aws

CDK Blocks Developer Guide

AWS CDK Blocks



Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS CDK Blocks: CDK Blocks Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS CDK Blocks? 1
Why CDK Blocks 1
CDK Blocks 2
Data and storage 2
Authentication 2
Compute and background 3
AI 3
Communication 3
Configuration 4
Observability 4
Hosting and deployment 4
How CDK Blocks works 5
Supported platforms 5
Related services and tools 5
Next steps 7
Choosing AWS CDK Blocks 8
Should I use CDK Blocks? 8
How does CDK Blocks compare with other approaches? 8
When should I choose CDK Blocks? 13
What does CDK Blocks require? 14
When is another approach a better fit? 14
Decision table 14
Next steps 15
Getting started 16
Prerequisites 16
Create your CDK Blocks project 16
Run the application locally 17
Explore the backend code 18
Explore the frontend code 21
Make a change 21
Available CDK Blocks 22
Optional: Deploy your application to AWS 23
Clean up 23
Next steps 23

Concepts ... **24**
CDK Blocks ... 24
Scope ... 24
The app definition ... 25
Conditional exports ... 26
ApiNamespace ... 27
BlocksContext ... 27
The CDK layer ... 28
Local development ... 29
Sandbox deployments ... 29
Terminology reference ... 30
Local development ... **31**
How local development works ... 31
Conditional exports ... 31
The dev server ... 33
Where local state lives ... 34
What each Block maps to locally ... 34
Testing with local implementations ... 38
Unit test business logic with CDK Blocks as parameters ... 38
Integration test against local implementations ... 39
Control and inspect state through `.bb-data/` ... 40
Test against a sandbox ... 40
Testing ... **42**
Where to run tests ... 42
Choosing where to run a test ... 42
Related resources ... 43
Supported platforms ... **44**
Web frameworks ... 44
Native and mobile platforms ... 45
How native codegen works ... 46
Writing APIs for native compatibility ... 46
Hosting and deployment ... 47
Framework auto-detection ... 47
Deploy to AWS ... **49**
Setting up AWS credentials ... 49
Sign up for an AWS account ... 49

Create a user with administrative access ... 49
Install and configure the AWS CLI ... 50
Bootstrap the AWS CDK ... 50
Deploy to a sandbox ... 51
Removing a sandbox ... 52
Recover from a failed sandbox deploy ... 53
Full deployment ... 53
Deleting the production stack ... 54
Running the CDK CLI yourself ... 54
Staged deployments with a pipeline ... 55
Next steps ... 56
CDK Blocks reference ... 57
Authentication ... 2
Data storage ... 57
Realtime and async ... 58
AI ... 3
Communication ... 3
Configuration ... 4
Observability ... 4
Hosting ... 60
Authentication ... 60
Choosing a sign-in method ... 60
Auth ... 61
Data storage ... 65
Choosing a data Block ... 65
KVStore ... 66
Database ... 68
FileBucket ... 73
Realtime and async ... 75
Choosing a realtime or async Block ... 75
Realtime ... 75
AsyncJob ... 75
CronJob ... 76
AI ... 76
Choosing an AI Block ... 76
Agent ... 77

KnowledgeBase ... 78
Communication ... 78
EmailClient ... 78
Configuration ... 79
AppSetting ... 79
Secret ... 79
Observability ... 81
Choosing an observability Block ... 82
Metrics ... 82
Logger ... 82
Tracer ... 83
Dashboard ... 83
Best practices ... 84
Project structure ... 84
Error handling ... 85
Authentication and authorization ... 86
Local development ... 87
Testing ... 88
Performance ... 88
Deployment ... 89
Working with coding agents ... 90
Example prompt for coding agents ... 91
Hosting ... 92
What Hosting provisions ... 92
Supported frameworks ... 93
Basic usage ... 94
Front door ... 97
Preview deployments ... 102
Custom domains ... 104
SSR compute ... 105
Security ... 107
Operations ... 108
Configuration reference ... 109
Custom CDK Blocks ... 113
Use cases for custom CDK Blocks ... 113
Create a Block with the scaffolder ... 113

Block structure ... 114
Configure package.json exports ... 115
Define shared types ... 117
Define named errors ... 117
Implement the runtime ... 117
Write the local implementation ... 119
Implement the infrastructure ... 120
Client middleware (optional) ... 121
Use your Block ... 122
Testing your Block ... 123
Publishing ... 124
Best practices for custom CDK Blocks ... 125
Existing apps ... 126
AWS Amplify Gen 2 apps ... 126
What the command adds ... 126
Calling the API with Amplify sign-in ... 128
Running and deploying ... 128
Existing AWS CDK apps ... 129
Related resources ... 130
Existing infrastructure ... 131
Choosing a pattern ... 131
Pattern 1: CDK in CDK Blocks ... 131
Example: BlocksStack with an SNS topic ... 132
Example: BlocksBackend inside an existing stack ... 133
Tradeoffs ... 134
Pattern 2: fromExisting ... 134
Example ... 135
Tradeoffs ... 136
Pattern 3: Custom CDK Block ... 136
When to use ... 136
Pattern 4: Vendorize ... 137
Tradeoffs ... 137
Pattern 5: Infrastructure outside the AWS CDK ... 137
When to use ... 138
When not to use ... 138
Tradeoffs ... 138

Raw HTTP routes and webhooks ... 139
CLI reference ... 141
Create a project ... 141
Local development ... 143
Sandbox deployment ... 143
Production deployment ... 144
Other project scripts ... 144
Related resources ... 146
Telemetry and identification ... 146
CDK Blocks CLI telemetry overview ... 147
How to opt out of CDK Blocks CLI telemetry ... 152
Send telemetry data to a local file ... 153
CloudFormation template metadata ... 154
Service request user agent ... 155
Versioning ... 157
Version numbers ... 157
Exceptions ... 157
Upgrade your packages ... 158
Examples ... 159
CRUD API with authentication ... 159
Realtime chat ... 160
Frontend integration ... 161
File uploads with metadata ... 162
SQL-backed application ... 163
Common patterns ... 165
Related resources ... 165
Security ... 167
How CDK Blocks manages permissions ... 167
Authentication ... 171
Data protection ... 172
Network security ... 172
API protection ... 174
Managing secrets ... 176
Least privilege for deployment ... 177
Troubleshooting ... 178
Local development issues ... 178

Port 3000 or 3001 already in use 178
Local data is stale or corrupted 178
TypeScript errors after changing the backend 178
Deployment issues 179
CDK bootstrap required 179
AWS credentials expired or missing 179
Sandbox deployment hangs or times out 179
Permission denied during deployment 180
Runtime issues 180
Behavior differs between local and deployed 180
API returns 500 Internal Server Error 181
Getting further help 182

What is AWS CDK Blocks?

AWS CDK Blocks is an expansion of the AWS Cloud Development Kit (CDK). Each CDK Block packages a CDK construct along with the connecting code applications need to use that infrastructure, incorporating AWS service-specific best practices and a local testing setup. AWS is releasing more than 17 CDK Blocks for common uses of AWS, including data storage and processing, background jobs, and building AI agents. Developers can build and test their applications locally and deploy the same code unchanged to their own AWS account.

- *CDK Blocks reference*
- *Getting started*
- *CLI reference*

Your entire application runs locally without an AWS account. When you're ready, deploy the same code to AWS without changing anything.

Why CDK Blocks

Your account, your infrastructure, and your standards

Every app deploys as a standard CDK stack to your own AWS account. Set a CDK Block's options, point it at an existing resource with `fromExisting`, change its infrastructure in the CDK layer, or copy its source into your project with `vendorize`. Add any CDK Construct next to the CDK Blocks.

Your standards as custom CDK Blocks

Platform developers package their own standards as custom CDK Blocks and share them through npm, so app teams start from the same approved pattern.

Your whole app on your machine, and the same code on AWS

Run one command to get a working application on your machine. You don't need an AWS account until you're ready to deploy.

End-to-end type safety

Define your backend once. Your frontend types update on their own. TypeScript carries types from your database to your UI, and a TypeScript frontend calls the backend with no code generation step. For Swift, Kotlin, and Dart, you generate a client from a spec file.

Guidance for coding agents

New projects include an `AGENTS.md` file with instructions for coding agents. Each CDK Block's documentation ships in the npm package, under `node_modules/aws-cdk-blocks/docs/`. No plugins needed.

CDK Blocks

A CDK Block packages an app capability: a CDK Construct for its infrastructure, its runtime code, and a local implementation. Each CDK Block is an npm package. You import it and use it in your backend code.

The following table lists all available CDK Blocks.

Data and storage

Block	Description	AWS service
`KVStore`	Key-value and structured data with conditional writes, indexes, and queries	Amazon DynamoDB
`Database`	SQL on PostgreSQL with migrations and transactions	Amazon Aurora DSQL, or Amazon Aurora PostgreSQL Serverless v2
`FileBucket`	File storage with presigned URLs	Amazon S3

Authentication

Block	Description	AWS service
`Auth`	Email and password sign-in with MFA, groups, and passkeys, plus social, OIDC, and SAML sign-in	Amazon Cognito, or no user pool with `store: 'builtin'`

Compute and background

Block	Description	AWS service
`AsyncJob`	Background work with retries, and optional status tracking	Amazon SQS + AWS Lambda or AWS Fargate
`CronJob`	Scheduled task execution	Amazon EventBridge Scheduler + AWS Lambda
`Compute`	Choose where a workload runs: serverless or container	AWS Lambda or AWS Fargate

AI

Block	Description	AWS service
`Agent`	AI agent with tools, HITL approval, conversation persistence	Amazon Bedrock AgentCore Runtime + Amazon Bedrock models
`KnowledgeBase`	Semantic document retrieval and RAG	Amazon Bedrock Knowledge Bases

Communication

Block	Description	AWS service
`Realtime`	WebSocket pub/sub channels	Amazon API Gateway WebSocket
`EmailClient`	Transactional email	Amazon SES

Configuration

Block	Description	AWS service
`AppSetting`	Configuration values that are not sensitive , such as feature flags and settings for the whole app	AWS Systems Manager Parameter Store
`Secret`	Production secrets, such as API keys and connection strings	AWS Secrets Manager

Observability

Block	Description	AWS service
`Logger`	Structured JSON logging with levels and context	Amazon CloudWatch Logs
`Metrics`	Custom application metrics	Amazon CloudWatch
`Tracer`	Distributed request tracing	AWS X-Ray
`Dashboard`	Dashboard with health and logs for each compute, plus the metrics you choose	Amazon CloudWatch

Hosting and deployment

Component	Description	AWS service
`Hosting`	Frontend deployment with SSR support (CDK layer, import from `aws-cdk-blocks/cdk`)	Amazon CloudFront + Amazon S3

For detailed API references for each CDK Block, see *CDK Blocks reference*.

How CDK Blocks works

CDK Blocks uses Node.js conditional exports to load different code for each context:

- **Local development**: CDK Blocks use in-memory and filesystem storage. Your app runs on your machine.
- **CDK synthesis**: CDK Blocks produce CDK Constructs. CDK Blocks creates a CloudFormation template.
- **AWS Lambda runtime**: CDK Blocks call AWS services through the SDK.

The same `new KVStore(scope, 'todos')` line becomes a local store in development, a DynamoDB table at deploy time, and SDK calls in production. You don't change any code.

Supported platforms

CDK Blocks supports web frameworks (Next.js, Nuxt, Astro, React, Vue, Svelte, Angular), native mobile (Swift, Kotlin, Dart/Flutter), and desktop apps. Type safety extends from your backend all the way to your client, regardless of platform.

For the full list of supported frameworks and versions, see *Supported platforms*.

Related services and tools

The following are some of the AWS services and tools that CDK Blocks works with. This is not an exhaustive list.

AWS Amplify

AWS Amplify is a set of tools and services for building and hosting apps. You can add CDK Blocks to an existing Amplify Gen 2 app. For more information, see *Existing apps*.

AWS Lambda

CDK Blocks runs your backend on AWS Lambda by default. With the `Compute` Block, you can run a workload as a container on AWS Fargate instead. For more information, see the AWS Lambda Developer Guide.

Amazon Elastic Container Service (Amazon ECS)

With the `Compute` Block set to `container`, CDK Blocks runs your backend as a long-lived process on AWS Fargate, the serverless compute engine for Amazon ECS. For more information, see the Amazon ECS Developer Guide.

Amazon API Gateway

CDK Blocks uses API Gateway to expose your backend APIs and WebSocket channels. For more information, see the API Gateway Developer Guide.

Amazon DynamoDB

Several CDK Blocks use DynamoDB for data persistence. For more information, see the Amazon DynamoDB Developer Guide.

Amazon Aurora

The Database Block uses Aurora DSQL by default, and Aurora PostgreSQL Serverless v2 for an `aurora` cluster. For more information, see the Amazon Aurora DSQL User Guide and the Amazon Aurora User Guide.

Amazon Bedrock

The Agent Block uses Amazon Bedrock models and runs its agent loop on Amazon Bedrock AgentCore Runtime. The KnowledgeBase Block uses Amazon Bedrock Knowledge Bases. For more information, see the Amazon Bedrock User Guide.

Amazon Cognito

The `Auth` Block uses a Cognito user pool by default. For more information, see the Amazon Cognito Developer Guide.

Amazon S3

`FileBucket` stores files in Amazon S3, and `Hosting` serves static assets from it. For more information, see the Amazon S3 User Guide.

Amazon SQS

`AsyncJob` queues work in Amazon SQS. For more information, see the Amazon SQS Developer Guide.

Amazon EventBridge

`CronJob` runs on an Amazon EventBridge Scheduler schedule. For more information, see the Amazon EventBridge Scheduler User Guide.

Amazon SES

`EmailClient` sends email through Amazon SES. For more information, see the Amazon SES Developer Guide.

AWS Secrets Manager

The `Secret` Block stores a secret in Secrets Manager. For more information, see the AWS Secrets Manager User Guide.

AWS Systems Manager Parameter Store

`AppSetting` stores configuration values in Parameter Store. For more information, see the AWS Systems Manager User Guide.

Amazon CloudWatch

`Logger`, `Metrics`, and `Dashboard` send data to CloudWatch. For more information, see the Amazon CloudWatch User Guide.

AWS X-Ray

`Tracer` sends traces to X-Ray. For more information, see the AWS X-Ray Developer Guide.

Amazon CloudFront

`Hosting` serves your frontend through CloudFront. For more information, see the Amazon CloudFront Developer Guide.

Next steps

- *Getting started*: Set up your environment and build your first application.
- *Concepts*: Learn about CDK Blocks, scopes, and the app definition.
- *CDK Blocks reference*: Browse all available CDK Blocks and their APIs.

Choosing AWS CDK Blocks

Use this page, or point your coding agent at it, to decide whether to build with AWS CDK Blocks. It explains how CDK Blocks builds on the AWS CDK, compares it with other ways to build on AWS, and lists what it requires. To choose a specific CDK Block, see the pages linked at the end.

Should I use CDK Blocks?

CDK Blocks builds on the AWS CDK. A CDK Construct defines AWS infrastructure, such as a queue or a database. A CDK Block adds two things to a construct. It includes the code your application calls to use that infrastructure, and it includes a local implementation, so you can run and test without an AWS account.

The code in each CDK Block follows AWS best practices. For example, it retries failed work, sends messages that keep failing to a dead-letter queue, and grants only the AWS Identity and Access Management (IAM) permissions it needs. You don't write that code yourself.

Use CDK Blocks to build TypeScript applications on AWS. These include APIs for frontends, other services, and command line tools, as well as background jobs, scheduled jobs, realtime capabilities, and AI agents. Every CDK Block contains a CDK Construct, so you can add any other CDK Construct to the same stack.

How does CDK Blocks compare with other approaches?

The following table compares CDK Blocks with CDK Constructs on their own, and with three other ways to build on AWS.

Question	CDK Blocks	AWS CDK Constructs	AWS SDK from your own server	A managed backend platform	Containers you run yourself
What do you write?	You create CDK Blocks and write the methods your callers use, all in one	You define every resource. You also write the code that calls each service,	You write a server that calls AWS services through the AWS SDK.	You write app code against the platform's APIs. The platform defines and	You write the application, build the container images, and configure

Question	CDK Blocks	AWS CDK Constructs	AWS SDK from your own server	A managed backend platform	Containers you run yourself
	`cdk-blocks/` directory. Each CDK Block provides its infrastructure, the code that calls its AWS service, and a local implementation.	connect your API, and write the IAM policies.	You create the resources some other way.	runs the backend.	orchestration, networking, and scaling.
Who writes retries, dead-letter queues, and batch failure handling?	The CDK Blocks do. For example, `AsyncJob` creates a dead-letter queue, reports which jobs in a batch failed, and sets the queue timeout to match the worker's timeout.	You do. Constructs create the infrastructure, but your application code handles failures.	You do.	It depends on the platform.	You do.
Where does it run?	In your own AWS account, as a CDK stack.	In your AWS account, on compute you define.	Wherever you host the server.	On the platform's infrastructure.	On the cluster or container service you operate.

Question	CDK Blocks	AWS CDK Constructs	AWS SDK from your own server	A managed backend platform	Containers you run yourself
Can you run the whole application locally without an AWS account?	Yes. `npm run dev` runs every CDK Block locally and stores data in `.bb-data/`. Three capabilities need a deployed stack: the `Dashboard` route, `Auth` providers that sign in through Amazon Cognito, and `KnowledgeBase` sources in Amazon S3.	No. You test against AWS, or against emulators you set up.	The server runs locally, but its AWS SDK calls go to AWS unless you stub them.	It depends on the platform.	The containers run locally, but the AWS services they call still need AWS or emulators.
Who writes IAM permissions?	The CDK Blocks do. Each CDK Block grants only the access it needs to one shared execution role.	You do, with grant methods or policy statements.	You do, for the server's role or credentials.	The platform does.	You do, for task or pod roles.

Question	CDK Blocks	AWS CDK Constructs	AWS SDK from your own server	A managed backend platform	Containers you run yourself
How do callers get a typed client?	TypeScript callers import types directly from the backend, so there is no schema to maintain. Dart, Swift, and Kotlin clients are generated from a spec file.	You define the API and the client, or generate the client from a schema.	You define the API and the client.	Usually through the platform's own client SDK.	You define the API and the client.

Question	CDK Blocks	AWS CDK Constructs	AWS SDK from your own server	A managed backend platform	Containers you run yourself
Where does your code run?	Your API methods run on AWS Lambda by default, or on a container with the `Compute` Block. Each call has about 29 seconds to respond. Work that takes longer goes in an `AsyncJob`, which runs in the background: on Lambda for up to 15 minutes, or on a container for longer jobs.	On any compute you define.	In your server process.	On whatever the platform provides.	In processes you size and run, in any language.

Question	CDK Blocks	AWS CDK Constructs	AWS SDK from your own server	A managed backend platform	Containers you run yourself
What if you need something the CDK Blocks don't provide?	Add any CDK Construct to the same stack, point a CDK Block at a resource you already have with `fromExisting`, or copy a CDK Block's source into your project with vendorize.	You already control every resource.	You control the server code. The infrastructure is managed elsewhere.	You are limited to the platform's extension points.	You control everything you run.

When should I choose CDK Blocks?

Choose CDK Blocks when:

- You build a TypeScript app on AWS, such as an API, a background worker, a scheduled job, a realtime capability, or an AI agent.
- You want the code that calls AWS services to follow AWS best practices, without writing it yourself.
- You want to build and test locally without an AWS account, and then deploy the same code.
- You want IAM permissions and resource settings to come from the CDK Blocks you use.
- You already have a CDK app. You can add CDK Blocks to one of its stacks with the `BlocksBackend` construct. For more information, see Pattern 1: CDK in CDK Blocks.
- You are a platform developer. You can package your organization's approved patterns, such as encryption or sign-in rules, as custom CDK Blocks that your app teams install from npm. For more information, see Creating custom CDK Blocks.

What does CDK Blocks require?

CDK Blocks fits when your application meets these requirements:

- **TypeScript on Node.js.** CDK Blocks are TypeScript packages. Your code runs on Node.js 24 on AWS, and local development needs Node.js 22 or later.
- **API calls that respond within about 29 seconds.** Put longer work in an `AsyncJob`. It runs in the background, and on a container it can run for longer than 15 minutes.
- **HTTP and WebSocket.** CDK Blocks serve requests over HTTP and WebSocket.
- **AWS as the deployment target.** CDK Blocks deploys as a CDK stack in your own AWS account.
- **One shared execution role.** All CDK Blocks code runs with one shared IAM role. You can limit that role with a permissions boundary, but you can't give each function its own role.

If part of your application doesn't meet these requirements, such as a service in another language, build that part with CDK Constructs or containers in the same stack. Use CDK Blocks for the rest.

When is another approach a better fit?

Choose AWS CDK Constructs without CDK Blocks when you need full control of every resource setting, and you're ready to write the service calls, IAM policies, and local testing setup yourself.

Choose the AWS SDK from your own server when you already run a server and only need to call a few AWS services from it.

Choose a managed backend platform when you accept the platform's limits so you don't have to run infrastructure, and you don't need the resources in your own AWS account.

Choose containers you run yourself when your API needs a server process that keeps running between requests, a language other than TypeScript, or a protocol other than HTTP and WebSocket.

Decision table

Start at the top row and stop at the first row that matches your application.

