppEU”, new StageProps { Env = new Environment { Account = “111111111111”, Region = “eu-west-1” } })); wave.AddStage(new MyPipelineAppStage(this, “MyAppUS”, new StageProps { Env = new Environment { Account = “111111111111”, Region = “us-west-1” } })); — ====

[#cdk-pipeline-validation] == Testing deployments

You can add steps to a CDK Pipeline to validate the deployments that you’re performing. For example, you can use the CDK Pipeline library’s +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.ShellStep.html[ShellStep]+ to perform tasks such as the following:

* Trying to access a newly deployed Amazon API Gateway backed by a Lambda function
* Checking a setting of a deployed resource by issuing an {aws} CLI command

In its simplest form, adding validation actions looks like this:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // stage was returned by pipeline.addStage

stage.addPost(new ShellStep(“validate”, { commands: [‘../tests/validate.sh’], })); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // stage was returned by pipeline.addStage

stage.addPost(new ShellStep(“validate”, { commands: [‘../tests/validate.sh’], })); —

Python:: + [source,python,subs=“verbatim,attributes”] —

= stage was returned by pipeline.add\_stage

stage.add\_post(ShellStep(“validate”, commands=[’‘../tests/validate.sh’’] )) —

Java:: + [source,java,subs=“verbatim,attributes”] — // stage was returned by pipeline.addStage

stage.addPost(ShellStep.Builder.create(“validate”) .commands(Arrays.asList(“‘../tests/validate.sh’”)) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // stage was returned by pipeline.addStage

stage.AddPost(new ShellStep(“validate”, new ShellStepProps { Commands = new string[] { “‘../tests/validate.sh’” } })); — ====

Many {aws} CloudFormation deployments result in the generation of resources with unpredictable names. Because of this, CDK Pipelines provide a way to read {aws} CloudFormation outputs after a deployment. This makes it possible to pass (for example) the generated URL of a load balancer to a test action.

To use outputs, expose the CfnOutput object you’re interested in. Then, pass it in a step’s envFromCfnOutputs property to make it available as an environment variable within that step.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // given a stack lbStack that exposes a load balancer construct as loadBalancer this.loadBalancerAddress = new cdk.CfnOutput(lbStack, ‘LbAddress’, { value: https://${lbStack.loadBalancer.loadBalancerDnsName}/ });

// pass the load balancer address to a shell step stage.addPost(new ShellStep(“lbaddr”, { envFromCfnOutputs: {lb\_addr: lbStack.loadBalancerAddress}, commands: [‘echo $lb\_addr’] })); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // given a stack lbStack that exposes a load balancer construct as loadBalancer this.loadBalancerAddress = new cdk.CfnOutput(lbStack, ‘LbAddress’, { value: https://${lbStack.loadBalancer.loadBalancerDnsName}/ });

// pass the load balancer address to a shell step stage.addPost(new ShellStep(“lbaddr”, { envFromCfnOutputs: {lb\_addr: lbStack.loadBalancerAddress}, commands: [‘echo $lb\_addr’] })); —

Python:: + [source,python,subs=“verbatim,attributes”] —

= given a stack lb\_stack that exposes a load balancer construct as load\_balancer

self.load\_balancer\_address = cdk.CfnOutput(lb\_stack, “LbAddress”, value=f”https://{lb\_stack.load\_balancer.load\_balancer\_dns\_name}/“)

= pass the load balancer address to a shell step

stage.add\_post(ShellStep(“lbaddr”, env\_from\_cfn\_outputs={“lb\_addr”: lb\_stack.load\_balancer\_address} commands=[“echo $lb\_addr”])) —

Java:: + [source,java,subs=“verbatim,attributes”] — // given a stack lbStack that exposes a load balancer construct as loadBalancer loadBalancerAddress = CfnOutput.Builder.create(lbStack, “LbAddress”) .value(String.format(“https://%s/”, lbStack.loadBalancer.loadBalancerDnsName)) .build();

stage.addPost(ShellStep.Builder.create(“lbaddr”) .envFromCfnOutputs( // Map.of requires Java 9 or later java.util.Map.of(“lbAddr”, loadBalancerAddress)) .commands(Arrays.asList(“echo $lbAddr”)) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // given a stack lbStack that exposes a load balancer construct as loadBalancer loadBalancerAddress = new CfnOutput(lbStack, “LbAddress”, new CfnOutputProps { Value = string.Format(“https://{0}/”, lbStack.loadBalancer.LoadBalancerDnsName) });

stage.AddPost(new ShellStep(“lbaddr”, new ShellStepProps { EnvFromCfnOutputs = new Dictionary<string, CfnOutput> { { “lbAddr”, loadBalancerAddress } }, Commands = new string[] { “echo $lbAddr” } })); — ====

You can write simple validation tests right in the ShellStep, but this approach becomes unwieldy when the test is more than a few lines. For more complex tests, you can bring additional files (such as complete shell scripts, or programs in other languages) into the ShellStep via the inputs property. The inputs can be any step that has an output, including a source (such as a GitHub repo) or another ShellStep.

Bringing in files from the source repository is appropriate if the files are directly usable in the test (for example, if they are themselves executable). In this example, we declare our GitHub repo as source (rather than instantiating it inline as part of the CodePipeline). Then, we pass this fileset to both the pipeline and the validation test.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const source = CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’);

const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: new ShellStep(‘Synth’, { input: source, commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’] }) });

const stage = pipeline.addStage(new MyPipelineAppStage(this, ‘test’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } }));

stage.addPost(new ShellStep(‘validate’, { input: source, commands: [‘sh ../tests/validate.sh’] })); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const source = CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’);

const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: new ShellStep(‘Synth’, { input: source, commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’] }) });

const stage = pipeline.addStage(new MyPipelineAppStage(this, ‘test’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } }));

stage.addPost(new ShellStep(‘validate’, { input: source, commands: [‘sh ../tests/validate.sh’] })); —

Python:: + [source,python,subs=“verbatim,attributes”] — source = CodePipelineSource.git\_hub(“OWNER/REPO”, “main”)

pipeline = CodePipeline(self, “Pipeline”, pipeline\_name=“MyPipeline”, synth=ShellStep(“Synth”, input=source, commands=[“npm install -g aws-cdk”, “python -m pip install -r requirements.txt”, “cdk synth”]))

stage = pipeline.add\_stage(MyApplicationStage(self, “test”, env=cdk.Environment(account=“111111111111”, region=“eu-west-1”)))

stage.add\_post(ShellStep(“validate”, input=source, commands=[“sh ../tests/validate.sh”], )) —

Java:: + [source,java,subs=“verbatim,attributes”] — final CodePipelineSource source = CodePipelineSource.gitHub(“OWNER/REPO”, “main”);

final CodePipeline pipeline = CodePipeline.Builder.create(this, “pipeline”) .pipelineName(“MyPipeline”) .synth(ShellStep.Builder.create(“Synth”) .input(source) .commands(Arrays.asList(“npm install -g aws-cdk”, “cdk synth”)) .build()) .build();

final StageDeployment stage = pipeline.addStage(new MyPipelineAppStage(this, “test”, StageProps.builder() .env(Environment.builder() .account(“111111111111”) .region(“eu-west-1”) .build()) .build()));

stage.addPost(ShellStep.Builder.create(“validate”) .input(source) .commands(Arrays.asList(“sh ../tests/validate.sh”)) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var source = CodePipelineSource.GitHub(“OWNER/REPO”, “main”);

var pipeline = new CodePipeline(this, “pipeline”, new CodePipelineProps { PipelineName = “MyPipeline”, Synth = new ShellStep(“Synth”, new ShellStepProps { Input = source, Commands = new string[] { “npm install -g aws-cdk”, “cdk synth” } }) });

var stage = pipeline.AddStage(new MyPipelineAppStage(this, “test”, new StageProps { Env = new Environment { Account = “111111111111”, Region = “eu-west-1” } }));

stage.AddPost(new ShellStep(“validate”, new ShellStepProps { Input = source, Commands = new string[] { “sh ../tests/validate.sh” } })); — ====

Getting the additional files from the synth step is appropriate if your tests need to be compiled, which is done as part of synthesis.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const synthStep = new ShellStep(‘Synth’, { input: CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’), commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’], });

const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: synthStep });

const stage = pipeline.addStage(new MyPipelineAppStage(this, ‘test’, { env: { account: ‘111111111111’, region: ‘eu-west-1’ } }));

// run a script that was transpiled from TypeScript during synthesis stage.addPost(new ShellStep(‘validate’, { input: synthStep, commands: [‘node tests/validate.js’] })); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const synthStep = new ShellStep(‘Synth’, { input: CodePipelineSource.gitHub(‘OWNER/REPO’, ‘main’), commands: [‘npm ci’, ‘npm run build’, ‘npx cdk synth’], });

const pipeline = new CodePipeline(this, ‘Pipeline’, { pipelineName: ‘MyPipeline’, synth: synthStep });

const stage = pipeline.addStage(new MyPipelineAppStage(this, “test”, { env: { account: “111111111111”, region: “eu-west-1” } }));

// run a script that was transpiled from TypeScript during synthesis stage.addPost(new ShellStep(‘validate’, { input: synthStep, commands: [‘node tests/validate.js’] })); —

Python:: + [source,python,subs=“verbatim,attributes”] — synth\_step = ShellStep(“Synth”, input=CodePipelineSource.git\_hub(“OWNER/REPO”, “main”), commands=[“npm install -g aws-cdk”, “python -m pip install -r requirements.txt”, “cdk synth”])

pipeline = CodePipeline(self, “Pipeline”, pipeline\_name=“MyPipeline”, synth=synth\_step)

stage = pipeline.add\_stage(MyApplicationStage(self, “test”, env=cdk.Environment(account=“111111111111”, region=“eu-west-1”)))

= run a script that was compiled during synthesis

stage.add\_post(ShellStep(“validate”, input=synth\_step, commands=[“node test/validate.js”], )) —

Java:: + [source,java,subs=“verbatim,attributes”] — final ShellStep synth = ShellStep.Builder.create(“Synth”) .input(CodePipelineSource.gitHub(“OWNER/REPO”, “main”)) .commands(Arrays.asList(“npm install -g aws-cdk”, “cdk synth”)) .build();

final CodePipeline pipeline = CodePipeline.Builder.create(this, “pipeline”) .pipelineName(“MyPipeline”) .synth(synth) .build();

final StageDeployment stage = pipeline.addStage(new MyPipelineAppStage(this, “test”, StageProps.builder() .env(Environment.builder() .account(“111111111111”) .region(“eu-west-1”) .build()) .build()));

stage.addPost(ShellStep.Builder.create(“validate”) .input(synth) .commands(Arrays.asList(“node ./tests/validate.js”)) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var synth = new ShellStep(“Synth”, new ShellStepProps { Input = CodePipelineSource.GitHub(“OWNER/REPO”, “main”), Commands = new string[] { “npm install -g aws-cdk”, “cdk synth” } });

var pipeline = new CodePipeline(this, “pipeline”, new CodePipelineProps { PipelineName = “MyPipeline”, Synth = synth });

var stage = pipeline.AddStage(new MyPipelineAppStage(this, “test”, new StageProps { Env = new Environment { Account = “111111111111”, Region = “eu-west-1” } }));

stage.AddPost(new ShellStep(“validate”, new ShellStepProps { Input = synth, Commands = new string[] { “node ./tests/validate.js” } })); — ====

[#cdk-pipeline-security] == Security notes

Any form of continuous delivery has inherent security risks. Under the {aws} https://aws.amazon.com/compliance/shared-responsibility-model/[Shared Responsibility Model], you are responsible for the security of your information in the {aws} Cloud. The CDK Pipelines library gives you a head start by incorporating secure defaults and modeling best practices.

However, by its very nature, a library that needs a high level of access to fulfill its intended purpose cannot assure complete security. There are many attack vectors outside of {aws} and your organization.

In particular, keep in mind the following:

* Be mindful of the software you depend on. Vet all third-party software you run in your pipeline, because it can change the infrastructure that gets deployed.
* Use dependency locking to prevent accidental upgrades. CDK Pipelines respects package-lock.json and yarn.lock to make sure that your dependencies are the ones you expect.
* CDK Pipelines runs on resources created in your own account, and the configuration of those resources is controlled by developers submitting code through the pipeline. Therefore, CDK Pipelines by itself cannot protect against malicious developers trying to bypass compliance checks. If your threat model includes developers writing CDK code, you should have external compliance mechanisms in place like https://aws.amazon.com/blogs/mt/proactively-keep-resources-secure-and-compliant-with-aws-cloudformation-hooks/[{aws} CloudFormation Hooks] (preventive) or https://aws.amazon.com/config/[{aws} Config] (reactive) that the {aws} CloudFormation Execution Role does not have permissions to disable.
* Credentials for production environments should be short-lived. After bootstrapping and initial provisioning, there is no need for developers to have account credentials at all. Changes can be deployed through the pipeline. Reduce the possibility of credentials leaking by not needing them in the first place.

[#cdk-pipeline-troubleshooting] == Troubleshooting

The following issues are commonly encountered while getting started with CDK Pipelines.

*Pipeline: Internal Failure*:: + — CREATE\_FAILED | {aws}::CodePipeline::Pipeline | Pipeline/Pipeline Internal Failure — + Check your GitHub access token. It might be missing, or might not have the permissions to access the repository.

*Key: Policy contains a statement with one or more invalid principals*:: + — CREATE\_FAILED | {aws}::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey Policy contains a statement with one or more invalid principals. — + One of the target environments has not been bootstrapped with the new bootstrap stack. Make sure all your target environments are bootstrapped.

*Stack is in ROLLBACK\_COMPLETE state and cannot be updated*:: + — Stack ++++++is in ROLLBACK\_COMPLETE state and cannot be updated. (Service: AmazonCloudFormation; Status Code: 400; Error Code: ValidationError; Request ID: ...) —- + The stack failed its previous deployment and is in a non-retryable state. Delete the stack from the {aws} CloudFormation console and retry the deployment.++++++

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#aspects] = Aspects and the {aws} CDK :info\_titleabbrev: Aspects :keywords: {aws} CDK, aspects

== [abstract]

## Aspects are a way to apply an operation to all constructs in a given scope. The aspect could modify the constructs, such as by adding tags. Or it could verify something about the state of the constructs, such as making sure that all buckets are encrypted.

// Content start

Aspects are a way to apply an operation to all constructs in a given scope. The aspect could modify the constructs, such as by adding tags. Or it could verify something about the state of the constructs, such as making sure that all buckets are encrypted.

To apply an aspect to a construct and all constructs in the same scope, call +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Aspects.html#static-ofscope[Aspects].of(<SCOPE>).add()+ with a new aspect, as shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Aspects.of(myConstruct).add(new SomeAspect(…)); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Aspects.of(myConstruct).add(new SomeAspect(…)); —

Python:: + [source,python,subs=“verbatim,attributes”] — Aspects.of(my\_construct).add(SomeAspect(…)) —

Java:: + [source,java,subs=“verbatim,attributes”] — Aspects.of(myConstruct).add(new SomeAspect(…)); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Aspects.Of(myConstruct).add(new SomeAspect(…)); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Aspects\_Of(stack).Add(awscdk.NewTag(…)) — ====

The {aws} CDK uses aspects to xref:tagging[tag resources], but the framework can also be used for other purposes. For example, you can use it to validate or change the {aws} CloudFormation resources that are defined for you by higher-level constructs.

[#aspects-detail] == Aspects in detail

Aspects employ the link:https://en.wikipedia.org/wiki/Visitor\_pattern[visitor pattern]. An aspect is a class that implements the following interface.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — interface IAspect { visit(node: IConstruct): void;} —

JavaScript:: JavaScript doesn’t have interfaces as a language feature. Therefore, an aspect is simply an instance of a class having a visit method that accepts the node to be operated on.

Python:: Python doesn’t have interfaces as a language feature. Therefore, an aspect is simply an instance of a class having a visit method that accepts the node to be operated on.

Java:: + [source,java,subs=“verbatim,attributes”] — public interface IAspect { public void visit(Construct node); } —

C#:: + [source,csharp,subs=“verbatim,attributes”] — public interface IAspect { void Visit(IConstruct node); } —

Go:: + [source,go,subs=“verbatim,attributes”] — type IAspect interface { Visit(node constructs.IConstruct) } — ====

When you call +Aspects.of(<SCOPE>).add(...)+, the construct adds the aspect to an internal list of aspects. You can obtain the list with Aspects.of(<SCOPE>).

During the xref:deploy-how-synth-app[prepare phase], the {aws} CDK calls the visit method of the object for the construct and each of its children in top-down order.

The visit method is free to change anything in the construct. In strongly typed languages, cast the received construct to a more specific type before accessing construct-specific properties or methods.

Aspects don’t propagate across Stage construct boundaries, because Stages are self-contained and immutable after definition. Apply aspects on the Stage construct itself (or lower) if you want them to visit constructs inside the Stage.

[#aspects-example] == Example

The following example validates that all buckets created in the stack have versioning enabled. The aspect adds an error annotation to the constructs that fail the validation. This results in the synth operation failing and prevents deploying the resulting cloud assembly.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — class BucketVersioningChecker implements IAspect { public visit(node: IConstruct): void { // See that we’re dealing with a CfnBucket if (node instanceof s3.CfnBucket) {

// Check for versioning property, exclude the case where the property // can be a token (IResolvable). if (!node.versioningConfiguration || (!Tokenization.isResolvable(node.versioningConfiguration) && node.versioningConfiguration.status !== ‘Enabled’)) { Annotations.of(node).addError(‘Bucket versioning is not enabled’); } } } }

// Later, apply to the stack Aspects.of(stack).add(new BucketVersioningChecker()); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class BucketVersioningChecker { visit(node) { // See that we’re dealing with a CfnBucket if ( node instanceof s3.CfnBucket) {

// Check for versioning property, exclude the case where the property // can be a token (IResolvable). if (!node.versioningConfiguration || !Tokenization.isResolvable(node.versioningConfiguration) && node.versioningConfiguration.status !== ‘Enabled’)) { Annotations.of(node).addError(‘Bucket versioning is not enabled’); } } } }

// Later, apply to the stack Aspects.of(stack).add(new BucketVersioningChecker()); —

Python:: + [source,python,subs=“verbatim,attributes”] — @jsii.implements(cdk.IAspect) class BucketVersioningChecker:

def visit(self, node): # See that we’re dealing with a CfnBucket if isinstance(node, s3.CfnBucket):

# Check for versioning property, exclude the case where the property # can be a token (IResolvable). if (not node.versioning\_configuration or not Tokenization.is\_resolvable(node.versioning\_configuration) and node.versioning\_configuration.status != “Enabled”): Annotations.of(node).add\_error(‘Bucket versioning is not enabled’)

= Later, apply to the stack

## Aspects.of(stack).add(BucketVersioningChecker())

Java:: + [source,java,subs=“verbatim,attributes”] — public class BucketVersioningChecker implements IAspect { @Override public void visit(Construct node) { // See that we’re dealing with a CfnBucket if (node instanceof CfnBucket) { CfnBucket bucket = (CfnBucket)node; Object versioningConfiguration = bucket.getVersioningConfiguration(); if (versioningConfiguration == null || !Tokenization.isResolvable(versioningConfiguration.toString()) && !versioningConfiguration.toString().contains(“Enabled”)) Annotations.of(bucket.getNode()).addError(“Bucket versioning is not enabled”); } } }

// Later, apply to the stack Aspects.of(stack).add(new BucketVersioningChecker()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — class BucketVersioningChecker : Amazon.Jsii.Runtime.Deputy.DeputyBase, IAspect { public void Visit(IConstruct node) { // See that we’re dealing with a CfnBucket if (node is CfnBucket) { var bucket = (CfnBucket)node; if (bucket.VersioningConfiguration is null || !Tokenization.IsResolvable(bucket.VersioningConfiguration) && !bucket.VersioningConfiguration.ToString().Contains(“Enabled”)) Annotations.Of(bucket.Node).AddError(“Bucket versioning is not enabled”); } } }

// Later, apply to the stack Aspects.Of(stack).add(new BucketVersioningChecker()); — ====

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#blueprints] = Configure constructs with CDK Blueprints :keywords: {aws} CDK, {aws} CloudFormation, IaC, Infrastructure as code, CDK projects, {aws} CDK concepts, blueprints, property injection, construct properties, organizational standards

== [abstract]

## CDK Blueprints allow organizations to standardize L2 construct configurations through property injection, making it easier for developers to follow best practices while maintaining flexibility.

// Content start

= [NOTE]

CDK Blueprints is in preview release and is subject to change.

====

Use {aws} CDK Blueprints to standardize and distribute L2 construct configurations across your organization. With Blueprints, you can ensure that {aws} resources are configured consistently according to your organizational standards and best practices. For example, you can automatically enable encryption for all Amazon S3 buckets, apply specific logging configurations to all {aws} Lambda functions, or enforce standard security rules for all security groups.

Blueprints are powered by *property injection*, a mechanism introduced in {aws} CDK link:https://github.com/aws/aws-cdk/releases/tag/v2.196.0[v2.196.0] that allows you to modify construct properties at instantiation time. A Blueprint is a collection of property injectors, where each property injector specifies optimal configuration for a specific L2 construct. The Blueprint represents the overall best practices for your organization.

Blueprints are not a compliance enforcement mechanism. Developers can still override the defaults if needed. For strict compliance enforcement, consider using {aws} CloudFormation Guard, Service Control Policies, or CDK Aspects in addition to Blueprints.

For detailed implementation information, see the link:https://github.com/aws/aws-cdk-rfcs/blob/main/text/0693-property-injection.md[Property Injection RFC].

[#blueprints-key-pieces] == Key components of Blueprints

Blueprints are collections of property injectors that apply default properties to constructs when they’re instantiated. A property injector is a component that implements the IPropertyInjector interface, which intercepts construct creation and modifies or adds properties before the construct is created.

* *IPropertyInjector* - An +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.IPropertyInjector.html[IPropertyInjector]+ defines a way to inject additional properties that are not specified in the props. It is specific to one L2 construct and operates on that construct’s properties.
* *PropertyInjectors* - +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.PropertyInjectors.html[PropertyInjectors]+ are a collection of injectors attached to the construct tree. Injectors can be attached to any construct, but in practice we expect most of them will be attached to App, Stage or Stack.

[#blueprints-use-cases] == Common use cases for Blueprints

You can use CDK Blueprints to standardize many aspects of your AWS resources. Here are some common use cases:

*Security standards*:: + –

* Ensure all Amazon S3 buckets have server-side encryption enabled.
* Configure all security groups to block public access by default.
* Apply least-privilege AWS Identity and Access Management (IAM) permissions to {aws} Lambda functions.
* {blank}
* == Enforce SSL for all network communications.

*Operational excellence*:: + –

* Configure standardized logging for all {aws} Lambda functions.
* Apply consistent tagging strategies across resources.
* Set up default monitoring and alerting thresholds.
* {blank}
* == Implement standard retention policies for logs and backups.

*Cost optimization*:: + –

* Configure appropriate instance sizes based on environment.
* Apply auto-scaling policies with organizational defaults.
* Set lifecycle rules for Amazon S3 buckets to transition objects to cheaper storage classes.
* {blank}
* == Configure default provisioned throughput for databases.

*Compliance requirements*:: + –

* Implement required encryption settings for regulated data.
* Apply necessary backup policies for data retention requirements.
* Configure default Amazon VPC settings that meet security requirements.
* {blank}
* == Ensure resources have required tags for cost allocation.

*Developer productivity*:: + –

* Provide sensible defaults that reduce the need for boilerplate code.
* Create organization-specific Stack classes with built-in injectors.
* Share best practices across teams through reusable injectors.
* {blank}
* == Simplify onboarding by encoding organizational knowledge in code.

[#blueprints-getting-started] == Getting started with Blueprints

Here’s a simple example of how to create and use a property injector:

First, create a property injector for Amazon S3 buckets:

== [source,typescript,subs=“verbatim,attributes”]

import { IPropertyInjector, InjectionContext } from ‘aws-cdk-lib’; import { Bucket, BucketProps, BlockPublicAccess } from ‘aws-cdk-lib/aws-s3’;

export class SecureBucketDefaults implements IPropertyInjector { public readonly constructUniqueId: string;

constructor() { this.constructUniqueId = Bucket.PROPERTY\_INJECTION\_ID; }

public inject(originalProps: BucketProps, \_context: InjectionContext): BucketProps { return { // Set security defaults blockPublicAccess: BlockPublicAccess.BLOCK\_ALL, enforceSSL: true,

// Include original props to allow overrides …originalProps, }; } } —-

Then, use the injector in your CDK application:

== [source,typescript,subs=“verbatim,attributes”]

import { App, Stack } from ‘aws-cdk-lib’; import { Bucket } from ‘aws-cdk-lib/aws-s3’; import { SecureBucketDefaults } from ‘./secure-bucket-defaults’;

// Attach injectors when creating the App const app = new App({ propertyInjectors: [new SecureBucketDefaults()] });

const stack = new Stack(app, ‘MyStack’);

// This bucket automatically gets the default properties const myBucket = new Bucket(stack, ‘MyBucket’); —

Alternatively, you can use the PropertyInjectors.of() method:

== [source,typescript,subs=“verbatim,attributes”]

import { App, Stack, PropertyInjectors } from ‘aws-cdk-lib’; import { SecureBucketDefaults } from ‘./secure-bucket-defaults’;

const app = new App(); PropertyInjectors.of(app).add(new SecureBucketDefaults());

const stack = new Stack(app, ‘MyStack’); const myBucket = new Bucket(stack, ‘MyBucket’); —

[#blueprints-best-practices] == Best practices

* Place default properties before +...originalProps+ to allow overrides.
* Place forced properties after +...originalProps+ to prevent overrides.
* Use a skip flag when creating resources to prevent infinite recursion. For an example, see link:https://github.com/aws/aws-cdk-rfcs/blob/main/text/0693-property-injection.md#what-happens-when-you-need-to-create-a-accesslogbucket-for-a-bucket[What happens when you need to create a accessLogBucket for a Bucket?] in the *Property Injection RFC*.
* Add logging for debugging.
* Use CDK context to enable/disable injectors for testing.

For more detailed information about property injection, including implementation details, troubleshooting tips, and reference information, see the link:https://github.com/aws/aws-cdk-rfcs/blob/main/text/0693-property-injection.md[Property Injection RFC].

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#cfn-layer] = Customize constructs from the {aws} Construct Library :info\_titleabbrev: Customize constructs

== [abstract]

## Customize constructs from the {aws} Construct Library through escape hatches, raw overrides, and custom resources.

// Content start

Customize constructs from the {aws} Construct Library through escape hatches, raw overrides, and custom resources.

[#develop-customize-escape] == Use escape hatches

The {aws} Construct Library provides xref:constructs[constructs] of varying levels of abstraction.

At the highest level, your {aws} CDK application and the stacks in it are themselves abstractions of your entire cloud infrastructure, or significant chunks of it. They can be parameterized to deploy them in different environments or for different needs.

Abstractions are powerful tools for designing and implementing cloud applications. The {aws} CDK gives you the power not only to build with its abstractions, but also to create new abstractions. Using the existing open-source L2 and L3 constructs as guidance, you can build your own L2 and L3 constructs to reflect your own organization’s best practices and opinions.

No abstraction is perfect, and even good abstractions cannot cover every possible use case. During development, you may find a construct that almost fits your needs, requiring a small or large customization.

For this reason, the {aws} CDK provides ways to *break out* of the construct model. This includes moving to a lower-level abstraction or to a different model entirely. Escape hatches let you *escape* the {aws} CDK paradigm and customize it in ways that suit your needs. Then, you can wrap your changes in a new construct to abstract away the underlying complexity and provide a clean API for other developers.

The following are examples of situations where you can use escape hatches:

* An {aws} service feature is available through {aws} CloudFormation, but there are no L2 constructs for it.
* An {aws} service feature is available through {aws} CloudFormation, and there are L2 constructs for the service, but these don’t yet expose the feature. Because L2 constructs are curated by the CDK team, they may not be immediately available for new features.
* The feature is not yet available through {aws} CloudFormation at all.
* To determine whether a feature is available through {aws} CloudFormation, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html[{aws} Resource and Property Types Reference].

[#develop-customize-escape-l1] === Develop escape hatches for L1 constructs

If L2 constructs are not available for the service, you can use the automatically generated L1 constructs. These resources can be recognized by their name starting with Cfn, such as CfnBucket or CfnRole. You instantiate them exactly as you would use the equivalent {aws} CloudFormation resource.

