: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(); } —- ….