If	Then
Your application isn't written in TypeScript for Node.js.	Use another approach, such as containers you run yourself.
Your application must run outside AWS.	Use another approach.
An API call needs more than about 29 seconds, or your API needs a server process that keeps running, or a protocol other than HTTP and WebSocket.	Build that API with containers. Long background work still fits CDK Blocks. Put it in an `AsyncJob`, and run it on a container if it takes more than 15 minutes.
You already have a CDK app.	Add CDK Blocks to one of its stacks with `BlocksBackend` . For more information, see Pattern 1: CDK in CDK Blocks.
CDK Blocks cover most of your application, but one resource has no CDK Block.	Use CDK Blocks, and add a CDK Construct for that resource to the same stack. For more information, see Existing infrastructure.
Your organization needs its own approved version of a capability.	Build a custom CDK Block. For more informati on, see Creating custom CDK Blocks.
Your application is TypeScript, runs on AWS, and there are CDK Blocks for what it needs.	Use CDK Blocks.

Next steps

- To choose a CDK Block, see Data storage, Authentication, and the full list in CDK Blocks reference.
- To learn when to go beyond a CDK Block, see Existing infrastructure.

Getting started with AWS CDK Blocks

In this tutorial, you set up your development environment, create a todo application with user authentication, data persistence, and a type-safe API. The application runs locally and can optionally deploy to AWS.

Prerequisites

To develop with AWS CDK Blocks, you need the following on your local machine:

- **Node.js** version 22 or later. Download from https://nodejs.org/.
- **npm** version 10 or later (included with Node.js).
- A code editor with TypeScript support, such as Visual Studio Code or Kiro.

To verify your Node.js and npm versions:

```
node --version
npm --version
```

For the optional deployment step, you also need:

- AWS CLI configured with credentials
- AWS CDK bootstrapped in your account

For deployment details, see *Deploy to AWS*.

Create your CDK Blocks project

Create a new CDK Blocks application:

```
npm create @aws-cdk-blocks/blocks-app@latest my-todo-app
cd my-todo-app
```

The command copies the `default` template into `my-todo-app` and installs its dependencies. The project has the following structure:

```
my-todo-app/
```

```
### .cdk-blocks/
#   ### config.json
### cdk-blocks/          2
#   ### index.ts
#   ### index.cdk.ts     3
#   ### index.handler.ts 4
#   ### client.js        5
#   ### package.json
#   ### tsconfig.json
### public/
#   ### favicon.svg
### src/                 6
#   ### index.ts
### test/
#   ### e2e.test.ts      7
### .gitignore
### AGENTS.md            8
### README.md
### cdk.json
### index.html
### package.json
### tsconfig.app.json
### tsconfig.json
### tsconfig.node.json
### vite.config.ts
```

A unique stack ID for this project. Deployments use it to name your stacks, so commit this file.

2 The backend: CDK Block instantiations and API definitions in `index.ts`.

3 The CDK app that defines your AWS deployment.

4 The AWS Lambda entry point for your backend.

5 The typed frontend client. The development server regenerates it, so do not edit it.

6 The frontend: imports and calls the backend API directly.

7 An end-to-end test. Run it with `npm run test:e2e`.

8 Instructions for coding agents that work in this project.

Run the application locally

Start the development server:

```
npm run dev
```

In your web browser, navigate to `http://localhost:3000`. You see a todo application with authentication, CRUD operations, priorities, and sorting.

All CDK Blocks are running with local implementations:

- `KVStore` uses file-backed storage in `.bb-data/` for structured data
- `Auth` runs a local user pool in `.bb-data/` and prints sign-up codes in the terminal instead of sending email
- `Realtime` runs a local WebSocket server for pub/sub channels
- `ApiNamespace` routes calls through a local HTTP server

No AWS account is needed. Changes to your code are reflected immediately through hot reload.

Explore the backend code

The `cdk-blocks/index.ts` file defines your backend and API in a single place. The listings in this section show its code without the template's comments. The first part creates the CDK Blocks:

```
import { ApiNamespace, ApiError, Scope, Auth, KVStore, Realtime } from 'aws-cdk-
blocks';
import { z } from 'zod';

const scope = new Scope('my-app');

const auth = new Auth(scope, 'auth', {
  users: { signInWith: ['email'], caseSensitive: false },
  codeDelivery: async (username, code, purpose) => {
    console.log(`[auth] ${purpose} code for ${username}: ${code}`);
  },
});
export const authApi = auth.createApi();

const todoSchema = z.object({
  userSub: z.string(),
  todoId: z.string(),
  title: z.string(),
  completed: z.boolean(),
  priority: z.number(),     // 1=high, 2=medium, 3=low
  version: z.number(),
  createdAt: z.number(),
```

```
});

const todos = new KVStore(scope, 'todos', {
  schema: todoSchema,
  key: { partitionKey: 'userSub', sortKey: 'todoId' },
});

const rt = new Realtime(scope, 'live', {
  namespaces: {
    todos: Realtime.namespace(z.object({
      action: z.enum(['created', 'updated', 'deleted']),
      todoId: z.string(),
    })),
  },
});

const todosChannel = (userSub: string) => encodeURIComponent(userSub);
```

This code creates three CDK Blocks:

- `new Auth(scope, 'auth', {…})` creates email and password sign-in, where users sign in with their email address. Sign-up confirms the address with a 6-digit code. Locally, `codeDelivery` prints the code in the terminal. On AWS, this provisions an Amazon Cognito user pool, which emails the code. The line `export const authApi = auth.createApi();` exposes the sign-up and sign-in API that the frontend uses. Exporting the `auth` instance itself would expose nothing.
- `new KVStore(scope, 'todos', {…})` creates structured data storage with a Zod schema for validation. The `key` option keys each todo by user and todo ID, so the API can list one user's todos. The `version` field supports optimistic locking. Locally, this is a file-backed store in the `.bb-data/` directory. On AWS, this provisions a DynamoDB table.
- `new Realtime(scope, 'live', {…})` creates a pub/sub namespace named `todos`. The API publishes an event to it whenever a todo changes, so your other open tabs reload their lists. Each user has their own channel. The `todosChannel` helper encodes the user ID, so a user ID can't add `/` segments that place one user's channel under another's. Locally, this runs a WebSocket server inside the development server. On AWS, this provisions an Amazon API Gateway WebSocket API and a DynamoDB table for connections.

The API methods use these CDK Blocks. This listing is shortened. The template also defines `updatePriority` and `deleteTodo`.

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async subscribeTodos() {
    const user = await auth.requireAuth(context);
    return rt.getChannel('todos', todosChannel(user.userSub));
  },

  async createTodo(title: string, priority: number = 2) {
    const user = await auth.requireAuth(context);
    const todoId = `${Date.now().toString(36)}-${Math.random().toString(36).slice(2,
 8)}`;
    const todo = {
      userSub: user.userSub,
      todoId,
      title,
      completed: false,
      priority,
      version: 1,
      createdAt: Date.now(),
    };
    await todos.put({ userSub: user.userSub, todoId }, todo);
    await rt.publish('todos', todosChannel(user.userSub), { action: 'created' as const,
 todoId });
    return todo;
  },

  async listTodos(sortBy?: 'priority' | 'title') {
    const user = await auth.requireAuth(context);
    const list = await Array.fromAsync(
      todos.query({ where: { userSub: { equals: user.userSub } } }),
      (entry) => entry.value,
    );
    if (sortBy === 'priority') return list.sort((a, b) => a.priority - b.priority);
    if (sortBy === 'title') return list.sort((a, b) => a.title.localeCompare(b.title));
    return list;
  },

  async toggleTodo(todoId: string) {
    const user = await auth.requireAuth(context);
    const todo = await todos.get({ userSub: user.userSub, todoId });
    if (!todo) throw new ApiError('Todo not found', 404, { name:
 'TodoNotFoundException' });
    await todos.put(
      { userSub: user.userSub, todoId },
```

```
      { ...todo, completed: !todo.completed, version: todo.version + 1 },
      { ifFieldEquals: { version: todo.version } },
    );
    await rt.publish('todos', todosChannel(user.userSub), { action: 'updated' as const,
 todoId });
    return { success: true };
  },

  // updatePriority and deleteTodo follow the same pattern.
}));
```

Every method starts with `auth.requireAuth(context)`. Each `ApiNamespace` method is a public endpoint until it calls `requireAuth`, and the local implementation does not warn you if you forget. The `toggleTodo` method throws an `ApiError` with status 404 so the client gets a named error, and it uses `ifFieldEquals` to reject a write if another request changed the todo first.

The template keys each user's data on `user.userSub`, which stays the same for the user's lifetime. Key your own data on it too.

Explore the frontend code

The frontend in `src/index.ts` imports the backend API directly:

```
import { api, authApi } from 'cdk-blocks';
```

There is no client generation step, no API URL configuration, and no SDK initialization. TypeScript provides full type safety. If you change a method signature in the backend, the frontend shows a compile error immediately.

Make a change

Add a method that deletes all completed todos. In `cdk-blocks/index.ts`, add it inside the `ApiNamespace` definition, after `deleteTodo`:

```
  async clearCompleted() {
    const user = await auth.requireAuth(context);
    const list = await Array.fromAsync(
      todos.query({ where: { userSub: { equals: user.userSub } } }),
      (entry) => entry.value,
```

```
    );
    const done = list.filter((todo) => todo.completed);
    for (const todo of done) {
      await todos.delete({ userSub: user.userSub, todoId: todo.todoId });
      await rt.publish('todos', todosChannel(user.userSub), { action: 'deleted' as
 const, todoId: todo.todoId });
    }
    return { cleared: done.length };
  },
```

The development server hot-reloads. You can immediately call the new method from the frontend with full type safety:

```
const { cleared } = await api.clearCompleted();
```

Available CDK Blocks

The following table lists the CDK Blocks used in this tutorial and other commonly used CDK Blocks:

Block	Purpose
`KVStore`	Key-value and structured data with indexes and queries
`Auth`	Email and password, social, OIDC, and SAML sign-in with sessions
`ApiNamespace`	Type-safe RPC from browser to backend
`Database`	SQL on PostgreSQL with migrations and transactions
`FileBucket`	File uploads and downloads
`Realtime`	WebSocket pub/sub channels
`AsyncJob`	Background job processing
`Agent`	AI agent with tool calling

For the complete list of all available CDK Blocks, see *CDK Blocks reference*.

Optional: Deploy your application to AWS

When you're ready to deploy to a real AWS environment, see *Deploy to AWS* for the full setup and deployment steps.

Clean up

To stop the local development server, press `Ctrl+C` in your terminal.

If you deployed to AWS and want to remove all resources:

```
# Remove production deployment
npm run destroy

# Remove sandbox environment
npm run sandbox:destroy
```

Next steps

- *Concepts*: Learn about CDK Blocks, scopes, the app definition, and how code maps to resources.
- *CDK Blocks reference*: Explore all available CDK Blocks and their APIs.
- *Examples*: See common application patterns and sample code.

AWS CDK Blocks concepts

This topic explains the key concepts you need to understand when building applications with AWS CDK Blocks.

CDK Blocks

A **CDK Block** is an npm package for a single capability. Each CDK Block packages a CDK Construct for the infrastructure, an interface, and the interface's AWS and local implementations into one package. When you create a CDK Block in your code, CDK Blocks sets up the matching AWS resources, wires permissions, and gives you a local implementation for development.

For example, `KVStore` is a CDK Block for key-value storage. A single instantiation gives you:

- A DynamoDB table (provisioned automatically during deployment)
- AWS SDK integration (used at runtime in Lambda)
- A local store (used during local development)

```
import { KVStore, Scope } from 'aws-cdk-blocks';

const scope = new Scope('my-app');
const cache = new KVStore<{ name: string }>(scope, 'cache', {});

// Same API works locally and in production
await cache.put('user:123', { name: 'Alice' });
const user = await cache.get('user:123');
```

Scope

A **Scope** is a namespace container for CDK Blocks. Every CDK Block must be instantiated within a scope. The scope provides identity and grouping. Each CDK Block's full identifier (`fullId`) joins its scope ID and the ID you provide with `-`.

```
const scope = new Scope('my-app');
const users = new KVStore(scope, 'users', {});   // Full ID: my-app-users
const sessions = new KVStore(scope, 'sessions', {}); // Full ID: my-app-sessions
```

In a deployed app, the full ID also starts with the stack name, for example `my-stack-my-app-users`. Resource names and the local `.bb-data/` folders are derived from it, so scopes keep resource names unique and predictable across your application.

> **Warning**
>
> Renaming a CDK Block ID (the second argument to the constructor) or its scope changes its full ID. On the next deployment, the CDK Block gets a new, empty AWS resource, and your app no longer sees the old data. What happens to the old resource depends on the removal policy. With `BlocksPresets.production`, stateful resources such as `KVStore`, `Database`, and `FileBucket` use a `RETAIN` policy, so the old resource is removed from the stack but kept in your account. With `BlocksPresets.sandbox`, the old resource is deleted along with its data. Always treat CDK Block IDs as immutable once deployed.

The app definition

The **app definition** is your backend entry point, the `cdk-blocks/index.ts` file. This is where you instantiate CDK Blocks and define your API. Each CDK Block includes a CDK Construct, so CDK Blocks creates CDK infrastructure from this code. To change what a CDK Block creates or add other resources, use the CDK layer.

```
// cdk-blocks/index.ts - the app definition
import { ApiNamespace, Scope, KVStore, Auth } from 'aws-cdk-blocks';

const scope = new Scope('my-app');

// Infrastructure is derived from these instantiations
const auth = new Auth(scope, 'auth');
const todos = new KVStore<{ title: string; done: boolean }>(scope, 'todos', {});

// API methods are callable from the frontend
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createTodo(title: string) {
    const user = await auth.requireAuth(context);
    await todos.put(`${user.userSub}:${title}`, { title, done: false });
  },
}));

// Exposes sign-up, sign-in, and sign-out to the frontend
```

```
export const authApi = auth.createApi();
```

The app definition serves three purposes simultaneously:

- **Local development**: CDK Blocks resolve to local implementations
- **Deployment**: CDK Blocks resolve to CDK Constructs that define your infrastructure
- **Production runtime**: CDK Blocks resolve to AWS SDK integrations running in Lambda

Conditional exports

CDK Blocks uses **Node.js conditional exports** to route the same `import` statement to different implementations depending on the execution context. This is the mechanism that makes a single codebase work across local development, deployment, and production.

When you write `import { KVStore } from 'aws-cdk-blocks'`, the resolved file depends on context:

Context	Resolved implementation	What happens
Local development	In-memory and filesystem	Your app runs entirely on localhost
CDK synthesis	CDK Construct (DynamoDB table)	Infrastructure is defined for CloudFormation
Lambda runtime	AWS SDK (DynamoDB client)	Real AWS service calls in production
TypeScript/IDE	Type definitions	Full IntelliSense and type checking

You never need to configure conditional exports manually. CDK Blocks sets up the build system to select the correct implementation for each context automatically.

ApiNamespace

An **ApiNamespace** defines type-safe backend methods that your frontend can call directly. It's the bridge between your backend logic and your client code.

```
// Backend: define the API
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async greet(name: string) {
    return { message: `Hello, ${name}!` };
  },
}));
```

```
// Frontend: call the API directly
import { api } from 'cdk-blocks';

const result = await api.greet('World');
// result.message === 'Hello, World!'
// TypeScript knows the return type
```

There is no code generation, no API client initialization, and no URL configuration. The frontend import is type-safe. If you change the backend method signature, TypeScript reports errors in the frontend immediately.

Locally, `ApiNamespace` routes calls through a local HTTP server. In production, calls go through API Gateway to Lambda.

BlocksContext

The **BlocksContext** is the request/response object provided to `ApiNamespace` handlers. CDK Blocks that need HTTP-level access (such as authentication CDK Blocks reading cookies or setting headers) accept `BlocksContext` as a parameter.

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async protectedAction() {
    // Auth blocks use context to read headers/cookies
    const user = await auth.requireAuth(context);
    return { username: user.username };
  },
}));
```

You don't construct `BlocksContext` yourself. It's provided automatically by the framework for each incoming request.

Every `ApiNamespace` method is a public endpoint until it calls `requireAuth`. Creating an `Auth` Block does not protect any method by itself. To make every method of a namespace require a session, define it with `auth.protect`. The local implementation does not check auth either, so a method you forget to gate still passes every local test.

The CDK layer

The **CDK layer** is the file `cdk-blocks/index.cdk.ts`. In a standalone project, it creates your stack and gives you direct access to CDK Constructs. Use it when you need to:

- Add AWS resources that don't have a CDK Block (such as Amazon SNS topics)
- Configure custom domains or other environment-specific settings
- Integrate CDK Blocks into an existing CDK application

```
// cdk-blocks/index.cdk.ts
import * as cdk from 'aws-cdk-lib';
import * as sns from 'aws-cdk-lib/aws-sns';
import { BlocksStack, BlocksPresets, registerConfig } from 'aws-cdk-blocks/cdk';
import { getStackName } from 'aws-cdk-blocks/scripts';
import { fileURLToPath } from 'node:url';
import { dirname, join } from 'node:path';

const __dirname = dirname(fileURLToPath(import.meta.url));

const app = new cdk.App();
const sandboxMode = app.node.tryGetContext('sandboxMode') === 'true';
const projectRoot = app.node.tryGetContext('projectRoot') || process.cwd();

const stack = await BlocksStack.create(app, getStackName({ sandbox: sandboxMode,
 projectRoot }), {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production,
});

// Add any CDK construct alongside your Blocks
const topic = new sns.Topic(stack, 'alerts');
```

```
topic.grantPublish(stack.executionRole);
registerConfig(stack, 'ALERTS_TOPIC_ARN', topic.topicArn);
```

All compute in your app shares one execution role, `stack.executionRole`, so a grant to it applies to every function. `registerConfig` passes a value to your runtime code, which reads it as `process.env.ALERTS_TOPIC_ARN`. Pass absolute paths to `BlocksStack.create`. A relative `backendCDKPath` resolves against the current working directory, not against this file.

For background work, use the `AsyncJob` and `CronJob` Blocks instead of adding queues or schedules in the CDK layer.

Every project needs this file. In a standalone project, `cdk.json` points the CDK app at it, and CDK Blocks does not generate one if it is missing. The project templates include it.

Local development

When you run `npm run dev`, CDK Blocks starts your entire application locally:

- CDK Blocks use local implementations (file-backed stores, a local user pool, embedded databases)
- The application runs on `http://localhost:3000` with hot reload
- No AWS account, no internet connection, and no cloud costs required

Local data persists in a `.bb-data/` directory at your project root. Each CDK Block gets its own subdirectory.

Sandbox deployments

A **sandbox** is a fast, ephemeral deployment to AWS for testing against real services. When you run `npm run sandbox`:

- CDK Blocks deploys your backend to Lambda with hot-swapping
- CDK Blocks resolve to real AWS services (DynamoDB, API Gateway, etc.)
- Each developer gets an isolated sandbox identified by a unique ID
- Remove it with `npm run sandbox:destroy`

Sandboxes are useful when you need to test behavior that differs between local implementations and real services, such as DynamoDB query performance or IAM permission boundaries.

Terminology reference

Term	Definition
CDK Block	An npm package for a single capability that includes a CDK Construct for its infrastructure, its runtime code, and a local implementation.
Scope	A namespace container that provides identity to CDK Blocks.
App definition	The backend entry point (`cdk-blocks/index.ts`) where CDK Blocks are instantiated and APIs are defined. Infrastructure is derived from this code.
CDK layer	The file (`cdk-blocks/index.cdk.ts`) that creates your stack in a standalone project and gives you direct CDK access for custom infrastructure.
ApiNamespace	A core primitive that defines type-safe RPC methods callable from the frontend. It is not a CDK Block.
BlocksContext	The request/response context object provided to API handlers.
Conditional exports	Node.js mechanism that routes imports to different files based on execution context.
Sandbox	A fast, ephemeral AWS deployment for testing against real services.
Local implementation	A version of a CDK Block that runs on your machine without AWS, backed by memory, the filesystem, or an embedded engine such as PGlite.

Local development

Your whole app runs on your machine. When you run `npm run dev`, your entire application runs on your local machine: backend, frontend, database, authentication, and background jobs. You don't need an AWS account, AWS credentials, an internet connection, or a running container daemon. The only prerequisite is Node.js version 22 or later.

Every CDK Block ships with a local implementation that exposes the same typed API as its AWS counterpart. At GA, each CDK Block runs locally on its own built-in implementation, so your app works without an AWS account. AWS Emulator support for AWS CDK Blocks is planned.

A few things need a network connection, because they call the real service:

- An `Auth` provider that signs in with a real identity provider.
- A `Database` on an existing PostgreSQL database (`DatabaseCluster.fromExisting()`).
- An `Agent` whose `model.local` names a hosted model. An Amazon Bedrock model in `model.local` also needs AWS credentials.

This page explains how local development works, what each CDK Block maps to locally, and how to use local implementations to test your application.