For example, to instantiate a low-level Amazon S3 bucket L1 with analytics enabled, you would write something like the following.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new s3.CfnBucket(this, ‘amzn-s3-demo-bucket’, { analyticsConfigurations: [ { id: ‘Config’, // … + } ] }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new s3.CfnBucket(this, ‘amzn-s3-demo-bucket’, { analyticsConfigurations: [ { id: ‘Config’ // … + } ] }); —

Python:: + [source,python,subs=“verbatim,attributes”] — s3.CfnBucket(self, “amzn-s3-demo-bucket”, analytics\_configurations: [ dict(id=“Config”, # … ) ] ) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnBucket.Builder.create(this, “amzn-s3-demo-bucket”) .analyticsConfigurations(Arrays.asList(java.util.Map.of( // Java 9 or later “id”, “Config”, // … ))).build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new CfnBucket(this, ‘amzn-s3-demo-bucket’, new CfnBucketProps { AnalyticsConfigurations = new Dictionary<string, string> { [“id”] = “Config”, // … } }); — ====

There might be rare cases where you want to define a resource that doesn’t have a corresponding CfnXxx class. This could be a new resource type that hasn’t yet been published in the {aws} CloudFormation resource specification. In cases like this, you can instantiate the cdk.CfnResource directly and specify the resource type and properties. This is shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new cdk.CfnResource(this, ‘amzn-s3-demo-bucket’, { type: ‘{aws}::S3::Bucket’, properties: { // Note the PascalCase here! These are CloudFormation identifiers. AnalyticsConfigurations: [ { Id: ‘Config’, // … } ] } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new cdk.CfnResource(this, ‘amzn-s3-demo-bucket’, { type: ‘{aws}::S3::Bucket’, properties: { // Note the PascalCase here! These are CloudFormation identifiers. AnalyticsConfigurations: [ { Id: ‘Config’ // … } ] } }); —

Python:: + [source,python,subs=“verbatim,attributes”] — cdk.CfnResource(self, ‘amzn-s3-demo-bucket’, type=“{aws}::S3::Bucket”, properties=dict( # Note the PascalCase here! These are CloudFormation identifiers. “AnalyticsConfigurations”: [ { “Id”: “Config”, # … } ] ) ) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnResource.Builder.create(this, “amzn-s3-demo-bucket”) .type(“{aws}::S3::Bucket”) .properties(java.util.Map.of( // Map.of requires Java 9 or later // Note the PascalCase here! These are CloudFormation identifiers “AnalyticsConfigurations”, Arrays.asList( java.util.Map.of(“Id”, “Config”, // … )))) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new CfnResource(this, “amzn-s3-demo-bucket”, new CfnResourceProps { Type = “{aws}::S3::Bucket”, Properties = new Dictionary<string, object> { // Note the PascalCase here! These are CloudFormation identifiers [“AnalyticsConfigurations”] = new Dictionary<string, string>[] { new Dictionary<string, string> { [“Id”] = “Config” } } } }); — ====

[#develop-customize-escape-l2] === Develop escape hatches for L2 constructs

If an L2 construct is missing a feature or you’re trying to work around an issue, you can modify the L1 construct that’s encapsulated by the L2 construct.

All L2 constructs contain within them the corresponding L1 construct. For example, the high-level Bucket construct wraps the low-level CfnBucket construct. Because the CfnBucket corresponds directly to the {aws} CloudFormation resource, it exposes all features that are available through {aws} CloudFormation.

The basic approach to get access to the L1 construct is to use construct.node.defaultChild (Python: default\_child), cast it to the right type (if necessary), and modify its properties. Again, let’s take the example of a Bucket.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // Get the CloudFormation resource const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties cfnBucket.analyticsConfiguration = [ { id: ‘Config’, // … + }]; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Get the CloudFormation resource const cfnBucket = bucket.node.defaultChild;

// Change its properties cfnBucket.analyticsConfiguration = [ { id: ‘Config’ // … + }]; —

Python:: + [source,python,subs=“verbatim,attributes”] —

= Get the CloudFormation resource

cfn\_bucket = bucket.node.default\_child

= Change its properties

cfn\_bucket.analytics\_configuration = [ { “id”: “Config”, # … }] —

Java:: + [source,java,subs=“verbatim,attributes”] — // Get the CloudFormation resource CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

cfnBucket.setAnalyticsConfigurations( Arrays.asList(java.util.Map.of( // Java 9 or later “Id”, “Config”, // … )); ) —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // Get the CloudFormation resource var cfnBucket = (CfnBucket)bucket.Node.DefaultChild;

cfnBucket.AnalyticsConfigurations = new List++++++{ new Dictionary<string, string> { [“Id”] = “Config”, // … } }; — ====++++++

You can also use this object to change {aws} CloudFormation options such as Metadata and UpdatePolicy.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — cfnBucket.cfnOptions.metadata = { MetadataKey: ‘MetadataValue’ }; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — cfnBucket.cfnOptions.metadata = { MetadataKey: ‘MetadataValue’ }; —

Python:: + [source,python,subs=“verbatim,attributes”] — cfn\_bucket.cfn\_options.metadata = { “MetadataKey”: “MetadataValue” } —

Java:: + [source,java,subs=“verbatim,attributes”] — cfnBucket.getCfnOptions().setMetadata(java.util.Map.of( // Java 9+ “MetadataKey”, “Metadatavalue”)); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — cfnBucket.CfnOptions.Metadata = new Dictionary<string, object> { [“MetadataKey”] = “Metadatavalue” }; — ====

[#develop-customize-unescape] == Use un-escape hatches

The {aws} CDK also provides the capability to go *up* an abstraction level, which we might refer to as an “un-escape” hatch. If you have an L1 construct, such as CfnBucket, you can create a new L2 construct (Bucket in this case) to wrap the L1 construct.

This is convenient when you create an L1 resource but want to use it with a construct that requires an L2 resource. It’s also helpful when you want to use convenience methods like .grantXxxxx() that aren’t available on the L1 construct.

You move to the higher abstraction level using a static method on the L2 class called .fromCfnXxxxx()–for example, Bucket.fromCfnBucket() for Amazon S3 buckets. The L1 resource is the only parameter.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — b1 = new s3.CfnBucket(this, “buck09”, { … }); b2 = s3.Bucket.fromCfnBucket(b1); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — b1 = new s3.CfnBucket(this, “buck09”, { …} ); b2 = s3.Bucket.fromCfnBucket(b1); —

Python:: + [source,python,subs=“verbatim,attributes”] — b1 = s3.CfnBucket(self, “buck09”, …) b2 = s3.from\_cfn\_bucket(b1) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnBucket b1 = CfnBucket.Builder.create(this, “buck09”) // …. .build(); IBucket b2 = Bucket.fromCfnBucket(b1); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var b1 = new CfnBucket(this, “buck09”, new CfnBucketProps { … }); var v2 = Bucket.FromCfnBucket(b1); — ====

L2 constructs created from L1 constructs are proxy objects that refer to the L1 resource, similar to those created from resource names, ARNs, or lookups. Modifications to these constructs do not affect the final synthesized {aws} CloudFormation template (since you have the L1 resource, however, you can modify that instead). For more information on proxy objects, see xref:resources-external[Referencing resources in your {aws} account].

To avoid confusion, do not create multiple L2 constructs that refer to the same L1 construct. For example, if you extract the CfnBucket from a Bucket using the technique in the xref:develop-customize-escape-l2[previous section], you shouldn’t create a second Bucket instance by calling Bucket.fromCfnBucket() with that CfnBucket. It actually works as you’d expect (only one +{aws}::S3::Bucket+ is synthesized) but it makes your code more difficult to maintain.

[#develop-customize-override] == Use raw overrides

If there are properties that are missing from the L1 construct, you can bypass all typing using raw overrides. This also makes it possible to delete synthesized properties.

Use one of the addOverride methods (Python: add\_override) methods, as shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // Get the CloudFormation resource const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Use dot notation to address inside the resource template fragment cfnBucket.addOverride(‘Properties.VersioningConfiguration.Status’, ‘NewStatus’); cfnBucket.addDeletionOverride(‘Properties.VersioningConfiguration.Status’);

// use index (0 here) to address an element of a list cfnBucket.addOverride(‘Properties.Tags.0.Value’, ‘NewValue’); cfnBucket.addDeletionOverride(‘Properties.Tags.0’);

// addPropertyOverride is a convenience function for paths starting with “Properties.” cfnBucket.addPropertyOverride(‘VersioningConfiguration.Status’, ‘NewStatus’); cfnBucket.addPropertyDeletionOverride(‘VersioningConfiguration.Status’); cfnBucket.addPropertyOverride(‘Tags.0.Value’, ‘NewValue’); cfnBucket.addPropertyDeletionOverride(‘Tags.0’); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Get the CloudFormation resource const cfnBucket = bucket.node.defaultChild ;

// Use dot notation to address inside the resource template fragment cfnBucket.addOverride(‘Properties.VersioningConfiguration.Status’, ‘NewStatus’); cfnBucket.addDeletionOverride(‘Properties.VersioningConfiguration.Status’);

// use index (0 here) to address an element of a list cfnBucket.addOverride(‘Properties.Tags.0.Value’, ‘NewValue’); cfnBucket.addDeletionOverride(‘Properties.Tags.0’);

// addPropertyOverride is a convenience function for paths starting with “Properties.” cfnBucket.addPropertyOverride(‘VersioningConfiguration.Status’, ‘NewStatus’); cfnBucket.addPropertyDeletionOverride(‘VersioningConfiguration.Status’); cfnBucket.addPropertyOverride(‘Tags.0.Value’, ‘NewValue’); cfnBucket.addPropertyDeletionOverride(‘Tags.0’); —

Python:: + [source,python,subs=“verbatim,attributes”] —

= Get the CloudFormation resource

cfn\_bucket = bucket.node.default\_child

= Use dot notation to address inside the resource template fragment

cfn\_bucket.add\_override(“Properties.VersioningConfiguration.Status”, “NewStatus”) cfn\_bucket.add\_deletion\_override(“Properties.VersioningConfiguration.Status”)

= use index (0 here) to address an element of a list

cfn\_bucket.add\_override(“Properties.Tags.0.Value”, “NewValue”) cfn\_bucket.add\_deletion\_override(“Properties.Tags.0”)

= addPropertyOverride is a convenience function for paths starting with “Properties.”

cfn\_bucket.add\_property\_override(“VersioningConfiguration.Status”, “NewStatus”) cfn\_bucket.add\_property\_deletion\_override(“VersioningConfiguration.Status”) cfn\_bucket.add\_property\_override(“Tags.0.Value”, “NewValue”) cfn\_bucket.add\_property\_deletion\_override(“Tags.0”) —

Java:: + [source,java,subs=“verbatim,attributes”] — // Get the CloudFormation resource CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

// Use dot notation to address inside the resource template fragment cfnBucket.addOverride(“Properties.VersioningConfiguration.Status”, “NewStatus”); cfnBucket.addDeletionOverride(“Properties.VersioningConfiguration.Status”);

// use index (0 here) to address an element of a list cfnBucket.addOverride(“Properties.Tags.0.Value”, “NewValue”); cfnBucket.addDeletionOverride(“Properties.Tags.0”);

// addPropertyOverride is a convenience function for paths starting with “Properties.” cfnBucket.addPropertyOverride(“VersioningConfiguration.Status”, “NewStatus”); cfnBucket.addPropertyDeletionOverride(“VersioningConfiguration.Status”); cfnBucket.addPropertyOverride(“Tags.0.Value”, “NewValue”); cfnBucket.addPropertyDeletionOverride(“Tags.0”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // Get the CloudFormation resource var cfnBucket = (CfnBucket)bucket.node.defaultChild;

// Use dot notation to address inside the resource template fragment cfnBucket.AddOverride(“Properties.VersioningConfiguration.Status”, “NewStatus”); cfnBucket.AddDeletionOverride(“Properties.VersioningConfiguration.Status”);

// use index (0 here) to address an element of a list cfnBucket.AddOverride(“Properties.Tags.0.Value”, “NewValue”); cfnBucket.AddDeletionOverride(“Properties.Tags.0”);

// addPropertyOverride is a convenience function for paths starting with “Properties.” cfnBucket.AddPropertyOverride(“VersioningConfiguration.Status”, “NewStatus”); cfnBucket.AddPropertyDeletionOverride(“VersioningConfiguration.Status”); cfnBucket.AddPropertyOverride(“Tags.0.Value”, “NewValue”); cfnBucket.AddPropertyDeletionOverride(“Tags.0”); — ====

[#develop-customize-custom] == Use custom resources

If the feature isn’t available through {aws} CloudFormation, but only through a direct API call, you must write an {aws} CloudFormation Custom Resource to make the API call you need. You can use the {aws} CDK to write custom resources and wrap them into a regular construct interface. From the perspective of a consumer of your construct, the experience will feel native.

Building a custom resource involves writing a Lambda function that responds to a resource’s CREATE, UPDATE, and DELETE lifecycle events. If your custom resource needs to make only a single API call, consider using the https://github.com/awslabs/aws-cdk/tree/master/packages/%40aws-cdk/custom-resources[AwsCustomResource]. This makes it possible to perform arbitrary SDK calls during an {aws} CloudFormation deployment. Otherwise, you should write your own Lambda function to perform the work you need to get done.

The subject is too broad to cover completely here, but the following links should get you started:

* https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-custom-resources.html[Custom Resources]
* https://github.com/aws-samples/aws-cdk-examples/tree/master/typescript/custom-resource/[Custom-Resource Example]
* For a more fully fledged example, see the https://github.com/awslabs/aws-cdk/blob/master/packages/@aws-cdk/aws-certificatemanager/lib/dns-validated-certificate.ts [DnsValidatedCertificate] class in the CDK standard library. This is implemented as a custom resource.

# AWS CDK Pipelines

Learn how to create and manage continuous deployment pipelines for your AWS CDK applications.

## 📚 Complete Pipelines Guide

The comprehensive CDK pipelines tutorial has been organized into focused sections:

**👉** [**Access the Complete Pipelines Guide**](cdk-pipelines/README.md)

## What are CDK Pipelines?

CDK Pipelines is a library that makes it easy to set up continuous deployment pipelines for CDK applications. It automatically handles:

* Building and testing your CDK application
* Deploying to multiple environments
* Managing permissions and security
* Cross-account deployments

## Learning Path

1. [**Introduction**](cdk-pipelines/01-pipeline-introduction.md) - Learn the basics
2. [**Initialize**](cdk-pipelines/03-pipeline-init.md) - Create your first pipeline
3. [**Configure**](cdk-pipelines/04-pipeline-define.md) - Set up stages and deployment
4. [**Advanced Topics**](cdk-pipelines/README.md) - Security, validation, troubleshooting

## Related Topics

* [Stages](stages.md) - Understanding deployment stages
* [Bootstrapping](../04-deployment/bootstrapping.md) - Environment preparation
* [Best Practices](../03-development/best-practices.md) - CDK best practices

include::attributes.txt[]

// Attributes [.topic] [#customize-permissions-boundaries] = Create and apply permissions boundaries for the {aws} CDK :info\_titleabbrev: Create and apply permissions boundaries :keywords: {aws} CDK, IAM, security, permissions, Permissions boundaries

== [abstract]

## A *permissions boundary* is an {aws} Identity and Access Management (IAM) advanced feature that you can use to set the maximum permissions that an IAM entity, such as a user or role, can have. You can use permissions boundaries to restrict the actions that IAM entities can perform when using the {aws} Cloud Development Kit ({aws} CDK).

// Content start A *permissions boundary* is an {aws} Identity and Access Management (IAM) advanced feature that you can use to set the maximum permissions that an IAM entity, such as a user or role, can have. You can use permissions boundaries to restrict the actions that IAM entities can perform when using the {aws} Cloud Development Kit ({aws} CDK).

To learn more about permissions boundaries, see link:https://docs.aws.amazon.com/IAM/latest/UserGuide/access\_policies\_boundaries.html[Permissions boundaries for IAM entities] in the *IAM User Guide*.

[#customize-permissions-boundaries-when] == When to use permissions boundaries with the {aws} CDK

Consider applying permissions boundaries when you need to restrict developers in your organization from performing certain actions with the {aws} CDK. For example, if there are specific resources in your {aws} environment that you don’t want developers to modify, you can create and apply a permissions boundary.

[#customize-permissions-boundaries-how] == How to apply permissions boundaries with the {aws} CDK

[#customize-permissions-boundaries-how-create] === Create the permissions boundary

First, you create the permissions boundary, using an {aws} managed policy or a customer managed policy to set the boundary for an IAM entity (user or role). This policy limits the maximum permissions for the user or role. For instructions on creating permissions boundaries, see link:https://docs.aws.amazon.com/IAM/latest/UserGuide/access\_policies\_boundaries.html[Permissions boundaries for IAM entities] in the *IAM User Guide*.

Permissions boundaries set the maximum permissions that an IAM entity can have, but don’t grant permissions on their own. You must use permissions boundaries with IAM policies to effectively limit and grant the proper permissions for your organization. You must also prevent IAM entities from being able to escape the boundaries that you set. For an example, see link:https://docs.aws.amazon.com/IAM/latest/UserGuide/access\_policies\_boundaries.html#access\_policies\_boundaries-delegate[Delegating responsibility to others using permissions boundaries] in the *IAM User Guide*.

[#customize-permissions-boundaries-how-apply] === Apply the permissions boundary during bootstrapping

After creating the permissions boundary, you can enforce it for the {aws} CDK by applying it during bootstrapping.

Use the xref:ref-cli-cmd-bootstrap-options-custom-permissions-boundary[--custom-permissions-boundary] option and specify the name of the permissions boundary to apply. The following is an example that applies a permissions boundary named cdk-permissions-boundary:

== [source,none,subs=“verbatim,attributes”]

$ cdk bootstrap –custom-permissions-boundary ++++++—-++++++

By default, the CDK uses the CloudFormationExecutionRole IAM role, defined in the bootstrap template, to receive permissions for performing deployments. By applying the custom permissions boundary during bootstrapping, the permissions boundary gets attached to this role. The permissions boundary will then set the maximum permissions that can be performed by developers in your organization when using the {aws} CDK. To learn more about this role, see xref:bootstrapping-env-roles[IAM roles created during bootstrapping].

When you apply permissions boundaries in this way, they are applied to the specific environment that you bootstrap. To use the same permissions boundary across multiple environments, you must apply the permissions boundary for each environment during bootstrapping. You can also apply different permissions boundaries for different environments.

[#customize-permissions-boundaries-learn] == Learn more

For more information on permissions boundaries, see https://aws.amazon.com/blogs/security/when-and-where-to-use-iam-permissions-boundaries/[When and where to use IAM permissions boundaries] in the *{aws} Security Blog*.

include::attributes.txt[]

[.topic] [#customize-synth] = Customize CDK stack synthesis :info\_titleabbrev: Customize CDK synthesis :keywords: {aws} CDK, {aws} CloudFormation stack, synthesis, cdk synth, {aws} CloudFormation template, Customize

== [abstract]

## You can customize {aws} Cloud Development Kit ({aws} CDK) stack synthesis by modifying the default synthesizer, using other available built-in synthesizers, or create your own synthesizer.

// Content start

You can customize {aws} Cloud Development Kit ({aws} CDK) stack synthesis by modifying the default synthesizer, using other available built-in synthesizers, or creating your own synthesizer.

The {aws} CDK includes the following built-in synthesizers that you can use to customize synthesis behavior:

* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.DefaultStackSynthesizer.html[DefaultStackSynthesizer] – If you don`’t specify a synthesizer, this one is used automatically. It supports cross-account deployments and deployments using the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines-readme.html[CDK Pipelines] construct. Its bootstrap contract requires an existing Amazon S3 bucket with a known name, an existing Amazon ECR repository with a known name, and five existing IAM roles with known names. The default bootstrapping template meets these requirements.
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CliCredentialsStackSynthesizer.html[CliCredentialsStackSynthesizer] – This synthesizer’s bootstrap contract requires an existing Amazon S3 bucket and existing Amazon ECR repository. It does not require any IAM roles. To perform deployments, this synthesizer relies on the permissions of the CDK CLI user and is recommend for organizations that want to restrict IAM deployment credentials. This synthesizer doesn’t support cross-account deployments or CDK Pipelines.
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.LegacyStackSynthesizer.html[LegacyStackSynthesizer] – This synthesizer emulates CDK v1 synthesis behavior. Its bootstrap contract requires an existing Amazon S3 bucket of an arbitrary name and expects the locations of assets to be passed in as CloudFormation stack parameters. If you use this synthesizer, you must use the CDK CLI to perform deployment.

If none of these built-in synthesizers are appropriate for your use case, you can write your own synthesizer as a class that implements IStackSynthesizer or look at link:https://constructs.dev/search?q=synthesizer&cdk=aws-cdk[synthesizers] from the Construct Hub.

[#bootstrapping-custom-synth-default] == Customize the DefaultStackSynthesizer

The DefaultStackSynthesizer is the default synthesizer for the {aws} CDK. It is designed to allow cross-account deployments of CDK applications, as well as deploying CDK apps from a CI/CD system that does not have explicit support for the {aws} CDK, but supports regular CloudFormation deployments, such as {aws} CodePipeline. This synthesizer is the best option for most use cases.

[#bootstrapping-custom-synth-default-contract] === DefaultStackSynthesizer bootstrap contract

DefaultStackSynthesizer requires the following bootstrap contract. These are the resources that must be created during bootstrapping:

[cols=“1,1,1,1”, frame=“all”, options=“header”] |=== | Bootstrap resource | Description | Default expected resource name | Purpose

|=== | Amazon S3 bucket | Staging bucket | cdk-hnb659fds-assets-++++++-++++++++++++++++++ | Stores file assets. |===

|=== | Amazon ECR repository | Staging repository | cdk-hnb659fds-container-assets-++++++-++++++++++++++++++ | Stores and manages Docker image assets. |===

|=== | IAM role | Deploy role | cdk-hnb659fds-deploy-role-++++++-++++++++++++++++++ | Assumed by the CDK CLI and potentially CodePipeline to assume other roles and start the {aws} CloudFormation deployment. |===

The trust policy of this role controls who can deploy with the {aws} CDK in this {aws} environment.

|=== | IAM role | {aws} CloudFormation execution role | cdk-hnb659fds-cfn-exec-role-++++++-++++++++++++++++++ | This role is used by {aws} CloudFormation to perform the deployment. |===

The policies of this role control what operations the CDK deployment can perform.

|=== | IAM role | Lookup role | cdk-hnb659fds-lookup-role-++++++-++++++++++++++++++ | This role is used when the CDK CLI needs to perform environmental context lookups. |===

The trust policy of this role controls who can look up information in the environment.

|=== | IAM role | File publishing role | cdk-hnb659fds-file-publishing-role-++++++-++++++++++++++++++ | This role is used to upload assets to the Amazon S3 staging bucket. It is assumed from the deploy role. |===

|=== | IAM role | Image publishing role | cdk-hnb659fds-image-publishing-role-++++++-++++++++++++++++++ | This role is used to upload Docker images to the Amazon ECR staging repository. It is assumed from the deploy role. |===

|=== | SSM parameter | Bootstrap version parameter | /cdk-bootstrap/hnb659fds/++++++++++++ | The version of the bootstrap template. It is used by the bootstrap template and the CDK CLI to validate requirements. |===

One way to customize CDK stack synthesis, is by modifying the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.DefaultStackSynthesizer.html[DefaultStackSynthesizer]. You can customize this synthesizer for a single CDK stack using the synthesizer property of your Stack instance. You can also modify DefaultStackSynthesizer for all stacks in your CDK app using the defaultStackSynthesizer property of your App instance.

[#bootstrapping-custom-synth-qualifiers] === Change the qualifier

The *qualifier* is added to the name of resources created during bootstrapping. By default, this value is hnb659fds. When you modify the qualifier during bootstrapping, you need to customize CDK stack synthesis to use the same qualifier.

To change the qualifier, configure the qualifier property of DefaultStackSynthesizer or configure the qualifier as a context key in your CDK project'scdk.json` file.

The following is an example of configuring the qualifier property of the DefaultStackSynthesizer:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new MyStack(this, ‘MyStack’, { synthesizer: new DefaultStackSynthesizer({ qualifier: ‘MYQUALIFIER’, }), }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new MyStack(this, ‘MyStack’, { synthesizer: new DefaultStackSynthesizer({ qualifier: ‘MYQUALIFIER’, }), }) —

Python:: + [source,python,subs=“verbatim,attributes”] — MyStack(self, “MyStack”, synthesizer=DefaultStackSynthesizer( qualifier=“MYQUALIFIER” )) —

Java:: + [source,java,subs=“verbatim,attributes”] — new MyStack(app, “MyStack”, StackProps.builder() .synthesizer(DefaultStackSynthesizer.Builder.create() .qualifier(“MYQUALIFIER”) .build()) .build(); ) —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new MyStack(app, “MyStack”, new StackProps { Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps { Qualifier = “MYQUALIFIER” }) }); —

Go:: + [source,go,subs=“verbatim,attributes”] — func NewMyStack(scope constructs.Construct, id string, props \*MyStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } stack := awscdk.NewStack(scope, &id, &sprops)

…. synth := awscdk.NewDefaultStackSynthesizer(&awscdk.DefaultStackSynthesizerProps{ Qualifier: jsii.String(“MYQUALIFIER”), })

stack.SetSynthesizer(synth)

return stack } —- ==== ….

The following is an example of configuring the qualifier as a context key in cdk.json:

== [source,json,subs=“verbatim,attributes”]

{ “app”: “…”, “context”: { “@aws-cdk/core:bootstrapQualifier”: “MYQUALIFIER” } } —

[#bootstrapping-custom-synth-names] === Change resource names

All of the other DefaultStackSynthesizer properties relate to the names of the resources in the bootstrap template. You only need to provide any of these properties if you modified the bootstrap template and changed the resource names or naming scheme.

All properties accept the special placeholders +${Qualifier}+, ${AWS::Partition}, ${AWS::AccountId}, and ${AWS::Region}. These placeholders are replaced with the values of the qualifier parameter and the {aws} partition, account ID, and {aws} Region values for the stack’s environment, respectively.

