

# AWS CDK

Version aws-cdk-lib 2.275 · verified 2026-11-02

Define AWS infrastructure in TypeScript that compiles to CloudFormation — constructs, synth, deploy, references, grants, testing and CI.

## Contents


Mental model ..... 2

App, stacks and constructs ..... 2

Synth, diff, deploy ..... 4

Environments and bootstrapping ..... 6

Cross-stack references ..... 8

Assets and Lambda bundling ..... 9

Grants and removal policies ..... 11

Aspects, tags and tests ..... 12

CI and drift ..... 14

Further reading ..... 15

## Mental model

A CDK app is a program that writes CloudFormation templates. `cdk synth` runs your code and emits templates; `cdk deploy` hands them to CloudFormation, which does the actual provisioning. So every CDK problem is either a synth-time problem (your code, types, lookups) or a deploy-time CloudFormation problem (permissions, limits, replacements) — the first fix is knowing which phase failed.

## App, stacks and constructs

| Level | What it is | Example |
|---|---|---|
| L1 | One-to-one CloudFormation resource | `s3.CfnBucket` |
| L2 | Curated resource, sane defaults | `s3.Bucket` |
| L3 | Pattern of several resources | `ecs_patterns.*` |

| Level | What it is | Example |
| --- | --- | --- |
| Stack | One CloudFormation stack | `new Stack(app, id)` |

Constructs form a tree: the `App` is the root, stacks are its children, and resources live inside stacks. Each construct takes `(scope, id, props)`; the path of ids becomes the CloudFormation logical ID. Prefer L2s and drop to L1 only for a property the L2 does not expose.

```
import { App, Stack } from "aws-cdk-lib";
import * as s3 from "aws-cdk-lib/aws-s3";

class StorageStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);
    this.bucket = new s3.Bucket(
      this, "Uploads", { versioned: true },
    );
  }
```

```
}


const app = new App();
new StorageStack(app, "Storage");
```

> **Gotcha:** Renaming a construct id or moving it to another parent changes its logical ID, and CloudFormation replaces the resource — a new, empty bucket or table. `cdk refactor` (experimental) exists to map such moves.

## Synth, diff, deploy

| Command | Does |
| --- | --- |
| `cdk synth` | Writes templates to `cdk.out/` |
| `cdk diff` | Compares app with deployed stack |
| `cdk deploy` | Deploys via a change set |
| `cdk deploy --hotswap` | Fast path, skips CloudFormation |

| Command | Does |
| --- | --- |
| `cdk destroy` | Deletes the stack |
| `cdk drift` | Reports out-of-band changes |

Run `cdk diff` before every deploy and read the replacement markers. `cdk deploy` creates and executes a change set by default; `--method=prepare-change-set` creates it for review without executing. `--require-approval destructive` stops for confirmation before anything is replaced or deleted; the default, `broadening`, stops only for widened IAM or security-group rules. In CI with no terminal, a required approval fails the deploy.

```
npx cdk synth Storage
npx cdk diff Storage
npx cdk deploy Storage \
  --require-approval destructive
```

> **Warning:** `--hotswap` deliberately introduces drift: it updates Lambda code and similar resources directly, behind CloudFormation's back. Use it in dev stacks only, never in production.

## Environments and bootstrapping

| `env` setting | Result |
|---|---|
| Omitted | Environment-agnostic stack |
| `CDK_DEFAULT_*` vars | Resolved from the CLI at synth |
| Literal account and region | Pinned — use for production |

`cdk bootstrap` deploys the `CDKToolkit` stack: an S3 bucket and ECR repository for assets, plus the roles deployments assume. CDK never bootstraps automatically; run it once per account and Region you deploy to. Lookups such as `Vpc.fromLookup` need a

concrete account and Region, and cache results in `cdk.context.json` — commit that file.

```
npx cdk bootstrap aws://123456789012/eu-west-1
```

```
new StorageStack(app, "Storage", {
  env: {
    account: "123456789012",
    region: "eu-west-1",
  },
});
```

> **Gotcha:** With `CDK_DEFAULT_ACCOUNT`, the template depends on whose credentials ran synth. Fine on a laptop; pin literals for production so CI and humans synthesize the same thing.