How local development works

Conditional exports

Every CDK Block is an npm package with several implementations behind a single import. CDK Blocks uses Node.js conditional exports to route the same `import` statement to the right implementation for the execution context. For example, the `KVStore` package declares:

```
{
  "exports": {
    ".": {
      "browser": "./dist/index.browser.js",
      "cdk": {
        "types": "./dist/index.cdk.d.ts",
        "default": "./dist/index.cdk.js"
      },
      "aws-runtime": "./dist/index.aws.js",
```

```
      "types": "./dist/index.mock.d.ts",
      "default": "./dist/index.mock.js"
    }
  }
}
```

Context	Export condition	What resolves
Local development	`default`	The local implementation. Your code runs in-process on your machine, backed by the filesystem, in-memory state, or an embedded engine.
Deployment (CDK synthesis)	`cdk`	CDK Constructs that define the CDK Block's AWS infrastructure.
Production runtime (Lambda)	`aws-runtime`	The AWS SDK implementation that calls real AWS services.
Browser	`browser`	The client-side code bundled into your frontend.

The local implementation is the default resolution, so any Node.js process that imports your backend gets the local implementations automatically. This includes the dev server, your test runner, and any script you run. You don't turn on a local mode. The local implementation is used whenever no deployment context applies. CDK Blocks applies the `aws-runtime` condition only when it bundles your backend for Lambda, so the AWS SDK is never loaded during local development.

You never configure this yourself. CDK Blocks sets up the build system to select the correct implementation for each context automatically.

Local implementations are for local development only. If a backend is bundled without the `aws-runtime` condition, it loads the local implementations instead of the AWS ones. To catch this, a CDK Block's local implementation checks where it runs when it loads, and throws an error on compute that CDK Blocks deployed. The error starts with `A Building Block mock entry (first detected: <package>) was loaded in a deployed environment`. It explains how to set the `aws-runtime` condition. The package it names is the first CDK Block whose

check ran, which can be a CDK Block that another CDK Block uses. CDK Blocks detects a deployed environment from the `BLOCKS_CONFIG_BUCKET` and `BLOCKS_CONFIG_KEY` environment variables, which it sets on every compute it provisions. On compute that CDK Blocks does not provision, the check does nothing, so the `aws-runtime` condition is your only protection there.

The dev server

`npm run dev` starts a local dev server in watch mode:

- Your backend runs in-process. There is no separate compute to provision, and code changes reload automatically.
- In templates with a frontend, open `http://localhost:3000`. The dev server listens on that port and forwards page requests to your frontend dev server (Vite or Next.js), which runs on port 3100.
- The templates without a frontend (`backend`, `api-only`, `sql`, `cron`, and `observability`) serve only the API, on port 3001. In an Amplify project, CDK Blocks adds `npm run blocks:dev` instead of `npm run dev`; it also serves the API on port 3001, and `npm run blocks:stop` stops it.
- `ApiNamespace` calls route through the local HTTP server with the same request and response semantics as Amazon API Gateway in production, including `BlocksContext`, cookies, and headers.
- `Realtime` channels connect over a WebSocket served by the dev server.
- `CronJob` schedules fire while the dev server is running.

The dev server listens on `127.0.0.1` only, so other machines on your network cannot reach it. This matters because it serves your whole API against local data. Anyone who can reach the port can create an account, or sign in as any user through a `stubIdp()` provider of an `Auth` Block. They can then read or change that data. To reach the dev server from a container or a remote development machine, set `BLOCKS_DEV_HOST`. For example, `BLOCKS_DEV_HOST=0.0.0.0 npm run dev` listens on every interface and prints a warning. Anyone who can reach that interface can call your API, so do this only where you control access to the port.

To stop a dev server that is running in another terminal or in the background, run `npm run dev:stop`. It stops the server by the process ID that `npm run dev` recorded in the project, so it never stops unrelated processes. If the server uses a port other than 3000, pass the port. For example, in a template without a frontend, run:

```
npm run dev:stop -- --port 3001
```

Where local state lives

CDK Blocks that persist data write to a `.bb-data/` directory in the directory where you start the dev server, which is your project root. Most CDK Block instances get their own subdirectory, named after the CDK Block's full ID: its scope ID joined to its parent scope IDs with `-`. For example, `new KVStore(new Scope('my-app'), 'orders')` writes to `.bb-data/my-app-orders/`. `AppSetting` and `Secret` values share one file, `.bb-data/settings.json`.

This has three practical benefits:

- Data persists across restarts. Stop and restart the dev server, and your users, files, and table rows are still there.
- You can inspect the state directly. The data is stored as plain files, so you can open your `EmailClient` outbox or a `KVStore's contents in your editor.
- You can reset at any time. Delete `.bb-data/`, or a single CDK Block's subdirectory, to start from a clean state.

Keep `.bb-data/` out of version control. Scaffolded CDK Blocks applications add it to `.gitignore` for you.

What each Block maps to locally

Every CDK Block exposes the same typed API locally and on AWS. The following table shows what backs that API in each environment.

Block	Locally	On AWS
`Auth`	A local user pool in `.bb-data/` . Sign-up, sign-in, password reset, MFA, and passkey flows work without any AWS calls, and verification codes go to the `codeDelivery` callback. On a `store: 'builtin'` Block, a	An Amazon Cognito user pool with your configured options, or no user pool with `store: 'builtin'` . `stubIdp()` providers work in sandbox deployments only, and synth fails if any other deployment uses one

Block	Locally	On AWS
	`type: 'oidc'` provider talks to the real identity provider, so register the local callback URL `http://localhost:3000/aws-blocks/auth/callback` with it. A `stubIdp()` provider signs in test users offline without credentials. Social, SAML, and OIDC providers on the default store need a deployed user pool, so their sign-in is unavailable under `npm run dev`	
`KVStore`	File-backed store in `.bb-data/` with the same key and index query semantics. The dev server keeps the store in memory and reads its file only when it creates the store, so another process that uses the same `.bb-data/` directory, such as a second dev server or a script, does not see its writes and can overwrite them	DynamoDB table, with Global Secondary Indexes for its `indexes`
`Database`	PGlite, an embedded WebAssembly PostgreSQL engine, running in-process with data in `.bb-data/` and one schema per CDK Block. Supports migrations, transactions, and Row Level Security. A `dsql` cluster also enforces Amazon Aurora DSQL's SQL rules and transaction limits, so unsupported SQL fails locally instead of at deploy time	Aurora DSQL, or Aurora PostgreSQL Serverless v2 for an `aurora` cluster

Block	Locally	On AWS
`FileBucket`	A directory in `.bb-data/`, mirroring S3 API behavior including presigned upload/download URLs	S3 bucket
`Realtime`	Typed pub/sub over a WebSocket served by the local dev server	API Gateway WebSocket API with DynamoDB connection management
`AsyncJob`	In-process queue. Handlers execute inside the dev server, and failed jobs retry up to `maxRetries` attempts (default 3). With `trackStatus: true`, `getStatus()` and `waitUntilComplete()` report the same states as in production: `queued`, `processing`, `complete`, and `failed`	SQS queue with a dead-letter queue, consumed by the app's default compute or by the `Compute` you assign
`CronJob`	In-process scheduler. `rate` and `cron` schedules fire on Node.js timers while the dev server runs, and each trigger is logged to the terminal	Amazon EventBridge Scheduler schedule that invokes a Lambda function
`Agent`	A canned, keyword-based provider that returns predictable responses without a real model, API key, or cloud cost. You can alternatively point an `openai-api` provider at Ollama or any OpenAI-compatible endpoint	An Amazon Bedrock AgentCore Runtime runs the agent loop and streams over a `Realtime` WebSocket. It calls Amazon Bedrock models, by default `BedrockModels.BALANCED` through a global inference profile. The runtime is billed for vCPU and memory while a session runs

Block	Locally	On AWS
`KnowledgeBase`	A local text index built over your documents in `.bb-data/` , so retrieval works offline	Amazon Bedrock Knowledge Bases with automatic ingestion, chunking, and embedding
`EmailClient`	Emails are captured to a local outbox (`.bb-data/…/emails.json`) instead of being sent, so you can inspect what your applicati on tried to send	Amazon SES
`AppSetting`	Values in `.bb-data/settings.json`	SSM Parameter Store (SecureString when `secret: true`)
`Secret`	Plaintext values in `.bb-data/settings.json` , with the previous version in `.bb-data/secret-versions.json` . Never use real credentials locally. As in production, a `Secret` is read-only unless you set `allowRuntimeWrite: true` , so set a local value by editing `settings.json`	AWS Secrets Manager secret
`Logger`	Structured logs to your terminal, using the same code path as production	Same output, captured by CloudWatch Logs. Entries include the X-Ray trace ID when the runtime provides one. To add a request ID, use `log.child({ requestId })`
`Metrics`	EMF-formatted JSON to your terminal, using the same code path as production	Same output, captured as CloudWatch metrics
`Tracer`	Traces recorded to files in `.bb-data/` for local inspection	AWS X-Ray distributed tracing

Block	Locally	On AWS
`Dashboard`	Not available locally. The dashboard route (`/aws-blocks/dashboard` by default) responds with HTTP 503 and a message directing you to deploy, because there is no CloudWatch dashboard to render	CloudWatch dashboard organized by compute, with health and logs sections, traces when the app has a `Tracer`, and widgets for the metrics you list in `metricConfigs`
`Compute`	No compute is provisioned. Every handler runs in-process in the dev server, whatever compute you assign it to	AWS Lambda function for a `serverless` compute, or an Amazon ECS service on AWS Fargate for a `container` compute

Note two things about this table:

- `Logger` and `Metrics` run the same code in both environments. Only the destination of the output differs: what you see in your terminal locally is what CloudWatch captures in production.
- The data CDK Blocks use real database engines locally. `Database` runs actual PostgreSQL through PGlite, and on a `dsql` cluster it also validates your SQL against Aurora DSQL restrictions, so incompatible SQL fails on your machine instead of during a deployment.

Testing with local implementations

Local implementations also form the basis for testing. A typical CDK Blocks test setup has three levels: unit tests for your business logic, integration tests against local implementations, and end-to-end tests against a sandbox. The first two levels run entirely on your machine.

Unit test business logic with CDK Blocks as parameters

Extract business logic from your API handlers into pure functions that accept CDK Block instances as parameters. Type each parameter with only the CDK Block methods the function calls, using `Pick`. The function then depends only on that part of the CDK Block's typed interface, so you can test it with a standard test runner such as Vitest, without starting the dev server:

```
// orders.ts - testable without the full framework
```

```
import { ApiError, type KVStore } from 'aws-cdk-blocks';

export interface OrderInput {
  title: string;
}

export interface Order extends OrderInput {
  id: string;
  userId: string;
}

// Depend only on the KVStore methods you call, so a test can pass a small stub.
export type OrderStore = Pick<KVStore<Order>, 'put'>;

export async function createOrder(store: OrderStore, userId: string, input:
 OrderInput): Promise<Order> {
  if (!input.title) throw new ApiError('Title required', 400);
  const order: Order = { id: crypto.randomUUID(), ...input, userId };
  await store.put(`${userId}:${order.id}`, order);
  return order;
}
```

```
// orders.test.ts
import { it, expect, vi } from 'vitest';
import { createOrder, type OrderStore } from './orders.js';

it('creates an order', async () => {
  const testStore: OrderStore = { put: vi.fn(async () => {}) };
  const result = await createOrder(testStore, 'user-1', { title: 'Test' });
  expect(result.title).toBe('Test');
  expect(testStore.put).toHaveBeenCalledWith(`user-1:${result.id}`, result);
});
```

The real `KVStore` satisfies `OrderStore`, so your API handler passes its store to `createOrder` unchanged.

For more information about structuring your application this way, see Testing in Best practices.

Integration test against local implementations

For tests that cover CDK Block behavior itself, run your application with `npm run dev` and test against it. CDK Blocks resolve to their local implementations automatically.

Projects created from most templates include such a test in `test/e2e.test.ts`. It uses the Node.js test runner (`node:test`) and calls your API through the same typed client as your frontend. Run it with `npm run test:e2e`. It uses a dev server that is already running, or starts one.

- Tests run in milliseconds because everything is in-process.
- Stateful CDK Blocks behave realistically: conditional writes on `KVStore` fail when they should, `Database` enforces your migrations and Row Level Security, and an `AsyncJob` with `trackStatus: true` records each job's status transitions.
- Tests run anywhere Node.js runs, including CI, without AWS credentials.

Control and inspect state through `.bb-data/`

Because local state is plain files, your tests can seed and assert on it directly:

- Reset between tests or runs by deleting `.bb-data/` or one CDK Block's subdirectory.
- Assert on side effects that are hard to observe in the cloud. For example, `EmailClient` writes every email to a local outbox file. A test can trigger a password reset and then read the outbox to verify the message content.

Test against a sandbox

Local implementations match the behavior of their AWS counterparts at the API level, but some things can only be verified against real services: IAM permission boundaries, service quotas and throttling, DynamoDB query performance at scale, or real model output from Amazon Bedrock.

For those cases, deploy a sandbox with `npm run sandbox` and run the same tests against it. A sandbox is a fast, ephemeral deployment to AWS that each developer gets in isolation. A useful pattern is to parameterize your end-to-end tests by target environment, so the same test suite runs against `localhost` during development and against a sandbox URL before you ship:

Run against	Use for
Local dev server	Day-to-day development, fast feedback, CI on every commit
Sandbox deployment	IAM and permission verification, performance characteristics, service-specific behavior, real AI model responses

Run against	Use for
Production deployment	Smoke tests after release

Do most of your development and testing against local implementations, and use a sandbox when you need to verify the behavior of the AWS services themselves.

For more information about how CDK Blocks are structured, see Conditional exports. For deploying to AWS, see Deploy to AWS.

Testing AWS CDK Blocks applications

You can run the tests for a AWS CDK Blocks application in two places: against each CDK Block's local implementation, or against a sandbox on AWS. This page explains what each place tests and how to choose between them.

Where to run tests

Where	What runs	What it catches	What it misses
Local implementations	Every CDK Block runs its local implementation. This is what `npm run dev` and your test runner use by default.	Mistakes in your business logic and in how you use each CDK Block, such as a failed conditional write, an expired item, or a payload that fails schema validation.	How AWS services respond. The AWS SDK is never loaded.
Sandbox	Your whole application, deployed to AWS with `npm run sandbox`.	Everything that depends on real AWS: IAM grants, the deployed API and job wiring, quotas, performance, and real model output.	Nothing on AWS, but each run takes a deploy and costs what the resources cost.

At GA, each CDK Block runs locally on its own built-in implementation, so your app works without an AWS account. AWS Emulator support for CDK Blocks is planned.

Choosing where to run a test

Run most tests against local implementations. They run in milliseconds, need no AWS account, and run in CI on every commit. For how to write them, see Testing with local implementations. Use a sandbox for what only AWS can show: the IAM grants that CDK Blocks adds, the deployed API and job wiring, quotas, performance, and model output. Also use a sandbox for any behavior that a CDK Block's `DESIGN.md` lists as different on AWS. Many CDK Blocks ship a `DESIGN.md` in their npm

package with a "Mock vs AWS Behavior Differences" table. Run your sandbox tests before you ship. For more information, see Test against a sandbox.

Related resources

- Testing with local implementations
- Deploy to a sandbox

Supported platforms

AWS CDK Blocks supports web, desktop, and mobile platforms. Your backend API is defined once in TypeScript, and type-safe clients are available for every supported platform.

Web frameworks

CDK Blocks auto-detects your web framework and configures the development server and deployment pipeline accordingly. The following frameworks are supported:

Framework	Minimum version	Notes
Next.js	15.x	Full SSR support with automatic route detection and middleware handling. If your project does not install a Next.js build adapter package itself (the `nextjs` template does), Hosting uses a pinned one. The pinned adapter supports Next.js 15.5.18 or a later 15.x release, and Next.js 16.2.6 or later.
Nuxt	4.x	SSR via Nitro adapter with automatic server routes
Astro	4.x	Static and SSR modes supported. Supported range: Astro 4 through 7.
SolidStart	1.x	Via Nitro adapter
TanStack Start	1.x	Via Nitro adapter
SvelteKit	2.x	SSR via `@sveltejs/adapter-node` . If your `svelte.config.js` sets no adapter, Hosting adds `adapter-node` for the build. Any other adapter fails synthesis.
React (SPA)	18.x	Static single-page application, any build tool (Vite, Create React App)

Framework	Minimum version	Notes
Vue (SPA)	3.x	Static single-page application via Vite
Svelte (SPA)	4.x	Static single-page application via Vite
Angular (SPA)	17.x	Static single-page application. Angular 17 and later write the build to a nested folder, so set `buildOutputDir` (for example, `dist/my-app/browser`).

For SPA frameworks, CDK Blocks serves static assets from CloudFront with S3 origin. For SSR frameworks (Next.js, Nuxt, Astro SSR, SvelteKit, SolidStart, TanStack Start), CDK Blocks deploys server-side rendering via Lambda compute.

Framework detection is automatic based on the dependencies, dev dependencies, and peer dependencies in your `package.json`. You can override it explicitly in your Hosting configuration.

Native and mobile platforms

CDK Blocks generates typed native clients from your backend API via OpenRPC codegen. You define your API once in TypeScript, and codegen produces a typed client for each native platform on top of that platform's SDK.

SDK	Platforms	Distribution
Swift	iOS 16 or later, macOS 13 or later	Swift Package Manager
Kotlin (Kotlin Multiplatform)	Android (API level 23 or later), iOS, and the JVM (for example, desktop apps)	Maven/Gradle
Dart (Flutter)	iOS, Android, web, and desktop. Dart 3.11 or later, and Flutter 3.41 or later for `blocks_runtime_flutter`	Dart packages `blocks_runtime`, `blocks_codegen`, and `blocks_runtime_flutter`

How native codegen works

The build pipeline generates an OpenRPC specification (`cdk-blocks/blocks.spec.json`) from your `ApiNamespace` methods. Native codegen tools read this specification to produce typed SDKs for each platform.

The codegen pipeline has two layers:

Platform SDK (hand-written, built once per platform)

Handles transport (HTTP JSON-RPC), authentication (cookie management, token injection), Realtime hydration (WebSocket connections), and FileBucket hydration (presigned URL operations).

Generated typed facade (app-specific, produced by codegen)

One typed method per `ApiNamespace` method. Calls into the platform SDK's transport layer. Contains no transport logic.

To generate the specification:

```
npx blocks-generate-spec
```

This reads `cdk-blocks/index.ts` and writes `cdk-blocks/blocks.spec.json`. Scaffolded apps run it with `npm run spec`. To regenerate it on every build, add it to your build script:

```
{
  "scripts": {
    "build:spec": "blocks-generate-spec",
    "build": "npm run build:spec && tsc && vite build"
  }
}
```

Writing APIs for native compatibility

TypeScript method signatures produce correct native types automatically. For richer types (format constraints, enums), use Zod schemas:

- Plain TypeScript types (tier 2): correct base types inferred from method signatures

- Zod schemas (tier 1): full JSON Schema with constraints (`format: uuid`, `maxLength`, `minLength`). Schema detection requires Zod 4.
- If neither is detected, parameters emit as `unknown` (generates `Any` in native clients)

For best results, use explicit return types and Zod validation on complex inputs.

Hosting and deployment

CDK Blocks provides `Hosting` (import from `aws-cdk-blocks/cdk`) for deploying your frontend alongside your backend. Hosting provisions:

- **Amazon CloudFront** distribution with global edge caching (the default front door; you can choose another)
- **Amazon S3** origin for static assets
- **Lambda compute** for server-side rendering (SSR frameworks only)
- **Optional WAF** integration for security rules
- **Custom domains** with automatic DNS and TLS
- **Skew protection** during rolling deployments

Hosting deploys your frontend when you run `npm run deploy`. A plain sandbox deploys only your backend, and `npm run sandbox:preview` adds Hosting in a fast preview shape. For more information, see Basic usage in the Hosting chapter.

Framework auto-detection

Hosting detects your framework from the `dependencies`, `devDependencies`, and `peerDependencies` in `package.json`. It checks in this order and uses the first match:

- `next` → Next.js adapter
- `nuxt`, `nitropack`, `@solidjs/start`, `@tanstack/start`, or one other package on Hosting's fixed Nitro list → Nitro adapter. For any other Nitro-based framework, set `framework: 'nitro'`
- `astro` → Astro adapter
- `@sveltejs/kit` → SvelteKit adapter
- None of the above → SPA/static adapter

Override detection by setting `framework` explicitly in your Hosting configuration.

Deploy your application to AWS

After developing and testing your application locally, you can deploy it to AWS. This page covers the one-time AWS account setup and the deployment commands.

Setting up AWS credentials

If you already have AWS credentials configured, skip to the section called “Bootstrap the AWS CDK”.

Sign up for an AWS account

If you don’t already have an AWS account, complete the following steps to create one.

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Create a user with administrative access

1. Sign in to the IAM console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.
2. Enable IAM Identity Center.

   For more information about enabling IAM Identity Center, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.
3. In IAM Identity Center, grant administrative access to a user.

For more information about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

> **Note**
>
> Administrative access is recommended for getting started. For production environments, use a least-privilege IAM policy scoped to only the resources that AWS CDK Blocks creates.

Install and configure the AWS CLI

1. Install the AWS CLI version 2. For more information about installing the AWS CLI, see Installing the AWS CLI.
2. Configure the AWS CLI with your credentials. For more information about configuring the AWS CLI, see Configuring the AWS CLI to use IAM Identity Center.
3. Verify your configuration by running the following command:

```
aws sts get-caller-identity
```

   This command should return your account ID, user ID, and ARN.