The following example shows the most commonly used properties for DefaultStackSynthesizer along with their default values, as if you were instantiating the synthesizer. For a complete list, see link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.DefaultStackSynthesizerProps.html#properties[DefaultStackSynthesizerProps]:

==== [role=“tablist”] TypeScript:: + [source,javascript] — new DefaultStackSynthesizer({ // Name of the S3 bucket for file assets fileAssetsBucketName: ‘cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}’, bucketPrefix: ’’,

// Name of the ECR repository for Docker image assets imageAssetsRepositoryName: ‘cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}’, dockerTagPrefix: ’’,

// ARN of the role assumed by the CLI and Pipeline to deploy here deployRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}’, deployRoleExternalId: ’’,

// ARN of the role used for file asset publishing (assumed from the CLI role) fileAssetPublishingRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}’, fileAssetPublishingExternalId: ’’,

// ARN of the role used for Docker asset publishing (assumed from the CLI role) imageAssetPublishingRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}’, imageAssetPublishingExternalId: ’’,

// ARN of the role passed to CloudFormation to execute the deployments cloudFormationExecutionRole: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}’,

// ARN of the role used to look up context information in an environment lookupRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}’, lookupRoleExternalId: ’’,

// Name of the SSM parameter which describes the bootstrap stack version number bootstrapStackVersionSsmParameter: ‘/cdk-bootstrap/${Qualifier}/version’,

// Add a rule to every template which verifies the required bootstrap stack version generateBootstrapVersionRule: true,

== })

JavaScript:: + [source,javascript] — new DefaultStackSynthesizer({ // Name of the S3 bucket for file assets fileAssetsBucketName: ‘cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}’, bucketPrefix: ’’,

// Name of the ECR repository for Docker image assets imageAssetsRepositoryName: ‘cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}’, dockerTagPrefix: ’’,

// ARN of the role assumed by the CLI and Pipeline to deploy here deployRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}’, deployRoleExternalId: ’’,

// ARN of the role used for file asset publishing (assumed from the CLI role) fileAssetPublishingRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}’, fileAssetPublishingExternalId: ’’,

// ARN of the role used for Docker asset publishing (assumed from the CLI role) imageAssetPublishingRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}’, imageAssetPublishingExternalId: ’’,

// ARN of the role passed to CloudFormation to execute the deployments cloudFormationExecutionRole: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}’,

// ARN of the role used to look up context information in an environment lookupRoleArn: ‘arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}’, lookupRoleExternalId: ’’,

// Name of the SSM parameter which describes the bootstrap stack version number bootstrapStackVersionSsmParameter: ‘/cdk-bootstrap/${Qualifier}/version’,

// Add a rule to every template which verifies the required bootstrap stack version generateBootstrapVersionRule: true, }) —

Python:: + [source,python] — DefaultStackSynthesizer( # Name of the S3 bucket for file assets file\_assets\_bucket\_name=“cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}”, bucket\_prefix=““,

# Name of the ECR repository for Docker image assets

image\_assets\_repository\_name=“cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}”, docker\_tag\_prefix=““,

# ARN of the role assumed by the CLI and Pipeline to deploy here

deploy\_role\_arn=“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}”, deploy\_role\_external\_id=““,

# ARN of the role used for file asset publishing (assumed from the CLI role)

file\_asset\_publishing\_role\_arn=“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}”, file\_asset\_publishing\_external\_id=““,

# ARN of the role used for Docker asset publishing (assumed from the CLI role)

image\_asset\_publishing\_role\_arn=“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}”, image\_asset\_publishing\_external\_id=““,

# ARN of the role passed to CloudFormation to execute the deployments

cloud\_formation\_execution\_role=“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}”,

# ARN of the role used to look up context information in an environment

lookup\_role\_arn=“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}”, lookup\_role\_external\_id=““,

# Name of the SSM parameter which describes the bootstrap stack version number

bootstrap\_stack\_version\_ssm\_parameter=“/cdk-bootstrap/${Qualifier}/version”,

# Add a rule to every template which verifies the required bootstrap stack version

generate\_bootstrap\_version\_rule=True, ) —

Java:: + [source,java] — DefaultStackSynthesizer.Builder.create() // Name of the S3 bucket for file assets .fileAssetsBucketName(“cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}”) .bucketPrefix(’’)

// Name of the ECR repository for Docker image assets .imageAssetsRepositoryName(“cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}”) .dockerTagPrefix(’’)

// ARN of the role assumed by the CLI and Pipeline to deploy here .deployRoleArn(“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}”) .deployRoleExternalId(““)

// ARN of the role used for file asset publishing (assumed from the CLI role) .fileAssetPublishingRoleArn(“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}”) .fileAssetPublishingExternalId(““)

// ARN of the role used for Docker asset publishing (assumed from the CLI role) .imageAssetPublishingRoleArn(“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}”) .imageAssetPublishingExternalId(““)

// ARN of the role passed to CloudFormation to execute the deployments .cloudFormationExecutionRole(“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}”)

.lookupRoleArn(“arn:{AWS::AccountId}:role/cdk-$\{Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}”) .lookupRoleExternalId(““)

// Name of the SSM parameter which describes the bootstrap stack version number .bootstrapStackVersionSsmParameter(“/cdk-bootstrap/${Qualifier}/version”)

// Add a rule to every template which verifies the required bootstrap stack version .generateBootstrapVersionRule(true) .build() —

C#:: + [source,csharp] — new DefaultStackSynthesizer(new DefaultStackSynthesizerProps { // Name of the S3 bucket for file assets FileAssetsBucketName = “cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}”, BucketPrefix = ““,

…. // Name of the ECR repository for Docker image assets ImageAssetsRepositoryName = “cdk-{AWS::AccountId}-${AWS::Region}”, DockerTagPrefix = ““,

// ARN of the role assumed by the CLI and Pipeline to deploy here DeployRoleArn = “arn:{AWS::AccountId}:role/cdk-{AWS::AccountId}-${AWS::Region}”, DeployRoleExternalId = ““,

// ARN of the role used for file asset publishing (assumed from the CLI role) FileAssetPublishingRoleArn = “arn:{AWS::AccountId}:role/cdk-{AWS::AccountId}-${AWS::Region}”, FileAssetPublishingExternalId = ““,

// ARN of the role used for Docker asset publishing (assumed from the CLI role) ImageAssetPublishingRoleArn = “arn:{AWS::AccountId}:role/cdk-{AWS::AccountId}-${AWS::Region}”, ImageAssetPublishingExternalId = ““,

// ARN of the role passed to CloudFormation to execute the deployments CloudFormationExecutionRole = “arn:{AWS::AccountId}:role/cdk-{AWS::AccountId}-${AWS::Region}”,

LookupRoleArn = “arn:{AWS::AccountId}:role/cdk-{AWS::AccountId}-${AWS::Region}”, LookupRoleExternalId = ““,

// Name of the SSM parameter which describes the bootstrap stack version number BootstrapStackVersionSsmParameter = “/cdk-bootstrap/${Qualifier}/version”,

// Add a rule to every template which verifies the required bootstrap stack version GenerateBootstrapVersionRule = true, }) —- ==== ….

[#bootstrapping-custom-synth-cli] == Use CliCredentialsStackSynthesizer

To modify the security credentials used to provide permissions during CDK deployments, you can customize synthesis by using link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CliCredentialsStackSynthesizer.html[CliCredentialsStackSynthesizer]. This synthesizer works with the default {aws} resources that are created during bootstrapping to store assets, such as the Amazon S3 bucket and Amazon ECR repository. Instead of using the default IAM roles created by the CDK during bootstrapping, it uses the security credentials of the actor initiating deployment. Therefore, the security credentials of the actor must have valid permissions to perform all deployment actions. The following diagram illustrates the deployment process when using this synthesizer:

image::./images/CliCredentialsStackSynthesizer-deploy-process\_cdk\_flowchart.png[“Deployment process using the CLICredentialsStackSynthesizer.”,scaledwidth=100%]

When using CliCredentialsStackSynthesizer:

* By default, CloudFormation performs API calls in your account using the permissions of the actor. Therefore, the current identity must have permission to make necessary changes to the {aws} resources in the CloudFormation stack, along with the permissions to perform necessary CloudFormation operations, such as CreateStack or UpdateStack. Deployment capabilities will be limited to the permissions of the actor.
* Asset publishing and CloudFormation deployments will be done using the current IAM identity. This identity must have sufficient permissions to both read from and write to the asset bucket and repository.
* Lookups are performed using the current IAM identity, and lookups are subject to its policies.

When using this synthesizer, you can use a separate CloudFormation execution role by specifying it using the xref:ref-cli-cmd-options-role-arn[--role-arn] option with any CDK CLI command.

[#bootstrapping-custom-synth-cli-contract] === CliCredentialsStackSynthesizer bootstrap contract

CliCredentialsStackSynthesizer requires the following bootstrap contract. These are the resources that must be created during bootstrapping:

[cols=“1,1,1,1”, frame=“all”, options=“header”] |=== | Bootstrap resource | Description | Default expected resource name | Purpose

|=== | Amazon S3 bucket | Staging bucket | cdk-hnb659fds-assets-++++++-++++++++++++++++++ | Stores file assets. |===

|=== | Amazon ECR repository | Staging repository | cdk-hnb659fds-container-assets-++++++-++++++++++++++++++ | Stores and manages Docker image assets. |===

The string hnb659fds in the resource name is called the *qualifier*. Its default value has no special significance. You can have multiple copies of the bootstrap resources in a single environment as long as they have a different qualifier. Having multiple copies can be useful for keeping assets of different applications in the same environment separated.

You can deploy the default bootstrap template to satisfy CliCredentialsStackSynthesizer’s bootstrap contract. The default bootstrap template will create IAM roles, but this synthesizer will not use them. You can also customize the bootstrap template to remove the IAM roles.

[#bootstrapping-custom-synth-cli-modify] === Modify CliCredentialsStackSynthesizer

If you change the qualifier or any of the default bootstrap resource names during bootstrapping, you have to modify the synthesizer to use the same names. You can modify the synthesizer for a single stack or for all stacks in your app. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript] — new MyStack(this, ‘MyStack’, { synthesizer: new CliCredentialsStackSynthesizer({ qualifier: ‘MYQUALIFIER’, }), }); —

JavaScript:: + [source,javascript] — new MyStack(this, ‘MyStack’, { synthesizer: new CliCredentialsStackSynthesizer({ qualifier: ‘MYQUALIFIER’, }), }) —

Python:: + [source,python] — MyStack(self, “MyStack”, synthesizer=CliCredentialsStackSynthesizer( qualifier=“MYQUALIFIER” )) —

Java:: + [source,java] — new MyStack(app, “MyStack”, StackProps.builder() .synthesizer(CliCredentialsStackSynthesizer.Builder.create() .qualifier(“MYQUALIFIER”) .build()) .build(); ) —

C#:: + [source,csharp] — new MyStack(app, “MyStack”, new StackProps { Synthesizer = new CliCredentialsStackSynthesizer(new CliCredentialsStackSynthesizerProps { Qualifier = “MYQUALIFIER” }) }); — ====

The following example shows the most commonly used properties for CliCredentialsStackSynthesizer along with their default values. For a complete list, see link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CliCredentialsStackSynthesizerProps.html[CliCredentialsStackSynthesizerProps]:

==== [role=“tablist”] TypeScript:: + [source,javascript] — new CliCredentialsStackSynthesizer({ // Value for ‘${Qualifier}’ in the resource names qualifier: ‘hnb659fds’,

// Name of the S3 bucket for file assets fileAssetsBucketName: ‘cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}’, bucketPrefix: ’’,

// Name of the ECR repository for Docker image assets imageAssetsRepositoryName: ‘cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}’, dockerTagPrefix: ’’, }) —

JavaScript:: + [source,javascript] — new CliCredentialsStackSynthesizer({ // Value for ‘${Qualifier}’ in the resource names qualifier: ‘hnb659fds’,

// Name of the S3 bucket for file assets fileAssetsBucketName: ‘cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}’, bucketPrefix: ’’,

// Name of the ECR repository for Docker image assets imageAssetsRepositoryName: ‘cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}’, dockerTagPrefix: ’’, }) —

Python:: + [source,python] — CliCredentialsStackSynthesizer( # Value for ‘${Qualifier}’ in the resource names qualifier=“hnb659fds”,

# Name of the S3 bucket for file assets

file\_assets\_bucket\_name=“cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}”, bucket\_prefix=““,

# Name of the ECR repository for Docker image assets

image\_assets\_repository\_name=“cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}”, docker\_tag\_prefix=““, ) —

Java:: + [source,java] — CliCredentialsStackSynthesizer.Builder.create() // Value for ‘${Qualifier}’ in the resource names .qualifier(“hnb659fds”)

// Name of the S3 bucket for file assets .fileAssetsBucketName(“cdk-$\{Qualifier}-assets-${AWS::AccountId}-${AWS::Region}”) .bucketPrefix(’’)

// Name of the ECR repository for Docker image assets .imageAssetsRepositoryName(“cdk-$\{Qualifier}-container-assets-${AWS::AccountId}-${AWS::Region}”) .dockerTagPrefix(’’) .build() —

C#:: + [source,csharp] — new CliCredentialsStackSynthesizer(new CliCredentialsStackSynthesizerProps {

…. // Value for ‘${Qualifier}’ in the resource names Qualifier = “hnb659fds”,

// Name of the S3 bucket for file assets FileAssetsBucketName = “cdk-{AWS::AccountId}-${AWS::Region}”, BucketPrefix = ““,

// Name of the ECR repository for Docker image assets ImageAssetsRepositoryName = “cdk-{AWS::AccountId}-${AWS::Region}”, DockerTagPrefix = ““, }) —- ==== ….

[#bootstrapping-custom-synth-legacy] == Use LegacyStackSynthesizer

The LegacyStackSynthesizer emulates the behavior of CDK v1 deployments. The security credentials of the actor performing deployment will be used to establish permissions. File assets will be uploaded to a bucket that must be created using a {aws} CloudFormation stack named CDKToolkit. The CDK CLI will create an unmanaged Amazon ECR repository named aws-cdk/assets to store Docker image assets. You will be responsible to clean up and manage this repository. Stacks synthesized using the LegacyStackSynthesizer can only be deployed using the CDK CLI.

You can use the LegacyStackSynthesizer if you are migrating from CDK v1 to CDK v2, and are unable to re-bootstrap your environments. For new projects, we recommend that you don’t use LegacyStackSynthesizer.

[#bootstrapping-custom-synth-legacy-contract] === LegacyStackSynthesizer bootstrap contract

LegacyStackSynthesizer requires the following bootstrap contract. These are the resources that must be created during bootstrapping:

[cols=“1,1,1,1”, frame=“all”, options=“header”] |=== | Bootstrap resource | Description | Default expected resource name | Purpose

|=== | Amazon S3 bucket | Staging bucket | cdk-hnb659fds-assets-++++++-++++++++++++++++++ | Stores file assets. |===

|=== | CloudFormation output | Bucket name output | Stack – CDKToolkit |===

Output name – BucketName |

A CloudFormation output describing the name of the staging bucket |===

The LegacyStackSynthesizer does not assume the existence of an Amazon S3 bucket with a fixed name. Instead, the synthesized CloudFormation template will contain three CloudFormation parameters for each file asset. These parameters will store the Amazon S3 bucket name, Amazon S3 object key, and artifact hash for each file asset.

[noloc]Docker image assets will be published to an Amazon ECR repository named aws-cdk/assets. This name can be changed per asset. The repositories will be created if they do not exist.

A CloudFormation stack must exist with the default name CDKToolkit. This stack must have a CloudFormation export named BucketName that refers to the staging bucket.

The default bootstrap template satisfies the LegacyStackSynthesizer bootstrap contract. However, only the Amazon S3 bucket from the bootstrap resources of the bootstrap template will be used. You can customize the bootstrap template to remove the Amazon ECR, IAM, and SSM bootstrap resources.

[[bootstrapping-custom-synth-legacy-deploy,bootstrapping-custom-synth-legacy-deploy.title]] === LegacyStackSynthesizer deployment process

When you use this synthesizer, the following process is performed during deployment:

* The CDK [noloc]CLI looks for a CloudFormation stack named CDKToolkit in your environment. From this stack, the CDK [noloc]CLI reads the CloudFormation output named BucketName. You can use the --toolkit-stack-name option with cdk deploy to specify a different stack name.
* The security credentials of the actor initiating deployment will be used to establish permissions for deployment. Therefore, the actor must have sufficient permissions to perform all deployment actions. This includes reading and writing to the Amazon S3 staging bucket, creating and writing to the Amazon ECR repository, starting and monitoring {aws} CloudFormation deployments, and performing any API calls necessary for deployment.
* If necessary, and if permissions are valid, file assets will be published to the Amazon S3 staging bucket.
* If necessary, and if permissions are valid, [noloc]Docker image assets are published to the repository named by the repositoryName property of the asset. The default value is 'aws-cdk/assets' if you don`’t provide a repository name.
* If permissions are valid, the {aws} CloudFormation deployment is performed. The locations of the Amazon S3 staging bucket and keys are passed as CloudFormation parameters.

:doctype: book

include::attributes.txt[]

// Attribues [.topic] [#get-cfn-param] = Use CloudFormation parameters to get a CloudFormation value :info\_titleabbrev: Use CloudFormation parameters

// Content start

Use {aws} CloudFormation parameters within {aws} Cloud Development Kit ({aws} CDK) applications to input custom values into your synthesized CloudFormation templates at deployment.

For an introduction, see xref:parameters[Parameters and the {aws} CDK].

[#parameters-define] == Define parameters in your CDK app

Use the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CfnParameter.html[CfnParameter] class to define a parameter. You’ll want to specify at least a type and a description for most parameters, though both are technically optional. The description appears when the user is prompted to enter the parameter’s value in the {aws} CloudFormation console. For more information on the available types, see link:https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html#parameters-section-structure-properties-type[Types].

= [NOTE]

You can define parameters in any scope. However, we recommend defining parameters at the stack level so that their logical ID doesn’t change when you refactor your code.

====

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const uploadBucketName = new CfnParameter(this, “uploadBucketName”, { type: “String”, description: “The name of the Amazon S3 bucket where uploaded files will be stored.”}); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const uploadBucketName = new CfnParameter(this, “uploadBucketName”, { type: “String”, description: “The name of the Amazon S3 bucket where uploaded files will be stored.”}); —

Python:: + [source,python,subs=“verbatim,attributes”] — upload\_bucket\_name = CfnParameter(self, “uploadBucketName”, type=“String”, description=“The name of the Amazon S3 bucket where uploaded files will be stored.”) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnParameter uploadBucketName = CfnParameter.Builder.create(this, “uploadBucketName”) .type(“String”) .description(“The name of the Amazon S3 bucket where uploaded files will be stored”) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var uploadBucketName = new CfnParameter(this, “uploadBucketName”, new CfnParameterProps { Type = “String”, Description = “The name of the Amazon S3 bucket where uploaded files will be stored” }); — ====

[#parameters-use,] == Use parameters

A CfnParameter instance exposes its value to your CDK app via a xref:tokens[token]. Like all tokens, the parameter’s token is resolved at synthesis time. But it resolves to a reference to the parameter defined in the {aws} CloudFormation template (which will be resolved at deploy time), rather than to a concrete value.

You can retrieve the token as an instance of the Token class, or in string, string list, or numeric encoding. Your choice depends on the kind of value required by the class or method that you want to use the parameter with.

==== [role=“tablist”] TypeScript:: + [cols=“1,1”, options=“header”] |=== | Property | kind of value

|=== | value | Token class instance |===

|=== | valueAsList | The token represented as a string list |===

|=== | valueAsNumber | The token represented as a number |===

|=== | valueAsString | The token represented as a string |===

JavaScript:: + [cols=“1,1”, options=“header”] |=== | Property | kind of value

|=== | value | Token class instance |===

|=== | valueAsList | The token represented as a string list |===

|=== | valueAsNumber | The token represented as a number |===

|=== | valueAsString | The token represented as a string |===

Python:: + [cols=“1,1”, options=“header”] |=== | Property | kind of value

|=== | value | Token class instance |===

|=== | value\_as\_list | The token represented as a string list |===

|=== | value\_as\_number | The token represented as a number |===

|=== | value\_as\_string | The token represented as a string |===

Java:: + [cols=“1,1”, options=“header”] |=== | Property | kind of value

|=== | getValue() | Token class instance |===

|=== | getValueAsList() | The token represented as a string list |===

|=== | getValueAsNumber() | The token represented as a number |===

|=== | getValueAsString() | The token represented as a string |===

C#:: + [cols=“1,1”, options=“header”] |=== | Property | kind of value

|=== | Value | Token class instance |===

|=== | ValueAsList | The token represented as a string list |===

|=== | ValueAsNumber | The token represented as a number |===

|ValueAsString |The token represented as a string |=== ====

For example, to use a parameter in a Bucket definition:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = new Bucket(this, “amzn-s3-demo-bucket”, { bucketName: uploadBucketName.valueAsString}); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = new Bucket(this, “amzn-s3-demo-bucket”, { bucketName: uploadBucketName.valueAsString}); —

Python:: + [source,python,subs=“verbatim,attributes”] — bucket = Bucket(self, “amzn-s3-demo-bucket”, bucket\_name=upload\_bucket\_name.value\_as\_string) —

Java:: + [source,java,subs=“verbatim,attributes”] — Bucket bucket = Bucket.Builder.create(this, “amzn-s3-demo-bucket”) .bucketName(uploadBucketName.getValueAsString()) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var bucket = new Bucket(this, “amzn-s3-demo-bucket”) { BucketName = uploadBucketName.ValueAsString }; — ====

[#parameters-deploy] == Deploy CDK apps containing parameters

When you deploy a generated {aws} CloudFormation template through the {aws} CloudFormation console, you will be prompted to provide the values for each parameter.

You can also provide parameter values using the CDK CLI cdk deploy command, or by specifying parameter values in your CDK project’s stack file.

[#parameters-deploy-cli] === Provide parameter values with [.noloc]cdk deploy

When you deploy using the CDK CLI cdk deploy command, you can provide parameter values at deployment with the --parameters option.

The following is an example of the cdk deploy command structure:

== [source,none,subs=“verbatim,attributes”]

$ cdk deploy ++++++–parameters ++++++:++++++=++++++—-++++++++++++++++++++++++

If your CDK app contains a single stack, you don’t have to provide the stack logical ID argument or the stack-name value in the --parameters option. The CDK CLI will automatically find and provide these values. The following is an example that specifies an uploadbucket value for the uploadBucketName parameter of the single stack in our CDK app:

== [source,none,subs=“verbatim,attributes”]

$ cdk deploy –parameters ++++++=++++++—-++++++++++++

[#parameters-deploy-cli-multi-stack] === Provide parameter values with cdk deploy for multi-stack applications

The following is an example CDK application in TypeScript that contains two CDK stacks. Each stack contains an Amazon S3 bucket instance and a parameter to set the Amazon S3 bucket name:

== [source,javascript,subs=“verbatim,attributes”]

import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import \* as s3 from ‘aws-cdk-lib/aws-s3’;

// Define the CDK app const app = new cdk.App();

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

…. // Set a default parameter name const bucketNameParam = new cdk.CfnParameter(this, ‘bucketNameParam’, { type: ‘String’, default: ‘myfirststackdefaultbucketname’ });

// Define an S3 bucket new s3.Bucket(this, ‘MyFirstBucket’, { bucketName: bucketNameParam.valueAsString }); } } ….

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

…. // Set a default parameter name const bucketNameParam = new cdk.CfnParameter(this, ‘bucketNameParam’, { type: ‘String’, default: ‘mysecondstackdefaultbucketname’ });

// Define an S3 bucket new s3.Bucket(this, ‘MySecondBucket’, { bucketName: bucketNameParam.valueAsString }); } } ….

// Instantiate the stacks new MyFirstStack(app, ‘MyFirstStack’, { stackName: ‘MyFirstDeployedStack’, });

new MySecondStack(app, ‘MySecondStack’, { stackName: ‘MySecondDeployedStack’, }); —

For CDK apps that contain multiple stacks, you can do the following:

* *Deploy one stack with parameters* – To deploy a single stack from a multi-stack application, provide the stack logical ID as an argument.
* The following is an example that deploys MySecondStack with mynewbucketname as the parameter value for bucketNameParam:

## [source,none,subs=“verbatim,attributes”]

* $ cdk deploy ++++++–parameters ++++++=‘++++++’ —-++++++++++++++++++
* *Deploy all stacks and specify parameter values for each stack* – Provide the '\*' wildcard or the --all option to deploy all stacks. Provide the --parameters option multiple times in a single command to specify parameter values for each stack. The following is an example:

## [source,none,subs=“verbatim,attributes”]

* $ cdk deploy <’\*‘> –parameters ++++++:++++++=’++++++’ –parameters ++++++:++++++=‘++++++’ —-++++++++++++++++++++++++++++++++++++
* *Deploy all stacks and specify parameter values for a single stack* – Provide the '\*' wildcard or the --all option to deploy all stacks. Then, specify the stack to define the parameter for in the --parameters option. The following are examples that deploys all stacks in a CDK app and specifies a parameter value for the MySecondDeployedStack {aws} CloudFormation stack. All other stacks will deploy and use the default parameter value:

## [source,none,subs=“verbatim,attributes”]

* $ cdk deploy <’\*‘> –parameters ++++++:++++++=’++++++’ $ cdk deploy <–all> –parameters ++++++:++++++=‘++++++’ —-++++++++++++++++++++++++++++++++++++

[#parameters-deploy-cli-nested-stack] === Provide parameter values with cdk deploy for applications with nested stacks

The CDK CLI behavior when working with applications containing nested stacks is similar to multi-stack applications. The main difference is, if you want to deploy all nested stacks, use the +'\\\*\*'+ wildcard. The '\*' wildcard deploys all stacks but will not deploy nested stacks. The +'\*\*'+ wildcard deploys all stacks, including nested stacks.

The following is an example that deploys nested stacks while specifying the parameter value for one nested stack:

== [source,none,subs=“verbatim,attributes”]

$ cdk deploy ’\*\*’ –parameters :++++++=‘++++++’ —-++++++++++++

For more information on cdk deploy command options, see xref:ref-cli-cmd-deploy[cdk deploy].

include::attributes.txt[]

// Attributes

[.topic] [[get-context-var,get-context-var.title]] = Save and retrieve context variable values :info\_titleabbrev: Get context value

// Content start

You can specify context variables with the {aws} Cloud Development Kit ({aws} CDK) CLI or in the cdk.json file. Then, use the TryGetContext method to retrieve values.

[#develop-context-specify] == Specify context variables

You can specify a context variable either as part of an {aws} CDK CLI command, or in cdk.json.

To create a command line context variable, use the --context (-c) option, as shown in the following example.

== [source,none,subs=“verbatim,attributes”]

## cdk synth -c bucket\_name=mygroovybucket

To specify the same context variable and value in the cdk.json file, use the following code.

== [source,json,subs=“verbatim,attributes”]

{ “context”: { “bucket\_name”: “myotherbucket” } } —

If you specify a context variable using both the {aws} CDK CLI and cdk.json file, the {aws} CDK CLI value takes precedence.

[#develop-context-retrieve] == Retrieve context variable values

To get the value of a context variable in your app, use the TryGetContext method in the context of a construct. (That is, when this, or self in Python, is an instance of some construct.)

In this example, we retrieve the value of the bucket\_name context variable. If the requested value is not defined, TryGetContext returns undefined (None in Python; null in Java and C#; nil in Go) rather than raising an exception.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket\_name = this.node.tryGetContext(‘bucket\_name’); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket\_name = this.node.tryGetContext(‘bucket\_name’); —

Python:: + [source,python,subs=“verbatim,attributes”] — bucket\_name = self.node.try\_get\_context(“bucket\_name”) —

Java:: + [source,java,subs=“verbatim,attributes”] — String bucketName = (String)this.getNode().tryGetContext(“bucket\_name”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var bucketName = this.Node.TryGetContext(“bucket\_name”); — ====

Outside the context of a construct, you can access the context variable from the app object, like this.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const app = new cdk.App(); const bucket\_name = app.node.tryGetContext(‘bucket\_name’) —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const app = new cdk.App(); const bucket\_name = app.node.tryGetContext(‘bucket\_name’); —

Python:: + [source,python,subs=“verbatim,attributes”] — app = cdk.App() bucket\_name = app.node.try\_get\_context(“bucket\_name”) —

Java:: + [source,java,subs=“verbatim,attributes”] — App app = App(); String bucketName = (String)app.getNode().tryGetContext(“bucket\_name”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — app = App(); var bucketName = app.Node.TryGetContext(“bucket\_name”); — ====

For more details on working with context variables, see xref:context[Context values and the {aws} CDK].

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#get-env-var] = Get a value from an environment variable :info\_titleabbrev: Get environment value

// Content start

To get the value of an environment variable, use code like the following. This code gets the value of the environment variable amzn-s3-demo-bucket.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] —

// Sets bucket\_name to undefined if environment variable not set var bucket\_name = process.env.amzn-s3-demo-bucket;

// Sets bucket\_name to a default if env var doesn’t exist var bucket\_name = process.env.amzn-s3-demo-bucket || “DefaultName”; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] —

// Sets bucket\_name to undefined if environment variable not set var bucket\_name = process.env.amzn-s3-demo-bucket;

// Sets bucket\_name to a default if env var doesn’t exist var bucket\_name = process.env.amzn-s3-demo-bucket || “DefaultName”; —

Python:: + [source,python,subs=“verbatim,attributes”] — import os

= Raises KeyError if environment variable doesn’t exist

bucket\_name = os.environ[“amzn-s3-demo-bucket”]

= Sets bucket\_name to None if environment variable doesn’t exist

bucket\_name = os.getenv(“amzn-s3-demo-bucket”)

= Sets bucket\_name to a default if env var doesn’t exist

## bucket\_name = os.getenv(“amzn-s3-demo-bucket”, “DefaultName”)

Java:: + [source,java,subs=“verbatim,attributes”] — // Sets bucketName to null if environment variable doesn’t exist String bucketName = System.getenv(“amzn-s3-demo-bucket”);

// Sets bucketName to a default if env var doesn’t exist String bucketName = System.getenv(“amzn-s3-demo-bucket”); if (bucketName == null) bucketName = “DefaultName”; —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using System;

// Sets bucket name to null if environment variable doesn’t exist string bucketName = Environment.GetEnvironmentVariable(“amzn-s3-demo-bucket”);

// Sets bucket\_name to a default if env var doesn’t exist string bucketName = Environment.GetEnvironmentVariable(“amzn-s3-demo-bucket”) ?? “DefaultName”; — ====

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#get-secrets-manager-value] = Get a value from {aws} Secrets Manager :info\_titleabbrev: Get Secrets Manager value

// Content start

To use values from {aws} Secrets Manager in your {aws} CDK app, use the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_secretsmanager.Secret.html#static-fromwbrsecretwbrattributesscope-id-attrs[fromSecretAttributes()] method. It represents a value that is retrieved from Secrets Manager and used at {aws} CloudFormation deployment time. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as sm from “aws-cdk-lib/aws-secretsmanager”;

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

const secret = sm.Secret.fromSecretAttributes(this, “ImportedSecret”, { secretCompleteArn: “arn:aws:secretsmanager:::secret:-” // If the secret is encrypted using a KMS-hosted CMK, either import or reference that key: // encryptionKey: … } ); } } —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const sm = require(“aws-cdk-lib/aws-secretsmanager”);

class SecretsManagerStack extends cdk.Stack { constructor(scope, id, props) { super(scope, id, props);

const secret = sm.Secret.fromSecretAttributes(this, “ImportedSecret”, { secretCompleteArn: “arn:aws:secretsmanager:::secret:-” // If the secret is encrypted using a KMS-hosted CMK, either import or reference that key: // encryptionKey: … }); } }

== module.exports = { SecretsManagerStack }

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk.aws\_secretsmanager as sm

class SecretsManagerStack(cdk.Stack): def *init*(self, scope: cdk.App, id: str, **kwargs): super().*init*(scope, name,** kwargs)

secret = sm.Secret.from\_secret\_attributes(self, “ImportedSecret”, secret\_complete\_arn=“arn:aws:secretsmanager:::secret:-”, # If the secret is encrypted using a KMS-hosted CMK, either import or reference that key: # encryption\_key=…. ) —-

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.secretsmanager.Secret; import software.amazon.awscdk.services.secretsmanager.SecretAttributes;

public class SecretsManagerStack extends Stack { public SecretsManagerStack(App scope, String id) { this(scope, id, null); }

public SecretsManagerStack(App scope, String id, StackProps props) { super(scope, id, props);

Secret secret = (Secret)Secret.fromSecretAttributes(this, "ImportedSecret", SecretAttributes.builder()  
 .secretCompleteArn("arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-<random-6-characters>")  
 // If the secret is encrypted using a KMS-hosted CMK, either import or reference that key:  
 // .encryptionKey(...)  
 .build());

} } —-

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK.{aws}.SecretsManager;

public class SecretsManagerStack : Stack { public SecretsManagerStack(App scope, string id, StackProps props) : base(scope, id, props) {

var secret = Secret.FromSecretAttributes(this, "ImportedSecret", new SecretAttributes {  
 SecretCompleteArn = "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-<random-6-characters>"  
 // If the secret is encrypted using a KMS-hosted CMK, either import or reference that key:  
 // encryptionKey = ...,  
 });

} } —- ====

= [TIP]

Use the {aws} CLI link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_secretsmanager.Secret.html[create-secret] CLI command to create a secret from the command line, such as when testing:

== [source,none,subs=“verbatim,attributes”]

## aws secretsmanager create-secret –name ImportedSecret –secret-string mygroovybucket

The command returns an ARN that you can use with the preceding example.