## Cross-stack references

| Strength | Mechanism | Coupling |
|---|---|---|
| Strong (default) | Export + `Fn::ImportValue` | Producer locked |
| Weak | `Fn::GetStackOutput` | None |
| Both | Transitional, for migration | Partial |

Pass a construct from one stack's props into another and CDK wires the reference for you, across Regions and accounts too. Strong references are enforced by CloudFormation: an export cannot be removed while another stack imports it.

```
const storage = new StorageStack(app, "Data");
new ApiStack(app, "Api", {
  bucket: storage.bucket,
});
// Data exports the bucket name;
// Api imports it with Fn::ImportValue
```

> **Gotcha:** Removing the bucket now fails with `Export ... cannot be deleted as it is in use by Api` — the "deadly embrace". Switch the reference to `ReferenceStrength.BOTH`, deploy, then `WEAK`, deploy, then remove it.

## Assets and Lambda bundling

| Prop | Purpose |
| --- | --- |
| `entry` | Handler source file (`.ts`, `.js`, ...) |
| `handler` | Export name, default `handler` |
| `bundling.externalModules` | Packages left out of the bundle |
| `bundling.forceDockerBundling` | Bundle in a container |

`NodejsFunction` bundles TypeScript or JavaScript with esbuild. If `esbuild` is installed locally it runs on your

machine; otherwise CDK bundles inside a Lambda-compatible Docker container, which is much slower. Assets are uploaded to the bootstrap bucket at deploy time.

```
import * as nodejs from
  "aws-cdk-lib/aws-lambda-nodejs";
import { Runtime }
  from "aws-cdk-lib/aws-lambda";

const fn = new nodejs.NodejsFunction(
  this, "Api", {
    entry: "src/api.ts",
    runtime: Runtime.NODEJS_LATEST,
  },
);
```

> **Tip:** `npm install --save-dev esbuild@0` so bundling never falls back to Docker. In a monorepo, install it in the root `package.json`.

## Grants and removal policies

| API | Effect |
| --- | --- |
| `bucket.grantRead(fn)` | Least-privilege IAM for `fn` |
| `table.grantReadWriteData(fn)` | Same, for DynamoDB |
| `RemovalPolicy.RETAIN` | Default for stateful resources |
| `RemovalPolicy.DESTROY` | Delete with the stack |
| `autoDeleteObjects: true` | Empties a bucket first |

`grant*` methods write the IAM policy statements — and KMS key permissions where needed — so you rarely author policy JSON. Stateful resources such as buckets and tables default to `RETAIN`, so `cdk destroy` orphans them instead of deleting your data.

```
import { RemovalPolicy } from "aws-cdk-lib";

const tmp = new s3.Bucket(this, "Scratch", {
  removalPolicy: RemovalPolicy.DESTROY,
```

```
  autoDeleteObjects: true,
});
tmp.grantReadWrite(fn);
```

> **Gotcha:** `DESTROY` alone does not delete a bucket that still has objects — CloudFormation refuses. Add `autoDeleteObjects: true`, which installs a custom resource that empties it.

## Aspects, tags and tests

| API | Use |
|---|---|
| `Tags.of(scope).add(k, v)` | Tag everything below scope |
| `Aspects.of(scope).add(a)` | Visit every construct |
| `Template.fromStack(stack)` | Synthesize for assertions |
| `hasResourceProperties` | Assert a resource exists |
| `resourceCountIs` | Assert a count |

An aspect is a visitor run over the construct tree before synthesis: use it to tag, enforce encryption, or fail the build on a policy violation. Unit-test stacks with the `assertions` module — it checks the synthesized template, so it catches CloudFormation shape, not runtime behavior.

```
import { Template, Match }
  from "aws-cdk-lib/assertions";

const t = Template.fromStack(stack);
t.resourceCountIs("AWS::S3::Bucket", 1);
t.hasResourceProperties("AWS::S3::Bucket", {
  VersioningConfiguration: {
    Status: "Enabled",
  },
  Tags: Match.anyValue(),
});
```

## CI and drift

| Concern | Approach |
| --- | --- |
| Pipeline | `pipelines.CodePipeline`, or any CI running `cdk deploy` |
| Review | `cdk diff` output on every PR |
| Drift | `cdk drift`, then fix in code |

CDK Pipelines is a self-mutating CodePipeline: the synth step produces `cdk.out`, and the pipeline updates itself before deploying your stages. Any CI works too, as long as it runs synth and deploy with credentials that can assume the bootstrap roles. Console edits create drift that the next deploy may silently overwrite.

```
npm ci
npx cdk synth
npx cdk diff --all
npx cdk deploy --all --require-approval never
npx cdk drift --fail
```

## Further reading

- AWS CDK v2 Developer Guide
- aws-cdk-lib API reference
- CDK Pipelines
- AWS CDK CLI README