`npm run sandbox` and `npm run deploy` deploy with the same credentials that the CDK uses. Before they deploy, they print the deployment target: the credentials (an AWS profile or access keys from the environment) and the Region. The Region comes from `cdk.json` (or `~/.cdk.json`), the `AWS_REGION` or `AWS_DEFAULT_REGION` environment variable, or your profile. Check this line before each deploy.

Bootstrap the AWS CDK

CDK Blocks uses the AWS CDK to deploy infrastructure. You must bootstrap your AWS account before your first deployment.

From your project directory, run the following command, replacing `ACCOUNT_ID` and `REGION` with your values:

```
npx cdk bootstrap aws://ACCOUNT_ID/REGION --app=
```

For example:

```
npx cdk bootstrap aws://123456789012/us-east-1 --app=
```

The empty `--app=` option stops the CDK from synthesizing your app to find environments, because the command already names the account and Region.

You need to bootstrap only one time per account and Region combination. `npm run sandbox` and `npm run deploy` check this before they synthesize. If the account and Region are not bootstrapped, they stop and print the command to run.

Some apps also deploy a support stack in `us-east-1`, so bootstrap `us-east-1` in the same account too. This happens when the stack that holds `Hosting` uses the CloudFront front door with monitoring on (the default) and sets an explicit account and an explicit Region other than `us-east-1`, or when the app has a Lambda@Edge function (for example a Next.js route with `runtime = 'edge'`). The check that runs before a deploy covers only your app's Region.

Deploy to a sandbox

A sandbox is your own temporary copy of the backend, for testing against real AWS services. To deploy your application to a sandbox, run the following command:

```
npm run sandbox
```

The sandbox:

- Deploys the backend with a full CloudFormation deployment. The first deploy can take several minutes.
- Then runs `cdk watch`. When you save a code change, it is hotswapped into the AWS Lambda function.
- Starts the local dev server at `http://localhost:3000`. It serves your frontend locally and sends API calls to the deployed backend. Templates without a frontend (`backend`, `api-only`, and `sql`) start the server on port 3001 instead.
- Uses real AWS services (DynamoDB, API Gateway, Lambda)

- Gives each developer an isolated environment. The stack is named `<stackId>-<sandboxId>`. AWS CDK Blocks keeps the sandbox ID in `.cdk-blocks-sandbox/sandbox-id.txt`, which you don't commit, so each machine gets its own stack.

`cdk watch` hotswaps only code. It skips an infrastructure change, such as a new CDK Block or a changed CDK Block option, and prints a warning about changes that can't be hotswapped. To deploy that change, stop the sandbox and run `npm run sandbox` again. Pressing **Ctrl+C** stops only the local processes. Your AWS resources keep running until you destroy them.

The same code that ran locally now runs on AWS. CDK Blocks automatically resolve to their AWS implementations. `KVStore` becomes a DynamoDB table, `Auth` provisions an Amazon Cognito user pool, and `ApiNamespace` creates an API Gateway endpoint backed by a Lambda function.

Other sandbox commands:

- `npm run sandbox:deploy` deploys the sandbox once and exits, with no `cdk watch` and no local dev server. Use it in scripts and CI.
- `npm run sandbox:preview` deploys the sandbox with your frontend hosted in preview shape: served through an API Gateway HTTP API, with no CloudFront distribution. It prints the preview URL and exits. Running it again hotswaps a change that touches only backend code. A plain `npm run sandbox` or `npm run sandbox:deploy` removes the hosted preview (it prints a warning first), and the next preview gets a new URL. Templates without a frontend do not have this script.
- `npm run sandbox:console` opens the CloudFormation console's stack list, filtered to the stack from your last deploy. It uses the Region from `AWS_REGION`, `AWS_DEFAULT_REGION`, or your AWS CLI configuration, and `us-east-1` if none is set.

Removing a sandbox

To remove sandbox resources, run the following command:

```
npm run sandbox:destroy
```

This deletes every stack the sandbox deployed, including any support stack in `us-east-1`. A support stack that holds a Lambda@Edge function cannot be deleted for a few hours after its CloudFront distribution is gone. In that case the command says so; run `npm run sandbox:destroy` again later.

Recover from a failed sandbox deploy

The sandbox usually deploys with CloudFormation Express Mode, which turns off automatic rollback. (When an existing sandbox stack has a Lambda Web Function endpoint, the sandbox prints `Express Mode off for this deploy` and deploys with rollback instead.) A failed deploy leaves the stack in a failed state, and the next `npm run sandbox` cannot update it. To recover, destroy the sandbox and deploy again:

```
npm run sandbox:destroy
npm run sandbox
```

If `sandbox:destroy` reports that the stack is in `UPDATE_ROLLBACK_FAILED`, run the following command first, using your sandbox stack name:

```
aws cloudformation continue-update-rollback --stack-name <your-sandbox-stack>
```

Full deployment

For a complete deployment including hosting, run the following command:

```
npm run deploy
```

This runs `cdk deploy --all`, a full CloudFormation deployment of every stack in your app. Most apps have one stack. The `us-east-1` support stack, if your app has one, deploys too. The deploy runs with `--require-approval never`, so the CDK does not stop to ask you to approve IAM or security group changes.

The production stack is named `<stackId>-prod`. The `stackId` comes from `.blocks/config.json`, which `npm create @aws-cdk-blocks/blocks-app` writes when it creates your project, for example `{ "stackId": "my-app-a1b2c3" }`. Commit this file. The command has no option for a stage or a stack name, so every `npm run deploy` to the same account and Region updates the same stack, whoever runs it. To deploy separate stages such as beta and prod, use a pipeline. For more information, see the section called “Staged deployments with a pipeline”.

When the deploy finishes, the command prints the API URL and the frontend URL. CloudFront can serve the previous frontend build for a short time after that. To wait until the new build is live, add `--wait-live`:

```
npm run deploy -- --wait-live
```

The command then waits up to 180 seconds for the new build. To change the limit, add `--wait-live-timeout=<seconds>` (1 to 3600). If the new build is not confirmed in time, the deploy has still succeeded, but the command exits with a nonzero code. `--wait-live` has no effect for an app without `Hosting`.

Deleting the production stack

To delete the production stack, run the following command:

```
npm run destroy
```

The production preset keeps stateful resources such as tables and buckets (`RETAIN`), so they stay in your account after the stack is deleted. Retained buckets can still lose their objects: if a delete attempt fails, the command empties every bucket in every stack of the app before it retries. An `Auth` user pool follows the preset too, unless you set `removalPolicy` on the CDK Block. Set `removalPolicy: 'retain'` on any user pool that holds real users, so a stack that uses another preset keeps it as well.

> **Warning**
>
> `npm run destroy` runs `cdk destroy` without `--all`. As a result, on an app with more than one top-level stack, such as an app with a `us-east-1` support stack for `Hosting`, the first attempt always fails. The command then empties the buckets in every stack before it retries, so you lose the objects in buckets the preset would keep. Even after the retries, the stacks are not deleted. For those apps, don't run `npm run destroy`. Run `cdk destroy` with the stack names yourself.

Running the CDK CLI yourself

The npm scripts run your CDK app with the `cdk` export condition. `cdk.json` runs `npx tsx -C cdk cdk-blocks/index.cdk.ts`, and the scripts set `NODE_OPTIONS=--conditions=cdk`. If you run `cdk synth` or `cdk deploy` another way, or load your backend from an existing CDK app, set the condition yourself:

```
NODE_OPTIONS="--conditions=cdk" npx cdk synth
```

Without it, Node.js resolves each CDK Block to its local implementation instead of its CDK Construct, and `BlocksStack.create()` stops with a `Missing --conditions=cdk` error.

Staged deployments with a pipeline

To deploy to more than one stage, define a pipeline with the `Pipeline` construct from `aws-cdk-blocks/cdk`. It is built on `@aws-cdk-blocks/pipeline`, which you can also use on its own without the CDK Blocks secret and configuration defaults.

`Pipeline` creates one self-mutating AWS CodePipeline for each branch. Each pipeline pulls your code from your source repository through an AWS CodeConnections connection and runs `cdk synth`. It updates itself when its own definition changes, then deploys each stage in order. A stage can target its own account and Region (`env`), wait for a manual approval (`requireApproval`), and wait for a bake time after it deploys (`bakeTime`). To deploy a stage to a different account than the pipeline, also set `crossAccountKeys: true`.

Create `cdk-blocks/pipeline.cdk.ts`:

```
import * as cdk from 'aws-cdk-lib';
import { Pipeline } from 'aws-cdk-blocks/cdk';

const app = new cdk.App();

await Pipeline.create(app, 'MyAppPipeline', {
  appFile: './index.cdk.ts',
  source: {
    repo: 'my-org/my-app',
    connectionArn: 'arn:aws:codeconnections:us-east-1:111122223333:connection/EXAMPLE',
  },
  synth: {
    commands: ['npm ci', 'npx cdk synth --app "npx tsx -C cdk cdk-blocks/
pipeline.cdk.ts"'],
  },
  branches: [
    {
      branch: 'main',
      stages: [
        { name: 'beta' },
```

```
        { name: 'prod', requireApproval: true },
      ],
    },
  ],
});
```

`appFile` points at your app's `index.cdk.ts`, relative to the pipeline file. The pipeline imports it once for each stage, and `BlocksStack.create()` adds the app's stack to that stage. Each stage gets its own stack, named after the pipeline, branch, and stage, for example `MyAppPipeline-main-Stage-beta-<stackId>-prod`. The `synth` commands point `cdk synth` at the pipeline file, because `cdk.json` still points at `index.cdk.ts` for `npm run deploy`.

Before the first run:

1. Create a connection to your source repository provider in the AWS CodeConnections console and finish authorizing it. A new connection stays in `PENDING` status until you do.
2. Commit and push the pipeline file, `package-lock.json`, and `.blocks/config.json`.
3. Deploy the pipeline one time from your machine:

   ```
   npx cdk deploy --app "npx tsx -C cdk cdk-blocks/pipeline.cdk.ts"
   ```

After that, each push to `main` runs the pipeline. For more options, such as steps that run after a deploy and build secrets, see pipeline package source.

Next steps

- To configure custom domains, VPC settings, or other environment-specific infrastructure, see The CDK layer in the Concepts topic.
- To integrate CDK Blocks into an existing CDK application, see Integrating with existing infrastructure.

CDK Blocks reference

AWS CDK Blocks ships with built-in CDK Blocks covering authentication, data storage, realtime messaging, AI, observability, and more. Each CDK Block provides a typed API, automatic resource provisioning, and a local implementation for offline development.

Import CDK Blocks from `aws-cdk-blocks`. Each entry below also names the package that contains the CDK Block, which has its own README.

For more information about source code and advanced configuration, see CDK Blocks on GitHub.

Authentication

A CDK Block for user identity and session management.

- **Auth** (`@aws-cdk-blocks/bb-auth`): Email and password sign-in backed by an Amazon Cognito user pool, with multi-factor authentication (MFA), groups, and passkeys. Social, SAML, and OIDC providers go in one `providers` record. With `store: 'builtin'`, your backend verifies OIDC and OAuth 2.0 sign-ins without a user pool, and `stubIdp()` gives you an offline identity provider for development. `Auth` replaces `AuthBasic`, `AuthCognito`, and `AuthOIDC`.

For more information, see Authentication.

Data storage

CDK Blocks for persisting and retrieving data.

- **KVStore** (`@aws-cdk-blocks/bb-kv-store`): Key-value and document storage backed by DynamoDB, with schema validation, composite keys, and secondary indexes. Use for caches, session stores, and app records that you list or query. For feature flags, use AppSetting.
- **Database** (`@aws-cdk-blocks/bb-database`): SQL on PostgreSQL with migrations, transactions, and a Kysely dialect. It runs on Amazon Aurora DSQL by default, or on Aurora PostgreSQL Serverless v2 for Row Level Security and the rest of PostgreSQL. Use for relational data.
- **FileBucket** (`@aws-cdk-blocks/bb-file-bucket`): File storage for uploads and downloads with presigned URLs. Use for user-generated content.

For more information, see Data storage.

Realtime and async

CDK Blocks for realtime communication and background processing.

- **Realtime** (`@aws-cdk-blocks/bb-realtime`): Typed WebSocket pub/sub channels. Use for live updates, chat, and notifications.
- **AsyncJob** (`@aws-cdk-blocks/bb-async-job`): Background work via Amazon SQS, with retries and a dead-letter queue. Jobs are fire-and-forget by default. Set `trackStatus: true` to read each job's status with `getStatus()` or `waitUntilComplete()`. Use for tasks that don't need immediate results.
- **CronJob** (`@aws-cdk-blocks/bb-cron-job`): Scheduled task execution via an Amazon EventBridge Scheduler schedule. Use for periodic maintenance and batch processing.
- **Compute** (`@aws-cdk-blocks/bb-compute`): Choose where a workload runs. A `serverless` compute runs on AWS Lambda, and a `container` compute runs on AWS Fargate. Pass it as the `compute` option of an `ApiNamespace`, `AsyncJob`, `CronJob`, or `Scope`, each of which runs on either compute type. `Realtime` and `RawRoute` always stay on the app's default compute, including inside a container `Scope`. Workloads without one use the app's default serverless compute.

For more information, see Realtime and async.

AI

CDK Blocks for AI capabilities.

- **Agent** (`@aws-cdk-blocks/bb-agent`): AI agent with streaming, tool calling, human-in-the-loop (HITL) approval, and conversation persistence. Locally, uses a canned keyword-based provider (no real model needed). On AWS, the agent loop runs on an Amazon Bedrock AgentCore Runtime, which is billed for vCPU and memory while a session runs. It streams to clients over a `Realtime` WebSocket. The default model is `BedrockModels.BALANCED`, called through a global inference profile.
- **KnowledgeBase** (`@aws-cdk-blocks/bb-knowledge-base`): Semantic document retrieval via Amazon Bedrock Knowledge Bases. Use for RAG and contextual search.

For more information, see AI.

Communication

CDK Blocks for sending messages to users.

- **EmailClient** (`@aws-cdk-blocks/bb-email-client`): Transactional email sending via Amazon SES. Locally, captures emails for testing.

For more information, see Communication.

Configuration

CDK Blocks for application settings and secrets.

- **AppSetting** (`@aws-cdk-blocks/bb-app-setting`): A single configuration value that is not sensitive, backed by SSM Parameter Store. Use for feature flags and runtime config for the whole app. Don't use it for API keys or other production secrets.
- **Secret** (`@aws-cdk-blocks/bb-secret`): A single application secret backed by AWS Secrets Manager, such as an API key or a connection string. It keeps the previous version and can wrap an existing secret with `Secret.fromExisting(arn)`. A `Secret` is read-only at runtime by default; set `allowRuntimeWrite: true` to add `put()`.

For more information, see Configuration.

Observability

CDK Blocks for monitoring, logging, and tracing.

- **Metrics** (`@aws-cdk-blocks/bb-metrics`): Custom application metrics via CloudWatch Embedded Metric Format.
- **Logger** (`@aws-cdk-blocks/bb-logger`): Structured JSON logging with levels and contextual metadata.
- **Tracer** (`@aws-cdk-blocks/bb-tracer`): Distributed tracing via AWS X-Ray.
- **Dashboard** (`@aws-cdk-blocks/bb-dashboard`): A CloudWatch dashboard organized by compute. Each compute gets health and logs sections, plus traces when the app has a Tracer.

Metric widgets appear for the metrics you list in `metricConfigs`. It works without Metrics or Logger.

For more information, see Observability.

Hosting

Frontend deployment (CDK layer only, import from `aws-cdk-blocks/cdk`).

- **Hosting** (`aws-cdk-blocks/cdk`, built on `@aws-cdk-blocks/hosting`): Frontend deployment with SSR support. Auto-detects framework (Next.js, Nuxt, Astro, SPA).

For more information, see Hosting.

Authentication

This section covers the `Auth` Block, which handles user identity and sessions. One `Auth` Block covers email and password sign-in, social sign-in, OpenID Connect (OIDC), and Security Assertion Markup Language (SAML) 2.0, and you configure it with one options object.

Choosing a sign-in method

Each `Auth` Block first chooses where its users live, with the `store` option. Every sign-in provider then goes in one `providers` record, keyed by the provider ID that the frontend uses, and tagged with its `type`.

Sign-in method	Configure with	Best for	Works under `npm run dev`
Email and password	On by default (`emailPassword`), default store	Your own user directory, with MFA, passkeys, and groups	Yes
Social sign-in	`providers` entry with `type: 'google'`, `'apple'`, `'facebook'`, or `'amazon'`, default store	Consumer sign-in through Amazon Cognito	No, deploy a sandbox to test it

Sign-in method	Configure with	Best for	Works under `npm run dev`
OIDC, verified by your backend	`store: 'builtin'` with `type: 'oidc'` entries, or `github()`, `customOauth2()`, and `stubIdp()`	Any OIDC or OAuth 2.0 provider, with no user pool	Yes
OIDC through Cognito	`type: 'oidc'` entry with a `clientSecret`, default store	OIDC users who must be in your user pool	No, deploy a sandbox to test it
SAML	`type: 'saml'` entry, default store	An identity provider that supports only SAML 2.0	No, deploy a sandbox to test it

For a workforce identity provider, prefer a `store: 'builtin'` Block: it creates no Cognito resources and you can test it offline with `stubIdp()`.

Auth

The `createApi()` method returns the sign-in API that the `Authenticator` UI component and your own forms call. Export its result to expose that API, for example `export const authApi = auth.createApi();`. Exporting the CDK Block instance itself exposes nothing.

With `store: 'cognito'`, the default, users are records in an Amazon Cognito user pool. Locally, the CDK Block runs a user pool in `.bb-data/`, so sign-up, sign-in, password reset, MFA, and passkeys work without AWS calls, and verification codes go to the optional `codeDelivery` callback instead of email. On AWS, Cognito sends the codes. Its default email sender sends at most 50 emails a day, so before launch, set the `email` option to send through your own Amazon Simple Email Service (Amazon SES) identity.

With `store: 'builtin'`, there is no user pool. Your backend verifies each sign-in and the CDK Block keeps only its sessions. A CDK Block holds one kind of user, so an app with both kinds uses two `Auth` Blocks.

```
import { Auth, Scope, Secret } from 'aws-cdk-blocks';
```

```
const scope = new Scope('my-app');
// `generate` gives the first deploy a placeholder. Set the real value after deploy.
const googleSecret = new Secret(scope, 'google-client-secret', { generate: true });

// The default store: email and password, plus a social provider through Cognito.
const auth = new Auth(scope, 'auth', {
  users: { signInWith: ['email'], caseSensitive: false, groups: ['admin'] },
  providers: {
    google: { type: 'google', clientId: 'your-client-id', clientSecret: googleSecret },
  },
  removalPolicy: 'retain',
});
export const authApi = auth.createApi();

// No user pool: your backend verifies the sign-in.
const sso = new Auth(scope, 'sso', {
  store: 'builtin',
  providers: {
    corp: { type: 'oidc', issuer: 'https://idp.example.com', clientId: 'your-client-
id' },
  },
  redirects: { callbackPath: '/aws-blocks/auth/sso/callback', signOutPath: '/aws-
blocks/auth/sso/signout' },
});
export const ssoApi = sso.createApi();
```

Two AWS CDK Blocks that both have `providers` need different `redirects` paths, because they would otherwise mount the same callback route. A provider secret is a reference, never a string. Use a `Secret`, and set its value as described in Setting a secret value.

The compiler checks the combinations. For example, `mfa` and a `type: 'google'` entry are compile errors on a `store: 'builtin'` Block. A method that the options turn off, such as `signIn` with `emailPassword: false`, is a compile error too.

Cognito fixes `users.signInWith` and `users.caseSensitive` when the user pool is created, so choose them before real users sign up.

Protecting API methods

Every `ApiNamespace` method is public until it awaits a guard. `await auth.requireAuth(context)` returns the signed-in user, or responds with a 401

`NotAuthenticatedException` error. `await auth.requireRole(context, 'admin')` also responds with a 403 `NotAuthorizedException` error when the user is not in the group. Declare the groups in `users.groups`. On the default store, `requireRole` reads groups from the user pool on each request. On a `store: 'builtin'` Block, it reads them from the provider's `groupsClaim`.

To make a forgotten guard impossible, define the namespace with `auth.protect`. Its `methods` run only after the session check, with the signed-in user, and its optional `public` methods need no session. The following example uses the `auth` Block from the earlier example and a `notes` store:

```
const notes = new KVStore(scope, 'notes');

export const notesApi = auth.protect(scope, 'notesApi', {
  public: () => ({
    async health() {
      return { ok: true };
    },
  }),
  methods: (context, user) => ({
    async addNote(text: string) {
      await notes.put(`${user.userSub}:${crypto.randomUUID()}`, text);
    },
  }),
}, { role: 'admin' });
```

Key each user's data on `user.userSub`, which is stable for the user's lifetime. On the default store, `userSub` is the Cognito `sub` and `userId` is the Cognito username. On a `store: 'builtin'` Block, both are the provider's issuer and subject.

To manage users from your own methods, opt in with `admin: {}`. The `auth.admin` handle is never served as an API method, so every method that calls it must check its caller first. To add the first admin, run `npx @aws-cdk-blocks/bb-auth users add-to-group <user> admin`, with `--stage prod --yes` for a deployed user pool. To keep end users from writing an attribute that you use for authorization, such as a role, declare it in `users.attributes` with `userWritable: false`.