====

Once you have created a Secret instance, you can get the secret’s value from the instance’s secretValue attribute. The value is represented by a link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.SecretValue.html[SecretValue] instance, a special type of xref:tokens[Tokens and the {aws} CDK]. Because it’s a token, it has meaning only after resolution. Your CDK app does not need to access its actual value. Instead, the app can pass the SecretValue instance (or its string or numeric representation) to whatever CDK method needs the value.

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [[get-ssm-value,get-ssm-value.title]] = Get a value from the Systems Manager Parameter Store :info\_titleabbrev: Get SSM value

// Content start

The {aws} Cloud Development Kit ({aws} CDK) can retrieve the value of {aws} Systems Manager Parameter Store attributes. During synthesis, the {aws} CDK produces a xref:tokens[token] that is resolved by {aws} CloudFormation during deployment.

The {aws} CDK supports retrieving both plain and secure values. You may request a specific version of either kind of value. For plain values, you may omit the version from your request to retrieve the latest version. For secure values, you must specify the version when requesting the value of the secure attribute.

= [NOTE]

This topic shows how to read attributes from the {aws} Systems Manager Parameter Store. You can also read secrets from the {aws} Secrets Manager (see xref:get-secrets-manager-value[Get a value from {aws} Secrets Manager]).

====

[#ssm-read-at-deploy] == Read Systems Manager values at deployment time

To read values from the Systems Manager Parameter Store, use the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ssm.StringParameter.html#static-valuewbrforwbrstringwbrparameterscope-parametername-version[valueForStringParameter] and link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ssm.StringParameter.html#static-valuewbrforwbrsecurewbrstringwbrparameterscope-parametername-version[valueForSecureStringParameter] methods. Choose a method based on whether the attribute you want is a plain string or a secure string value. These methods return xref:tokens[tokens], not the actual value. The value is resolved by {aws} CloudFormation during deployment. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as ssm from ‘aws-cdk-lib/aws-ssm’;

// Get latest version or specified version of plain string attribute const latestStringToken = ssm.StringParameter.valueForStringParameter( this, ‘my-plain-parameter-name’); // latest version const versionOfStringToken = ssm.StringParameter.valueForStringParameter( this, ‘my-plain-parameter-name’, 1); // version 1

// Get specified version of secure string attribute const secureStringToken = ssm.StringParameter.valueForSecureStringParameter( this, ‘my-secure-parameter-name’, 1); // must specify version —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const ssm = require(‘aws-cdk-lib/aws-ssm’);

// Get latest version or specified version of plain string attribute const latestStringToken = ssm.StringParameter.valueForStringParameter( this, ‘my-plain-parameter-name’); // latest version const versionOfStringToken = ssm.StringParameter.valueForStringParameter( this, ‘my-plain-parameter-name’, 1); // version 1

// Get specified version of secure string attribute const secureStringToken = ssm.StringParameter.valueForSecureStringParameter( this, ‘my-secure-parameter-name’, 1); // must specify version —

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk.aws\_ssm as ssm

= Get latest version or specified version of plain string attribute

latest\_string\_token = ssm.StringParameter.value\_for\_string\_parameter( self, “my-plain-parameter-name”) latest\_string\_token = ssm.StringParameter.value\_for\_string\_parameter( self, “my-plain-parameter-name”, 1)

= Get specified version of secure string attribute

secure\_string\_token = ssm.StringParameter.value\_for\_secure\_string\_parameter( self, “my-secure-parameter-name”, 1) # must specify version —

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.ssm.StringParameter;

//Get latest version or specified version of plain string attribute String latestStringToken = StringParameter.valueForStringParameter( this, “my-plain-parameter-name”); // latest version String versionOfStringToken = StringParameter.valueForStringParameter( this, “my-plain-parameter-name”, 1); // version 1

//Get specified version of secure string attribute String secureStringToken = StringParameter.valueForSecureStringParameter( this, “my-secure-parameter-name”, 1); // must specify version —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK.{aws}.SSM;

// Get latest version or specified version of plain string attribute var latestStringToken = StringParameter.ValueForStringParameter( this, “my-plain-parameter-name”); // latest version var versionOfStringToken = StringParameter.ValueForStringParameter( this, “my-plain-parameter-name”, 1); // version 1

// Get specified version of secure string attribute var secureStringToken = StringParameter.ValueForSecureStringParameter( this, “my-secure-parameter-name”, 1); // must specify version — ====

A link:https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/dynamic-references.html#template-parameters-dynamic-patterns-resources[limited number of {aws} services] currently support this feature.

[#ssm-read-at-synth] == Read Systems Manager values at synthesis time

At times, it’s useful to provide a parameter at synthesis time. By doing this, the {aws} CloudFormation template will always use the same value instead of resolving the value during deployment.

To read a value from the Systems Manager Parameter Store at synthesis time, use the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ssm.StringParameter.html#static-valuewbrfromwbrlookupscope-parametername[valueFromLookup] method (Python: value\_from\_lookup). This method returns the actual value of the parameter as a xref:context[Context values and the {aws} CDK] value. If the value is not already cached in cdk.json or passed on the command line, it is retrieved from the current {aws} account. For this reason, the stack *must* be synthesized with explicit {aws} environment information.

The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as ssm from ‘aws-cdk-lib/aws-ssm’;

== const stringValue = ssm.StringParameter.valueFromLookup(this, ‘my-plain-parameter-name’);

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const ssm = require(‘aws-cdk-lib/aws-ssm’);

== const stringValue = ssm.StringParameter.valueFromLookup(this, ‘my-plain-parameter-name’);

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk.aws\_ssm as ssm

== string\_value = ssm.StringParameter.value\_from\_lookup(self, “my-plain-parameter-name”)

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.ssm.StringParameter;

== String stringValue = StringParameter.valueFromLookup(this, “my-plain-parameter-name”);

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK.{aws}.SSM;

== var stringValue = StringParameter.ValueFromLookup(this, “my-plain-parameter-name”);

====

Only plain Systems Manager strings may be retrieved. Secure strings cannot be retrieved. The latest version will always be returned. Specific versions cannot be requested.

= [IMPORTANT]

The retrieved value will end up in your synthesized {aws} CloudFormation template. This might be a security risk, depending on who has access to your {aws} CloudFormation templates and what kind of value it is. Generally, don’t use this feature for passwords, keys, or other values you want to keep private.

====

[#ssm-write] == Write values to Systems Manager

You can use the {aws} CLI, the {aws} Management Console, or an {aws} SDK to set Systems Manager parameter values. The following examples use the link:https://docs.aws.amazon.com/cli/latest/reference/ssm/put-parameter.html[ssm put-parameter] CLI command.

== [source,none,subs=“verbatim,attributes”]

aws ssm put-parameter –name “parameter-name” –type “String” –value “parameter-value” aws ssm put-parameter –name “secure-parameter-name” –type “SecureString” –value “secure-parameter-value” —

When updating an SSM value that already exists, also include the --overwrite option.

== [source,none,subs=“verbatim,attributes”]

aws ssm put-parameter –overwrite –name “parameter-name” –type “String” –value “parameter-value” aws ssm put-parameter –overwrite –name “secure-parameter-name” –type “SecureString” –value “secure-parameter-value” —

include::attributes.txt[]

[.topic] [#libraries] = The {aws} CDK libraries :info\_titleabbrev: Libraries :keywords: {aws} CDK, {aws} CDK libraries, constructs, {aws} Construct Library

== [abstract]

## Learn about the core libraries that you will use with the {aws} Cloud Development Kit ({aws} CDK).

// Content start

Learn about the core libraries that you will use with the {aws} Cloud Development Kit ({aws} CDK).

[#libraries-cdk] == The {aws} CDK Library

The {aws} CDK Library, also referred to as aws-cdk-lib, is the main library that you will use to develop applications with the {aws} CDK. It is developed and maintained by {aws}. This library contains base classes, such as https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.App.html[App] and https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html[Stack]. It also contains the libraries you will use to define your infrastructure through constructs.

[#libraries-construct] == The {aws} Construct Library

The {aws} Construct Library is a part of the {aws} CDK Library. It contains a collection of xref:constructs[constructs] that are developed and maintained by {aws}. It is organized into various modules for each {aws} service. Each module includes constructs that you can use to define your {aws} resources and properties.

[#libraries-constructs] == The Constructs library

The Constructs library, commonly referred to as constructs, is a library for defining and composing cloud infrastructure components. It contains the core https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct] class, which represents the construct building block. This class is the foundational base class of all constructs from the {aws} Construct Library. The Constructs library is a separate general-purpose library that is used by other construct-based tools such as *CDK for Terraform* and *CDK for Kubernetes*.

[#libraries-reference] == The {aws} CDK API reference

The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-construct-library.html[{aws} CDK API reference] contains official reference documentation for the {aws} CDK Library, including the {aws} Construct Library and Constructs library. A version of the API reference is provided for each supported programming language.

* For {aws} CDK Library (aws-cdk-lib) documentation, see https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib-readme.html[aws-cdk-lib module].
* Documentation for constructs in the {aws} Construct Library are organized by {aws} service in the following format: aws-cdk-lib.<service>. For example, construct documentation for Amazon Simple Storage Service (Amazon S3), can be found at https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3-readme.html[aws-cdk-lib.aws\_s3 module].
* For Constructs library (constructs) documentation, see https://docs.aws.amazon.com/cdk/api/v2/docs/constructs-readme.html[constructs module].

[#libraries-reference-contribute] === Contribute to the {aws} CDK API reference

The {aws} CDK is open-source and we welcome you to contribute. Community contributions positively impact and improve the {aws} CDK. For instructions on contributing specifically to {aws} CDK API reference documentation, see https://github.com/aws/aws-cdk/blob/main/CONTRIBUTING.md#documentation[Documentation] in the *aws-cdk [.noloc]GitHub repository*.

[#libraries-learn,libraries-learn.title] == Learn more

For instructions on importing and using the CDK Library, see xref:work-with[Work with the CDK library].

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#migrate] = Migrate existing resources and {aws} CloudFormation templates to the {aws} CDK :info\_titleabbrev: Migrate to the {aws} CDK :keywords: {aws} CDK, {aws} CDK CLI, {aws} CloudFormation, Migrate, {aws} resources, Infrastructure as Code, IaC

== [abstract]

## Use the {aws} Cloud Development Kit ({aws} CDK) Command Line Interface ({aws} CDK CLI) to migrate deployed {aws} resources, deployed {aws} CloudFormation stacks, and local {aws} CloudFormation templates to {aws} CDK.

// Content start

[cols=“1”, frame=“all”] |===

|=== | The CDK Migrate feature is in preview release for {aws} CDK and is subject to change. |===

Use the {aws} Cloud Development Kit ({aws} CDK) Command Line Interface ({aws} CDK CLI) to migrate deployed {aws} resources, deployed {aws} CloudFormation stacks, and local {aws} CloudFormation templates to {aws} CDK.

[#migrate-intro] == How migration works

Use the {aws} CDK CLI cdk migrate command to migrate from the following sources:

–

* Deployed {aws} resources.
* Deployed {aws} CloudFormation stacks.
* {blank}
* == Local {aws} CloudFormation templates.

*Deployed {aws} resources*:: + You can migrate deployed {aws} resources from a specific environment ({aws} account and {aws} Region) that are not associated with an {aws} CloudFormation stack. + The {aws} CDK CLI utilizes the *IaC generator* service to scan for resources in your {aws} environment to gather resource details. To learn more about IaC generator, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/generate-IaC.html[Generating templates for existing resources] in the *{aws} CloudFormation User Guide*. + After gathering resource details, the {aws} CDK CLI creates a new CDK app that includes a single stack containing your migrated resources.

*Deployed {aws} CloudFormation stacks*:: + You can migrate a single {aws} CloudFormation stack into a new {aws} CDK app. The {aws} CDK CLI will retrieve the {aws} CloudFormation template of your stack and create a new CDK app. The CDK app will consist of a single stack that contains your migrated {aws} CloudFormation stack.

*Local {aws} CloudFormation templates*:: + You can migrate from a local {aws} CloudFormation template. Local templates may or may not contain deployed resources. The {aws} CDK CLI will create a new CDK app that contains a single stack with your resources. + After migrating, you can manage, modify, and deploy your CDK app to {aws} CloudFormation to provision or update your resources.

[#migrate-benefits] == Benefits of CDK Migrate

Migrating resources into {aws} CDK has historically been a manual process that requires time and expertise with {aws} CloudFormation and {aws} CDK to even begin. With CDK Migrate, the {aws} CDK CLI facilitates a majority of the migration effort for you in a fraction of the time. CDK Migrate will quickly get you started with using the {aws} CDK to develop and manage new and existing applications on {aws}.

[#migrate-considerations] == Considerations

[#migrate-considerations-general] === General considerations

*CDK Migrate vs. CDK Import*:: + The cdk import command can import deployed resources into a new or existing CDK app. When importing, each resource will have to manually be defined as an L1 construct in your app. We recommend using cdk import to import one or more resources at a time into a new or existing CDK app. To learn more, see xref:cli-import[Import existing resources into a stack]. + The cdk migrate command migrates from deployed resources, deployed {aws} CloudFormation stacks, or local {aws} CloudFormation templates into a new CDK app. During migration, the {aws} CDK CLI uses cdk import to import your resources into the new CDK app. The {aws} CDK CLI also generates L1 constructs for each resource for you. We recommend using cdk migrate when importing from a supported migration source into a new {aws} CDK app.

*CDK Migrate creates L1 constructs only*:: + The newly created CDK app will include L1 constructs only. You can add higher-level constructs to your app after migration.

*CDK Migrate creates CDK apps that contain a single stack*:: + The newly created CDK app will contain a single stack. + When migrating deployed resources, all migrated resources will be contained within a single stack in the new CDK app. + When migrating {aws} CloudFormation stacks, you can only migrate a single {aws} CloudFormation stack into a single stack in the new CDK app.

*Migrating assets*:: + Project assets, such as {aws} Lambda code, will not directly migrate into the new CDK app. After migration, you can specify asset values to include them in the CDK app.

*Migrating stateful resources*:: + When migrating stateful resources, such as databases and Amazon Simple Storage Service (Amazon S3) buckets, you’d most often want to migrate the existing resource instead of creating a new resource. + To migrate and preserve stateful resources, do the following: + – \*\* Verify that your stateful resource supports import. For more information, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/resource-import-supported-resources.html[Resource type support] in the *{aws} CloudFormation User Guide*. \*\* After migration, verify that the migrated resource`’s logical ID in the new CDK app matches the logical ID of the deployed resource. \*\* If migrating from an {aws} CloudFormation stack, verify that the stack name in the new CDK app matches the {aws} CloudFormation stack. \*\* Deploy the CDK app using the same {aws} account and {aws} Region of the migrated resource. –

[#migrate-considerations-template] === Considerations when migrating from an {aws} CloudFormation template

*CDK Migrate supports single template migration*:: + When migrating {aws} CloudFormation templates, you can select a single template to migrate. Nested templates are not supported.

*Migrating templates with intrinsic functions*:: + When migrating from an {aws} CloudFormation template that uses intrinsic functions, the {aws} CDK CLI will attempt to migrate your logic into the CDK app with the Fn class. To learn more, see https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Fn.html[class Fn] in the *{aws} Cloud Development Kit ({aws} CDK) API Reference*.

[#migrate-considerations-resources] === Considerations when migrating from deployed resources

*Scan limitations*:: + When scanning your environment for resources, IaC generator has specific limitations on the data it can retrieve and quota limitations when scanning. To learn more, see link:https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/generate-IaC.html#generate-template-considerations[Considerations] in the *{aws} CloudFormation User Guide*.

[#migrate-prerequisites] == Prerequisites

Before using the cdk migrate command, complete all set up steps in xref:getting-started[Getting started with the {aws} CDK].

[#migrate-gs] == Get started with CDK Migrate

To begin, run the {aws} CDK CLI cdk migrate command from a directory of your choice. Provide required and optional options, depending on the type of migration you are performing.

For a full list and description of options that you can use with cdk migrate, see xref:ref-cli-cdk-migrate[cdk migrate].

The following are some important options that you may want to provide.

*Stack name*:: + The only required option is --stack-name. Use this option to specify a name for the stack that will be created within the {aws} CDK app after migration. The stack name will also be used as the name of your {aws} CloudFormation stack at deployment.

*Language*:: + Use --language to specify the programming language of the new CDK app.

*{aws} account and {aws} Region*:: + The {aws} CDK CLI retrieves {aws} account and {aws} Region information from default sources. For more information, see xref:environments[Environments for the {aws} CDK]. You can use --account and --region options with cdk migrate to provide other values.

*Output directory of your new CDK project*:: + By default, the {aws} CDK CLI will create a new CDK project in your working directory and use the value you provide with --stack-name to name the project folder. If a folder with the same name currently exists, the {aws} CDK CLI will overwrite that folder. + You can specify a different output path for the new CDK project folder with the --output-path option.

*Migration source*:: + Provide an option to specify the source you are migrating from. + –

* --from-path – Migrate from a local {aws} CloudFormation template.
* --from-scan – Migrate from deployed resources in an {aws} account and {aws} Region.
* {blank}
* == --from-stack – Migrate from an {aws} CloudFormation stack.
* Depending on your migration source, you can provide additional options to customize the cdk migrate command.

[#migrate-stack] == Migrate from an {aws} CloudFormation stack

To migrate from a deployed {aws} CloudFormation stack, provide the --from-stack option. Provide the name of your deployed {aws} CloudFormation stack with --stack-name. The following is an example:

== [source,bash,subs=“verbatim,attributes”]

## $ cdk migrate –from-stack –stack-name “myCloudFormationStack”

The {aws} CDK CLI will do the following:

. Retrieve the {aws} CloudFormation template of your deployed stack. . Run cdk init to initialize a new CDK app. . Create a stack within the CDK app that contains your migrated {aws} CloudFormation stack.

When you migrate from a deployed {aws} CloudFormation stack, the {aws} CDK CLI attempts to match deployed resource logical IDs and the deployed {aws} CloudFormation stack name to the migrated resources and stack in the new CDK app.

After migration, you can manage and modify your CDK app normally. When you deploy, {aws} CloudFormation will identify the deployment as an {aws} CloudFormation stack update due to the matching {aws} CloudFormation stack name. Resources with matching logical IDs will be updated. For more information on deploying, see xref:migrate-manage[Manage and deploy your CDK app].

[#migrate-template] == Migrate from an {aws} CloudFormation template

CDK Migrate supports migrating from {aws} CloudFormation templates formatted in JSON or YAML.

To migrate from a local {aws} CloudFormation template, use the --from-path option and provide a path to the local template. You must also provide the required --stack-name option. The following is an example:

== [source,bash,subs=“verbatim,attributes”]

## $ cdk migrate –from-path “./template.json” –stack-name “myCloudFormationStack”

The {aws} CDK CLI will do the following:

. Retrieve your local {aws} CloudFormation template. . Run cdk init to initialize a new CDK app. . Create a stack within the CDK app that contains your migrated {aws} CloudFormation template.

After migration, you can manage and modify your CDK app normally. At deployment, you have the following options:

* *Update an {aws} CloudFormation stack* – If the local {aws} CloudFormation template was previously deployed, you can update the deployed {aws} CloudFormation stack.
* *Deploy a new {aws} CloudFormation stack* – If the local {aws} CloudFormation template was never deployed, or if you want to create a new stack from a previously deployed template, you can deploy a new {aws} CloudFormation stack.

[#migrate-template-sam] === Migrate from an {aws} SAM template

To migrate from an {aws} Serverless Application Model ({aws} SAM) template, you must first convert it to an {aws} CloudFormation template or deploy to create an {aws} CloudFormation stack.

To convert an {aws} SAM template to {aws} CloudFormation, you can use the {aws} SAM CLI sam validate --debug command. You may have to set lint to false in your samconfig.toml file before running this command.

To convert to an {aws} CloudFormation stack, deploy the {aws} SAM template using the {aws} SAM CLI. Then migrate from the deployed stack.

[#migrate-resources] == Migrate from deployed resources

To migrate from deployed {aws} resources, provide the --from-scan option. You must also provide the required --stack-name option. The following is an example:

== [source,bash,subs=“verbatim,attributes”]

## $ cdk migrate –from-scan –stack-name “myCloudFormationStack”

The {aws} CDK CLI will do the following:

. *Scan your account for resource and property details* – The {aws} CDK CLI utilizes IaC generator to scan your account and gather details. . *Generate an {aws} CloudFormation template* – After scanning, the {aws} CDK CLI utilizes IaC generator to create an {aws} CloudFormation template. . *Initialize a new CDK app and migrate your template* – The {aws} CDK CLI runs cdk init to initialize a new {aws} CDK app and migrates your {aws} CloudFormation template into the CDK app as a single stack.

[#migrate-resources-filters] === Use filters

By default, the {aws} CDK CLI will scan the entire {aws} environment and migrate resources up to the maximum quota limit of IaC generator. You can provide filters with the {aws} CDK CLI to specify a criteria for which resources get migrated from your account to the new CDK app. To learn more, see xref:ref-cli-cdk-migrate-options-filter[--filter].

[#migrate-resources-scan] === Scanning for resources with IaC generator

Depending on the number of resources in your account, scanning may take a few minutes. A progress bar will display during the scanning process.

[#migrate-resources-supported] *Supported resource types*:: + The {aws} CDK CLI will migrate resources supported by the IaC generator. For a full list, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/resource-import-supported-resources.html[Resource type support] in the *{aws} CloudFormation User Guide*.

[#migrate-resources-writeonly] === Resolve write-only properties

Some supported resources contain write-only properties. These properties can be written to, to configure the property, but can’t be read by IaC generator or {aws} CloudFormation to obtain the value. For example, a property used to specify a database password may be write-only for security reasons.

When scanning resources during migration, IaC generator will detect resources that may contain write-only properties and will categorize them into any of the following types:

* MUTUALLY\_EXCLUSIVE\_PROPERTIES – These are write-only properties for a specific resource that are interchangeable and serve a similar purpose. One of the mutually exclusive properties are required to configure your resource. For example, the S3Bucket, ImageUri, and ZipFile properties for an +{aws}::Lambda::Function+ resource are mutually exclusive write-only properties. Any one of them can be used to specify your function assets, but you must use one.
* MUTUALLY\_EXCLUSIVE\_TYPES – These are required write-only properties that accept multiple configuration types. For example, the Body property of an +{aws}::ApiGateway::RestApi+ resource accepts an object or string type.
* UNSUPPORTED\_PROPERTIES – These are write-only properties that don’t fall under the other two categories. They are either optional properties or required properties that accept an array of objects.

For more information on write-only properties and how IaC generator manages them when scanning for deployed resources and creating {aws} CloudFormation templates, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/generate-IaC-write-only-properties.html[IaC generator and write-only properties] in the *{aws} CloudFormation User Guide*.

After migration, you must specify write-only property values in the new CDK app. The {aws} CDK CLI will append a *Warnings* section to the CDK project’s ReadMe file to document any write-only properties that were identified by IaC generator. The following is an example:

== [source,markdown,subs=“verbatim,attributes”]

= Welcome to your CDK TypeScript project

…

== Warnings

=== Write-only properties

Write-only properties are resource property values that can be written to but can’t be read by {aws} CloudFormation or CDK Migrate. For more information, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/generate-IaC-write-only-properties.html[IaC generator and write-only properties].

Write-only properties discovered during migration are organized here by resource ID and categorized by write-only property type. Resolve write-only properties by providing property values in your CDK app. For guidance, see https://docs.aws.amazon.com/cdk/v2/guide/migrate.html#migrate-resources-writeonly[Resolve write-only properties].

=== MyLambdaFunction

* *UNSUPPORTED\_PROPERTIES*: \*\* SnapStart/ApplyOn: Applying SnapStart setting on function resource type.Possible values: [PublishedVersions, None] This property can be replaced with other types \*\* Code/S3ObjectVersion: For versioned objects, the version of the deployment package object to use. This property can be replaced with other exclusive properties
* *MUTUALLY\_EXCLUSIVE\_PROPERTIES*: \*\* Code/S3Bucket: An Amazon S3 bucket in the same {aws} Region as your function. The bucket can be in a different {aws} account. This property can be replaced with other exclusive properties \*\* Code/S3Key: The Amazon S3 key of the deployment package. This property can be replaced with other exclusive properties —

–

* Warnings are organized under headings that identify the resource`’s logical ID that they are associated with.
* {blank}
* == Warnings are categorized by type. These types come directly from IaC generator.

*To resolve write-only properties*:: + . Identify write-only properties to resolve from the *Warnings* section of your CDK project'sReadMefile. Here, you can take note of the resources in your CDK app that may contain write-only properties and identify the write-only property types that were discovered. + .. For ```MUTUALLY\_EXCLUSIVE\_PROPERTIES+, determine which mutually exclusive property to configure in your {aws} CDK app. .. For +MUTUALLY\_EXCLUSIVE\_TYPES, determine which accepted type that you will use to configure the property. .. For UNSUPPORTED\_PROPERTIES```+, determine if the property is optional or required. Then, configure as necessary. . Use guidance from https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/generate-IaC-write-only-properties.html[IaC generator and write-only properties] to reference what the warning types mean. . In your CDK app, write-only property values to resolve will also be specified in the +Props` section of your app. Provide the correct values here. For property descriptions and guidance, you can reference the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-construct-library.html[{aws} CDK API Reference]. +

The following is an example of the Props section within a migrated CDK app with two write-only properties to resolve: + [source,javascript,subs=“verbatim,attributes”] — export interface MyTestAppStackProps extends cdk.StackProps { /\*\*

* The Amazon S3 key of the deployment package. \*/ readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Keym8P82: string; /\*\*
* An Amazon S3 bucket in the same {aws} Region as your function. The bucket can be in a different {aws} account. \*/ readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Bucketzidw8: string; } —

Once you resolve all write-only property values, you’re ready to prepare for deployment.

[#migrate-resources-file] === The migrate.json file

The {aws} CDK CLI creates a migrate.json file in your {aws} CDK project during migration. This file contains reference information on your deployed resources. When you deploy your CDK app for the first time, the {aws} CDK CLI uses this file to reference your deployed resources, associates your resources with the new {aws} CloudFormation stack, and deletes the file.

[#migrate-manage] == Manage and deploy your CDK app

When migrating to {aws} CDK, the new CDK app may not be deployment-ready immediately. This topic describes action items to consider while managing and deploying your new CDK app.

[#migrate-manage-prepare] === Prepare for deployment

Before deploying, you must prepare your CDK app.

*Synthesize your app*:: Use the cdk synth command to synthesize the stack in your CDK app into an {aws} CloudFormation template. + If you migrated from a deployed {aws} CloudFormation stack or template, you can compare the synthesized template to the migrated template to verify resource and property values. + To learn more about cdk synth, see xref:cli-synth[Synthesize stacks].

*Perform a diff*:: If you migrated from a deployed {aws} CloudFormation stack, you can use the cdk diff command to compare with the stack in your new CDK app. + To learn more about cdk diff, see xref:cli-diff[Compare stacks].

*Bootstrap your environment*:: If you are deploying from an {aws} environment for the first time, use cdk bootstrap to prepare your environment. To learn more, see xref:bootstrapping[{aws} CDK bootstrapping].

[#migrate-manage-deploy] === Deploy your CDK app

When you deploy a CDK app, the {aws} CDK CLI utilizes the {aws} CloudFormation service to provision your resources. Resources are bundled into a single stack in the CDK app and are deployed as a single {aws} CloudFormation stack.

Depending on where you migrated from, you can deploy to create a new {aws} CloudFormation stack or update an existing {aws} CloudFormation stack.

*Deploy to create a new {aws} CloudFormation stack*:: If you migrated from deployed resources, the {aws} CDK CLI will automatically create a new {aws} CloudFormation stack at deployment. Your deployed resources will be included in the new {aws} CloudFormation stack. + If you migrated from a local {aws} CloudFormation template that was never deployed, the {aws} CDK CLI will automatically create a new {aws} CloudFormation stack at deployment. + If you migrated from a deployed {aws} CloudFormation stack or local {aws} CloudFormation template that was previously deployed, you can deploy to create a new {aws} CloudFormation stack. To create a new stack, do the following: + –

* Deploy to a new {aws} environment. This consists of using a different {aws} account or deploying to a different {aws} Region.
* {blank}
* == If you want to deploy a new stack to the same {aws} environment of the migrated stack or template, you must modify the stack name in your CDK app to a new value. You must also modify all logical IDs of resources in your CDK app. Then, you can deploy to the same environment to create a new stack and new resources.

*Deploy to update an existing {aws} CloudFormation stack*:: If you migrated from a deployed {aws} CloudFormation stack or local {aws} CloudFormation template that was previously deployed, you can deploy to update the existing {aws} CloudFormation stack. + Verify that the stack name in your CDK app matches the stack name of the deployed {aws} CloudFormation stack and deploy to the same {aws} environment.

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#plugins] = Configure plugins for the CDK Toolkit :info\_titleabbrev: Configure plugins :keywords: CDK Command Line Interface, CDK CLI, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), credentials, plugins, CDK Toolkit, CDK Toolkit Library

== [abstract]

## The {aws} CDK Toolkit provides a plugin system that enables developers to extend its functionality.

// Content start

The {aws} CDK Toolkit supports plugins that add new capabilities to your CDK workflows. Plugins are primarily designed for use with the CDK Command Line Interface (CDK CLI), though they can also be used with the CDK Toolkit Library programmatically. They provide a standardized way to extend functionality, such as alternative credential sources.

= [NOTE]

# While the plugin system works with both the CDK CLI and CDK Toolkit Library, it is primarily intended for CLI usage. When using the CDK Toolkit Library programmatically, there are often simpler, more direct ways to accomplish the same tasks without plugins, particularly for credential management.

Currently, the CDK Toolkit supports one plugin capability:

* *Custom {aws} credential providers* - Create alternative methods to obtain {aws} credentials beyond the built-in mechanisms.

[#plugins-create] == How to create plugins

To create a CDK Toolkit plugin, you first create a Node.js module, authored in TypeScript or JavaScript, that can be loaded by the CDK Toolkit. This module exports an object with a specific structure that the CDK Toolkit can recognize. At minimum, the exported object must include a version identifier and an initialization function that receives an IPluginHost instance, which the plugin uses to register its capabilities.

==== [role=“tablist”] TypeScript:: + *CommonJS*::: + [source,typescript,subs=“verbatim,attributes”] — // Example plugin structure import type { IPluginHost, Plugin } from ‘@aws-cdk/cli-plugin-contract’;

const plugin: Plugin = { // Version of the plugin infrastructure (currently always ‘1’) version: ‘1’,

// Initialization function called when the plugin is loaded init(host: IPluginHost): void { // Register your plugin functionality with the host // For example, register a custom credential provider } };

== export = plugin;

* *ESM*:::

## [source,typescript,subs=“verbatim,attributes”]

* // Example plugin structure import type { IPluginHost, Plugin } from ‘@aws-cdk/cli-plugin-contract’;

const plugin: Plugin = { // Version of the plugin infrastructure (currently always ‘1’) version: ‘1’,

// Initialization function called when the plugin is loaded init(host: IPluginHost): void { // Register your plugin functionality with the host // For example, register a custom credential provider } };

== export { plugin as ‘module.exports’ };

JavaScript:: + *CommonJS*::: + [source,javascript,subs=“verbatim,attributes”] — const plugin = { // Version of the plugin infrastructure (currently always ‘1’) version: ‘1’,

// Initialization function called when the plugin is loaded init(host) { // Register your plugin functionality with the host // For example, register a custom credential provider } };

== module.exports = plugin;

* *ESM*:::

## [source,javascript,subs=“verbatim,attributes”]

* const plugin = { // Version of the plugin infrastructure (currently always ‘1’) version: ‘1’,
* // Initialization function called when the plugin is loaded init(host) { // Register your plugin functionality with the host // For example, register a custom credential provider } };

== export { plugin as ‘module.exports’ };

====

= [NOTE]

CDK Toolkit plugins are loaded as CommonJS modules. You can build your plugin as an ECMAScript module (ESM), but have to link:https://nodejs.org/api/modules.html#loading-ecmascript-modules-using-require[adhere to some constraints]:

–

* The module cannot contain top-level await, or import any modules that contain top-level await.
* {blank}
* == The module must ensure compatibility by exporting the plugin object under the module.exports export name: export { plugin as 'module.exports' }.
* ====

[#plugins-load-cli] == How to load plugins with the CDK CLI

You have two options to specify plugins:

*Option 1: Use the --plugin command line option*:: + [source,bash,subs=“verbatim,attributes”] —

= Load a single plugin

$ cdk list –plugin=my-custom-plugin

= Load multiple plugins

## $ cdk deploy –plugin=custom-plugin-1 –plugin=custom-plugin-2

* The value of the --plugin argument should be a JavaScript file that, when imported via the Node.js require() function, returns an object implementing the Plugin interface.

*Option 2: Add entries to configuration files*:: + You can add plugin specifications to the project-specific cdk.json file or the ~/.cdk.json global configuration file: + [source,json,subs=“verbatim,attributes”] — { “plugin”: [ “custom-plugin-1”, “custom-plugin-2” ] } — + With either approach, the CDK CLI will load the specified plugins before running commands.

[#plugins-load-library] === Load plugins in code with the CDK Toolkit Library

When working with the CDK Toolkit Library programmatically, you can load plugins directly in your code using the PluginHost instance from the @aws-cdk/toolkit-lib package. The PluginHost provides a load() method for loading plugins by module name or path.

==== [role=“tablist”] TypeScript:: + [source,typescript,subs=“verbatim,attributes”] — import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Create a Toolkit instance const toolkit = new Toolkit();

// Load a plugin by module name or path // The module must export an object matching the Plugin interface await toolkit.pluginHost.load(‘my-custom-plugin’);

// You can load multiple plugins if needed await toolkit.pluginHost.load(‘./path/to/another-plugin’);

// Now proceed with other CDK Toolkit Library operations // The plugin’s functionality will be available to the toolkit —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Toolkit } = require(‘@aws-cdk/toolkit-lib’);

// Create a Toolkit instance const toolkit = new Toolkit();

// Load a plugin by module name or path // The module must export an object matching the Plugin interface await toolkit.pluginHost.load(‘my-custom-plugin’);

// You can load multiple plugins if needed await toolkit.pluginHost.load(‘./path/to/another-plugin’);

// Now proceed with other CDK Toolkit Library operations // The plugin’s functionality will be available to the toolkit — ====

The load() method takes a single parameter, moduleSpec, which is the name or path of the plugin module to load. This can be either:

* A Node.js module name installed in the node\_modules directory.
* A relative or absolute file path to a JavaScript or TypeScript module.

[#plugins-credentials] == Implementing credential provider plugins

The current primary use case for plugins is to create custom {aws} credential providers. Like the {aws} CLI, the CDK CLI needs {aws} credentials for authentication and authorization. However, there are several scenarios where the standard credential resolution might fail:

* The initial set of credentials cannot be obtained.
* The account to which the initial credentials belong cannot be obtained.
* The account associated with the credentials is different from the account on which the CLI is trying to operate.

To address these scenarios, the CDK Toolkit supports credential provider plugins. These plugins implement the CredentialProviderSource interface from the @aws-cdk/cli-plugin-contract package and are registered to the Toolkit using the registerCredentialProviderSource method. This enables the CDK Toolkit to obtain {aws} credentials from non-standard sources like specialized authentication systems or custom credential stores.

To implement a custom credential provider, create a class that implements the required interface:

==== [role=“tablist”] TypeScript:: + *CommonJS*::: + [source,typescript,subs=“verbatim,attributes”] — import type { CredentialProviderSource, ForReading, ForWriting, IPluginHost, Plugin, PluginProviderResult, SDKv3CompatibleCredentials } from ‘@aws-cdk/cli-plugin-contract’;

class CustomCredentialProviderSource implements CredentialProviderSource { // Friendly name for the provider, used in error messages public readonly name: string = ‘custom-credential-provider’;

// Check if this provider is available on the current system public async isAvailable(): Promise++++++{ // Return false if the plugin cannot be used // For example, if it depends on files not present on the host return true; }++++++

// Check if this provider can provide credentials for a specific account public async canProvideCredentials(accountId: string): Promise++++++{ // Return false if the plugin cannot provide credentials for this account // For example, if the account is not managed by this credential system return true; // You can use patterns to filter specific accounts // return accountId.startsWith(‘123456’); }++++++

// Get credentials for the specified account and access mode // Returns PluginProviderResult which can be one of: // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024 and will reach end-of-life on Sept 8, 2025) // - SDKv3CompatibleCredentialProvider // - SDKv3CompatibleCredentials public async getProvider(accountId: string, mode: ForReading | ForWriting): Promise++++++{ // The access mode can be used to provide different credential sets const readOnly = mode === 0 satisfies ForReading;++++++

…. // Create appropriate credentials based on your authentication mechanism const credentials: SDKv3CompatibleCredentials = { accessKeyId: ‘AKIAIOSFODNN7EXAMPLE’, secretAccessKey: ‘wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY’, // Add sessionToken if using temporary credentials // sessionToken: ‘AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk’, // expireTime: new Date(Date.now() + 3600 \* 1000), // 1 hour from now };

return credentials; } } ….

const plugin: Plugin = { version: ‘1’, init(host: IPluginHost): void { // Register the credential provider to the PluginHost. host.registerCredentialProviderSource(new CustomCredentialProviderSource()); } };

== export = plugin;

* *ESM*:::

## [source,typescript,subs=“verbatim,attributes”]

* import type { CredentialProviderSource, ForReading, ForWriting, IPluginHost, Plugin, PluginProviderResult, SDKv3CompatibleCredentials } from ‘@aws-cdk/cli-plugin-contract’;

class CustomCredentialProviderSource implements CredentialProviderSource { // Friendly name for the provider, used in error messages public readonly name: string = ‘custom-credential-provider’;

// Check if this provider is available on the current system public async isAvailable(): Promise++++++{ // Return false if the plugin cannot be used // For example, if it depends on files not present on the host return true; }++++++

// Check if this provider can provide credentials for a specific account public async canProvideCredentials(accountId: string): Promise++++++{ // Return false if the plugin cannot provide credentials for this account // For example, if the account is not managed by this credential system return true; // You can use patterns to filter specific accounts // return accountId.startsWith(‘123456’); }++++++

// Get credentials for the specified account and access mode // Returns PluginProviderResult which can be one of: // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024 and will reach end-of-life on Sept 8, 2025) // - SDKv3CompatibleCredentialProvider // - SDKv3CompatibleCredentials public async getProvider(accountId: string, mode: ForReading | ForWriting): Promise++++++{ // The access mode can be used to provide different credential sets const readOnly = mode === 0 satisfies ForReading;++++++

…. // Create appropriate credentials based on your authentication mechanism const credentials: SDKv3CompatibleCredentials = { accessKeyId: ‘AKIAIOSFODNN7EXAMPLE’, secretAccessKey: ‘wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY’, // Add sessionToken if using temporary credentials // sessionToken: ‘AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk’, // expireTime: new Date(Date.now() + 3600 \* 1000), // 1 hour from now };

return credentials; } } ….

const plugin: Plugin = { version: ‘1’, init(host: IPluginHost): void { // Register the credential provider to the PluginHost. host.registerCredentialProviderSource(new CustomCredentialProviderSource()); } };

== export { plugin as ‘module.exports’ };

JavaScript:: + *CommonJS*::: + [source,javascript,subs=“verbatim,attributes”] — // Implement the CredentialProviderSource interface class CustomCredentialProviderSource { constructor() { // Friendly name for the provider, used in error messages this.name = ‘custom-credential-provider’; }

// Check if this provider is available on the current system async isAvailable() { // Return false if the plugin cannot be used // For example, if it depends on files not present on the host return true; }

// Check if this provider can provide credentials for a specific account async canProvideCredentials(accountId) { // Return false if the plugin cannot provide credentials for this account // For example, if the account is not managed by this credential system return true; // You can use patterns to filter specific accounts // return accountId.startsWith(‘123456’); }

// Get credentials for the specified account and access mode // Returns PluginProviderResult which can be one of: // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024 and will reach end-of-life on Sept 8, 2025) // - SDKv3CompatibleCredentialProvider // - SDKv3CompatibleCredentials async getProvider(accountId, mode) { // The access mode can be used to provide different credential sets const readOnly = mode === 0; // 0 indicates ForReading; 1 indicates ForWriting

…. // Create appropriate credentials based on your authentication mechanism const credentials = { accessKeyId: ‘ASIAIOSFODNN7EXAMPLE’, secretAccessKey: ‘wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY’, // Add sessionToken if using temporary credentials // sessionToken: ‘AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk’, // expireTime: new Date(Date.now() + 3600 \* 1000), // 1 hour from now };

return credentials; } } ….

const plugin = { version: ‘1’, init(host) { // Register the credential provider to the PluginHost. host.registerCredentialProviderSource(new CustomCredentialProviderSource()); } };

== module.exports = plugin;

* *ESM*:::

## [source,javascript,subs=“verbatim,attributes”]

* // Implement the CredentialProviderSource interface class CustomCredentialProviderSource { constructor() { // Friendly name for the provider, used in error messages this.name = ‘custom-credential-provider’; }

// Check if this provider is available on the current system async isAvailable() { // Return false if the plugin cannot be used // For example, if it depends on files not present on the host return true; }

// Check if this provider can provide credentials for a specific account async canProvideCredentials(accountId) { // Return false if the plugin cannot provide credentials for this account // For example, if the account is not managed by this credential system return true; // You can use patterns to filter specific accounts // return accountId.startsWith(‘123456’); }

// Get credentials for the specified account and access mode // Returns PluginProviderResult which can be one of: // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024 and will reach end-of-life on Sept 8, 2025) // - SDKv3CompatibleCredentialProvider // - SDKv3CompatibleCredentials async getProvider(accountId, mode) { // The access mode can be used to provide different credential sets const readOnly = mode === 0; // 0 indicates ForReading; 1 indicates ForWriting

…. // Create appropriate credentials based on your authentication mechanism const credentials = { accessKeyId: ‘ASIAIOSFODNN7EXAMPLE’, secretAccessKey: ‘wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY’, // Add sessionToken if using temporary credentials // sessionToken: ‘AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk’, // expireTime: new Date(Date.now() + 3600 \* 1000), // 1 hour from now };

return credentials; } } ….

const plugin = { version: ‘1’, init(host) { // Register the credential provider to the PluginHost. host.registerCredentialProviderSource(new CustomCredentialProviderSource()); } };

== export { plugin as ‘module.exports’ };

====

= [IMPORTANT]

# Credentials obtained from providers are cached by the CDK Toolkit. It’s strongly recommended that credential objects returned by your provider are self-refreshing to prevent expiration issues during long-running operations. For more information, see link:https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/Package/-aws-sdk-client-sts/Interface/Credentials/[Credentials] in the *{aws} SDK for JavaScript v3 documentation*.

[#plugins-learn] == Learn more

To learn more about CDK Toolkit plugins, see the link:https://github.com/aws/aws-cdk-cli/tree/main/packages/%40aws-cdk/cli-plugin-contract[{aws} CDK Toolkit Plugin Contract] in the *aws-cdk-cli GitHub repository*.

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#policy-validation-synthesis] = {aws} CDK policy validation at synthesis time :info\_titleabbrev: Policy validation :keywords: cdk, policy, validation

== [abstract]

## By using the appropriate policy validation plugin, you can make the {aws} CDK application check the generated {aws} CloudFormation template against your policies immediately after synthesis.

// Content start

[#policy-validation] == Policy validation at synthesis time

If you or your organization use any policy validation tool, such as https://docs.aws.amazon.com/cfn-guard/latest/ug/what-is-guard.html[{aws} CloudFormation Guard] or https://www.openpolicyagent.org/[OPA], to define constraints on your {aws} CloudFormation template, you can integrate them with the {aws} CDK at synthesis time. By using the appropriate policy validation plugin, you can make the {aws} CDK application check the generated {aws} CloudFormation template against your policies immediately after synthesis. If there are any violations, the synthesis will fail and a report will be printed to the console.

The validation performed by the {aws} CDK at synthesis time validate controls at one point in the deployment lifecycle, but they can’t affect actions that occur outside synthesis. Examples include actions taken directly in the console or via service APIs. They aren’t resistant to alteration of {aws} CloudFormation templates after synthesis. Some other mechanism to validate the same rule set more authoritatively should be set up independently, like https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/hooks.html[{aws} CloudFormation hooks] or https://docs.aws.amazon.com/config/latest/developerguide/WhatIsConfig.html[{aws} Config]. Nevertheless, the ability of the {aws} CDK to evaluate the rule set during development is still useful as it will improve detection speed and developer productivity.

The goal of {aws} CDK policy validation is to minimize the amount of set up needed during development, and make it as easy as possible.

= [NOTE]

This feature is considered experimental, and both the plugin API and the format of the validation report are subject to change in the future.

====

[#for-application-developers] == For application developers

To use one or more validation plugins in your application, use the policyValidationBeta1 property of Stage:

== [source,javascript,subs=“verbatim,attributes”]

import { CfnGuardValidator } from ‘@cdklabs/cdk-validator-cfnguard’; const app = new App({ policyValidationBeta1: [ new CfnGuardValidator() ], }); // only apply to a particular stage const prodStage = new Stage(app, ‘ProdStage’, { policyValidationBeta1: […], }); —

Immediately after synthesis, all plugins registered this way will be invoked to validate all the templates generated in the scope you defined. In particular, if you register the templates in the App object, all templates will be subject to validation.

= [WARNING]

Other than modifying the cloud assembly, plugins can do anything that your {aws} CDK application can. They can read data from the filesystem, access the network etc. It’s your responsibility as the consumer of a plugin to verify that it’s secure to use.

====

[#cfnguard-plugin] === {aws} CloudFormation Guard plugin

Using the link:https://github.com/cdklabs/cdk-validator-cfnguard[CfnGuardValidator] plugin allows you to use https://github.com/aws-cloudformation/cloudformation-guard[{aws} CloudFormation Guard] to perform policy validations. The CfnGuardValidator plugin comes with a select set of https://docs.aws.amazon.com/controltower/latest/userguide/proactive-controls.html[{aws} Control Tower proactive controls] built in. The current set of rules can be found in the https://github.com/cdklabs/cdk-validator-cfnguard/blob/main/README.md[project documentation]. As mentioned in xref:policy-validation[Policy validation at synthesis time], we recommend that organizations set up a more authoritative method of validation using https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/hooks.html[{aws} CloudFormation hooks].

For link:https://docs.aws.amazon.com/controltower/latest/userguide/what-is-control-tower.html[{aws} Control Tower] customers, these same proactive controls can be deployed across your organization. When you enable {aws} Control Tower proactive controls in your {aws} Control Tower environment, the controls can stop the deployment of non-compliant resources deployed via {aws} CloudFormation. For more information about managed proactive controls and how they work, see the https://docs.aws.amazon.com/controltower/latest/userguide/proactive-controls.html[{aws} Control Tower documentation].

These {aws} CDK bundled controls and managed {aws} Control Tower proactive controls are best used together. In this scenario you can configure this validation plugin with the same proactive controls that are active in your {aws} Control Tower cloud environment. You can then quickly gain confidence that your {aws} CDK application will pass the {aws} Control Tower controls by running cdk synth locally.

[#validation-report] === Validation Report

When you synthesize the {aws} CDK app the validator plugins will be called and the results will be printed. An example report is showing below.

== [source,none,subs=“verbatim,attributes”]

## Validation Report (CfnGuardValidator)

(Summary) ╔═══════════╤════════════════════════╗ ║ Status │ failure ║ ╟───────────┼────────────────────────╢ ║ Plugin │ CfnGuardValidator ║ ╚═══════════╧════════════════════════╝ (Violations) Ensure S3 Buckets are encrypted with a KMS CMK (1 occurrences) Severity: medium Occurrences:

…. - Construct Path: MyStack/MyCustomL3Construct/Bucket - Stack Template Path: ./cdk.out/MyStack.template.json - Creation Stack: └── MyStack (MyStack) │ Library: aws-cdk-lib.Stack │ Library Version: 2.50.0 │ Location: Object. (/home/johndoe/tmp/cdk-tmp-app/src/main.ts:25:20) └── MyCustomL3Construct (MyStack/MyCustomL3Construct) │ Library: N/A - (Local Construct) │ Library Version: N/A │ Location: new MyStack (/home/johndoe/tmp/cdk-tmp-app/src/main.ts:15:20) └── Bucket (MyStack/MyCustomL3Construct/Bucket) │ Library: aws-cdk-lib/aws-s3.Bucket │ Library Version: 2.50.0 │ Location: new MyCustomL3Construct (/home/johndoe/tmp/cdk-tmp-app/src/main.ts:9:20) - Resource Name: amzn-s3-demo-bucket - Locations: > BucketEncryption/ServerSideEncryptionConfiguration/0/ServerSideEncryptionByDefault/SSEAlgorithm Recommendation: Missing value for key SSEAlgorithm - must specify aws:kms How to fix: > Add to construct properties for cdk-app/MyStack/Bucket encryption: BucketEncryption.KMS ….

== Validation failed. See above reports for details

By default, the report will be printed in a human readable format. If you want a report in JSON format, enable it using the @aws-cdk/core:validationReportJson via the CLI or passing it directly to the application:

== [source,javascript,subs=“verbatim,attributes”]

const app = new App({ context: { ‘@aws-cdk/core:validationReportJson’: true }, }); —

Alternatively, you can set this context key-value pair using the cdk.json or cdk.context.json files in your project directory (see xref:context[Context values and the {aws} CDK]).

If you choose the JSON format, the {aws} CDK will print the policy validation report to a file called policy-validation-report.json in the cloud assembly directory. For the default, human-readable format, the report will be printed to the standard output.

[#plugin-authors] == For plugin authors

[#plugins] === Plugins

The {aws} CDK core framework is responsible for registering and invoking plugins and then displaying the formatted validation report. The responsibility of the plugin is to act as the translation layer between the {aws} CDK framework and the policy validation tool. A plugin can be created in any language supported by {aws} CDK. If you are creating a plugin that might be consumed by multiple languages then it’s recommended that you create the plugin in TypeScript so that you can use JSII to publish the plugin in each {aws} CDK language.

[#creating-plugins] === Creating plugins

The communication protocol between the {aws} CDK core module and your policy tool is defined by the IPolicyValidationPluginBeta1 interface. To create a new plugin you must write a class that implements this interface. There are two things you need to implement: the plugin name (by overriding the name property), and the validate() method.

The framework will call validate(), passing an IValidationContextBeta1 object. The location of the templates to be validated is given by templatePaths. The plugin should return an instance of ValidationPluginReportBeta1. This object represents the report that the user wil receive at the end of the synthesis.

== [source,javascript,subs=“verbatim,attributes”]

validate(context: IPolicyValidationContextBeta1): PolicyValidationReportBeta1 { // First read the templates using context.templatePaths… // …then perform the validation, and then compose and return the report. // Using hard-coded values here for better clarity: return { success: false, violations: [{ ruleName: ‘CKV\_AWS\_117’, description: ‘Ensure that {aws} Lambda function is configured inside a VPC’, fix: ‘https://docs.bridgecrew.io/docs/ensure-that-aws-lambda-function-is-configured-inside-a-vpc-1’, violatingResources: [{ resourceName: ‘MyFunction3BAA72D1’, templatePath: ‘/home/johndoe/myapp/cdk.out/MyService.template.json’, locations: ‘Properties/VpcConfig’, }], }], }; } —

Note that plugins aren’t allowed to modify anything in the cloud assembly. Any attempt to do so will result in synthesis failure.

If your plugin depends on an external tool, keep in mind that some developers may not have that tool installed in their workstations yet. To minimize friction, we highly recommend that you provide some installation script along with your plugin package, to automate the whole process. Better yet, run that script as part of the installation of your package. With npm, for example, you can add it to the postinstall link:https://docs.npmjs.com/cli/v9/using-npm/scripts[script] in the package.json file.

[#handling-exemptions] === Handling Exemptions

If your organization has a mechanism for handling exemptions, it can be implemented as part of the validator plugin.

An example scenario to illustrate a possible exemption mechanism:

* An organization has a rule that public Amazon S3 buckets aren’t allowed, *except* for under certain scenarios.
* A developer is creating an Amazon S3 bucket that falls under one of those scenarios and requests an exemption (create a ticket for example).
* Security tooling knows how to read from the internal system that registers exemptions

In this scenario the developer would request an exception in the internal system and then will need some way of “registering” that exception. Adding on to the guard plugin example, you could create a plugin that handles exemptions by filtering out the violations that have a matching exemption in an internal ticketing system.

See the existing plugins for example implementations.

* https://github.com/cdklabs/cdk-validator-cfnguard[@cdklabs/cdk-validator-cfnguard]

include::attributes.txt[]

// Attributes [.topic] [#how-to-set-cw-alarm] = Set a CloudWatch alarm :info\_titleabbrev: Set CloudWatch alarm

// Content start

Use the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_cloudwatch-readme.html[aws-cloudwatch] package to set up Amazon CloudWatch alarms on CloudWatch metrics. You can use predefined metrics or create your own.

[#how-to-set-cw-alarm-use-metric] == Use an existing metric

Many {aws} Construct Library modules let you set an alarm on an existing metric by passing the metric’s name to a convenience method on an instance of an object that has metrics. For example, given an Amazon SQS queue, you can get the metric *ApproximateNumberOfMessagesVisible* from the queue’s link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_sqs.Queue.html#metricmetricname-props[metric()] method:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const metric = queue.metric(“ApproximateNumberOfMessagesVisible”); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const metric = queue.metric(“ApproximateNumberOfMessagesVisible”); —

Python:: + [source,python,subs=“verbatim,attributes”] — metric = queue.metric(“ApproximateNumberOfMessagesVisible”) —

Java:: + [source,java,subs=“verbatim,attributes”] — Metric metric = queue.metric(“ApproximateNumberOfMessagesVisible”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var metric = queue.Metric(“ApproximateNumberOfMessagesVisible”); — ====

[#how-to-set-cw-alarm-new-metric] == Create your own metric

Create your own link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_cloudwatch.Metric.html[metric] as follows, where the *namespace* value should be something like *{aws}/SQS* for an Amazon SQS queue. You also need to specify your metric’s name and dimension:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const metric = new cloudwatch.Metric({ namespace: ‘MyNamespace’, metricName: ‘MyMetric’, dimensionsMap: { MyDimension: ‘MyDimensionValue’ } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const metric = new cloudwatch.Metric({ namespace: ‘MyNamespace’, metricName: ‘MyMetric’, dimensionsMap: { MyDimension: ‘MyDimensionValue’ } }); —

Python:: + [source,python,subs=“verbatim,attributes”] — metric = cloudwatch.Metric( namespace=“MyNamespace”, metric\_name=“MyMetric”, dimensionsMap=dict(MyDimension=“MyDimensionValue”) ) —

Java:: + [source,java,subs=“verbatim,attributes”] — Metric metric = Metric.Builder.create() .namespace(“MyNamespace”) .metricName(“MyMetric”) .dimensionsMap(java.util.Map.of( // Java 9 or later “MyDimension”, “MyDimensionValue”)) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var metric = new Metric(this, “Metric”, new MetricProps { Namespace = “MyNamespace”, MetricName = “MyMetric”, Dimensions = new Dictionary<string, object> { { “MyDimension”, “MyDimensionValue” } } }); — ====

[#how-to-set-cw-alarm-create] == Create the alarm

Once you have a metric, either an existing one or one you defined, you can create an alarm. In this example, the alarm is raised when there are more than 100 of your metric in two of the last three evaluation periods. You can use comparisons such as less-than in your alarms via the comparisonOperator property. Greater-than-or-equal-to is the {aws} CDK default, so we don’t need to specify it.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const alarm = new cloudwatch.Alarm(this, ‘Alarm’, { metric: metric, threshold: 100, evaluationPeriods: 3, datapointsToAlarm: 2, }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const alarm = new cloudwatch.Alarm(this, ‘Alarm’, { metric: metric, threshold: 100, evaluationPeriods: 3, datapointsToAlarm: 2 }); —

Python:: + [source,python,subs=“verbatim,attributes”] — alarm = cloudwatch.Alarm(self, “Alarm”, metric=metric, threshold=100, evaluation\_periods=3, datapoints\_to\_alarm=2 ) —

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.cloudwatch.Alarm; import software.amazon.awscdk.services.cloudwatch.Metric;

Alarm alarm = Alarm.Builder.create(this, “Alarm”) .metric(metric) .threshold(100) .evaluationPeriods(3) .datapointsToAlarm(2).build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var alarm = new Alarm(this, “Alarm”, new AlarmProps { Metric = metric, Threshold = 100, EvaluationPeriods = 3, DatapointsToAlarm = 2 }); — ====

An alternative way to create an alarm is using the metric’s link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_cloudwatch.Metric.html#createwbralarmscope-id-props[createAlarm()] method, which takes essentially the same properties as the Alarm constructor. You don’t need to pass in the metric, because it’s already known.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — metric.createAlarm(this, ‘Alarm’, { threshold: 100, evaluationPeriods: 3, datapointsToAlarm: 2, }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — metric.createAlarm(this, ‘Alarm’, { threshold: 100, evaluationPeriods: 3, datapointsToAlarm: 2, }); —

Python:: + [source,python,subs=“verbatim,attributes”] — metric.create\_alarm(self, “Alarm”, threshold=100, evaluation\_periods=3, datapoints\_to\_alarm=2 ) —

Java:: + [source,java,subs=“verbatim,attributes”] — metric.createAlarm(this, “Alarm”, new CreateAlarmOptions.Builder() .threshold(100) .evaluationPeriods(3) .datapointsToAlarm(2) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — metric.CreateAlarm(this, “Alarm”, new CreateAlarmOptions { Threshold = 100, EvaluationPeriods = 3, DatapointsToAlarm = 2 }); — ====

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: CDK stages :info\_abstract: An {aws} Cloud Development Kit ({aws} CDK) stage represents a group of one or more CDK stacks that are configured to + deploy together. Use stages to deploy the same grouping of stacks to multiple environments, such as development, testing, + and production. :keywords: {aws} CDK, CDK stages, Deployment, CDK stacks

[#stages] = Introduction to {aws} CDK stages

== [abstract]

## An {aws} Cloud Development Kit ({aws} CDK) *stage* represents a group of one or more CDK stacks that are configured to deploy together. Use stages to deploy the same grouping of stacks to multiple environments, such as development, testing, and production.