Local sign-in with stubIdp

`stubIdp()` runs a small OIDC provider inside your app. It needs no credentials, works offline, and shows a page where you pick a test user. It works only under `npm run dev` and in sandbox

deployments, and synth fails for any other deployment that uses it. To keep the stub locally and a real provider in production, choose the entry with `getDeploymentType()`, which returns `'local'`, `'sandbox'`, or `'production'`. The following example uses the stub everywhere except production:

```
import { Auth, Scope, getDeploymentType, stubIdp } from 'aws-cdk-blocks';

const scope = new Scope('my-app');

const auth = new Auth(scope, 'auth', {
  store: 'builtin',
  providers: {
    corp:
      getDeploymentType() === 'production'
        ? { type: 'oidc', issuer: 'https://login.example.com', clientId: 'your-client-
id' }
        : stubIdp({ users: [{ sub: 'u-1', email: 'jane_doe@example.com', name: 'Jane
 Doe' }] }),
  },
});
export const authApi = auth.createApi();
```

Providers on the default store sign in through Cognito managed login, which needs a deployed user pool, so test them in a sandbox.

Native apps and other clients

Sessions are HTTP-only cookies. With `allowBearerAuth: true`, native apps and command line clients can send `Authorization: Bearer <access token>` instead: a Cognito access token on the default store, or the provider's access token on a `store: 'builtin'` Block. To accept tokens from other app clients of the same user pool, pass `allowBearerAuth: { clientIds }`. For native apps that sign in with a federated provider, also export `auth.createOidcApi()`.

Keep the Block ID

The CDK Block ID, `'auth'` in `new Auth(scope, 'auth')`, is part of every resource name. If you change it or move the CDK Block under another scope, CloudFormation creates a new user pool and deletes the old one with every user in it. At synth, the CDK Block refuses user pool changes that Cognito can't apply and changes that would remove the user pool, such as renaming the CDK Block. It compares against a baseline file in `cdk-blocks/baselines/`, so commit that file. The

user pool and session table follow the stack's `removalPolicy` default unless you set the option. Set `removalPolicy: 'retain'` for any user pool that holds real users.

Migrating from AuthBasic, AuthCognito, or AuthOIDC

`Auth` replaces the `AuthBasic`, `AuthCognito`, and `AuthOIDC` Blocks, which are removed. A codemod does the mechanical part of the migration. It never changes a CDK Block ID, and it leaves a `TODO(aws-blocks-auth-migrate)` comment wherever you need to decide something. Run these shell commands from your app's root directory:

```
npx @aws-cdk-blocks/bb-auth migrate --dry-run   # print the changes, write nothing
npx @aws-cdk-blocks/bb-auth migrate             # rewrite the files
```

An `AuthCognito` app that keeps the same CDK Block ID keeps its user pool and users, but each user signs in once more. `AuthBasic` users can't be moved and must sign up again. For more information about each CDK Block's steps and what happens on the first deploy, see the Auth migration guide on GitHub.

For more information, see bb-auth on GitHub.

Data storage

This section covers AWS CDK Blocks for persisting and retrieving data.

Choosing a data Block

Block	Best for	Avoid when
`KVStore`	Caches and session stores, and app records that you read by key, list in order inside a group, or read by another field through an index	You need JOINs, transactions, or complex SQL
`Database`	Relational data, complex queries, JOINs, transactions, migrations, with no virtual private cloud (VPC) to manage. Put it on an `aurora` cluster	Your workload is mostly key-value lookups, which `KVStore` serves with no cluster to manage

Block	Best for	Avoid when
	for triggers, extensions, Row Level Security, or large transactions	
`FileBucket`	File uploads, user-generated content, large binary objects	You need structured queries over file metadata (pair with `KVStore` for metadata)

Feature flags and runtime config for the whole app go in `AppSetting`, not in a data CDK Block. `KVStore` is for data your app writes. For more information, see Configuration.

KVStore

Key-value and document storage backed by DynamoDB. Pick one of three forms by how you read the data:

- No options: a string key and a string value (or a type argument), for caches, sessions, and other data where you always know the key.
- A `schema` and `indexes`: typed values on a string key, for data that you also read by another field.
- A `schema` and a `key` (`{ partitionKey, sortKey? }`): typed records on an object key, for data that you list or query as a collection, such as the todos of one user. The key is an object with those fields, each a string or a number.

A store fixes its key when you create it, so declare a `key` for any records that you list. The following example declares a key and lists the todos of one user:

```
const todos = new KVStore(scope, 'todos', {
  schema: z.object({ userId: z.string(), todoId: z.string(), title: z.string(), done:
 z.boolean() }),
  key: { partitionKey: 'userId', sortKey: 'todoId' },
});

await todos.put({ userId: 'alice', todoId: '001' }, { title: 'Write the docs', done:
 false });
for await (const { value } of todos.query({ where: { userId: { equals: 'alice' } },
 limit: 20 })) {
```

```
  console.log(value.title);
}
```

Every call takes the key first, as in `put(key, value, options)`. `query()` reads one partition in sort order, by the key or by an index. `query()` and `scan()` return `{ key, value, expiresAt }` entries and a `cursor` for the next page. KVStore also has batch calls and conditional writes, and no transaction API.

`get()` is strongly consistent by default. Pass `{ consistentRead: false }` to use half the read capacity where a stale value is acceptable. On AWS, `query()`, `scan()`, `getBatch()`, and index reads are eventually consistent.

Locally, KVStore persists data in the `.bb-data/` directory. The local implementation is always consistent and does not throttle requests, so test those behaviors in a sandbox. The local implementation applies DynamoDB's limits as AWS does: the 400 KB item size, measured as DynamoDB measures it, the size limits on key fields, at most 32 levels of nested lists and maps, and no repeated key within one batch request. On AWS, KVStore checks the key, nesting, and batch limits before it calls DynamoDB. Each of these limits throws a `KVStoreErrors` error in both places, so match those names rather than DynamoDB's `ValidationException`. A value that DynamoDB can't store, such as a `Date` or `NaN`, fails locally as it does on AWS. On AWS, it provisions a DynamoDB table with a global secondary index for each entry in `indexes`. To wrap an existing table, pass `table: KVStore.fromExisting(tableName)` and a `key` that matches the table.

KVStore has these durability options:

- Expiry: write an item with `put(key, value, { ttlSeconds })` or `{ expiresAt }`. Every read hides an expired item, and DynamoDB deletes it later. `ttl` names a number field of the value to hold the expiry.
- `pointInTimeRecovery`: continuous backups. On by default with the production preset and off with the sandbox preset.
- `encryption`: the AWS managed key (the default), a dedicated customer managed key, or a key you already own.
- `protection`: `'disposable'`, `'retained'`, or `'locked'`. By default, the production preset keeps and protects tables, and the sandbox preset deletes them with the stack.

Upgrading from DistributedTable or an earlier KVStore

`DistributedTable` is now part of `KVStore`. To keep a `DistributedTable`'s deployed table and items, declare it as `new KVStore(scope, id, { schema, key, indexes, ttl })` with the same ID. Then pass the key first to `put()`, read `{ key, value }` entries from `query()`, and use `KVStoreErrors`.
A `KVStore` deployed with `@aws-cdk-blocks/bb-kv-store` 0.4.0 or earlier stores items in another format, so the deploy fails if a new `KVStore` takes over its table.
To upgrade, change the class of that store to `LegacyKVStore` and keep its ID and options. Add a `KVStore` with a new ID, copy the data to it, and then remove the `LegacyKVStore`. In a sandbox, you can destroy and redeploy the stack instead. In the options of the new store, don't use `ttl: true`, and use `protection` instead of `removalPolicy` and `deletionProtection`. Replace `ifValueEquals` on an object value with `ifFieldEquals`. For more information about the upgrade, see the KVStore upgrade steps on GitHub.

For more information, see bb-kv-store on GitHub.

Database

SQL on PostgreSQL, with migrations, transactions, foreign keys, and joins. Each `Database` is a PostgreSQL schema on a cluster. With no options, the CDK Block owns an Amazon Aurora DSQL cluster, which scales to zero and needs no VPC. Write parameterized queries with the `sql` tagged template, which turns every `${value}` into a bound parameter. The query code is the same on every cluster.

```
// Owns an Aurora DSQL cluster. Its migrations are in cdk-blocks/migrations/shop/.
const shop = new Database(scope, 'shop');

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async addToOrder(orderId: number, amount: number) {
    const { userId } = await auth.requireAuth(context);
    await shop.transaction(async (tx) => {
      const order = await tx.queryOne<{ total: number }>(
        sql`SELECT total FROM orders WHERE id = ${orderId} AND customer_id = ${userId}
 FOR UPDATE`,
      );
      if (!order) throw new ApiError('Order not found', 404);
```

```
      await tx.execute(sql`UPDATE orders SET total = ${order.total + amount} WHERE id =
 ${orderId}`);
    }, { retryOnConflict: true, signal: context.request.signal });
  },
}));
```

⚠ Upgrading from the earlier Database and DistributedDatabase

`Database` replaces the earlier `Database` and `DistributedDatabase` Blocks.
A `DistributedDatabase` keeps its Aurora DSQL cluster and its rows. Declare it as a `Database` with the same ID, in the same scope and stack, and CloudFormation treats it as the same cluster. `Database` reads the `_migrations` table that `DistributedDatabase` wrote, so keep the migration files and their names. No applied file runs a second time.
Before you deploy, run `cdk diff`. Check that the cluster (`AWS::DSQL::Cluster`) shows as changed (`[~]`), not as removed and added (`[-]` then `[+]`). A removed cluster means the ID, the scope, or the stack name is not the one the `DistributedDatabase` had. Under the sandbox preset's `destroy` removal policy, the deploy then deletes that cluster with its rows.
An app that deployed the `Database` of `@aws-blocks/bb-data` keeps its Aurora cluster, its secret, and its VPC. Give a `LegacyDatabaseCluster` the ID and options of the old Block, then pass it as the `cluster` of a `Database` with a new ID. That class is deprecated and frozen, and `asUser()` is an error on it. The old package name keeps its original scope here because the package no longer exists.
`asUser()` and `asRole()` replace `withRLS()`, and `crud()` has no replacement. An app that used `fromExisting()` keeps its migration files and `_migrations` table. Pass `DatabaseCluster.fromExisting({ name: 'db-url' })` as `cluster`, point `migrationsPath` at the same directory, and no file runs again.
For more information, see Upgrading an older app on GitHub.

Clusters

To put several `Database` Blocks on one cluster, or to use full PostgreSQL, pass a `DatabaseCluster` as the `cluster` option. Each CDK Block on the cluster gets its own schema, named after its ID:

```
const main = new DatabaseCluster(scope, 'main', { type: 'aurora' });
```

```
const notes = new Database(scope, 'notes', { cluster: main });
```

The cluster `type` is one of these:

- `'dsql'`: Aurora DSQL, the engine that a `Database` without `cluster` owns. A transaction can change at most 3,000 rows and 10 MiB of data. Triggers, extensions, and Row Level Security are not available.
- `'aurora'`: Amazon Aurora PostgreSQL Serverless v2 with the express configuration. It has all of PostgreSQL, runs outside any VPC with IAM authentication, and scales to zero by default (`minCapacity: 0`). The first request after five idle minutes then waits about 20 seconds. Set `minCapacity: 0.5` to keep it warm, and `maxCapacity` (default 2) to raise the ceiling.

Schemas separate CDK Blocks by name, not by permission: every CDK Block on a cluster connects as one database role that can read and write every schema on it. Put a CDK Block whose data must stay separate on its own cluster.

The data stays on the cluster that a CDK Block first deploys to. If you rename a `Database` or `DatabaseCluster` ID, or change the `cluster` option, the CDK Block points at a new, empty cluster or schema, and nothing copies the rows. With the production preset, a cluster that leaves the stack is retained. With the sandbox preset, it is deleted in the same deploy.

Migrations

Create tables in migration files. Your app's own statements can only read and write rows. Each CDK Block runs the `.sql` files in `cdk-blocks/migrations/<id>/`, relative to the project root, in name order and once each. `migrationsPath` sets another directory. Locally, a CDK Block applies pending files before its first query. On AWS, they run during the deploy, before any function in the stack is updated.

```
-- cdk-blocks/migrations/shop/001_init.sql
CREATE TABLE orders (
  id          BIGINT GENERATED BY DEFAULT AS IDENTITY (CACHE 65536) PRIMARY KEY,
  customer_id TEXT NOT NULL,
  total       INTEGER NOT NULL
);

CREATE INDEX ASYNC orders_customer_id ON orders (customer_id);
```

On a `dsql` cluster, write the DSQL forms that the Aurora DSQL documentation shows, such as `CREATE INDEX ASYNC`. CDK Blocks checks every file with `dsql-lint` at `npm run dev` and at `cdk synth`, and the error names the file and the form to use instead. A deploy that stops in a file resumes it where it stopped, so don't edit a file that has partly run. Put the change in a new file. On an `aurora` cluster, write plain PostgreSQL. Each file runs in one transaction and, on AWS, must finish within ten minutes.

Queries and transactions

`query()` returns every row, `queryOne()` the first row or `null`, and `execute()` returns `{ rowCount }`. Pass a row type, as in `query<T>()`, or pass a Standard Schema such as a Zod object as `{ schema }` to validate every row. A `bigint` value, such as an identity column, comes back as a JavaScript number.

`transaction(fn)` commits when `fn` resolves and rolls back when it throws. An `ApiError` thrown in `fn` reaches the client unchanged. Aurora DSQL never waits on a lock: when two transactions change the same row, one fails at commit with `SerializationFailure`. With `retryOnConflict: true`, the CDK Block runs `fn` again after a random wait, up to `maxRetries` times (default 8), so keep side effects such as sending email out of `fn`. Pass `context.request.signal` as `signal` so that no retry commits after the request times out.

For the Kysely query builder, install `kysely` and pass `new DatabaseDialect(db)` from `@aws-cdk-blocks/bb-database/kysely` as its dialect. `0` writes a `Tables` interface from the CDK Block's migration files.

Row level access

On an `aurora` cluster, Row Level Security policies that you write in a migration can limit each user to their own rows. `db.asUser(user)` returns the CDK Block with each transaction run for that user: it first sets `request.jwt.claims` to `{ sub: user.userId }`, plus any claims you pass as the second argument. Pass the result of `requireAuth()` as `user`.

```
-- cdk-blocks/migrations/notes/001_notes.sql
CREATE TABLE notes (id SERIAL PRIMARY KEY, owner_id TEXT NOT NULL, body TEXT NOT NULL);
ALTER TABLE notes ENABLE ROW LEVEL SECURITY;
CREATE POLICY notes_owner ON notes
  USING (owner_id = current_setting('request.jwt.claims', true)::jsonb ->> 'sub');
```

With this policy, +notes.asUser(user).query(sql`SELECT id, body FROM notes)+` returns only that user's notes, and the same statement on the CDK Block itself returns none. For

policies written for a database role, such as `TO authenticated`, use `asRole(role, claims?)`. Both throw `DatabaseConfigurationError` on a `dsql` cluster.

Existing PostgreSQL databases

To use a PostgreSQL database that you already run, with its tables and data, pass `cluster: DatabaseCluster.fromExisting({ name: 'db-url', ca })` and `migrationsPath: './migrations'`. The CDK Block provisions nothing, and a Lambda function outside any VPC must be able to reach the database over IPv4 with TLS. The CDK Block always verifies the server certificate, against `ca` (PEM text) or the Node.js trust store.

Never write the connection string in code. `0` reads it from a hidden prompt, checks the database, and stores it for local mode. Add `--stage production` to store it for a stage as the SSM SecureString `/<stackName>-db-url`. You can pass your own `Secret` as `connectionString` instead of `name`.

Locally, the CDK Block connects to that database and runs no migration files, so use a development database. `0` applies the pending files in `./migrations` to it, and a deploy applies them to the stage, one transaction per file. `1` records a file, such as a dump of the existing schema, without running it.

Local mode

Locally, each cluster is one PGlite database (PostgreSQL compiled to WebAssembly) in `.bb-data/`, with one schema per CDK Block. A `dsql` cluster applies the DSQL checks and limits, so SQL that Aurora DSQL rejects fails on your machine. Statements run with the grants of the deployed app, and Row Level Security applies as on AWS. PGlite runs one statement or transaction at a time. To test the retry path, call `simulateConflict(db)` from `@aws-cdk-blocks/bb-database/testing`.

Database errors

Common request errors reach the client as named 4xx errors instead of a generic 500. Check for them with `isBlocksError()` on the server or the client, for example `isBlocksError(e, DatabaseErrors.UniqueConstraintViolation)`.

Error	Cause	HTTP status
`DatabaseErrors.PermissionDenied`	A Row Level Security policy rejected a statement from `asUser()` or `asRole()`	403
`DatabaseErrors.CheckViolation`	A value fails a `CHECK` constraint	400
`DatabaseErrors.NotNullViolation`	A required column is missing	400
`DatabaseErrors.UniqueConstraintViolation`	A duplicate key	409
`DatabaseErrors.ForeignKeyViolation`	The referenced row does not exist, or other rows still reference the row being deleted	409
`DatabaseErrors.SerializationFailure`	A conflict with a concurrent transaction. Safe to retry.	409

Other failures reach the client as status 500, with the name `QueryFailed` (any other SQL error, including DDL outside a migration), `ConnectionFailed`, `TransactionFailed`, `TransactionRowLimitExceeded`, or `ResultValidationFailed` (a row did not match `schema`).

For more information, see bb-database on GitHub.

FileBucket

File storage for uploads and downloads. Store files by path, generate presigned URLs for direct browser uploads and downloads, and list files by prefix. Versioning is on by default, so you can list and restore earlier versions of a file. Noncurrent versions expire after 90 days. Change this with `noncurrentVersionExpirationDays`.

Browser uploads and downloads that use presigned URLs need CORS rules after you deploy. A FileBucket has no CORS rules by default, and it does not use your stack's allowed origins. The local dev server accepts any origin, so a missing rule fails only after deploy. List your frontend origin in `corsRules`. A sandbox frontend on `http://localhost:3000` also calls the deployed bucket, so add that origin for sandboxes only, for example with `getDeploymentType()`:

```
const origins = ['https://app.example.com'];
if (getDeploymentType() === 'sandbox') origins.push('http://localhost:3000');

const uploads = new FileBucket(scope, 'uploads', {
  corsRules: [{
    allowedOrigins: origins,
    allowedMethods: ['GET', 'PUT'],
    allowedHeaders: ['Content-Type'],
  }],
});
```

A rule with the wildcard origin `'*'` cannot allow PUT, POST, or DELETE. CDK Blocks rejects it.

FileBucket also has these security options:

- `encryption`: S3 managed keys (the default), a dedicated customer managed AWS KMS key (`'customer-managed'`), or a key you already own (`FileBucket.fromKmsKey(keyArn)`).
- `objectLock`: write-once-read-many retention, for example `{ mode: 'governance', retentionDays: 365 }`. Decide when you create the bucket, because you cannot add Object Lock to a deployed bucket or turn it off. The bucket must be retained, and presigned uploads are not supported, so upload with `put()`.
- `accessLogging`: S3 server access logs in a separate log bucket. Off by default. To turn it on for every FileBucket, set `storageAccessLogging: true` in your stack's `defaults`. `defaults.accessLogging: true` also turns it on, unless `storageAccessLogging` is `false`.

Locally, FileBucket stores files in the `.bb-data/` directory on your filesystem, mirroring the S3 API behavior. On AWS, it provisions an S3 bucket. You can also wrap an existing S3 bucket with `FileBucket.fromExisting(bucketName)`. A wrapped bucket gets none of the FileBucket defaults, such as versioning, encryption, or blocked public access, and FileBucket ignores `corsRules` for it. Configure those on the bucket itself.

For more information, see bb-file-bucket on GitHub.

Realtime and async

This section covers AWS CDK Blocks for realtime communication and background processing.

Choosing a realtime or async Block

Block	Best for	Avoid when
`Realtime`	Live updates, chat, notifications, collaborative features via WebSocket	Your use case is request/response only (no persistent connections needed)
`AsyncJob`	Long-running background tasks, event-driven processing, queue-based workloads	The task completes in under a few seconds (handle it synchronously)
`CronJob`	Scheduled recurring tasks, periodic cleanup, report generation	You need immediate execution triggered by an event (use AsyncJob)

Realtime

Typed WebSocket pub/sub channels. Define each namespace with a schema for type-safe message broadcasting. Any Standard Schema library works, such as Zod. Publish messages from the server and subscribe from the client. Each namespace can have multiple channels for topic-based routing.

Locally, Realtime uses an in-process EventEmitter with a local WebSocket server. On AWS, it provisions one shared API Gateway WebSocket API for the app, with a DynamoDB table for connection management. Best for live updates, chat, notifications, and collaborative features.

For more information, see bb-realtime on GitHub.

AsyncJob

Background job processing. Submit work that runs asynchronously outside the request lifecycle. Define a handler function that processes each job payload. By default, jobs are fire-and-forget from the caller's perspective, with automatic retries on failure. Set `trackStatus: true` to record each job's state (`queued`, `processing`, `complete`, or `failed`) and read it with `getStatus()`

or `waitUntilComplete()`. A job is `failed` when the handler throws on its last attempt. Status tracking adds a DynamoDB table.