// Content start

An {aws} Cloud Development Kit ({aws} CDK) *stage* represents a group of one or more CDK stacks that are configured to deploy together. Use stages to deploy the same grouping of stacks to multiple environments, such as development, testing, and production.

To configure a CDK stage, import and use the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stage.html[Stage] construct.

The following is a basic example that defines a CDK stage named MyAppStage. We add two CDK stacks, named AppStack and DatabaseStack to our stage. For this example, AppStack contains application resources and DatabaseStack contains database resources. We then create two instances of MyAppStage, for development and production environments:

==== [role=“tablist”] TypeScript:: In cdk-demo-app/lib/app-stack.ts: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’;

// Define the app stack export class AppStack extends cdk.Stack { constructor(scope: Construct, id: string, props?: cdk.StackProps) { super(scope, id, props); // The code that defines your application goes here } } — +

In cdk-demo-app/lib/database-stack.ts: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’;

// Define the database stack export class DatabaseStack extends cdk.Stack { constructor(scope: Construct, id: string, props?: cdk.StackProps) { super(scope, id, props); // The code that defines your database goes here } } — +

In cdk-demo-app/lib/my-stage.ts: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import { Stage } from ‘aws-cdk-lib’; import { AppStack } from ‘./app-stack’; import { DatabaseStack } from ‘./database-stack’;

// Define the stage export class MyAppStage extends Stage { constructor(scope: Construct, id: string, props?: cdk.StageProps) { super(scope, id, props);

// Add both stacks to the stage new AppStack(this, ‘AppStack’); new DatabaseStack(this, ‘DatabaseStack’); } } —- +

In cdk-demo-app/bin/cdk-demo-app.ts: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node import ‘source-map-support/register’; import \* as cdk from ‘aws-cdk-lib’; import { MyAppStage } from ‘../lib/my-stage’;

// Create a CDK app const app = new cdk.App();

// Create the development stage new MyAppStage(app, ‘Dev’, { env: { account: ‘123456789012’, region: ‘us-east-1’ } });

// Create the production stage new MyAppStage(app, ‘Prod’, { env: { account: ‘098765432109’, region: ‘us-east-1’ } }); —

JavaScript:: In cdk-demo-app/lib/app-stack.js: + [source,javascript,subs=“verbatim,attributes”] — const { Stack } = require(‘aws-cdk-lib’);

class AppStack extends Stack { constructor(scope, id, props) { super(scope, id, props);

// The code that defines your application goes here } }

== module.exports = { AppStack }

In cdk-demo-app/lib/database-stack.js: + [source,javascript,subs=“verbatim,attributes”] — const { Stack } = require(‘aws-cdk-lib’);

class DatabaseStack extends Stack { constructor(scope, id, props) { super(scope, id, props);

// The code that defines your database goes here } }

== module.exports = { DatabaseStack }

In cdk-demo-app/lib/my-stage.js: + [source,javascript,subs=“verbatim,attributes”] — const { Stage } = require(‘aws-cdk-lib’); const { AppStack } = require(‘./app-stack’); const { DatabaseStack } = require(‘./database-stack’);

// Define the stage class MyAppStage extends Stage { constructor(scope, id, props) { super(scope, id, props);

// Add both stacks to the stage new AppStack(this, ‘AppStack’); new DatabaseStack(this, ‘DatabaseStack’); } }

== module.exports = { MyAppStage };

In cdk-demo-app/bin/cdk-demo-app.js: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node

const cdk = require(‘aws-cdk-lib’); const { MyAppStage } = require(‘../lib/my-stage’);

// Create the CDK app const app = new cdk.App();

// Create the development stage new MyAppStage(app, ‘Dev’, { env: { account: ‘123456789012’, region: ‘us-east-1’, }, });

// Create the production stage new MyAppStage(app, ‘Prod’, { env: { account: ‘098765432109’, region: ‘us-east-1’, }, }); —

Python:: In cdk-demo-app/cdk\_demo\_app/app\_stack.py: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import Stack from constructs import Construct

= Define the app stack

class AppStack(Stack): def *init*(self, scope: Construct, construct\_id: str, **kwargs) -> None: super().*init*(scope, construct\_id,** kwargs)

# The code that defines your application goes here ---- +

In cdk-demo-app/cdk\_demo\_app/database\_stack.py: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import Stack from constructs import Construct

= Define the database stack

class DatabaseStack(Stack): def *init*(self, scope: Construct, construct\_id: str, **kwargs) -> None: super().*init*(scope, construct\_id,** kwargs)

# The code that defines your database goes here ---- +

In cdk-demo-app/cdk\_demo\_app/my\_stage.py: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import Stage from constructs import Construct from .app\_stack import AppStack from .database\_stack import DatabaseStack

= Define the stage

class MyAppStage(Stage): def *init*(self, scope: Construct, id: str, **kwargs) -> None: super().*init*(scope, id,** kwargs)

# Add both stacks to the stage  
 AppStack(self, "AppStack")  
 DatabaseStack(self, "DatabaseStack") ---- +

In cdk-demo-app/app.py: + [source,python,subs=“verbatim,attributes”] — #!/usr/bin/env python3 import os

import aws\_cdk as cdk

from cdk\_demo\_app.my\_stage import MyAppStage

= Create a CDK app

app = cdk.App()

= Create the development stage

MyAppStage(app, ‘Dev’, env=cdk.Environment(account=‘123456789012’, region=‘us-east-1’), )

= Create the production stage

MyAppStage(app, ‘Prod’, env=cdk.Environment(account=‘098765432109’, region=‘us-east-1’), )

== app.synth()

Java:: In cdk-demo-app/src/main/java/com/myorg/AppStack.java: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps;

public class AppStack extends Stack { public AppStack(final Construct scope, final String id) { this(scope, id, null); }

…. public AppStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

// The code that defines your application goes here

} } —- + ….

In cdk-demo-app/src/main/java/com/myorg/DatabaseStack.java: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps;

public class DatabaseStack extends Stack { public DatabaseStack(final Construct scope, final String id) { this(scope, id, null); }

…. public DatabaseStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

// The code that defines your database goes here

} } —- + ….

In cdk-demo-app/src/main/java/com/myorg/MyAppStage.java: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.amazon.awscdk.Stage; import software.amazon.awscdk.StageProps; import software.constructs.Construct;

// Define the stage public class MyAppStage extends Stage { public MyAppStage(final Construct scope, final String id, final software.amazon.awscdk.Environment env) { super(scope, id, StageProps.builder().env(env).build());

// Add both stacks to the stage  
 new AppStack(this, "AppStack");  
 new DatabaseStack(this, "DatabaseStack");

} } —- +

In cdk-demo-app/src/main/java/com/myorg/CdkDemoAppApp.java: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.amazon.awscdk.App; import software.amazon.awscdk.Environment; import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class CdkDemoAppApp { public static void main(final String[] args) {

…. // Create a CDK app App app = new App();

// Create the development stage  
new MyAppStage(app, "Dev", Environment.builder()  
 .account("123456789012")  
 .region("us-east-1")  
 .build());  
  
// Create the production stage  
new MyAppStage(app, "Prod", Environment.builder()  
.account("098765432109")  
.region("us-east-1")  
.build());  
  
app.synth();

} } —- ….

C#:: In cdk-demo-app/src/CdkDemoApp/AppStack.cs: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs;

namespace CdkDemoApp { public class AppStack : Stack { internal AppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // The code that defines your application goes here } } } — +

In cdk-demo-app/src/CdkDemoApp/DatabaseStack.cs: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs;

namespace CdkDemoApp { public class DatabaseStack : Stack { internal DatabaseStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // The code that defines your database goes here } } } — +

In cdk-demo-app/src/CdkDemoApp/MyAppStage.cs: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs;

namespace CdkDemoApp { // Define the stage public class MyAppStage : Stage { internal MyAppStage(Construct scope, string id, Environment env) : base(scope, id, new StageProps { Env = env }) { // Add both stacks to the stage new AppStack(this, “AppStack”); new DatabaseStack(this, “DatabaseStack”); } } } — +

In cdk-demo-app/src/CdkDemoApp/program.cs: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using System; using System.Collections.Generic; using System.Linq;

namespace CdkDemoApp { sealed class Program { public static void Main(string[] args) { // Create a CDK app var app = new App();

…. // Create the development stage new MyAppStage(app, “Dev”, new Amazon.CDK.Environment { Account = “123456789012”, Region = “us-east-1” });

// Create the production stage  
 new MyAppStage(app, "Prod", new Amazon.CDK.Environment  
 {  
 Account = "098765432109",  
 Region = "us-east-1"  
 });  
  
 app.Synth();  
}

} } —- ….

Go:: In cdk-demo-app/cdk-demo-app.go: + [source,go,subs=“verbatim,attributes”] — package main

import ( “github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/constructs-go/constructs/v10” “github.com/aws/jsii-runtime-go” )

// Define the app stack type AppStackProps struct { awscdk.StackProps }

func NewAppStack(scope constructs.Construct, id string, props \*AppStackProps) awscdk.Stack { stack := awscdk.NewStack(scope, &id, &props.StackProps)

…. // The code that defines your application goes here

return stack } ….

// Define the database stack type DatabaseStackProps struct { awscdk.StackProps }

func NewDatabaseStack(scope constructs.Construct, id string, props \*DatabaseStackProps) awscdk.Stack { stack := awscdk.NewStack(scope, &id, &props.StackProps)

…. // The code that defines your database goes here

return stack } ….

// Define the stage type MyAppStageProps struct { awscdk.StageProps }

func NewMyAppStage(scope constructs.Construct, id string, props \*MyAppStageProps) awscdk.Stage { stage := awscdk.NewStage(scope, &id, &props.StageProps)

…. // Add both stacks to the stage NewAppStack(stage, “AppStack”, &AppStackProps{ StackProps: awscdk.StackProps{ Env: props.Env, }, })

NewDatabaseStack(stage, “DatabaseStack”, &DatabaseStackProps{ StackProps: awscdk.StackProps{ Env: props.Env, }, })

return stage } ….

func main() { defer jsii.Close()

…. // Create a CDK app app := awscdk.NewApp(nil)

// Create the development stage NewMyAppStage(app, “Dev”, &MyAppStageProps{ StageProps: awscdk.StageProps{ Env: &awscdk.Environment{ Account: jsii.String(“123456789012”), Region: jsii.String(“us-east-1”), }, }, })

// Create the production stage NewMyAppStage(app, “Prod”, &MyAppStageProps{ StageProps: awscdk.StageProps{ Env: &awscdk.Environment{ Account: jsii.String(“098765432109”), Region: jsii.String(“us-east-1”), }, }, })

app.Synth(nil) } ….

func env() \*awscdk.Environment { return nil } — ====

When we run cdk synth, two cloud assemblies are created in cdk.out. These two cloud assemblies contain the synthesized {aws} CloudFormation template and assets for each stage. The following is snippet of our project directory:

==== [role=“tablist”] TypeScript:: + [source,none,subs=“verbatim,attributes”] — cdk-demo-app ├── bin │ └── cdk-demo-app.ts ├── cdk.out │ ├── assembly-Dev │ │ ├── DevAppStack++++++.assets.json │ │ ├── DevAppStack++++++.template.json │ │ ├── DevDatabaseStack++++++.assets.json │ │ ├── DevDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── assembly-Prod │ │ ├── ProdAppStack++++++.assets.json │ │ ├── ProdAppStack++++++.template.json │ │ ├── ProdDatabaseStack++++++.assets.json │ │ ├── ProdDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json └── lib ├── app-stack.ts ├── database-stack.ts └── my-stage.ts —-++++++++++++++++++++++++++++++++++++++++++++++++

JavaScript:: + [source,none,subs=“verbatim,attributes”] — cdk-demo-app ├── bin │ └── cdk-demo-app.js ├── cdk.out │ ├── assembly-Dev │ │ ├── DevAppStack++++++.assets.json │ │ ├── DevAppStack++++++.template.json │ │ ├── DevDatabaseStack++++++.assets.json │ │ ├── DevDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── assembly-Prod │ │ ├── ProdAppStack++++++.assets.json │ │ ├── ProdAppStack++++++.template.json │ │ ├── ProdDatabaseStack++++++.assets.json │ │ ├── ProdDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json └── lib ├── app-stack.js ├── database-stack.js └── my-stage.js —-++++++++++++++++++++++++++++++++++++++++++++++++

Python:: + [source,none,subs=“verbatim,attributes”] — cdk-demo-app ├── app.py ├── cdk.out │ ├── assembly-Dev │ │ ├── DevAppStack++++++.assets.json │ │ ├── DevAppStack++++++.template.json │ │ ├── DevDatabaseStack++++++.assets.json │ │ ├── DevDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── assembly-Prod │ │ ├── ProdAppStack++++++.assets.json │ │ ├── ProdAppStack++++++.template.json │ │ ├── ProdDatabaseStack++++++.assets.json │ │ ├── ProdDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── cdk.out │ ├── manifest.json │ └── tree.json └── cdk\_demo\_app ├── **init**.py ├── app\_stack.py ├── database\_stack.py └── my\_stage.py —-++++++++++++++++++++++++++++++++++++++++++++++++

Java:: + [source,none,subs=“verbatim,attributes”] — cdk-demo-app ├── cdk.out │ ├── assembly-Dev │ │ ├── DevAppStack++++++.assets.json │ │ ├── DevAppStack++++++.template.json │ │ ├── DevDatabaseStack++++++.assets.json │ │ ├── DevDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── assembly-Prod │ │ ├── ProdAppStack++++++.assets.json │ │ ├── ProdAppStack++++++.template.json │ │ ├── ProdDatabaseStack++++++.assets.json │ │ ├── ProdDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── cdk.out │ ├── manifest.json │ └── tree.json └── src └── main └── java └── com └── myorg ├── AppStack.java ├── CdkDemoAppApp.java ├── DatabaseStack.java └── MyAppStage.java —-++++++++++++++++++++++++++++++++++++++++++++++++

C#:: + [source,none,subs=“verbatim,attributes”] — cdk-demo-app ├── cdk.out │ ├── assembly-Dev │ │ ├── DevAppStack++++++.assets.json │ │ ├── DevAppStack++++++.template.json │ │ ├── DevDatabaseStack++++++.assets.json │ │ ├── DevDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── assembly-Prod │ │ ├── ProdAppStack++++++.assets.json │ │ ├── ProdAppStack++++++.template.json │ │ ├── ProdDatabaseStack++++++.assets.json │ │ ├── ProdDatabaseStack++++++.template.json │ │ ├── cdk.out │ │ └── manifest.json │ ├── cdk.out │ ├── manifest.json │ └── tree.json └── src └── CdkDemoApp ├── AppStack.cs ├── DatabaseStack.cs ├── MyAppStage.cs └── Program.cs —-++++++++++++++++++++++++++++++++++++++++++++++++

Go:: + [source,none,subs=“verbatim,attributes”] — cdk-demo-app ├── cdk-demo-app.go └── cdk.out ├── assembly-Dev │ ├── DevAppStack++++++.assets.json │ ├── DevAppStack++++++.template.json │ ├── DevDatabaseStack++++++.assets.json │ ├── DevDatabaseStack++++++.template.json │ ├── cdk.out │ └── manifest.json ├── assembly-Prod │ ├── ProdAppStack++++++.assets.json │ ├── ProdAppStack++++++.template.json │ ├── ProdDatabaseStack++++++.assets.json │ ├── ProdDatabaseStack++++++.template.json │ ├── cdk.out │ └── manifest.json ├── cdk.out ├── manifest.json └── tree.json —-++++++++++++++++++++++++++++++++++++++++++++++++

When we list our stacks with cdk list, we see a total of four stacks:

== [source,none,subs=“verbatim,attributes”]

$ cdk list Dev/AppStack (Dev-AppStack) Dev/DatabaseStack (Dev-DatabaseStack) Prod/AppStack (Prod-AppStack) Prod/DatabaseStack (Prod-DatabaseStack) —

To deploy a specific stage, we run cdk deploy and provide the stacks to deploy. The following is an example that uses the /\* wildcard to deploy both stacks in our Dev stage:

== [source,none,subs=“verbatim,attributes”]

$ cdk deploy <“Dev/\*“>

✨ Synthesis time: 3.18s

Dev/AppStack (Dev-AppStack) Dev/AppStack (Dev-AppStack): deploying… [1/2]

✅ Dev/AppStack (Dev-AppStack)

✨ Deployment time: 1.11s

Stack ARN: …

✨ Total time: 4.29s

Dev/DatabaseStack (Dev-DatabaseStack) Dev/DatabaseStack (Dev-DatabaseStack): deploying… [2/2]

✅ Dev/DatabaseStack (Dev-DatabaseStack)

✨ Deployment time: 1.09s

Stack ARN: …

== ✨ Total time: 4.27s

====

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-actions] = Configuring CDK Toolkit programmatic actions :info\_titleabbrev: Configure programmatic actions :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, synthesize

== [abstract]

## The {aws} CDK Toolkit Library provides programmatic interfaces for application lifecycle actions such as synthesis, deployment, and stack management. This guide explains how to use each action in your code.

// Content start

The {aws} CDK Toolkit Library provides programmatic interfaces for application lifecycle actions such as synthesis, deployment, and stack management. This guide explains how to use each action in your code.

[#toolkit-library-actions-synth] == Generating cloud assemblies with synth

The synth action generates a cloud assembly from your cloud assembly source. For more information about synthesis, see xref:configure-synth[Configure and perform CDK stack synthesis]. A cloud assembly contains the following deployment artifacts from your CDK app:

* {aws} CloudFormation templates that define your infrastructure.
* Assets such as Lambda function code or Docker images.
* Deployment metadata and configuration.

Here’s how to use the synth action to create a cloud assembly:

== [source,typescript,subs=“verbatim,attributes”]

// Create a toolkit instance const toolkit = new Toolkit();

// Create a cloud assembly source from a TypeScript app const cloudAssemblySource = await toolkit.fromCdkApp(“ts-node app.ts”);

// Generate a cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

// Use the cloud assembly for operations await toolkit.list(cloudAssembly); await toolkit.deploy(cloudAssembly); await toolkit.diff(cloudAssembly);

// Query information from the cloud assembly const template = cloudAssembly.getStack(“my-stack”).template; —

= [TIP]

# Using a cloud assembly can optimize performance when you need to perform multiple operations, since synthesis only happens once. For more information about managing cloud assemblies, including caching and disposal, see xref:toolkit-library-configure-ca-cache[Create and manage cloud assemblies].

[#toolkit-library-actions-list] == Viewing stack information with list

The list action retrieves information about the stacks in your CDK application, including their dependencies and current status. Use this action to inspect your infrastructure before deployment or to generate reports.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Get information about specific stacks const stackDetails = await toolkit.list(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“my-stack”], // Only include stacks matching this pattern } });

// Process the returned stack information for (const stack of stackDetails) { console.log(Stack: ${stack.id}, Dependencies: ${stack.dependencies}); } —

[#toolkit-library-actions-deploy] == Provisioning infrastructure with deploy

The deploy action provisions or updates your infrastructure in {aws} using the cloud assembly produced during synthesis. For an introduction to deploying, see xref:deploy[Deploy {aws} CDK applications]. You can control deployment options such as stack selection, parameter values, and rollback behavior.

== [source,typescript,subs=“verbatim,attributes”]

// Deploy stacks with parameter values await toolkit.deploy(cloudAssemblySource, { parameters: StackParameters.exactly({ “MyStack”: { “BucketName”: “amzn-s3-demo-bucket” } }) }); —

The deploy action supports different deployment methods to accommodate various workflows. For most scenarios, especially in production environments, we recommend using the default deployment method which uses CloudFormation change sets. For development environments where iteration speed is important, you can use alternative methods like hotswap.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Deploy using default deployment method (recommended for production) await toolkit.deploy(cloudAssemblySource, { parameters: StackParameters.exactly({ “MyStack”: { “BucketName”: “amzn-s3-demo-bucket” } }) });

// For development environments only: Deploy with hotswap for faster iterations // Note: We recommend using default deployment methods for production environments await toolkit.deploy(cloudAssemblySource, { deploymentMethod: { method: “hotswap”, fallback: true }, // Faster but introduces drift stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“dev-stack”] } }); —

[#toolkit-library-actions-rollback] == Reverting failed deployments with rollback

The rollback action returns a stack to its last stable state when a deployment fails and cannot be automatically reversed. Use this action to recover from failed deployments that require manual intervention.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Roll back stacks to their last stable state await toolkit.rollback(cloudAssemblySource, { orphanFailedResources: false, // When true, removes failed resources from CloudFormation management stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“failed-stack”] } }); —

[#toolkit-library-actions-watch] == Monitoring changes with watch

The watch action continuously monitors your CDK app for local file changes and automatically performs deployments or hotswaps. This creates a file watcher that runs until your code exits or is terminated.

= [WARNING]

# Hotswap deployments update resources directly without going through CloudFormation when possible, making updates faster during development. This is enabled by default for the watch command. While this speeds up the development cycle, it introduces drift between your CloudFormation templates and deployed resources. Therefore, we recommend that you don’t use hotswaps in production environments.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Start watching for changes const watcher = await toolkit.watch(cloudAssemblySource, { include: [“lib/\_\*/\_.ts”], // Only watch TypeScript files in the lib directory exclude: [“\_\*/\_.test.ts”], // Ignore test files deploymentMethod: { method: “hotswap” }, // This is the default, shown here for clarity stacks: { strategy: StackSelectionStrategy.ALL // Watch all stacks } });

// Later in your code, you can explicitly stop watching: // await watcher.dispose(); —

The watch function returns an IWatcher object that allows you to explicitly control when to stop watching. Call the dispose() method on this object when you want to end the watch process.

[#toolkit-library-actions-destroy] == Removing infrastructure with destroy

The destroy action removes CDK stacks and their associated resources from {aws}. Use this action to clean up resources when they’re no longer needed.

= [IMPORTANT]

# The destroy action permanently removes resources without prompting for confirmation, unlike the CLI version of this command. Make sure you have backups of any important data before destroying stacks.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Remove specific stacks and their resources await toolkit.destroy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“dev-stack”], // Only destroy stacks matching this pattern } }); —

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-configure-ca] = Managing {aws} CDK Toolkit Library cloud assembly sources :info\_titleabbrev: Managing cloud assembly sources :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, synthesize, cloud assembly, cloud assembly source

== [abstract]

## Use the {aws} CDK Toolkit Library to configure cloud assembly sources and customize how you deploy your CDK applications. Learn how to configure cloud assembly sources to meet your deployment requirements and workflow needs.

// Content start

Use the {aws} CDK Toolkit Library to configure cloud assembly sources and customize how you deploy your CDK applications. This guide shows you how to configure cloud assembly sources to meet your deployment requirements and workflow needs.

Before you use the CDK Toolkit, specify a cloud assembly source. A *cloud assembly source* provides instructions for generating a cloud assembly from your CDK app. The resulting cloud assembly contains the synthesized infrastructure artifacts that the CDK Toolkit deploys to {aws}.

The CDK Toolkit Library offers several approaches to configure cloud assembly sources, each suited for different scenarios and workflows.

[#toolkit-library-configure-ca-options] == Selecting a cloud assembly source

[cols=“1,1,1”, options=“header”] |=== |Method |Best For |Consideration

|=== | fromCdkApp | Working with existing CDK applications in any supported language. | Requires the appropriate language runtime to be installed. |===

|=== | fromAssemblyBuilder | Creating CDK constructs inline with full control over the synthesis process. | Provides low-level access to CDK functionality and can be used to build custom versions of other methods like fromCdkApp. |===

|=== | fromAssemblyDirectory | Using pre-synthesized cloud assemblies. | Faster execution as synthesis step is skipped. |===

|=== | Custom Source | Extremely specialized scenarios requiring complete custom implementation. | Requires implementing the ICloudAssemblySource interface from scratch; lacks built-in functionality like context lookups; rarely needed for most use cases. |===

[#toolkit-library-configure-ca-how] == Configuring your cloud assembly source

[#toolkit-library-configure-ca-how-app] === From an existing CDK app

Use the fromCdkApp method to work with CDK apps written in any supported language. This approach is ideal when you have an existing CDK application and want to deploy it programmatically.

== [source,typescript,subs=“verbatim,attributes”]

import { App } from ‘aws-cdk-lib’; import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Create a toolkit instance const toolkit = new Toolkit();

// TypeScript app const cloudAssemblySource = await toolkit.fromCdkApp(“ts-node app.ts”);

// Deploy a specific stack from the assembly await toolkit.deploy(cloudAssemblySource, { stacks: [‘MyStack’] });

// Other language examples: // JavaScript app // const cloudAssemblySource = await toolkit.fromCdkApp(“node app.js”);

// Python app // const cloudAssemblySource = await toolkit.fromCdkApp(“python app.py”);

// Java app // const cloudAssemblySource = await toolkit.fromCdkApp(“mvn -e -q exec:java -Dexec.mainClass=com.mycompany.app.App”); —

[#toolkit-library-configure-ca-how-builder] === From an inline assembly builder

Create a CDK app directly in your code using an assembly builder function. This approach is useful for simple deployments or testing scenarios where you want to define your infrastructure inline.

== [source,typescript,subs=“verbatim,attributes”]

import { App, Stack, RemovalPolicy, StackProps } from ‘aws-cdk-lib’; import { Bucket } from ‘aws-cdk-lib/aws-s3’; import { Toolkit } from ‘@aws-cdk/toolkit-lib’; import { Construct } from ‘constructs’;

// Create a cloud assembly source from an inline CDK app const cloudAssemblySource = await toolkit.fromAssemblyBuilder(async () => { const app = new App();

…. // Define a simple stack with an S3 bucket class MyStack extends Stack { constructor(scope: Construct, id: string, props?: StackProps) { super(scope, id, props);

// Create an S3 bucket  
 new Bucket(this, 'MyBucket', {  
 versioned: true,  
 removalPolicy: RemovalPolicy.DESTROY,  
 autoDeleteObjects: true  
 });  
}

}

// Instantiate the stack new MyStack(app, ‘MyInlineStack’);

return app.synth(); }); ….

// Deploy using the cloud assembly source await toolkit.deploy(cloudAssemblySource, { stacks: [‘MyInlineStack’] }); —

[#toolkit-library-configure-ca-how-directory] === From an existing assembly directory

If you already have a synthesized cloud assembly, you can use it directly. This is useful when you’ve already run cdk synth or when working with cloud assemblies generated by CI/CD pipelines.