Locally, AsyncJob executes handlers in-process. On AWS, it provisions an SQS queue with a Lambda consumer, or a container consumer when you assign it to a container `Compute`. Each job gets 3 delivery attempts (`maxRetries`) before it moves to a dead-letter queue that keeps messages for 14 days. The consumer receives up to 10 jobs per batch (`batchSize`) and waits up to 5 seconds to fill a batch (`maxBatchingWindowSeconds`). The queue's visibility timeout is the 15-minute handler timeout plus the batching window, so 905 seconds by default. Best for tasks that don't need an immediate response: sending emails, processing uploads, generating reports.

For more information, see bb-async-job on GitHub.

CronJob

Scheduled task execution. Define a handler function and a cron expression or rate schedule. The handler runs automatically at the specified interval. Cron expressions use UTC unless you set `timezone` to an IANA time zone name, such as `America/Los_Angeles`. Set `enabled: false` to create the schedule turned off. Use `input` to pass the same typed payload to the handler on every run.

Locally, CronJob executes on schedule using Node.js timers. On AWS, it provisions an Amazon EventBridge Scheduler schedule that invokes a Lambda function. Best for periodic maintenance, batch processing, cleanup jobs, and report generation.

For more information, see bb-cron-job on GitHub.

AI

This section covers AWS CDK Blocks for AI capabilities: conversational agents and semantic search.

Choosing an AI Block

Block	Best for	Avoid when
`Agent`	Conversational AI with tool calling, streaming, and human-in-the-loop approval	You only need document search without conversation (use KnowledgeBase)

Block	Best for	Avoid when
`KnowledgeBase`	Semantic search over documents, RAG pipelines, context retrieval	You need multi-turn conversation or tool calling (use Agent, optionally with KnowledgeBase as a tool)

Agent

AI agent with streaming responses, tool calling, human-in-the-loop approval, and conversation persistence. Powered by the Strands Agents SDK. Define a system prompt and a set of tools, then stream messages to the agent. The agent can call your tools, ask for human approval before executing sensitive actions, and persist conversations across sessions.

Locally, Agent uses a canned keyword-based provider that returns predictable responses without calling any real model. No API keys or cloud costs needed. You can alternatively configure an `openai-api` provider pointing to Ollama or any OpenAI-compatible endpoint for testing with real models. On AWS, the agent loop runs on Amazon Bedrock AgentCore Runtime and streams responses to the browser over a `Realtime` WebSocket channel. The default model is `BedrockModels.BALANCED` (model ID `global.anthropic.claude-sonnet-4-6`), called through a global cross-Region inference profile. If your data must stay in specific Regions, set an inference profile limited to those Regions instead. You pay for model tokens in Amazon Bedrock, and for AgentCore Runtime vCPU and memory while a session is active.

Before you can call the default model, your account must submit use case details once in the Amazon Bedrock console. The model is also activated once per account through AWS Marketplace. To activate it, invoke it once, for example from the Amazon Bedrock console playground. Use a principal with the `aws-marketplace:Subscribe`, `aws-marketplace:Unsubscribe`, and `aws-marketplace:ViewSubscriptions` permissions. Until then, or when no configured model is available, the agent reports `AgentErrors.ModelUnavailable`. To filter content, attach an existing Amazon Bedrock guardrail with the `guardrail` model option. Set `untrustedOutput: true` on any tool that returns content you don't control, such as retrieved documents. The agent then marks that content as untrusted data and tells the model not to follow instructions in it.

For more information, see bb-agent on GitHub.

KnowledgeBase

Semantic document retrieval. Point it at a folder of documents and query with natural language to get the most relevant chunks ranked by relevance. Use it to build RAG (Retrieval-Augmented Generation) pipelines or contextual search features.

Locally, KnowledgeBase builds a text index over your documents in the `.bb-data/` directory, so retrieval works offline. On AWS, it provisions an Amazon Bedrock knowledge base with automatic document ingestion, chunking, and embedding. Vectors are stored in Amazon S3 Vectors. Documents are embedded with Amazon Titan Text Embeddings V2 (1024 dimensions by default) and split with semantic chunking unless you choose another `chunking` strategy. On AWS, the source can also be an `s3://` URI. Local development accepts only a local folder.

For more information, see bb-knowledge-base on GitHub.

Communication

This section covers AWS CDK Blocks for sending messages to users.

EmailClient

Transactional email sending. Configure a from address and send emails with text or HTML bodies. Use `sendBatch()` to send many emails at once. It takes up to 100 messages per call by default (`maxBatchSize`) and sends them to SES in groups of 50. Each message can have at most 50 recipients (To, CC, and BCC combined). Both `send()` and `sendBatch()` check every address, the recipient count, and the message size (40 MB) before calling SES.

Locally, EmailClient writes emails to an outbox file (`.bb-data/<id>/emails.json`, where `<id>` is the CDK Block's full ID) instead of sending them, so you can inspect them during development. On AWS, it sends via Amazon SES. Best for signup confirmations, password resets, notifications, and any transactional communication.

Before you deploy, verify the sender's domain or address as an identity in Amazon SES. A new account starts in the SES sandbox, which sends only to verified email addresses and domains and limits sending (200 messages per 24 hours, 1 per second) until you request production access. Sandbox status is per AWS Region. EmailClient grants your app's execution role only `ses:SendEmail` and `ses:SendBulkEmail`.

For more information, see bb-email-client on GitHub.

Configuration

This section covers AWS CDK Blocks for application settings and secrets.

AppSetting

A single configuration value that is not sensitive, such as a feature flag, an API URL, or a threshold. Read and update values at runtime. Values are typed and can be read in your API handlers without environment variable boilerplate.

Feature flags and runtime config for the whole app go in `AppSetting`. `KVStore` is for data your app writes. Production secrets, such as API keys, go in Secret.

Locally, AppSetting persists values to the filesystem in the `.bb-data/` directory. On AWS, it provisions an SSM Parameter Store parameter. Best for feature flags and any configuration that might change without a redeploy.

You can mark a setting as `secret: true` to store it as an SSM SecureString. This is a fallback for a secret of little value in a prototype, not for production secrets. A `secret: true` setting cannot take a `value` in code. CDK Blocks seeds it with a random string on first deploy, and you set the real value later.

For more information, see bb-app-setting on GitHub.

Secret

A single secret stored in AWS Secrets Manager. Use `Secret` for production credentials such as third-party API keys, OAuth client secrets, connection strings, and signing keys. This includes the client secret of an `Auth` provider. Each `Secret` is one Secrets Manager secret. The declaration holds only the name, never the value.

```
// Read-only (the default). Set the value after deploy.
const paymentsKey = new Secret(scope, 'payments-api-key');

// Random value created on first deploy, for keys nobody needs to read.
const signingKey = new Secret(scope, 'session-signing-key', { generate: true });

// The app writes this value itself, so it opts in to put().
const partnerToken = new Secret(scope, 'partner-token', { allowRuntimeWrite: true });
```

```
// A secret created and managed outside this app.
const sharedKey = new Secret(scope, 'shared-key', {
  secret: Secret.fromExisting('arn:aws:secretsmanager:us-
east-1:111122223333:secret:shared-key-AbCdEf'),
});

export const billing = new ApiNamespace(scope, 'billing', () => ({
  async charge(amount: number) {
    const key = await paymentsKey.get();
    if (key === null) throw new ApiError('Payments are not configured', 503);
    // Call the payment API with key and amount.
  },
}));
```

`get()` returns `null` when no value is set. A new secret starts empty unless you pass `generate: true`. Don't combine `generate` with `schema`, because a random string is not valid JSON.

Runtime access

A `Secret` is read-only at runtime by default. The object has only `get()` and `listVersions()`, and the app's execution role can only read the secret. Pass `allowRuntimeWrite: true` to add `put()` and grant write access. This also applies to a secret wrapped with `fromExisting()`.

Secret options

`Secret` also takes these options:

- `name`: a fixed Secrets Manager name, so a script or CI pipeline can set the value with the AWS CLI. By default, CDK Blocks derives a name that is unique in your stack.
- `schema`: a schema from any Standard Schema library, such as Zod, for a typed JSON secret. Values are checked on `get()` and `put()`.
- `cacheTtlSeconds`: how long the runtime reuses a fetched value before it reads again. The default is 300 seconds, so a value rotated outside the app is picked up within 5 minutes without a redeploy. Set `0` to turn off caching. A `put()` from the app clears the cache.
- `secret`: pass `Secret.fromExisting(arn)` to use a secret that already exists, such as one shared by many apps. CDK Blocks does not create, seed, tag, or delete it. It only reads it, writes it if you allow, and grants access.

Secret versions

Each change keeps the old value as the previous version. Locally, only changes that you make with `put()` do. A hand edit of `.bb-data/settings.json` does not. Read it with `get({ version: 'previous' })`, for example to accept tokens signed with the old key during a rotation. `listVersions()` returns the current and previous versions (metadata only), and you can pass one back to `get({ version })`.

Setting a secret value

To set a value after deploy, open the `/aws-blocks/settings` route of your API. It redirects to your stack's settings resource group in the AWS Resource Groups console, where `Secret` values appear next to `AppSetting` parameters. You can also give the secret a fixed `name` and set it with `aws secretsmanager put-secret-value`. Scaffolded apps include an `npm run secret` script, but it manages a separate set of secrets under the app's `blocks/<stackId>/secrets/` prefix. It does not set `Secret` values.

Secrets in local development

Locally, `Secret` stores the current value as plain text in `.bb-data/settings.json`, the same file `AppSetting` uses, and the previous value in `.bb-data/secret-versions.json`. The local `/aws-blocks/settings` route opens that file in your editor. `generate: true` seeds a random value locally too, and `cacheTtlSeconds` has no effect. Never use real credentials locally. `Secret` runs only on the server. `aws-cdk-blocks` does not export it to browser code, so browser code that imports it from there fails to build or to load. The browser build of `@aws-cdk-blocks/bb-secret` throws `SecretErrors.NotSupported` when you construct a `Secret`.

For more information, see bb-secret package source.

Observability

This section covers AWS CDK Blocks for monitoring, logging, and tracing.

Choosing an observability Block

Block	Best for	Avoid when
`Metrics`	Custom application metrics with dimensions for filtering and aggregation	You only need request-level tracing (use Tracer)
`Logger`	Structured JSON logging with log levels and request context	You need numeric aggregation or alerting (use Metrics)
`Tracer`	Request tracing in your backend code, latency analysis	You only need simple log output (use Logger)
`Dashboard`	A prebuilt view of compute health, logs, traces, and the metrics you choose	You need custom charts or interacti ve data exploration (use the CloudWatch console)

Metrics

Custom application metrics. Emit named metric data points with dimensions for filtering and aggregation. Supports child emitters with merged dimensions for scoped metric contexts. Uses CloudWatch Embedded Metric Format to emit metrics.

Locally, Metrics writes to the console. On AWS, metrics flow to Amazon CloudWatch. Best for tracking business KPIs, API latencies, error rates, and custom application counters.

For more information, see bb-metrics on GitHub.

Logger

Structured JSON logging with levels and contextual metadata. Supports debug, info, warn, and error levels. Attach arbitrary metadata to log entries for filtering and analysis.

Locally, Logger writes to the console. On AWS, structured JSON flows to CloudWatch Logs. Each entry includes the X-Ray trace ID (`traceId`) when Lambda provides one. To tag entries with a request ID or other context, create a child logger with `log.child({ requestId })`. Best for application logging with searchable structured data.

For more information, see bb-logger on GitHub.

Tracer

Distributed request tracing. Wrap operations in traced segments to visualize request flow across your application. Add searchable annotations for debugging and add metadata for detailed inspection.

Locally, Tracer records traces to files in the `.bb-data/` directory for inspection. On AWS, it records segments in AWS X-Ray. Adding a `Tracer` turns on X-Ray active tracing for every compute Lambda function in the app, unless you set `enabled: false`. X-Ray bills per trace. Helper functions, such as database migrations, are not traced. Only Lambda functions are traced. API Gateway stage tracing is not turned on. Best for debugging latency issues and understanding request paths in production.

For more information, see bb-tracer on GitHub.

Dashboard

Auto-generated CloudWatch dashboard. It works without any other observability CDK Block: each compute in your app gets a health section and a logs section. A traces section appears when the app has an enabled `Tracer`. To chart your own metrics, pass a `Metrics` Block and list the metric names in `metricConfigs`. Dashboard does not create alarms.

Locally, Dashboard is not available; its route responds with a message directing you to deploy. On AWS, it provisions a CloudWatch Dashboard with those sections, and its route (`/aws-blocks/dashboard` by default) redirects to the dashboard in the CloudWatch console. Best for teams that want out-of-the-box monitoring without manual dashboard setup.

For more information, see bb-dashboard on GitHub.

Best practices for AWS CDK Blocks

This topic covers best practices for structuring, securing, testing, and scaling AWS CDK Blocks applications.

Project structure

Organize your backend code for clarity and maintainability.

Keep the app definition thin. Your `cdk-blocks/index.ts` should contain CDK Block instantiations and API definitions. Move complex business logic into separate modules.

```
// cdk-blocks/index.ts - thin orchestration layer
import { ApiNamespace, Scope, KVStore } from 'aws-cdk-blocks';
import { createOrder, listOrders, type Order, type OrderInput } from './orders.js';

const scope = new Scope('my-app');
const store = new KVStore<Order>(scope, 'orders', {});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  createOrder: (input: OrderInput) => createOrder(store, context, input),
  listOrders: () => listOrders(store, context),
}));
```

```
// cdk-blocks/orders.ts - business logic, testable in isolation
import type { BlocksContext, KVStore } from 'aws-cdk-blocks';

export interface OrderInput { title: string }
export interface Order extends OrderInput { id: string; userId: string }

export async function createOrder(store: KVStore<Order>, context: BlocksContext, input:
 OrderInput) {
  // validation, business rules, persistence
}
```

Use a single Scope per application. Multiple scopes in one backend create separate resource namespaces, which adds complexity without benefit for most applications.

Co-locate related CDK Blocks. If your app has distinct domains (orders, users, notifications), group them logically but keep them in the same app definition file unless the file becomes unwieldy.

Error handling

CDK Blocks sends an error's `name` and message to the frontend only for errors it knows are safe to show: an `ApiError`, or an error created with `blocksError()`. Any other error thrown in an API method, including a plain `Error` with a custom `name`, reaches the client as a nameless error with status 500 and the message `Internal error`. Throw an `ApiError` with a name to enable structured error handling on the client.

```
// Backend
import { ApiError } from 'aws-cdk-blocks';

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async getItem(id: string) {
    const item = await store.get(id);
    if (!item) {
      throw new ApiError(`Item ${id} not found`, 404, { name: 'NotFoundError' });
    }
    return item;
  },
}));
```

```
// Frontend
import { api } from 'cdk-blocks';
import { isBlocksError } from 'aws-cdk-blocks/client';

try {
  await api.getItem('abc');
} catch (err) {
  if (isBlocksError(err, 'NotFoundError')) {
    // Handle not found
  }
}
```

Best practices for errors:

- Throw `ApiError(message, status, { name })` with a descriptive name (`NotFoundError`, `ValidationError`, `UnauthorizedError`). Avoid a plain `Error`, because its name and message never reach the client.
- For validation errors, throw `ApiError(message, 400)` so the client gets the message and a 400 status.

- In frontend code, import `isBlocksError` from `aws-cdk-blocks/client`. The main `aws-cdk-blocks` entry exports nothing in a browser build.
- Never expose internal details (stack traces, AWS resource ARNs) in error messages. The `Error.cause` property stays server-side and is never sent to the client.
- Let unhandled errors bubble up. CDK Blocks returns them as generic 500 responses without leaking internals.

Authentication and authorization

Always validate the user in every API method that accesses user data. Don't rely on frontend-only checks. Every `ApiNamespace` method is a public endpoint until it calls `requireAuth`. The local implementation does not check auth either, so a method you forget to gate still passes every local test.

```
export const api = new ApiNamespace(scope, 'api', (context) => ({
  async getMyData() {
    // Always authenticate server-side
    const user = await auth.requireAuth(context);
    return store.get(`user:${user.userSub}`);
  },
}));
```

Scope data access by user. Use the user's stable ID (`user.userSub`) as a key prefix to prevent users from accessing each other's data. It stays the same for the user's lifetime. Don't key data on an email address, which a user can change.

```
// Good: scoped to user
await store.put(`${user.userSub}:${itemId}`, data);

// Bad: no user scoping - any authenticated user can access any item
await store.put(itemId, data);
```

Make protection the default. Define namespaces that hold signed-in methods with `auth.protect`, so each method runs only after the session check. Gate admin methods with `requireRole` or the `role` option of `auth.protect`.

Choose where users live before the first deploy. The `Auth` Block keeps users in an Amazon Cognito user pool by default. Use `store: 'builtin'` when an OIDC provider is the only source

of users. Changing the ID of a deployed CDK Block, or switching it from the default store to `store: 'builtin'`, removes its user pool and every user in it.

Local development

Develop locally first. Use `npm run dev` as your primary development loop. Only deploy to a sandbox when you need to test behavior specific to real AWS services.

Understand local implementation limitations. Local implementations replicate the API surface but not all production behavior:

Block	Local behavior	Production difference
`KVStore`	Local filesystem store	`query()`, `scan()`, `getBatch()`, and index reads are eventually consistent. `get()` is strongly consistent by default.
`Database`	PGlite (embedded PostgreSQL), one statement or transaction at a time	Concurrent transactions on Aurora DSQL can fail at commit, which local runs don't reproduce. An `aurora` cluster at `minCapacity: 0` takes about 20 seconds to resume
`Auth`	Local user pool in `.bb-data/`. Social, SAML, and OIDC providers on the default store can't sign in	Sessions and users are not portable between local and deployed environments. Cognito sends codes by email
`Realtime`	Local WebSocket server	API Gateway WebSocket has connection limits and message size limits

Clear local data when needed. Delete the `.bb-data/` directory to reset all local state:

```
rm -rf .bb-data
```

Testing

Unit test your business logic independently. Extract logic from API handlers into pure functions that accept CDK Block instances as parameters.

```
// orders.ts - testable without the full framework
import { ApiError, type KVStore } from 'aws-cdk-blocks';

export interface OrderInput { title: string }
export interface Order extends OrderInput { id: string; userId: string }

export async function createOrder(store: KVStore<Order>, userId: string, input:
 OrderInput) {
  if (!input.title) throw new ApiError('Title required', 400);
  const order: Order = { id: crypto.randomUUID(), ...input, userId };
  await store.put(`${userId}:${order.id}`, order);
  return order;
}
```

```
// orders.test.ts
import type { KVStore } from 'aws-cdk-blocks';
import { createOrder, type Order } from './orders.js';

it('creates an order', async () => {
  const mockStore = { put: vi.fn(), get: vi.fn() };
  const result = await createOrder(mockStore as unknown as KVStore<Order>, 'user-1',
 { title: 'Test' });
  expect(result.title).toBe('Test');
  expect(mockStore.put).toHaveBeenCalled();
});
```

Integration test locally. Run your full application with `npm run dev` and test the API endpoints. Local implementations are deterministic and fast.

End-to-end test with sandbox. For critical paths, deploy to a sandbox and run tests against real AWS services to catch behavior differences.

Performance

Minimize CDK Block instantiations. Each CDK Block maps to AWS resources that you deploy and manage. CDK Blocks creates DynamoDB tables with on-demand billing, so an extra table costs little

while idle. Each one is still another resource to deploy, monitor, and back up. Don't create a new `KVStore` for every data type. Use key prefixes to partition data within a single store.

```
// Good: one store, partitioned by prefix
const store = new KVStore(scope, 'data', {});
await store.put(`users:${id}`, userData);
await store.put(`orders:${id}`, orderData);

// Avoid: a separate store for each entity (more resources to deploy and manage)
const userStore = new KVStore(scope, 'users', {});
const orderStore = new KVStore(scope, 'orders', {});
```

Declare a key for query-heavy workloads. If you need to list or sort records, or read them by other attributes, give the `KVStore` a `schema` and a `key`, and add `indexes`. A `query()` reads one partition, which is more efficient than a `scan()` of the whole table.

Keep API methods focused. By default, every `ApiNamespace` method in your app runs on one shared AWS Lambda function. To give a namespace more memory, pass a `Compute` Block in the namespace's `compute` option. Avoid methods that do too many things. Split them into focused operations.

Deployment

Use sandbox for development, full deploy for production. Sandboxes use Lambda hot-swapping for speed but aren't suitable for production traffic.

Set up separate AWS accounts for development, staging, and production. CDK Blocks deploys the same code to any account. The resources are derived from your code, not from environment-specific configuration.

Configure environment-specific settings in the CDK layer. Custom domains, VPC configuration, and other environment differences belong in `cdk-blocks/index.cdk.ts`, not in your runtime code.

```
// cdk-blocks/index.cdk.ts
const isProd = process.env.DEPLOY_ENV === 'production';

const stack = await BlocksStack.create(app, stackName, {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
```

```
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: isProd ? BlocksPresets.production : BlocksPresets.sandbox,
});

if (isProd) {
  // Production-only: custom domain, WAF, etc.
  new Hosting(stack, 'Hosting', {
    root: join(__dirname, '..'),
    api: stack,
    frontDoor: {
      features: {
        domain: {
          domainName: 'app.example.com',
          hostedZone: 'example.com',
          hostedZoneId: 'Z0123456789EXAMPLE',
        },
      },
    },
  });
}
```

Working with coding agents

CDK Blocks gives coding agents guidance. Steering files ship with each CDK Block.

To get the best results from coding agents:

- **Keep your app definition clean and readable.** Coding agents use it as context to understand your app.
- **Use descriptive CDK Block IDs.** `new KVStore(scope, 'user-sessions', {})` gives agents more context than `new KVStore(scope, 's1', {})`.
- **Write JSDoc comments on API methods.** Agents use these to understand intent when generating frontend code.
- **Install CDK Block packages.** Their `README.md` and type definitions in `node_modules` provide agents with API documentation and usage examples.
- **Use the Agent Block for AI capabilities.** The Agent Block provides tool calling, HITL approval, and conversation persistence. Locally, it uses a canned provider for predictable testing without API keys.

Example prompt for coding agents

The following prompt shows how a coding agent can add several capabilities in one request:

```
Add a FileBucket for user avatars, an AsyncJob that generates thumbnails on upload, and
 a Realtime channel that notifies the UI when processing completes.
```

Hosting

The Hosting construct provides frontend deployment with server-side rendering (SSR) support. Hosting auto-detects your frontend framework (Next.js, Nuxt and other Nitro-based frameworks, Astro, SvelteKit, or SPA) and provisions the required AWS resources. Import from `aws-cdk-blocks/cdk` and use in your CDK layer file (`cdk-blocks/index.cdk.ts`).

Locally, the development server serves your frontend at `http://localhost:3000` with hot reload. On AWS, Hosting provisions an Amazon CloudFront distribution with an Amazon S3 origin, AWS Lambda compute for SSR, an optional AWS WAF, Amazon CloudWatch alarms, and DNS records. Hosting also supports custom domains and skew protection during rolling deployments. CloudFront is the default front door. You can choose a different one with the `frontDoor` prop. For more information, see the section called “Front door”.

Use Hosting in the CDK layer, not in the app definition. The `npm run deploy` command deploys Hosting. A plain sandbox (`npm run sandbox`) deploys only your backend, and `npm run sandbox:preview` deploys a sandbox that includes Hosting in a fast preview shape. Hosting itself does not check for a sandbox. The project template’s CDK layer file decides when to add Hosting. For more information, see the section called “Basic usage”.

What Hosting provisions

On AWS, a single `Hosting` construct with the default front door provisions these frontend resources:

- **Amazon CloudFront** – a global content delivery network (CDN) distribution that serves your site from edge locations close to your users
- **Amazon S3** – an origin bucket that stores your static assets (HTML, CSS, JavaScript, images)
- **AWS Lambda** – server-side rendering (SSR) compute for framework routes that render on the server. Applies to SSR frameworks only. Hosting skips this resource for single-page applications (SPAs) and static sites.
- **Custom domains with TLS** – optional Amazon Route 53 records and an AWS Certificate Manager (ACM) certificate for your own domain over HTTPS
- **AWS WAF** – optional web application firewall rules, including rate limiting

- **Amazon CloudWatch monitoring** – alarms for the Amazon CloudFront 5xx error rate, SSR AWS Lambda errors and throttles, image optimization AWS Lambda errors, and the revalidation dead-letter queue depth (enabled by default)
- **Skew protection** – cookie-based pinning that keeps a user's session on a single build during rolling deployments (enabled by default)

When you provide the `api` prop, the front door also proxies backend API requests through the same domain as your frontend. With the default front door, Hosting adds Amazon CloudFront behaviors for this. Your frontend calls the API with relative URLs (for example, `/aws-blocks/...`). Because the frontend and API share a single origin, you avoid cross-origin resource sharing (CORS) concerns.

Supported frameworks

Hosting detects your framework from the `dependencies`, `devDependencies`, and `peerDependencies` in your `package.json` and selects the matching adapter. SSR frameworks deploy with AWS Lambda compute. SPA frameworks deploy as static assets that Amazon CloudFront serves from an Amazon S3 origin.

Framework	Rendering	Detected from
Next.js	SSR	`next` in dependencies
Nuxt	SSR	`nuxt` or `nitropack` in dependencies
Astro	Static or SSR	`astro` in dependencies
Nitro-based (SolidStart, TanStack Start, and others)	SSR	`@solidjs/start`, `@tanstack/start`, `nitropack`, or one of the other packages on Hosting's fixed Nitro list, in dependencies. For a Nitro-based framework that is not detected, set `framework: 'nitro'`
SvelteKit 2	SSR	`@sveltejs/kit` in dependencies
React, Vue, Svelte, Angular (SPA)	Static single-page application	None of the above (falls back to the SPA/static adapter)

If more than one package matches, the first match in this order wins: Next.js, Nitro-based, Astro, SvelteKit. For SvelteKit, Hosting builds your app with `@sveltejs/adapter-node` and runs the resulting server on AWS Lambda. If your `svelte.config.js` sets no adapter, Hosting replaces the file with one that uses `adapter-node` for the duration of the build, then restores your original. If it sets any other adapter, including `adapter-auto`, synthesis fails with `SvelteKitIncompatibleAdapterError`. Switch to `adapter-node` or remove the adapter.

Framework detection is automatic. To override it, set the `framework` prop explicitly. It accepts `nextjs`, `nuxt`, `nitro`, `astro`, `sveltekit`, `spa`, and `static`. Use `spa` for a single-page app that routes on the client, and `static` for a multi-page static site.

```
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(stack, 'Web', {
  root: join(__dirname, '..'),
  framework: 'nextjs',
});
```

For a framework that Hosting does not support, pass your own adapter function with the `customAdapter` prop.

For the full list of supported framework versions and native platform clients, see Supported platforms.

Basic usage

Before you begin, make sure you have a CDK layer file (`cdk-blocks/index.cdk.ts`) and a frontend app with a `package.json` in your project.

Add Hosting to your CDK layer file. At minimum, provide the `root` path to your frontend app and a `buildCommand`. Pass your backend stack as `api` to serve the frontend and API from the same domain.

The following example has the same shape as the CDK layer file that `npm create @aws-cdk-blocks/blocks-app` generates:

```
// cdk-blocks/index.cdk.ts
```

```
import * as cdk from 'aws-cdk-lib';
import { BlocksPresets, BlocksStack, Hosting, HostingPresets } from 'aws-cdk-blocks/
cdk';
import { getStackName } from 'aws-cdk-blocks/scripts';
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

const app = new cdk.App();
const sandboxMode = app.node.tryGetContext('sandboxMode') === 'true';
const previewMode = app.node.tryGetContext('previewMode') === 'true';
const projectRoot = app.node.tryGetContext('projectRoot') || process.cwd();

const stackName = getStackName({ sandbox: sandboxMode, projectRoot });
export const blocksStack = await BlocksStack.create(app, stackName, {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production,
});

// Hosting for `npm run deploy` and `npm run sandbox:preview`.
// A plain `npm run sandbox` deploys the backend only.
if (!sandboxMode || previewMode) {
  new Hosting(blocksStack, 'Hosting', {
    ...(previewMode ? HostingPresets.preview : HostingPresets.production),
    root: join(__dirname, '..'),
    buildCommand: 'npm run build',
    api: blocksStack,
  });
}
```

The example does the following:

- It defines *dirname*`dirname` is not defined by default.
- It names the stack with `getStackName()`, which ends the name in `-prod` for a deploy and in your sandbox ID for a sandbox, so a sandbox never overwrites your production stack.
- It adds Hosting only for `npm run deploy` (`sandboxMode` is not set) and for `npm run sandbox:preview` (`previewMode` is set). Without this check, every sandbox would deploy Hosting too.

- It spreads `HostingPresets.preview` for a preview sandbox and `HostingPresets.production` otherwise. For more information, see the section called "Preview deployments".

Omit the `api` prop to deploy a static-only site with no backend. If you already built your app, omit `buildCommand` and Hosting deploys the existing output.

The `Hosting` construct exposes the resources it creates as public readonly members, so you can reference them elsewhere in your CDK layer:

Member	Description
`bucket`	The Amazon S3 bucket storing static assets.
`distribution`	The Amazon CloudFront distribution serving the site. Available only with the default CloudFront front door or the stacked door. With any other front door, reading it throws `FrontDoorNotCloudFrontError`; use `url` instead.
`url`	The public URL of the deployed site.
`ssrFunction`	The primary SSR AWS Lambda function, when the framework uses SSR.
`ssrWebFunction`	The SSR server, when `compute.type` is `'web-function'`. Set instead of `ssrFunction`.
`buildCacheBucket`	The build-cache Amazon S3 bucket, when `buildCache.enabled` is `true`.
`monitoring`	With the CloudFront front door, when monitoring is enabled, an object with `alarms` (every Amazon CloudWatch alarm Hosting created) and `alarmTopics` (the Amazon Simple Notification Service topics the alarms notify).

Front door

The front door is the service that receives your users' requests and routes them to your static assets, SSR compute, and API. By default, Hosting uses an Amazon CloudFront distribution. To use a different front door, set the `frontDoor` prop:

`frontDoor` value	Serves your app through	Use it when	Settings it accepts in `features`
Omitted, `'cloudfront'`, or `{ features`	Amazon CloudFront, a global CDN with an edge cache (the default)	Your app is public, has users in many places, and needs the edge cache, skew protection, or geo restriction	`domain`, `waf`, `logging`, `monitoring`, `skewProtection`, `priceClass`, `contentSecurityPolicy`, `geoRestriction`, `ssrDefaultTtl`, `cacheKeyCookies`, `cacheKeyHeaders`, `responseHeadersPolicy`, `quotas`
`{ kind: 'alb' }`	A regional Application Load Balancer in a VPC (Hosting creates a VPC unless you pass `vpc`). No edge cache. Set `internal: true` for a private load balancer in private subnets. A VPC that you pass must have private subnets for this. With `features.domain`, the load balancer serves HTTPS at your domain, using `certificate` if you	Your load balancer must be in a VPC, or your app must be private	`domain`, `waf`, `logging`, `monitoring`

`frontDoor` value	Serves your app through	Use it when	Settings it accepts in `features`
	pass one. Without a domain, it serves plain HTTP.		
`{ kind: 'apiGateway' }`	Amazon API Gateway, with no VPC and no edge cache. Set `api: 'http'` (the default) for an HTTP API or `api: 'rest'` for a REST API. A REST API needs `domain`. Pass `certificate` to use your own regional certificate for the domain.	You want inexpensive serverless hosting with no VPC or CDN, such as previews and small apps	`domain`

`frontDoor` value	Serves your app through	Use it when	Settings it accepts in `features`
`{ kind: 'stacked', edge: 'cloudfront', router: 'alb' }`	An Amazon CloudFront edge in front of an Application Load Balancer. The edge serves static files from Amazon S3 and sends dynamic requests through the load balancer, which accepts traffic only from the edge. Takes `vpc`, `internal`, and `certificate`, as the `alb` door does. Without `features.router.domain`, the hop from the edge to the load balancer is plain HTTP, and synthesis warns about it for an internet-facing load balancer.	You want the CloudFront edge, with dynamic traffic through a load balancer that only the edge can reach	`edge: { waf, contentSecurityPolicy }` and `router: { domain, waf, logging, monitoring }`

`frontDoor` value	Serves your app through	Use it when	Settings it accepts in `features`
`{ kind: 'custom', door }`	A front door you write with `defineFrontDoor()` from `@aws-cdk-blocks/hosting/constructs`. Hosting passes the settings to your door as `input.features` and does not apply them itself, except that `logging` also controls the access logs of the asset bucket.	A platform team shares one front door across many apps	`domain`, `waf`, `logging`, `monitoring`, `geoRestriction`, `skewProtection`
`'none'`	An Amazon S3 static website over HTTP. Static sites and SPAs only. Your frontend calls the API from a different origin.	You serve a static demo over plain HTTP	None

The following behavior differs from the default CloudFront front door:

- **Edge features** – the `alb` and `apiGateway` front doors have no edge cache, skew protection, or geo restriction. The stacked front door has the edge cache, but no skew protection or geo restriction.
- **File size limit** – the `alb` and `apiGateway` front doors serve static files through an AWS Lambda proxy that returns the whole file, base64 encoded, in its response. The response limit is 1 MB behind an Application Load Balancer and 6 MB behind API Gateway, and base64 encoding makes each file about a third larger. The proxy measures a compressible file after gzip compression. Synthesis lists the files over the limit in an `AssetExceedsProxyLimit` warning.
- **Buffered server responses** – the HTTP API buffers SSR responses, with a limit of about 6 MB each. The Application Load Balancer, including the one behind the stacked front door, also buffers them, with a limit of 1 MB. A REST API streams them.

- **Next.js middleware and edge routes** – middleware runs only behind the CloudFront front door. Edge routes (`runtime: 'edge'`) need the CloudFront front door, or `edgeRoutes: 'regional'` behind the HTTP API, which is the default there. Otherwise, synthesis fails. The Next.js server streams its responses, so synthesis also fails behind the `alb` and stacked front doors, with `StreamingServerNeedsStreamingDoorError`.

Every built-in front door answers 404 for a missing static file. Behind every built-in front door except `'none'`, a multi-page static site with no 404 page of its own gets a default 404 page.

Pass the settings for the chosen front door in `features`. A setting the front door does not implement is a type error. The following example keeps the default CloudFront front door and turns on AWS WAF:

```
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(blocksStack, 'Web', {
  root: join(__dirname, '..'),
  buildCommand: 'npm run build',
  api: blocksStack,
  frontDoor: {
    features: {
      waf: { enabled: true },
    },
  },
});
```

The following example serves the same app behind an API Gateway HTTP API instead:

```
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(blocksStack, 'Web', {
  root: join(__dirname, '..'),
  buildCommand: 'npm run build',
  api: blocksStack,
  frontDoor: { kind: 'apiGateway' },
```

```
});
```

A custom front door is an object with a `service` ID and a `build` function. Hosting calls `build` once at synthesis, after it creates the bucket, compute, and API. The `url` that `build` returns becomes your app's address. Hosting does not check that your door applies `features`. It checks only that the door's routing reaches the static assets and the API (`wiringCheck`, on by default). The check sees routing wired with the `cf` helpers from `@aws-cdk-blocks/hosting/constructs`. If you wire routing another way, declare it with `cf.recordWired()` or set `wiringCheck: false`. The same module also exports the asset proxy and API proxy helpers that the built-in doors use.

```
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';
import { defineFrontDoor } from '@aws-cdk-blocks/hosting/constructs';

const __dirname = dirname(fileURLToPath(import.meta.url));

const myDoor = defineFrontDoor({
  service: 'my-door',
  build(scope, { app, origins, features }) {
    // app: routes, SPA fallback, redirects, header rules, error pages
    // origins: assets (bucket and build prefix), server, image, api
    // features: the settings from frontDoor.features
    // ...create the door...
    return { url: 'https://app.example.com' };
  },
});

new Hosting(blocksStack, 'Web', {
  root: join(__dirname, '..'),
  buildCommand: 'npm run build',
  api: blocksStack,
  frontDoor: { kind: 'custom', door: myDoor },
});
```

Preview deployments

`HostingPresets` holds two preset groups of Hosting props. Spread one into the props first, then add or override props after it:

- `HostingPresets.production` sets nothing, so you get the defaults: the CloudFront front door, image optimization, incremental regeneration, Lambda@Edge for Next.js edge routes, and a new build ID for every deploy.
- `HostingPresets.preview` is a fast, inexpensive shape for temporary environments such as previews of pull requests. It sets `frontDoor: { kind: 'apiGateway' }` (an API Gateway HTTP API, with no CloudFront), `imageOptimization: false`, `incrementalRegeneration: false`, `edgeRoutes: 'regional'`, and `buildId: 'fixed'`. Routing, server rendering, prerendered pages, cookie sign-in, and the API keep working. Server responses are buffered, with a limit of about 6 MB each. A preview has no CDN, AWS WAF, or skew protection, serves images unoptimized, and has the file size limit of the `apiGateway` front door. Its URL is public, and Hosting adds no authentication to it.

To keep one preview setting at its default, override it after the spread:

```
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(blocksStack, 'Hosting', {
  ...HostingPresets.preview,
  imageOptimization: true,
  root: join(__dirname, '..'),
  api: blocksStack,
});
```

`imageOptimization: false` and `edgeRoutes: 'regional'` need the API Gateway HTTP API front door. `buildId: 'fixed'` is not available behind the CloudFront front door or the stacked front door. This is because the edge cache would keep serving files that each deploy overwrites. When a setting does not fit the front door, synthesis fails with `UnsupportedHostingSettingError`.

`npm run sandbox:preview` deploys a sandbox with Hosting in the preview shape, when your CDK layer file is set up as in the section called “Basic usage”. It deploys once, prints the site URL, and exits. It does not watch for changes or start a local server. Run it again to redeploy. A change that touches only backend code is hotswapped. The plain sandbox commands deploy the same stack without Hosting, so running `npm run sandbox` after a preview removes the preview site. Remove a preview sandbox with `npm run sandbox:destroy`. For a pull request preview, run `npm`

`run sandbox:preview` from CI, and run `npm run sandbox:destroy` when the pull request closes. Keep `.cdk-blocks-sandbox/sandbox-id.txt` between the two runs, such as in a CI cache, because it names the sandbox stack.

Custom domains

Configure a custom domain with `frontDoor.features.domain`. With the default CloudFront front door, the ACM certificate must be in the `us-east-1` AWS Region. This requirement applies regardless of where you deploy the rest of your stack.

You can set up a domain in two ways.

Amazon Route 53 (automatic) – provide a `hostedZone` you control. Hosting looks the zone up, so the stack needs `env: { account, region }`, or you also pass the zone's `hostedZoneId`. Hosting requests a DNS-validated ACM certificate and creates the A and AAAA alias records automatically:

```
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(blocksStack, 'Web', {
  root: join(__dirname, '..'),
  buildCommand: 'npm run build',
  api: blocksStack,
  frontDoor: {
    features: {
      domain: {
        domainName: 'app.example.com',
        hostedZone: 'example.com',
      },
    },
  },
});
```

The following fragments go inside `frontDoor.features`. To serve both an apex domain and its `www` subdomain, pass an array of domain names and set `wwwRedirect`:

```
domain: {
```

```
  domainName: ['example.com', 'www.example.com'],
  hostedZone: 'example.com',
  wwwRedirect: 'toApex',
}
```

Bring your own DNS (manual) – if you manage DNS outside Amazon Route 53, omit `hostedZone` and supply a pre-validated `certificate` in `us-east-1`. Hosting does not create DNS records. Instead, it outputs the Amazon CloudFront distribution domain name. Point a CNAME (or an ALIAS/ANAME record for an apex domain) at that domain name:

```
domain: {
  domainName: ['app.example.com'],
  certificate: myPreValidatedCert, // ACM certificate in us-east-1
}
```

> **Note**
>
> If you provide neither a hosted zone nor a certificate, synthesis fails fast with a `MissingCertificateError` rather than waiting on an unvalidated certificate at deploy time. A certificate outside `us-east-1` fails with an `InvalidCertificateRegionError`.

SSR compute

For SSR frameworks, tune the AWS Lambda function that renders your server-side routes with the `compute` prop. Hosting ignores this prop for SPA and static deployments.

```
import * as cdk from 'aws-cdk-lib';
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(blocksStack, 'Web', {
  root: join(__dirname, '..'),
  buildCommand: 'npm run build',
  framework: 'nextjs',
  api: blocksStack,
  compute: {
    memorySize: 2048,
```

```
    timeout: cdk.Duration.seconds(30),
  },
});
```

Option	Default	Description
`type`	`'lambda'`	Where the SSR server runs: `'lambda'` or `'web-function'` . See **Lambda Web Functions** after this table.
`webFunction`	None	Tuning for Lambda Web Functions, such as `maxEnvironments` . Used only when `type` is `'web-function'` .
`memorySize`	1024	AWS Lambda memory size, in MB.
`timeout`	30 seconds	AWS Lambda timeout. Accepts a `cdk.Duration` or a plain number of seconds.
`reservedConcurrency`	None	Reserved concurrent executions for the SSR AWS Lambda function.
`imageOptimization.reservedConcurrency`	None	Reserved concurrency for the image-optimization AWS Lambda function. Left unreserved by default so deployments succeed on fresh AWS accounts.
`logRetention`	One month	Amazon CloudWatch log retention for the SSR AWS Lambda function.

Lambda Web Functions – set `compute.type` to `'web-function'` to run your framework's own Node.js server directly on AWS Lambda Web Functions. One execution environment serves many concurrent requests, which means fewer cold starts and lower cost for SSR that mostly waits on I/O. Web Functions work with Next.js, Nuxt and other Nitro-based frameworks, Astro, and SvelteKit. They need the default CloudFront front door and one of these Regions: `us-east-1`, `us-east-2`, `us-west-2`, `ap-northeast-1`, `eu-west-1`. With another front door, synthesis fails with `UnsupportedComputeFrontDoorError`. `memorySize` and `reservedConcurrency` do not apply; tune the server with `webFunction` instead:

```
compute: {
  type: 'web-function',
  webFunction: { maxEnvironments: 20 },
},
```

Security

Hosting provides several controls for protecting your distribution. Set the WAF, Content Security Policy, and geo-restriction settings inside `frontDoor.features`.