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Create a toolkit instance const toolkit = new Toolkit();

// Use an existing cloud assembly directory const cloudAssemblySource = await toolkit.fromAssemblyDirectory(“cdk.out”);

// Deploy using the cloud assembly source await toolkit.deploy(cloudAssemblySource, { stacks: [‘MyStack’] }); —

[#toolkit-library-configure-ca-cache] == Working with cached cloud assemblies

When working with cloud assemblies, you have two options:

| . Use a cloud assembly source directly (simple but may be slower): + [source,typescript,subs=“verbatim,attributes”] |
| --- |
| // Each operation triggers a new synthesis await toolkit.deploy(cloudAssemblySource, { /\* options */ }); await toolkit.list(cloudAssemblySource, { /* options \_/ }); |

* . Cache the cloud assembly (faster for multiple operations):

## [source,typescript,subs=“verbatim,attributes”]

* // Synthesize once and reuse const cloudAssembly = await toolkit.synth(cloudAssemblySource); try { // Multiple operations use the same assembly await toolkit.deploy(cloudAssembly, { /\_ options */ }); await toolkit.list(cloudAssembly, { /* options \*/ }); } finally { // Clean up when done await cloudAssembly.dispose(); } — –

Use cached assemblies when:

* You’re performing multiple operations (deploy, list, diff, etc.).
* Your CDK app doesn’t change frequently during operations.
* You want faster performance.

Use cloud assembly sources directly when:

* You’re performing a single operation.
* Your CDK app changes frequently.
* You want simpler code and don’t need to prioritze Toolkit operation speed.

= [IMPORTANT]

# Most Toolkit interactions should use a cached assembly for better performance. The only time to avoid caching is when your source changes frequently and checking for changes would be expensive.

[#toolkit-library-configure-ca-cache-how] === How to create, cache, and reuse cloud assemblies

After you create a cloud assembly source, you can generate a cloud assembly by synthesizing it. A cloud assembly contains the {aws} CloudFormation templates and assets needed for deployment.

We recommend that you generate a cloud assembly once and reuse it for multiple Toolkit operations. This caching approach is more efficient than regenerating the assembly for each operation. Consider regenerating the assembly only when your source changes frequently.

Here’s how to create a cached cloud assembly:

== [source,typescript,subs=“verbatim,attributes”]

// Generate a cloud assembly from your source const cloudAssembly = await toolkit.synth(cloudAssemblySource); —

You can then perform various Toolkit actions on the cached cloud assembly, such as list(), deploy(), and diff(). By caching cloud assemblies, subsequent Toolkit actions are performed faster since synthesis occurs only once. For more information, see xref:toolkit-library-actions-synth[synth - Generate cloud assemblies].

[#toolkit-library-configure-ca-cache-dispose] === Dispose of cloud assembly resources

Always dispose of cloud assemblies when you’re done using them to clean up temporary resources. We recommend using a try/finally block to ensure proper cleanup, especially when performing multiple operations:

== [source,typescript,subs=“verbatim,attributes”]

// Generate a cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try { // Use the cloud assembly for multiple operations await toolkit.list(cloudAssembly); await toolkit.deploy(cloudAssembly); } finally { // Always dispose when done await cloudAssembly.dispose(); } —

Here’s an example showing how to create and dispose of a cached cloud assembly:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

const toolkit = new Toolkit();

// Create cloud assembly source from a CDK app const cloudAssemblySource = await toolkit.fromCdkApp(“ts-node app.ts”);

// Create cloud assembly from source const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try { // List stacks in the assembly await toolkit.list(cloudAssembly);

…. // Check for changes await toolkit.diff(cloudAssembly);

// Deploy if needed await toolkit.deploy(cloudAssembly); } finally { // Always dispose when done await cloudAssembly.dispose(); } —- ….

[#toolkit-library-configure-ca-cache-lifetime] === Understanding cloud assembly lifetimes

When you create a cached cloud assembly using synth(), you get a special type that serves as both a readable CloudAssembly and a CloudAssemblySource. Any cloud assemblies produced from this cached assembly (for example, from list or deploy operations) are tied to the parent’s lifetime:

* Only the parent’s dispose() call actually cleans up resources
* Cloud assemblies from list/deploy operations are managed by their parent
* Failing to dispose of a cached cloud assembly is considered a bug

[#toolkit-library-configure-ca-best-practices] == Best practices for cloud assembly sources

When working with cloud assembly sources, consider these best practices:

* *Choose the right source method*: Select the approach that best fits your workflow and requirements.
* *Cache cloud assemblies*: Generate a cloud assembly once using synth() and reuse it for multiple operations to avoid unnecessary synthesis, especially for large applications.
* *Error handling*: Implement basic error handling to catch and display errors to users. Keep error handling simple and focus on providing clear error messages.
* *Version compatibility*: Ensure that your CDK Toolkit Library version can support the cloud assemblies you’re working with. If the Construct Library used to create the cloud assembly is newer than what your Toolkit Library supports, you’ll receive an error.
* *Environment variables*: Be aware that certain environment variables can affect cloud assembly synthesis and deployment. Variables like CDK\_DEFAULT\_ACCOUNT, CDK\_DEFAULT\_REGION, CDK\_OUTDIR, and CDK\_CONTEXT\_JSON can override default behaviors. Ensure these are set appropriately for your deployment environment.

The following example demonstrates how to implement error handling and proper cleanup while reusing a cloud assembly for multiple operations:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Example with error handling and proper cleanup async function deployInfrastructure(): Promise++++++{ let cloudAssembly;++++++

try { // Generate a cloud assembly once cloudAssembly = await toolkit.synth(cloudAssemblySource);

// Reuse the same cloud assembly for multiple operations  
 await toolkit.list(cloudAssembly); // Uses existing assembly  
 await toolkit.deploy(cloudAssembly); // Uses existing assembly  
 await toolkit.diff(cloudAssembly); // Uses existing assembly

} catch (error) { console.error(“Failed to deploy:”, error); } finally { // Always dispose when done if (cloudAssembly) { await cloudAssembly.dispose(); } } }

// Call the async function deployInfrastructure().catch(error => { console.error(“Deployment failed:”, error); process.exit(1); }); —

[#toolkit-library-configure-ca-troubleshooting] == Resolving potential issues

Follow these steps to resolve potential issues with cloud assembly sources:

* *Install missing dependencies*: Run npm install to install required dependencies for your CDK app.
* *Fix path issues*: Check that paths to CDK apps and assembly directories exist and are accessible.
* *Resolve version mismatches*: Update your CDK Toolkit Library version to match your CDK app version.
* *Fix synthesis errors*: Review your CDK app code for syntax errors or invalid configurations.

When errors occur during Toolkit operations, keep error handling simple and focus on providing clear error messages to users. Always dispose of cloud assemblies when you’re done using them. Here’s an example showing basic error handling with proper cleanup:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Example with simple error handling try { // Create the cloud assembly source const cloudAssemblySource = await toolkit.fromCdkApp(“ts-node app.ts”);

…. // Synthesize the cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

// Use the cloud assembly await toolkit.list(cloudAssembly); } catch (error) { // Display the error message console.error(“Operation failed:”, error.message); } finally { // Clean up resources await cloudAssembly.dispose(); } —- ….

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-configure-messages] = Configuring CDK Toolkit messages and interactions :info\_titleabbrev: Configuring messages and interactions :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, io host, IIoHost, messages, interactions

== [abstract]

## The {aws} CDK Toolkit Library provides the IIoHost interface to customize how messages and interactions are handled during CDK operations, enabling you to control the display of deployment progress, error messages, and user prompts to better integrate with your application’s user experience.

// Content start

The {aws} CDK Toolkit Library provides the +link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/Package/toolkit-lib/Interface/IIoHost/[IIoHost]+ interface to customize how messages and interactions are handled during CDK operations, enabling you to control the display of deployment progress, error messages, and user prompts to better integrate with your application’s user experience.

Before performing operations like deployment or synthesis, you need to understand how the CDK Toolkit communicates with users. The IIoHost interface serves as the communication channel between the CDK Toolkit and your application, handling both outgoing messages and incoming user responses.

When the CDK Toolkit runs operations, it communicates through two primary mechanisms:

* *Messages*: Informational outputs that notify you about operation progress (like “Starting deployment” or “Resource created”).
* *Requests*: Decision points that require input or confirmation (like “Do you want to deploy these changes?”), giving you an opportunity to provide information that wasn’t known to be needed beforehand.

[#toolkit-library-configure-messages-iiohost] == Using the IIoHost interface

The IIoHost interface consists of two primary methods:

. notify: Handles one-way informational messages. . requestResponse: Handles interactive requests that require a response.

== [source,typescript,subs=“verbatim,attributes”]

import { IoMessage, IoRequest } from ‘@aws-cdk/toolkit-lib’;

interface IIoHost { // Handle informational messages notify(message: IoMessage): Promise++++++;++++++

// Handle requests that need responses requestResponse(request: IoRequest): Promise++++++; } —-++++++

[#toolkit-library-configure-messages-levels] == Message levels and request types

The CDK Toolkit generates several types of messages and requests:

=== Message levels

* *Debug*: Detailed messages for troubleshooting.
* *Error*: Error messages that may affect operation.
* *Info*: General informational messages.
* *Result* - Primary message of an operation.
* *Trace*: Very detailed execution flow information.
* WARNING: Warning messages that don’t prevent operation.

For a complete list, see link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/message-registry/[IoMessages Registry] in the *{aws} CDK Toolkit Library API Reference*.

=== Request types

The CDK Toolkit sends requests when it needs input or confirmation from the user. These are special messages that allow for a response. If no response is provided, the Toolkit will use a default response when available.

[#toolkit-library-configure-messages-basic] == Basic IIoHost implementation

Here’s a simple example of implementing a custom io host:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Create a toolkit with custom message handling const toolkit = new Toolkit({ ioHost: { // Implementing the IIoHost interface // Handle informational messages notify: async function (msg) { // Example: Handle different message levels appropriately switch (msg.level) { case ‘error’: console.error([${msg.time}] ERROR: ${msg.message}); break; case ‘warning’: console.warn([${msg.time}] WARNING: ${msg.message}); break; case ‘info’: console.info([${msg.time}] INFO: ${msg.message}); break; case ‘debug’: console.debug([${msg.time}] DEBUG: ${msg.message}); break; case ‘trace’: console.debug([${msg.time}] TRACE: ${msg.message}); break; default: console.log([${msg.time}] ${msg.level}: ${msg.message}); } },

// Handle requests that need responses requestResponse: async function (msg) { // Example: Log the request and use default response console.log(Request: ${msg.message}, using default: ${msg.defaultResponse}); return msg.defaultResponse;

// Or implement custom logic to provide responses // if (msg.type === ‘deploy’) { // return promptUserForDeployment(msg); // } } } as IIoHost // Explicitly cast to IIoHost interface }); —-

The CDK Toolkit awaits the completion of each call, allowing you to perform asynchronous operations like HTTP requests or user prompts when handling messages.

[#toolkit-library-configure-messages-iiohost-default] == Default IIoHost behavior

If you don’t provide a custom io host, the CDK Toolkit Library uses a default non-interactive implementation:

* Console output for messages (using appropriate colors for different message types).
* Fully non-interactive with no prompts for user input.
* Automatically uses default responses where possible (equivalent to answering “yes” to prompts).
* Fails when input is required but no default response is available.

This default behavior is suitable for unattended operations, but not for interactive command-line applications. For command-line applications that require user interaction, you’ll need to implement a custom io host. Custom implementations are also useful for integrating with logging systems, UIs, or other specialized environments.

[#toolkit-library-configure-messages-advanced] == Advanced io host implementation

For more complex scenarios, we recommend extending the NonInteractiveIoHost class as a starting point. This approach allows you to leverage the existing non-interactive implementation while customizing only the specific behaviors you need to change.

Here’s an example of a custom io host that extends the base implementation:

== [source,typescript,subs=“verbatim,attributes”]

import { NonInteractiveIoHost } from ‘@aws-cdk/toolkit-lib’;

class MyCustomIoHost extends NonInteractiveIoHost { // Override only the methods you need to customize

// Example: Custom implementation for notify public async notify(msg: IoMessage++++++): Promise++++++{ // Add custom notification handling logic if (msg.level === ‘error’) { console.error(ERROR: ${msg.message}); // Optionally log to a service or notify a monitoring system await this.logToMonitoringService(msg); } else { await super.notify(msg); } }++++++++++++

// Example: Custom implementation for requestResponse public async requestResponse<T, U>(request: IoRequest<T, U>): Promise++++++{ // Implement custom request handling console.log(Received request: ${request.message}); return request.defaultResponse; }++++++

private async logToMonitoringService(msg: IoMessage++++++): Promise++++++{ // Implementation for monitoring service integration console.log(Logging to monitoring service: ${msg.level} - ${msg.message}); } } —-++++++++++++

This approach is more maintainable than implementing the entire IIoHost interface from scratch, as you only need to override the specific methods that require custom behavior.

[#toolkit-library-configure-messages-integration] == Integrating with different environments

=== Web application integration

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Example for integrating with a web application const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { // Send message to frontend via WebSocket webSocketServer.send(JSON.stringify({ type: ‘cdk-notification’, messageLevel: msg.level, message: msg.message, time: msg.time })); },

requestResponse: async function (msg) { // Create a promise that will be resolved when the user responds return new Promise((resolve) => { const requestId = generateUniqueId();

// Store the resolver function  
 pendingRequests[requestId] = resolve;  
  
 // Send request to frontend  
 webSocketServer.send(JSON.stringify({  
 type: 'cdk-request',  
 requestId: requestId,  
 requestType: msg.type,  
 message: msg.message,  
 defaultResponse: msg.defaultResponse  
 }));  
  
 // Frontend would call an API endpoint with the response,  
 // which would then call pendingRequests[requestId](response)

}); } } as IIoHost // Explicitly cast to IIoHost interface }); —-

=== CI/CD environment integration

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Example for CI/CD environments (non-interactive) const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { // Log all messages with appropriate level switch (msg.level) { case ‘error’: console.error(msg.message); break; case ‘warning’: console.warn(msg.message); break; default: console.log(msg.message); } },

requestResponse: async function (msg) { // In CI/CD, always use default responses or predefined answers console.log(Auto-responding to request: ${msg.message} with ${msg.defaultResponse}); return msg.defaultResponse; } } as IIoHost // Explicitly cast to IIoHost interface }); —-

[#toolkit-library-configure-messages-best-practices] == Best practices for io host implementation

When implementing a custom io host, consider these best practices:

* *Error handling*: Implement robust error handling in your io host methods to prevent failures from affecting CDK operations.
* *Timeouts*: Consider implementing timeouts for user interactions to prevent indefinite waiting.
* *Logging*: Store important messages in logs for troubleshooting, especially in non-interactive environments.
* *Default responses*: Provide sensible default responses for automated environments where user interaction isn’t possible.
* *Progress indication*: For long-running operations, provide clear progress indicators to improve user experience.

The following example demonstrates these best practices by implementing a custom io host with error handling, timeouts for user interactions, and proper logging. This implementation is suitable for interactive applications where you need to balance user responsiveness with reliable operation:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Example with error handling and timeouts const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { try { console.log([${msg.time}] ${msg.level}: ${msg.message}); // Additional logging or UI updates } catch (error) { // Ensure errors in notification handling don’t break the CDK operation console.error(“Error handling notification:”, error); } },

requestResponse: async function (msg) { try { // Implement timeout for user response const response = await Promise.race([ getUserResponse(msg), new Promise(resolve => setTimeout(() => resolve(msg.defaultResponse), 60000)) ]); return response; } catch (error) { console.error(“Error handling request:”, error); return msg.defaultResponse; } } } as IIoHost // Explicitly cast to IIoHost interface }); —-

[#toolkit-library-configure-messages-troubleshooting] == Troubleshooting

Considerations when implementing a custom io host:

* *Unhandled promise rejections*: Ensure all async operations are properly handled with try/catch blocks.
* *Infinite waiting*: Implement timeouts for all user interactions to prevent the application from hanging.
* *Missing message levels*: Be prepared to handle new message levels that might be added in future CDK versions.
* *Inconsistent responses*: Ensure your requestResponse implementation returns values in the expected format.

The following example demonstrates a robust approach to handling message levels, including graceful handling of unknown message types that might be introduced in future CDK versions. This implementation ensures your io host remains compatible with CDK updates while maintaining proper logging:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Example of handling unknown message levels const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { // Handle known levels const knownLevels = [‘info’, ‘warning’, ‘error’, ‘debug’, ‘trace’, ‘status’];

…. if (knownLevels.includes(msg.level)) { // Handle known level handleKnownMessageLevel(msg); } else { // Handle unknown level as info console.log(Unknown message level "${msg.level}": ${msg.message}); } },

requestResponse: async function (msg) { // Default implementation return msg.defaultResponse; } } as IIoHost // Explicitly cast to IIoHost interface }); —- ….

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-configure] = Configuring your CDK Toolkit instance :info\_titleabbrev: Configure the CDK Toolkit :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, synthesize

== [abstract]

## Learn how to customize your {aws} CDK Toolkit Library instance with options for message handling, {aws} profile selection, and stack selection strategies. This guide explains the available configuration options and how to implement them effectively to meet your specific deployment requirements.

// Content start

Learn how to customize your {aws} CDK Toolkit Library instance with options for message handling, {aws} profile selection, and stack selection strategies. This guide explains the available configuration options and how to implement them effectively to meet your specific deployment requirements.

[#toolkit-library-configure-profile] == Configuring your {aws} profile

When you use the CDK Toolkit Library, it makes API calls to {aws} using the SDK. While authentication is loaded automatically from your environment, you can explicitly specify which profile to use:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

// Create a toolkit instance with a specific AWS profile const toolkit = new Toolkit({ sdkConfig: { profile: “my-profile” }, }); —

[#toolkit-library-configure-stacks] == Configuring stack selection

Most CDK Toolkit actions require you to specify which stacks to operate on. The +link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/Package/toolkit-lib/Interface/StackSelector/[StackSelector]+ configuration controls this selection.

[#toolkit-library-configure-stacks-all] === Select all stacks

Use this when you want to operate on every stack in your CDK app:

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Select all stacks in the cloud assembly await toolkit.deploy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.ALL\_STACKS } }); —

[#toolkit-library-configure-stacks-main] === Select only main assembly stacks

Use this to select only the top-level stacks from the main assembly:

== [source,typescript,subs=“verbatim,attributes”]

// Select only top-level stacks await toolkit.deploy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.MAIN\_ASSEMBLY } }); —

[#toolkit-library-configure-stacks-single] === Select a single stack

Use this when your assembly contains exactly one stack and you want to assert this condition. If the assembly includes a single stack, it returns that stack. Otherwise, it throws an exception:

== [source,typescript,subs=“verbatim,attributes”]

// Ensure there’s exactly one stack and select it await toolkit.deploy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.ONLY\_SINGLE } }); —

[#toolkit-library-configure-stacks-pattern] === Select stacks by pattern

Use this to select specific stacks by name pattern:

== [source,typescript,subs=“verbatim,attributes”]

// Select stacks matching specific patterns await toolkit.deploy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“Dev-\*“,”Test-Backend”], // Supports wildcards } }); —

= [TIP]

# Use PATTERN\_MUST\_MATCH\_SINGLE to ensure exactly one stack matches your patterns, or PATTERN\_MATCH if it’s acceptable for no stacks to match. Pattern matching supports wildcards like “\*” to match multiple stacks with similar names.

[#toolkit-library-configure-errors] == Configuring error handling

The CDK Toolkit uses structured errors to help you identify and handle issues. Each error includes:

* A *source* indicating where the error originated (toolkit or user).
* A specific *error type* (authentication, validation, etc.).
* A descriptive *message*.

[#toolkit-library-configure-errors-how] === Handling errors

Use the helper methods provided by the CDK Toolkit to detect and handle specific error types:

== [source,typescript,subs=“verbatim,attributes”]

import { ToolkitError } from ‘@aws-cdk/toolkit-lib’;

try { // Attempt a CDK Toolkit operation await toolkit.deploy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.ALL\_STACKS } });

} catch (error) { // Handle specific error types if (ToolkitError.isAuthenticationError(error)) { // Example: AWS credentials are missing or invalid console.error(‘Authentication failed. Check your AWS credentials.’);

} else if (ToolkitError.isAssemblyError(error)) { // Example: Your CDK app has errors in stack definitions console.error(‘CDK app error:’, error.message);

} else if (ToolkitError.isDeploymentError(error)) { // Example: CloudFormation deployment failed console.error(‘Deployment failed:’, error.message);

} else if (ToolkitError.isToolkitError(error)) { // Handle all other Toolkit errors console.error(‘CDK Toolkit error:’, error.message);

} else { // Handle unexpected errors console.error(‘Unexpected error:’, error); } } —

= [IMPORTANT]

# Don’t rely on instanceof checks for error types, as they can behave unexpectedly when working with multiple copies of the same package. Always use the provided helper methods like ToolkitError.isAuthenticationError().

[#toolkit-library-configure-actions] == Configuring Toolkit actions

Each CDK Toolkit action (deploy, synth, list, etc.) has its own specific configuration options. These actions allow you to manage the complete lifecycle of your CDK infrastructure. For detailed information on configuring individual actions, see xref:toolkit-library-actions[Configure CDK Toolkit programmatic actions].

= [TIP]

# When building automation workflows, consider combining multiple actions in sequence. For example, you might want to synth your app, list the stacks to verify what will be deployed, and then deploy the infrastructure.

include::attributes.txt[]

// Page attributes

[.topic] [#toolkit-library-examples] = Advanced CDK Toolkit Library examples :info\_titleabbrev: Advanced examples :keywords: CDK Toolkit Library, Examples, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code

== [abstract]

## Learn how to use advanced features of the {aws} CDK Toolkit Library through practical examples. This guide provides detailed code samples for error handling, deployment monitoring, and cloud assembly management that build upon the basic concepts covered in other sections.

// Content start

Learn how to use advanced features of the {aws} CDK Toolkit Library through practical examples. This guide provides detailed code samples for error handling, deployment monitoring, and cloud assembly management that build upon the basic concepts covered in other sections.

[#toolkit-library-examples-integration] == Integrating features

The following example demonstrates how to combine cloud assembly sources, custom io host implementation, and deployment options:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit, StackSelectionStrategy, IIoHost } from ‘@aws-cdk/toolkit-lib’;

async function deployApplication(appPath, environment, options = {}) { // Create toolkit with custom message handling const toolkit = new Toolkit({ ioHost: { notify: async (msg) => { // Add environment to all messages console.log(+[${environment}][${msg.time}] ${msg.level}: ${msg.message}+); }, requestResponse: async (msg) => { // In production environments, use default responses if (environment === ‘production’) { console.log(Auto-approving for production: ${msg.message}); return msg.defaultResponse; }

// For other environments, implement custom approval logic  
 return promptForApproval(msg);

} } as IIoHost });

try { // Create cloud assembly source from the CDK app console.log(+Creating cloud assembly source from ${appPath}+); const cloudAssemblySource = await toolkit.fromCdkApp(appPath);

…. // Synthesize the cloud assembly console.log(Synthesizing cloud assembly); const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try { // Deploy with environment-specific options console.log(Deploying to ${environment} environment); return await toolkit.deploy(cloudAssembly, { stacks: options.stacks || { strategy: StackSelectionStrategy.ALL\_STACKS }, parameters: options.parameters || {}, tags: { Environment: environment, DeployedBy: ‘CDK-Toolkit-Library’, DeployTime: new Date().toISOString() } }); } finally { // Always dispose when done await cloudAssembly.dispose(); } } catch (error) { console.error(Deployment to ${environment} failed:, error); throw error; } } ….

// Example usage await deployApplication(‘ts-node app.ts’, ‘staging’, { parameters: { MyStack: { InstanceType: ‘t3.small’ } } }); —

[#toolkit-library-examples-progress] == Tracking deployment progress

Track deployment progress with detailed status updates:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit, StackSelectionStrategy, IIoHost } from ‘@aws-cdk/toolkit-lib’;

// Create a progress tracker class DeploymentTracker { private startTime: Date; private resources = new Map<string, string>();

constructor() { this.startTime = new Date(); }

onStackEvent(stackName: string, event: string, timestamp: string) { // Calculate elapsed time if needed, or use the provided timestamp const elapsed = (new Date().getTime() - this.startTime.getTime()) / 1000; console.log(+[${timestamp}] (${elapsed.toFixed(1)}s elapsed) Stack ${stackName}: ${event}+); }

onResourceEvent(resourceId: string, status: string) { this.resources.set(resourceId, status); this.printProgress(); }

private printProgress() { console.log(‘Status:’); for (const [id, status] of this.resources.entries()) { console.log(+- ${id}: ${status}+); } console.log(); } }

// Use the tracker with the toolkit const tracker = new DeploymentTracker(); const toolkit = new Toolkit({ ioHost: { notify: async (msg) => { if (msg.code.startsWith(‘CDK\_DEPLOY’)) { // Track deployment events if (msg.data && ‘stackName’ in msg.data) { tracker.onStackEvent(msg.data.stackName, msg.message, msg.time); } } else if (msg.code.startsWith(‘CDK\_RESOURCE’)) { // Track resource events if (msg.data && ‘resourceId’ in msg.data) { tracker.onResourceEvent(msg.data.resourceId, msg.message); } } } } as IIoHost });

// Example usage with progress tracking async function deployWithTracking(cloudAssemblySource: any) { try { // Synthesize the cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try { // Deploy using the cloud assembly await toolkit.deploy(cloudAssembly, { stacks: { strategy: StackSelectionStrategy.ALL\_STACKS } }); } finally { // Always dispose when done await cloudAssembly.dispose(); } } catch (error) { // Display the error message console.error(“Operation failed:”, error.message); throw error; } } —-

[#toolkit-library-examples-error] == Handling errors with recovery

Implement robust error handling with recovery strategies:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit, ToolkitError, StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

async function deployWithRetry(toolkit: Toolkit, cloudAssemblySource: any) { try { // Synthesize the cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try { // Deploy using the cloud assembly await toolkit.deploy(cloudAssembly, { stacks: { strategy: StackSelectionStrategy.ALL\_STACKS } }); } finally { // Always dispose when done await cloudAssembly.dispose(); } } catch (error) { // Simply show the error to the user console.error(“Operation failed:”, error.message); throw error; } }

// Example usage try { await deployWithRetry(toolkit, cloudAssemblySource); } catch (error) { console.error(“Operation failed:”, error.message); process.exit(1); } —

[#toolkit-library-examples-cicd] == Integrating with CI/CD pipelines

Integrate the CDK Toolkit Library into a CI/CD pipeline:

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit, StackSelectionStrategy, IIoHost } from ‘@aws-cdk/toolkit-lib’; import \* as fs from ‘fs’; import \* as path from ‘path’;

async function cicdDeploy() { // Create a non-interactive toolkit for CI/CD environments const toolkit = new Toolkit({ ioHost: { notify: async (msg) => { // Write to both console and log file const logMessage = ${msg.time} [${msg.level}] ${msg.message}; console.log(logMessage);

// Append to deployment log file  
 fs.appendFileSync('deployment.log', logMessage + '\n');

}, requestResponse: async (msg) => { // Always use default responses in CI/CD console.log(Auto-responding to: ${msg.message} with: ${msg.defaultResponse}); return msg.defaultResponse; } } as IIoHost });

// Determine environment from CI/CD variables const environment = process.env.DEPLOYMENT\_ENV || ‘development’;

// Load environment-specific parameters const paramsPath = path.join(process.cwd(), +params.${environment}.json+); const parameters = fs.existsSync(paramsPath) ? JSON.parse(fs.readFileSync(paramsPath, ‘utf8’)) : {};

try { // Use pre-synthesized cloud assembly from build step const cloudAssemblySource = await toolkit.fromAssemblyDirectory(‘cdk.out’);

…. // Synthesize the cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try { // Deploy with CI/CD specific options const result = await toolkit.deploy(cloudAssembly, { stacks: { strategy: StackSelectionStrategy.ALL\_STACKS }, parameters, tags: { Environment: environment, BuildId: process.env.BUILD\_ID || ‘unknown’, CommitHash: process.env.COMMIT\_HASH || ‘unknown’ } });

// Write outputs to a file for subsequent pipeline steps fs.writeFileSync( ‘stack-outputs.json’, JSON.stringify(result.outputs, null, 2) );

return result; } finally { // Always dispose when done await cloudAssembly.dispose(); } } catch (error) { // Display the error message console.error(“Operation failed:”, error.message); process.exit(1); } } ….

// Run the CI/CD deployment cicdDeploy().then(() => { console.log(‘CI/CD deployment completed successfully’); }); —

[#toolkit-library-examples-resources] == Additional resources

For more detailed information on specific components used in these examples, refer to:

* xref:toolkit-library-configure-ca[Managing cloud assembly sources] - Learn how to create and manage cloud assembly sources.
* xref:toolkit-library-configure-messages[Configuring messages and interactions] - Detailed guide on customizing the IIoHost interface.

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-gs] = Getting started with the CDK Toolkit Library :info\_titleabbrev: Getting started :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, synthesize

== [abstract]

## Get started with using the {aws} CDK Toolkit Library to programmatically perform CDK actions, such as synthesis and deployment, in your code.

// Content start

Get started with using the {aws} CDK Toolkit Library to programmatically perform CDK actions, such as synthesis and deployment, in your code.

[#toolkit-library-gs-prerequisites] == Prerequisites

. Supported version of Node.js installed. . {aws} credentials configured. . Basic familiarity with the {aws} CDK.

For more information, see xref:prerequisites[{aws} CDK prerequisites].

[#toolkit-library-gs-install] == Step 1: Installing the CDK Toolkit Library

Install the CDK Toolkit Library package in your project’s development environment by running the following:

== [source,none,subs=“verbatim,attributes”]

## npm install –save @aws-cdk/toolkit-lib

[#toolkit-library-gs-initialize] == Step 2: Initializing the CDK Toolkit Library

Create a CDK Toolkit instance to perform programmatic actions on your CDK app.

== [source,typescript,subs=“verbatim,attributes”]

import { Toolkit } from ‘@aws-cdk/toolkit-lib’;

const toolkit = new Toolkit({ // Optional configuration options go here }); —

You can customize the CDK Toolkit instance during creation. For instructions, see xref:toolkit-library-configure[Configure your CDK Toolkit instance].

[#toolkit-library-gs-ca] == Step 3: Creating a cloud assembly source for your CDK app

A cloud assembly source provides instructions for generating CloudFormation templates from your CDK app. You can create one in multiple ways. The following are a few examples:

| . *An inline assembly builder function*: + [source,typescript,subs=“verbatim,attributes”] |
| --- |
| import \* as cdk from ‘aws-cdk-lib’; |
| const cloudAssemblySource = await toolkit.fromAssemblyBuilder(async () => { const app = new cdk.App(); new MyStack(app, ‘MyStack’); return app.synth(); }); |

. *An existing CDK app file*: + [source,typescript,subs=“verbatim,attributes”] — const cloudAssemblySource = await toolkit.fromCdkApp(“ts-node app.ts”); — –

For more information, see xref:toolkit-library-configure-ca[Configure cloud assembly sources].

[#toolkit-library-gs-define] == Step 4: Defining programmatic actions for your CDK app

Now that you’ve created a CDK Toolkit instance and cloud assembly source, you can start to define programmatic actions. The following is a basic example that creates a deployment of the MyStack stack:

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

await toolkit.deploy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, // Deploy only stacks that exactly match the provided patterns patterns: [“MyStack”], }, }); —

[#toolkit-library-gs-customize] == Step 5: Customizing the CDK Toolkit further

You can configure and customize the CDK Toolkit further for your needs:

* *Messages and interactions* - Configure how the CDK Toolkit communicates with users and applications. See xref:toolkit-library-configure-messages[Configure messages & interactions].
* *Error handling* - Implement structured error handling for CDK operations. See xref:toolkit-library-configure-errors[Configure error handling].

[#toolkit-library-gs-resources] == Additional resources

For more information on the CDK Toolkit Library npm package, see the link:https://www.npmjs.com/package/@aws-cdk/toolkit-lib [ReadMe] in the \_@aws-cdk/toolkit-lib\_ npm package.

For API reference information, see the link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/[CDK Toolkit Library API reference].

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library] = Perform programmatic actions using the CDK Toolkit Library :info\_titleabbrev: Use the CDK Toolkit Library :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, synthesize

== [abstract]

## The {aws} Cloud Development Kit ({aws} CDK) Toolkit Library enables you to perform CDK actions programmatically instead of using the CLI.

// Content start

[#toolkit-library-intro] == Understanding the CDK Toolkit Library

The CDK Toolkit Library enables you to perform CDK actions programmatically through code instead of using CLI commands. You can use this library to create custom tools, build specialized CLI applications, and integrate CDK capabilities into your development workflows.

*Manage your infrastructure lifecycle with programmatic control*::

The CDK Toolkit Library provides programmatic interfaces for the following CDK actions: + –

* *Synthesis* - Generate {aws} CloudFormation templates and deployment artifacts.
* *Deployment* - Provision or update infrastructure using CloudFormation templates.
* *List* - View information about stacks and their dependencies.
* *Watch* - Monitor CDK apps for local changes.
* *Rollback* - Return stacks to their last stable state.
* {blank}
* == *Destroy* - Remove CDK stacks and associated resources.

*Enhance and customize your infrastructure management*:: + –

* *Control through code* - Integrate infrastructure management directly into your applications and build responsive deployment pipelines.
* *Manage cloud assemblies* - Create, inspect, and transform your infrastructure definitions before deployment.
* *Customize deployments* - Configure parameters, rollback behavior, and monitoring to match your requirements.
* *Handle errors precisely* - Implement structured error handling with detailed diagnostic information.
* *Tailor communications* - Configure custom progress indicators and logging through IoHost implementations.
* {blank}
* == *Connect with {aws}* - Configure profiles, Regions, and authentication flows programmatically.

[#toolkit-library-intro-when] == Choosing when to use the CDK Toolkit Library

The CDK Toolkit Library is particularly valuable when you need to:

* Automate infrastructure deployments as part of CI/CD pipelines.
* Build custom deployment tools tailored to your organization’s needs.
* Integrate CDK actions into existing applications or platforms.
* Create specialized deployment workflows with custom validation or approval steps.
* Implement advanced infrastructure management patterns across multiple environments.

[#toolkit-library-intro-example] == Using the CDK Toolkit Library

The following example shows how to create and deploy a simple S3 bucket using the CDK Toolkit Library:

== [source, typescript, subs=“verbatim,attributes”]

// Import required packages import { Toolkit } from ‘@aws-cdk/toolkit-lib’; import { App, Stack } from ‘aws-cdk-lib’; import \* as s3 from ‘aws-cdk-lib/aws-s3’;

// Create and configure the CDK Toolkit const toolkit = new Toolkit();

// Create a cloud assembly source with an inline app const cloudAssemblySource = await toolkit.fromAssemblyBuilder(async () => { const app = new App(); const stack = new Stack(app, ‘SimpleStorageStack’);

// Create an S3 bucket in the stack new s3.Bucket(stack, ‘MyFirstBucket’, { versioned: true });

return app.synth(); });

// Deploy the stack await toolkit.deploy(cloudAssemblySource); —

*What you can do next*:: + –

* *Automate deployments* - Trigger deployments programmatically and add pre/post deployment steps.
* *Integrate with systems* - Connect with CI/CD workflows, custom tools, and monitoring solutions.
* *Control deployment details* - Configure fine-grained options for stack selection and multi-environment deployments.
* {blank}
* == *Enhance reliability* - Implement production-ready error handling and deployment progress tracking.

[#toolkit-library-intro-next] == Next steps

To begin using the CDK Toolkit Library, see xref:toolkit-library-gs[Get started with the CDK Toolkit Library].

[#toolkit-library-intro-learn] == Learn more

To learn more about the CDK Toolkit Library, see the following:

* link:https://www.npmjs.com/package/@aws-cdk/toolkit-lib [ReadMe] in the \_@aws-cdk/toolkit-lib\_ npm package.
* link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/[{aws} CDK Toolkit Library API Reference].

include::toolkit-library-gs.adoc[leveloffset=+1]

include::toolkit-library-actions.adoc[leveloffset=+1]

include::toolkit-library-configure.adoc[leveloffset=+1]

include::toolkit-library-configure-ca.adoc[leveloffset=+1]

include::toolkit-library-configure-messages.adoc[leveloffset=+1]

include::toolkit-library-examples.adoc[leveloffset=+1]

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#use-cfn-public-registry] = Use resources from the {aws} CloudFormation Public Registry :info\_titleabbrev: Use resources from the CloudFormation Public Registry

// Content start

The {aws} CloudFormation Public Registry lets you manage extensions, both public and private, such as resources, modules, and hooks that are available for use in your {aws} account. You can use public resource extensions in your {aws} Cloud Development Kit ({aws} CDK) applications with the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CfnResource.html[CfnResource] construct.

To learn more about the {aws} CloudFormation Public Registry, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/registry.html[Using the {aws} CloudFormation registry] in the *{aws} CloudFormation User Guide*.

All public extensions published by {aws} are available to all accounts in all Regions without any action on your part. However, you must activate each third-party extension you want to use, in each account and Region where you want to use it.

= [NOTE]

When you use {aws} CloudFormation with third-party resource types, you will incur charges. Charges are based on the number of handler operations you run per month and handler operation duration. See https://aws.amazon.com/cloudformation/pricing/[CloudFormation pricing] for complete details.

====

To learn more about public extensions, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/registry-public.html[Using public extensions in CloudFormation] in the *{aws} CloudFormation User Guide*

[#use-cfn-public-registry-activate] == Activate a third-party resource in your account and Region

Extensions published by {aws} do not require activation. They are always available in every account and Region. You can activate a third-party extension through the {aws} Management Console, via the {aws} Command Line Interface, or by deploying a special {aws} CloudFormation resource.

*To activate a third-party extension through the {aws} Management Console or see what resources are available*:: + image::./images/activate-cfn-extension.png[scaledwidth=100%] + . Sign in to the {aws} account in which you want to use the extension, then switch to the Region where you want to use it. . Navigate to the CloudFormation console via the *Services* menu. . Choose *Public extensions* on the navigation bar, then activate the *Third party* radio button under *Publisher*. A list of the available third-party public extensions appears. (You may also choose *{aws}* to see a list of the public extensions published by {aws}, though you don’t need to activate them.) . Browse the list and find the extension you want to activate. Alternatively, search for it, then activate the radio button in the upper right corner of the extension’s card. . Choose the *Activate* button at the top of the list to activate the selected extension. The extension’s *Activate* page appears. . In the *Activate* page, you can override the extension’s default name and specify an execution role and logging configuration. You can also choose whether to automatically update the extension when a new version is released. When you have set these options as you like, choose *Activate extension* at the bottom of the page.

*To activate a third-party extension using the {aws} CLI*:: +

* Use the activate-type command. Substitute the ARN of the custom type you want to use where indicated.
* The following is an example:

## [source,none,subs=“verbatim,attributes”]

* aws cloudformation activate-type –public-type-arn ++++++–auto-update-activated —-++++++

*To activate a third-party extension through CloudFormation or CDK*:: + . Deploy a resource of type +{aws}::CloudFormation::TypeActivation+ and specify the following properties: + – .. TypeName - The name of the type, such as AWSQS::EKS::Cluster. .. MajorVersion - The major version number of the extension that you want. Omit if you want the latest version. .. AutoUpdate - Whether to automatically update this extension when a new minor version is released by the publisher. (Major version updates require explicitly changing the MajorVersion property.) .. ExecutionRoleArn - The ARN of the IAM role under which this extension will run. .. LoggingConfig - The logging configuration for the extension. – + The TypeActivation resource can be deployed by the CDK using the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CfnResource.html[CfnResource] construct. This is shown for the actual extensions in the following section.

[#use-cfn-public-registry-add] == Add a resource from the {aws} CloudFormation Public Registry to your CDK app

Use the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CfnResource.html[CfnResource] construct to include a resource from the {aws} CloudFormation Public Registry in your application. This construct is in the CDK’s aws-cdk-lib module.

For example, suppose that there is a public resource named MY::S5::UltimateBucket that you want to use in your {aws} CDK application. This resource takes one property: the bucket name. The corresponding CfnResource instantiation looks like this.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const ubucket = new CfnResource(this, ‘MyUltimateBucket’, { type: ‘MY::S5::UltimateBucket::MODULE’, properties: { BucketName: ‘UltimateBucket’ } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const ubucket = new CfnResource(this, ‘MyUltimateBucket’, { type: ‘MY::S5::UltimateBucket::MODULE’, properties: { BucketName: ‘UltimateBucket’ } }); —

Python:: + [source,python,subs=“verbatim,attributes”] — ubucket = CfnResource(self, “MyUltimateBucket”, type=“MY::S5::UltimateBucket::MODULE”, properties=dict( BucketName=“UltimateBucket”)) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnResource.Builder.create(this, “MyUltimateBucket”) .type(“MY::S5::UltimateBucket::MODULE”) .properties(java.util.Map.of( // Map.of requires Java 9+ “BucketName”, “UltimateBucket”)) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new CfnResource(this, “MyUltimateBucket”, new CfnResourceProps { Type = “MY::S5::UltimateBucket::MODULE”, Properties = new Dictionary<string, object> { [“BucketName”] = “UltimateBucket” } }); — ====

:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#use-cfn-template] = Import an existing {aws} CloudFormation template :info\_titleabbrev: Import an {aws} CloudFormation template

// Content start

Import resources from an {aws} CloudFormation template into your {aws} Cloud Development Kit ({aws} CDK) applications by using the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include-readme.html[cloudformation-include.CfnInclude] construct to convert resources to L1 constructs.

After import, you can work with these resources in your app in the same way that you would if they were originally defined in {aws} CDK code. You can also use these L1 constructs within higher-level {aws} CDK constructs. For example, this can let you use the L2 permission grant methods with the resources they define.

The cloudformation-include.CfnInclude construct essentially adds an {aws} CDK API wrapper to any resource in your {aws} CloudFormation template. Use this capability to import your existing {aws} CloudFormation templates to the {aws} CDK a piece at a time. By doing this, you can manage your existing resources using {aws} CDK constructs to utilize the benefits of higher-level abstractions. You can also use this feature to vend your {aws} CloudFormation templates to {aws} CDK developers by providing an {aws} CDK construct API.

= [NOTE]

{aws} CDK v1 also included link:https://docs.aws.amazon.com/cdk/api/latest/docs/aws-cdk-lib.CfnInclude.html[aws-cdk-lib.CfnInclude], which was previously used for the same general purpose. However, it lacks much of the functionality of cloudformation-include.CfnInclude.

====

[#use-cfn-template-import]] == Import an {aws} CloudFormation template

The following is a sample {aws} CloudFormation template that we will use to provide examples in this topic. Copy and save the template as my-template.json to follow along. After working through these examples, you can explore further by using any of your existing deployed {aws} CloudFormation templates. You can obtain them from the {aws} CloudFormation console.

== [source,json,subs=“verbatim,attributes”]

{ “Resources”: { “amzn-s3-demo-bucket”: { “Type”: “{aws}::S3::Bucket”, “Properties”: { “BucketName”: “amzn-s3-demo-bucket”, } } } } —

You can work with either JSON or YAML templates. We recommend JSON if available since YAML parsers can vary slightly in what they accept.

The following is an example of how to import the sample template into your {aws} CDK app using cloudformation-include. Templates are imported within the context of an CDK stack.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import \* as cfninc from ‘aws-cdk-lib/cloudformation-include’; import { Construct } from ‘constructs’;

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

const template = new cfninc.CfnInclude(this, ‘Template’, { templateFile: ‘my-template.json’, }); } } —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const cfninc = require(‘aws-cdk-lib/cloudformation-include’);

class MyStack extends cdk.Stack { constructor(scope, id, props) { super(scope, id, props);

const template = new cfninc.CfnInclude(this, ‘Template’, { templateFile: ‘my-template.json’, }); } }

== module.exports = { MyStack }

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk from aws\_cdk import cloudformation\_include as cfn\_inc from constructs import Construct

class MyStack(cdk.Stack):

…. def **init**(self, scope: Construct, id: str, \*\*kwargs) -> None: super().\_\_init\_\_(scope, id, \*\*kwargs)

template = cfn\_inc.CfnInclude(self, "Template",  
 template\_file="my-template.json") ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.cloudformation.include.CfnInclude; import software.constructs.Construct;

public class MyStack extends Stack { public MyStack(final Construct scope, final String id) { this(scope, id, null); }

…. public MyStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

CfnInclude template = CfnInclude.Builder.create(this, "Template")  
 .templateFile("my-template.json")  
 .build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using cfnInc = Amazon.CDK.CloudFormation.Include;

namespace MyApp { public class MyStack : Stack { internal MyStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { var template = new cfnInc.CfnInclude(this, “Template”, new cfnInc.CfnIncludeProps { TemplateFile = “my-template.json” }); } } } — ====

By default, importing a resource preserves the resource’s original logical ID from the template. This behavior is suitable for importing an {aws} CloudFormation template into the {aws} CDK, where logical IDs must be retained. {aws} CloudFormation needs this information to recognize these imported resources as the same resources from the {aws} CloudFormation template.

If you are developing an {aws} CDK construct wrapper for the template so that it can be used by other {aws} CDK developers, have the {aws} CDK generate new resource IDs instead. By doing this, the construct can be used multiple times in a stack without name conflicts. To do this, set the preserveLogicalIds property to false when importing the template. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const template = new cfninc.CfnInclude(this, ‘MyConstruct’, { templateFile: ‘my-template.json’, preserveLogicalIds: false }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const template = new cfninc.CfnInclude(this, ‘MyConstruct’, { templateFile: ‘my-template.json’, preserveLogicalIds: false }); —

Python:: + [source,python,subs=“verbatim,attributes”] — template = cfn\_inc.CfnInclude(self, “Template”, + template\_file=“my-template.json”, preserve\_logical\_ids=False) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnInclude template = CfnInclude.Builder.create(this, “Template”) .templateFile(“my-template.json”) .preserveLogicalIds(false) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var template = new cfnInc.CfnInclude(this, “Template”, new cfn\_inc.CfnIncludeProps { TemplateFile = “my-template.json”, PreserveLogicalIds = false }); — ====

To put imported resources under the control of your {aws} CDK app, add the stack to the App:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { MyStack } from ‘../lib/my-stack’;

const app = new cdk.App(); new MyStack(app, ‘MyStack’); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const { MyStack } = require(‘../lib/my-stack’);

const app = new cdk.App(); new MyStack(app, ‘MyStack’); —

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk from mystack.my\_stack import MyStack

app = cdk.App() MyStack(app, “MyStack”) —

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.App;

public class MyApp { public static void main(final String[] args) { App app = new App();

new MyStack(app, "MyStack");

} } —-

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK;

namespace CdkApp { sealed class Program { public static void Main(string[] args) { var app = new App(); new MyStack(app, “MyStack”); } } } — ====

To verify that there won’t be any unintended changes to the {aws} resources in the stack, you can perform a diff. Use the {aws} CDK CLI cdk diff command and omit any {aws} CDK-specific metadata. The following is an example:

== [source,none,subs=“verbatim,attributes”]

## cdk diff –no-version-reporting –no-path-metadata –no-asset-metadata

After you import an {aws} CloudFormation template, the {aws} CDK app should become the source of truth for your imported resources. To make changes to your resources, modify them in your {aws} CDK app and deploy with the {aws} CDK CLI cdk deploy command.

[#use-cfn-template-cfninclude-access] == Access imported resources

The name template in the example code represents the imported {aws} CloudFormation template. To access a resource from it, use the object’s link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrresourcelogicalid[getResource()] method. To access the returned resource as a specific kind of resource, cast the result to the desired type. This isn’t necessary in Python or JavaScript. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const cfnBucket = template.getResource(‘amzn-s3-demo-bucket’) as s3.CfnBucket; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const cfnBucket = template.getResource(‘amzn-s3-demo-bucket’); —

Python:: + [source,python,subs=“verbatim,attributes”] — cfn\_bucket = template.get\_resource(“amzn-s3-demo-bucket”) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnBucket cfnBucket = (CfnBucket)template.getResource(“amzn-s3-demo-bucket”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var cfnBucket = (CfnBucket)template.GetResource(“amzn-s3-demo-bucket”); — ====

From this example, cfnBucket is now an instance of the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html[aws-s3.CfnBucket] class. This is an L1 construct that represents the corresponding {aws} CloudFormation resource. You can treat it like any other resource of its type. For example, you can get its ARN value with the bucket.attrArn property.

To wrap the L1 CfnBucket resource in an L2 link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html[aws-s3.CfnBucket] instance instead, use the static methods link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html#static-fromwbrbucketwbrarnscope-id-bucketarn[fromBucketArn()], link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html#static-fromwbrbucketwbrattributesscope-id-attrs[fromBucketAttributes()], or link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html#static-fromwbrbucketwbrnamescope-id-bucketname[fromBucketName()]. Usually, the fromBucketName() method is most convenient. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = s3.Bucket.fromBucketName(this, ‘Bucket’, cfnBucket.ref); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = s3.Bucket.fromBucketName(this, ‘Bucket’, cfnBucket.ref); —

Python:: + [source,python,subs=“verbatim,attributes”] — bucket = s3.Bucket.from\_bucket\_name(self, “Bucket”, cfn\_bucket.ref) —

Java:: + [source,java,subs=“verbatim,attributes”] — Bucket bucket = (Bucket)Bucket.fromBucketName(this, “Bucket”, cfnBucket.getRef()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var bucket = (Bucket)Bucket.FromBucketName(this, “Bucket”, cfnBucket.Ref); — ====

Other L2 constructs have similar methods for creating the construct from an existing resource.

When you wrap an L1 construct in an L2 construct, it doesn’t create a new resource. From our example, we are not creating a second S3; bucket. Instead, the new Bucket instance encapsulates the existing CfnBucket.

From the example, the bucket is now an L2 Bucket construct that behaves like any other L2 construct. For example, you can grant an {aws} Lambda function write access to the bucket by using the bucket’s convenient link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html#grantwbrwriteidentity-objectskeypattern[grantWrite()] method. You don’t have to define the necessary {aws} Identity and Access Management (IAM) policy manually. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — bucket.grantWrite(lambdaFunc); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — bucket.grantWrite(lambdaFunc); —

Python:: + [source,python,subs=“verbatim,attributes”] — bucket.grant\_write(lambda\_func) —

Java:: + [source,java,subs=“verbatim,attributes”] — bucket.grantWrite(lambdaFunc); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — bucket.GrantWrite(lambdaFunc); — ====

[#use-cfn-template-cfninclude-params] == Replace parameters

If your {aws} CloudFormation template contains parameters, you can replace them with build time values at import by using the parameters property. In the following example, we replace the UploadBucket parameter with the ARN of a bucket defined elsewhere in our {aws} CDK code.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const template = new cfninc.CfnInclude(this, ‘Template’, { templateFile: ‘my-template.json’, parameters: { ‘UploadBucket’: bucket.bucketArn, }, }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const template = new cfninc.CfnInclude(this, ‘Template’, { templateFile: ‘my-template.json’, parameters: { ‘UploadBucket’: bucket.bucketArn, }, }); —

Python:: + [source,python,subs=“verbatim,attributes”] — template = cfn\_inc.CfnInclude(self, “Template”, + template\_file=“my-template.json”, parameters=dict(UploadBucket=bucket.bucket\_arn) ) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnInclude template = CfnInclude.Builder.create(this, “Template”) .templateFile(“my-template.json”) .parameters(java.util.Map.of( // Map.of requires Java 9+ “UploadBucket”, bucket.getBucketArn())) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var template = new cfnInc.CfnInclude(this, “Template”, new cfnInc.CfnIncludeProps { TemplateFile = “my-template.json”, Parameters = new Dictionary<string, string> { { “UploadBucket”, bucket.BucketArn } } }); — ====

[#use-cfn-template-cfninclude-other] == Import other template elements

You can import any {aws} CloudFormation template element, not just resources. The imported elements become a part of the {aws} CDK stack. To import these elements, use the following methods of the CfnInclude object:

* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrconditionconditionname[getCondition()] – {aws} CloudFormation https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/conditions-section-structure.html[conditions].
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrhookhooklogicalid[getHook()] – {aws} CloudFormation https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/blue-green.html[hooks] for blue/green deployments.
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrmappingmappingname[getMapping()] – {aws} CloudFormation https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/mappings-section-structure.html[mappings].
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbroutputlogicalid[getOutput()] – {aws} CloudFormation https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/outputs-section-structure.html[outputs].
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrparameterparametername[getParameter()] – {aws} CloudFormation https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html[parameters].
* link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrrulerulename[getRule()] – {aws} CloudFormation link:https://docs.aws.amazon.com/servicecatalog/latest/adminguide/reference-template\_constraint\_rules.html[rules] for {aws} Service Catalog templates.

Each of these methods return an instance of a class that represents the specific type of {aws} CloudFormation element. These objects are mutable. Changes that you make to them will appear in the template that gets generated from the {aws} CDK stack. The following is an example that imports a parameter from the template and modifies its default value:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const param = template.getParameter(‘MyParameter’); param.default = “{aws} CDK” —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const param = template.getParameter(‘MyParameter’); param.default = “{aws} CDK” —

Python:: + [source,python,subs=“verbatim,attributes”] — param = template.get\_parameter(“MyParameter”) param.default = “{aws} CDK” —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnParameter param = template.getParameter(“MyParameter”); param.setDefaultValue(“{aws} CDK”) —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var cfnBucket = (CfnBucket)template.GetResource(“amzn-s3-demo-bucket”); var param = template.GetParameter(“MyParameter”); param.Default = “{aws} CDK”; — ====

[#use-cfn-template-cfninclude-nested] == Import nested stacks

You can import https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-nested-stacks.html[nested stacks] by specifying them either when you import their main template, or at some later point. The nested template must be stored in a local file, but referenced as a NestedStack resource in the main template. Also, the resource name used in the {aws} CDK code must match the name used for the nested stack in the main template.

Given this resource definition in the main template, the following code shows how to import the referenced nested stack both ways.

== [source,json,subs=“verbatim,attributes”]

“NestedStack”: { “Type”: “{aws}::CloudFormation::Stack”, “Properties”: { “TemplateURL”: “https://my-s3-template-source.s3.amazonaws.com/nested-stack.json” } } —

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // include nested stack when importing main stack const mainTemplate = new cfninc.CfnInclude(this, ‘MainStack’, { templateFile: ‘main-template.json’, loadNestedStacks: { ‘NestedStack’: { templateFile: ‘nested-template.json’, }, }, });

// or add it some time after importing the main stack const nestedTemplate = mainTemplate.loadNestedStack(‘NestedTemplate’, { templateFile: ‘nested-template.json’, }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // include nested stack when importing main stack const mainTemplate = new cfninc.CfnInclude(this, ‘MainStack’, { templateFile: ‘main-template.json’, loadNestedStacks: { ‘NestedStack’: { templateFile: ‘nested-template.json’, }, }, });

// or add it some time after importing the main stack const nestedTemplate = mainTemplate.loadNestedStack(‘NestedStack’, { templateFile: ‘my-nested-template.json’, }); —

Python:: + [source,python,subs=“verbatim,attributes”] —

= include nested stack when importing main stack

main\_template = cfn\_inc.CfnInclude(self, “MainStack”, + template\_file=“main-template.json”, load\_nested\_stacks=dict(NestedStack= cfn\_inc.CfnIncludeProps(template\_file=“nested-template.json”)))

= or add it some time after importing the main stack

nested\_template = main\_template.load\_nested\_stack(“NestedStack”, template\_file=“nested-template.json”) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnInclude mainTemplate = CfnInclude.Builder.create(this, “MainStack”) .templateFile(“main-template.json”) .loadNestedStacks(java.util.Map.of( // Map.of requires Java 9+ “NestedStack”, CfnIncludeProps.builder() .templateFile(“nested-template.json”).build())) .build();

// or add it some time after importing the main stack IncludedNestedStack nestedTemplate = mainTemplate.loadNestedStack(“NestedTemplate”, CfnIncludeProps.builder() .templateFile(“nested-template.json”) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // include nested stack when importing main stack var mainTemplate = new cfnInc.CfnInclude(this, “MainStack”, new cfnInc.CfnIncludeProps { TemplateFile = “main-template.json”, LoadNestedStacks = new Dictionary<string, cfnInc.ICfnIncludeProps> { { “NestedStack”, new cfnInc.CfnIncludeProps { TemplateFile = “nested-template.json” } } } });

// or add it some time after importing the main stack var nestedTemplate = mainTemplate.LoadNestedStack(“NestedTemplate”, new cfnInc.CfnIncludeProps { TemplateFile = ‘nested-template.json’ }); — ====

You can import multiple nested stacks with either methods. When importing the main template, you provide a mapping between the resource name of each nested stack and its template file. This mapping can contain any number of entries. To do it after the initial import, call loadNestedStack() once for each nested stack.

After importing a nested stack, you can access it using the main template’s link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrnestedwbrstacklogicalid[getNestedStack()] method.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const nestedStack = mainTemplate.getNestedStack(‘NestedStack’).stack; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const nestedStack = mainTemplate.getNestedStack(‘NestedStack’).stack; —

Python:: + [source,python,subs=“verbatim,attributes”] — nested\_stack = main\_template.get\_nested\_stack(“NestedStack”).stack —

Java:: + [source,java,subs=“verbatim,attributes”] — NestedStack nestedStack = mainTemplate.getNestedStack(“NestedStack”).getStack(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var nestedStack = mainTemplate.GetNestedStack(“NestedStack”).Stack; — ====

The getNestedStack() method returns an link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.cloudformation\_include.CfnInclude.html#getwbrnestedwbrstacklogicalid[IncludedNestedStack] instance. From this instance, you can access the {aws} CDK https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.NestedStack.html[NestedStack] instance via the stack property, as shown in the example. You can also access the original {aws} CloudFormation template object via includedTemplate, from which you can load resources and other {aws} CloudFormation elements.