WAF – enable AWS WAF with the `waf` setting to add managed protections and rate limiting. You can also associate an existing WebACL by ARN (the WebACL must be in `us-east-1`):

```
waf: {
  enabled: true,
  rateLimit: 1000, // requests per 5-minute window per IP
}
```

Content Security Policy – set `contentSecurityPolicy` to override the restrictive default `Content-Security-Policy` header. Hosting applies its security headers, including this one, to the API and auth paths it proxies as well as to your site:

```
contentSecurityPolicy: "default-src 'self'; img-src 'self' data:;",
```

Geo-restriction – use `geoRestriction` to restrict access by country. Set `countries` to an array of ISO country codes and `type` to `'allowlist'` (allow only those countries) or `'denylist'` (block them):

```
geoRestriction: { type: 'allowlist', countries: ['US', 'CA', 'GB'] },
```

> **Warning**
>
> Hosting writes values from `backendConfig` to `/.cdk-blocks-sandbox/config.json` and injects them into the `BLOCKS_CONFIG` AWS Lambda environment variable. Both locations are publicly accessible. Do not include secrets, API keys, or other sensitive configuration. For sensitive values, use the `environment` prop with `secret()`, as described next.

Secrets and configuration for SSR code – the `environment` prop sets environment variables on every regional compute function, including image optimization. Lambda@Edge functions do not get them. A plain string is stored as is, in the CloudFormation template and the function configuration. A `secret('KEY')` value is stored in AWS Secrets Manager, and a `config('KEY')` value in AWS Systems Manager Parameter Store. For those, Hosting injects only the location of the value, never the value, and grants the function read access to it. The variable name must match the key:

```
import { config, secret } from 'aws-cdk-blocks/cdk';
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = dirname(fileURLToPath(import.meta.url));

new Hosting(blocksStack, 'Web', {
  root: join(__dirname, '..'),
  buildCommand: 'npm run build',
  api: blocksStack,
  environment: {
    FEATURE_FLAG: 'on',
    PAYMENTS_API_KEY: secret('PAYMENTS_API_KEY'),
    SUPPORT_EMAIL: config('SUPPORT_EMAIL'),
  },
});
```

In your SSR code, read the values with `await getSecret('PAYMENTS_API_KEY')` and `await getConfig('SUPPORT_EMAIL')` from `@aws-cdk-blocks/hosting`. Locally, each getter reads `process.env` first, so you can put the value in a `.env` file.

Operations

Hosting includes operational features to keep deployments observable and reliable.

Monitoring – Amazon CloudWatch alarms are enabled by default. They cover the Amazon CloudFront 5xx error rate, SSR AWS Lambda errors and throttles, image optimization AWS Lambda errors, and the revalidation dead-letter queue depth. Hosting creates the Amazon SNS topics the alarms notify. If your stack sets `env: { account, region }` and is not in `us-east-1`, the CloudFront alarm and its topic are created in `us-east-1`. Without `env`, they are created in your stack, and synth prints a warning. With the CloudFront front door, the construct's `monitoring` member exposes the alarms as `monitoring.alarms` and the

topics as `monitoring.alarmTopics`. To get notified, add email or URL subscriptions in `frontDoor.features`:

```
import { EmailSubscription } from 'aws-cdk-lib/aws-sns-subscriptions';
```

```
monitoring: { subscriptions: [new EmailSubscription('ops@example.com')] },
```

Set `monitoring: { enabled: false }` to opt out.

Skew protection – enabled by default. During a rolling deployment, viewers who started a session continue to receive assets from the same build. This prevents asset mismatches. Set `skewProtection: { enabled: false }` in `frontDoor.features` to disable it. Hosting keeps superseded builds in Amazon S3 for 30 days. Change this with the `buildRetentionDays` prop.

Access logging – enabled by default. Amazon CloudFront access logs go to a dedicated Amazon S3 bucket. Use the `logging` setting in `frontDoor.features` to change how long logs are kept, or set `logging: { enabled: false }` to turn them off:

```
logging: { enabled: true, retentionDays: 90 },
```

Build cache – enable `buildCache` to preserve framework build caches (such as `.next/cache`) between deployments in an Amazon S3 bucket, reducing cold-build times in CI:

```
buildCache: { enabled: true },
```

Configuration reference

The following table summarizes the top-level `Hosting` properties. All properties except `root` are optional. Deprecated properties still work and are merged into `frontDoor.features`. For more information, see the section called “Front door”.

> **Note**
>
> The top-level props `domain`, `waf`, `logging`, `monitoring`, `skewProtection`, `priceClass`, `contentSecurityPolicy`, `geoRestriction`, `ssrDefaultTtl`, `cacheKeyCookies`, `cacheKeyHeaders`, and `quotas` are deprecated aliases for the `frontDoor.features` settings of the same name. Setting the same setting

both ways fails synthesis with `FrontDoorSettingConflictError`. A top-level prop that the chosen front door does not implement fails synthesis with `UnsupportedFrontDoorSettingError`.

Property	Default	Description
`root`	None (required)	Path to the frontend app root directory. Required.
`buildCommand`	None	Shell command to build the frontend during CDK synthesis. Omit if you pre-built the app.
`framework`	Auto-detected	Framework type. Overrides detection from `package.json` .
`buildOutputDir`	Auto-detected	Path to the build output directory.
`customAdapter`	None	Your own adapter function, for a framework Hosting does not support.
`basePath`	None	URL prefix for the entire site (for example, `/app`).
`api`	None	Backend stack to proxy through the same domain (single-origin, no CORS).
`backendConfig`	None	Extra config merged into `config.json` . Publicly accessible, so do not include secrets.
`compute`	See the section called "SSR compute"	AWS Lambda compute configuration for SSR frameworks.
`environment`	None	Environment variables for SSR compute. Values are plain strings, `secret()`, or `config()`. See the section called "Security".
`frontDoor`	Amazon CloudFront	The front door and its `features` settings. See the section called "Front door".

Property	Default	Description
`imageOptimization`	`true`	Set `false` to build no image optimization function. Setting it to `false` needs the API Gateway HTTP API front door.
`incrementalRegeneration`	`true`	Set `false` to skip the Next.js incremental regeneration resources. Prerendered pages are still served but never regenerate.
`edgeRoutes`	`'edge'` (`'regional'` behind the API Gateway HTTP API front door)	Where Next.js edge routes run: Lambda@Edge (`'edge'`) or regional AWS Lambda functions (`'regional'`).
`buildId`	`'per-deploy'`	`'fixed'` keeps the build ID while the build output is unchanged, so a redeploy can skip uploads. Not available behind CloudFront.
`domain`	None	Deprecated. Use `frontDoor.features.domain`. Custom domain configuration.
`waf`	Disabled	Deprecated. Use `frontDoor.features.waf`. AWS WAF configuration.
`priceClass`	`PRICE_CLASS_100`	Deprecated. Use `frontDoor.features.priceClass`. Amazon CloudFront price class.
`geoRestriction`	None	Deprecated. Use `frontDoor.features.geoRestriction`. Country allow or deny list for the distribution.
`buildCache`	Disabled	Amazon S3-backed framework build cache.
`errorPages`	None	Custom 404/500 HTML pages. Incompatible with SPA client-side routing.

Property	Default	Description
`logging`	Enabled for CloudFront and asset bucket logs. With the ALB front door, ALB access logs are off unless you set it	Deprecated. Use `frontDoor.features.logging`. Amazon CloudFront access logging.
`monitoring`	Enabled	Deprecated. Use `frontDoor.features.monitoring`. Amazon CloudWatch alarms wired to Amazon SNS topics.
`skewProtection`	Enabled	Deprecated. Use `frontDoor.features.skewProtection`. Cookie-based build pinning during rolling deployments.
`buildRetentionDays`	30	Days to keep superseded builds in Amazon S3. The current build is never removed.
`retainOnDelete`	`false`	Retain the Amazon S3 bucket when you delete the stack.

For the full construct API, adapter internals, and advanced configuration, see the Hosting package source code on GitHub.

Creating custom CDK Blocks

When the AWS CDK Blocks that AWS provides don't cover your use case, or when you want every app team to start from your own standards, create a custom CDK Block. A custom CDK Block packages infrastructure, runtime logic, and a local implementation into a single reusable module. Platform developers share custom CDK Blocks with their app teams through npm.

Use cases for custom CDK Blocks

Create a custom CDK Block when:

- You're repeating the same AWS resource + SDK boilerplate across multiple API methods
- You want local development support for a resource that doesn't have a built-in CDK Block
- You want to share a reusable capability across projects or app teams

Don't create a custom CDK Block when:

- You only need the resource in one place. Use the The CDK layer directly instead
- A built-in CDK Block already does what you need

Create a Block with the scaffolder

Start with the `@aws-cdk-blocks/create-block` scaffolder. It generates a complete CDK Block package that builds and passes its tests before you change anything:

```
npm create @aws-cdk-blocks/block@latest Notifications
```

Pass the class name in PascalCase, without a BB prefix. The package is named `bb-` plus the name in kebab case, for example `@my-org/bb-notifications`.

What the scaffolder does depends on where you run it:

Where you run it	Result
Inside an npm workspaces repository	Creates `packages/bb-<name>` , adds it to the root `workspaces` , runs `npm install`, then builds and tests the

Where you run it	Result
	new package. Your app can import it without publishing. Your app code is not changed. The npm scope comes from the name in the root `package.json` , or is `app` if that name has no scope.
Anywhere else	Creates a standalone `@<scope>/bb-<name>` package in `./bb-<name>` and runs `npm install` in it. The scope defaults to `your-org`.

Option	Meaning
`--scope <npm-scope>`	npm scope for the package name
`--dir <path>`	Target directory for a standalone package
`--yes`, `-y`	Skip the confirmation prompt
`--skip-install`	Don't run `npm install`
`--skip-verify`	Don't build and test the package afterward
`--dry-run`	Print what would be generated, and write nothing

The generated code is a skeleton with one example method (`echo`) and `TODO` markers. Replace it with your CDK Block's API. The rest of this page walks through each file, using an email notifications CDK Block as the example.

Block structure

A CDK Block is an npm package with one entry for each place its code runs:

```
bb-notifications/
### src/
#   ### index.mock.ts
#   ### index.aws.ts      2
#   ### index.cdk.ts      3
```

```
#   ### index.browser.ts  4
#   ### types.ts          5
#   ### errors.ts         6
#   ### version.ts        7
#   ### index.test.ts     8
#   ### parity.test.ts    9
### scripts/
#   ### generate-version.mjs
### package.json
### tsconfig.json
### api-extractor.json
### README.md
### DESIGN.md
### LICENSE
```

Local: Runs during `npm run dev` and in tests. Its types are the CDK Block's public types.

2 **Runtime**: Calls AWS with the SDK. Runs in deployed compute.

3 **Infrastructure**: CDK Constructs and IAM grants. Runs during `cdk synth`.

4 **Browser**: A stub that exports the public types and error names.

5 **Shared types**: Option and input types. Types only.

6 **Errors**: The CDK Block's error names.

7 **Version**: `BB_NAME` and `BB_VERSION`. The `prebuild` script generates this file.

8 **Tests** for the local implementation.

9 **Parity test**: Checks that every entry exports the same error names.

Each implementation class extends `Scope`. The runtime and local classes call `super(id, { parent: scope, bbName, bbVersion })`; the CDK class passes `bbName` only. Extending `Scope` gives the CDK Block its `fullId` and `buildUserAgentChain()`, which you pass to AWS SDK clients as `customUserAgent`. In the CDK class, it also gives the CDK Block a place in the construct tree and the shared execution role. CDK Blocks telemetry counts custom CDK Blocks but never sends their names.

Configure package.json exports

The `package.json` uses conditional exports to route imports to the correct file:

```
{
  "name": "@my-org/bb-notifications",
  "version": "0.1.0",
  "type": "module",
```

```
  "keywords": ["aws-blocks"],
  "exports": {
    ".": {
      "browser": "./dist/index.browser.js",
      "cdk": {
        "types": "./dist/index.cdk.d.ts",
        "default": "./dist/index.cdk.js"
      },
      "aws-runtime": "./dist/index.aws.js",
      "types": "./dist/index.mock.d.ts",
      "default": "./dist/index.mock.js"
    }
  },
  "files": ["dist", "src", "README.md", "DESIGN.md", "LICENSE"],
  "dependencies": {
    "@aws-cdk-blocks/core": "^0.8.0",
    "@aws-sdk/client-sesv2": "^3.1161.0"
  },
  "peerDependencies": {
    "aws-cdk-lib": "^2.257.0",
    "constructs": "^10.6.0"
  }
}
```

Node.js uses the first condition that matches, so keep `browser`, `cdk`, and `aws-runtime` before `types` and `default`. The nested `types` under `cdk` lets TypeScript see the CDK class when it checks CDK code.

Export condition	File	When it's used
`browser`	`index.browser.js`	Bundling frontend code
`cdk`	`index.cdk.js`	CDK synthesis (`--conditions=cdk`)
`aws-runtime`	`index.aws.js`	Deployed compute (AWS Lambda by default)
`types`	`index.mock.d.ts`	IDE IntelliSense and TypeScript compilation
`default`	`index.mock.js`	Local development (`npm run dev`) and tests

Define shared types

Start by defining the options and inputs your CDK Block accepts. Every entry imports them from `types.ts`, which has no runtime code.

```
// src/types.ts
export interface NotificationsOptions {
  /** Sender address. It must be a verified identity in Amazon SES. */
  fromAddress: string;
}

export interface SendEmailInput {
  to: string;
  subject: string;
  body: string;
}
```

The class type that app code sees comes from `index.mock.ts`, because the `types` condition points to it. Keep the public methods the same in every entry.

Define named errors

List your CDK Block's error names in `errors.ts`:

```
// src/errors.ts
export const NotificationsErrors = {
  InvalidRecipient: 'InvalidRecipientException',
} as const;
```

Throw them with `blocksError(name, message)` from `@aws-cdk-blocks/core`. A branded error keeps its `name` and `message` when it reaches the client. Callers can then match it with `isBlocksError(e, NotificationsErrors.InvalidRecipient)` on the server or in the browser. A plain `Error`, or an error from an AWS SDK, reaches the client as a nameless error with status 500 and the message "Internal error". Don't put raw SDK error text in a branded message.

Implement the runtime

The runtime implementation uses the AWS SDK to interact with real services. It runs in deployed compute.

```
// src/index.aws.ts
import { SESv2Client, SendEmailCommand } from '@aws-sdk/client-sesv2';
import { Scope, blocksError } from '@aws-cdk-blocks/core';
import type { ScopeParent } from '@aws-cdk-blocks/core';
import { NotificationsErrors } from './errors.js';
import type { NotificationsOptions, SendEmailInput } from './types.js';
import { BB_NAME, BB_VERSION } from './version.js';

export { NotificationsErrors } from './errors.js';
export type { NotificationsOptions, SendEmailInput } from './types.js';

export class Notifications extends Scope {
  private readonly client: SESv2Client;
  private readonly fromAddress: string;

  constructor(scope: ScopeParent, id: string, options: NotificationsOptions) {
    super(id, { parent: scope, bbName: BB_NAME, bbVersion: BB_VERSION });
    this.client = new SESv2Client({ customUserAgent: this.buildUserAgentChain() });
    this.fromAddress = options.fromAddress;
  }

  async sendEmail(input: SendEmailInput): Promise<{ messageId: string }> {
    if (!input.to.includes('@')) {
      throw blocksError(NotificationsErrors.InvalidRecipient, 'The recipient is not an
 email address.');
    }
    const result = await this.client.send(new SendEmailCommand({
      FromEmailAddress: this.fromAddress,
      Destination: { ToAddresses: [input.to] },
      Content: {
        Simple: {
          Subject: { Data: input.subject },
          Body: { Text: { Data: input.body } },
        },
      },
    }));
    return { messageId: result.MessageId ?? 'unknown' };
  }
}
```

Pass `this.buildUserAgentChain()` as `customUserAgent` so the SDK's user agent names CDK Blocks. Only official CDK Blocks add their own name to the chain, so a custom CDK Block is not named in it.

If your CDK Block owns a named resource, such as a table, name it from `this.fullId` in the CDK layer. In the runtime and local entries, call `registerSdkIdentifiers(this.fullId, { tableName })` in the constructor. Read the name back with `getSdkIdentifiers(this)` when a method runs, not in the constructor. Both functions come from `@aws-cdk-blocks/core`. Each layer derives the same name, so the CDK layer does not need to pass it to the runtime.

Write the local implementation

The local implementation provides the same API but runs without AWS. Use in-memory data structures, the filesystem, or embedded databases.

```
// src/index.mock.ts
import { randomUUID } from 'node:crypto';
import { writeFileSync } from 'node:fs';
import { join } from 'node:path';
import { Scope, blocksError } from '@aws-cdk-blocks/core';
import type { ScopeParent } from '@aws-cdk-blocks/core';
import { assertNotMockInDeployedEnv, getMockDataDir } from '@aws-cdk-blocks/core/bb-
utils';
import { NotificationsErrors } from './errors.js';
import type { NotificationsOptions, SendEmailInput } from './types.js';
import { BB_NAME, BB_VERSION } from './version.js';

// Fail closed if the local implementation is ever loaded in a deployed environment.
assertNotMockInDeployedEnv('@my-org/bb-notifications');

export { NotificationsErrors } from './errors.js';
export type { NotificationsOptions, SendEmailInput } from './types.js';

export class Notifications extends Scope {
  private readonly fromAddress: string;

  constructor(scope: ScopeParent, id: string, options: NotificationsOptions) {
    super(id, { parent: scope, bbName: BB_NAME, bbVersion: BB_VERSION });
    this.fromAddress = options.fromAddress;
  }

  async sendEmail(input: SendEmailInput): Promise<{ messageId: string }> {
    if (!input.to.includes('@')) {
      throw blocksError(NotificationsErrors.InvalidRecipient, 'The recipient is not an
 email address.');
    }
```

```
    const messageId = randomUUID();
    // .bb-data/<fullId>/ belongs to this Block instance.
    const file = join(getMockDataDir(this), `${messageId}.json`);
    writeFileSync(file, JSON.stringify({ from: this.fromAddress, ...input,
 messageId }));
    console.log(`[mock] Email to ${input.to}: "${input.subject}"`);
    return { messageId };
  }
}
```

During local development, emails are logged to the console and saved as JSON files instead of being sent through Amazon SES. `getMockDataDir(this)` returns `.bb-data/<fullId>/` in the project folder and creates it. Pass `this`, not the parent scope, so each instance of the CDK Block gets its own folder.

`assertNotMockInDeployedEnv` makes the module throw if it loads in deployed compute. If the backend is ever bundled without the `aws-runtime` condition, it fails at load instead of running the local implementation.

Implement the infrastructure

The infrastructure export defines CDK Constructs that are synthesized during deployment. Grant permissions to `this.executionRole`, the one IAM role that all compute in the app shares.

```
// src/index.cdk.ts
import * as iam from 'aws-cdk-lib/aws-iam';
import { Scope, synthGuard } from '@aws-cdk-blocks/core/cdk';
import type { ScopeParent } from '@aws-cdk-blocks/core';
import type { NotificationsOptions } from './types.js';
import { BB_NAME } from './version.js';

export { NotificationsErrors } from './errors.js';
export type { NotificationsOptions, SendEmailInput } from './types.js';

export class Notifications extends Scope {
  constructor(scope: ScopeParent, id: string, options: NotificationsOptions) {
    super(id, { parent: scope, bbName: BB_NAME });
    // All compute in the app shares one execution role. Grant to it.
    this.executionRole.addToPrincipalPolicy(new iam.PolicyStatement({
      actions: ['ses:SendEmail'],
      resources: ['*'],
```

```
      conditions: {
        StringEquals: { 'ses:FromAddress': options.fromAddress },
      },
    }));
  }

  // Runtime methods don't exist during synth. A call fails with a clear error.
  sendEmail(..._args: unknown[]): never {
    return synthGuard('Notifications', 'sendEmail');
  }
}
```

Stub every runtime method with `synthGuard`, as the CDK Blocks that ship with CDK Blocks do. If app code calls `sendEmail` at the top level of the backend module, synth stops with an error that says where to call it instead.

If the runtime needs a value that only the CDK layer knows, other than a resource name, pass it with `registerConfig(this, 'KEY', value)` from `@aws-cdk-blocks/core/cdk`. The runtime reads it as `process.env.KEY`.

Client middleware (optional)

Most CDK Blocks return plain JSON data and need no browser code. Add client middleware only if a method must give the browser a live object, such as a channel to subscribe to. This uses the Transferable pattern:

1. The server method returns an object whose `toJSON()` returns a descriptor tagged with `__blocks`.
2. The CDK Block calls `this.registerClientMiddleware()` in the constructor of its local and runtime entries. The generated frontend client imports the module it names.
3. That module calls `registerMiddleware` from `@aws-cdk-blocks/core/client`. Its `onResponse` hook turns the descriptor into the live object. An optional `onRequest` hook can change outgoing requests, for example to add a header.

```
// src/index.aws.ts and src/index.mock.ts (constructor)
this.registerClientMiddleware('@my-org/bb-notifications/middleware');

// A method that returns a Transferable
async trackDelivery(messageId: string) {
```

```
  const eventsUrl = `${this.eventsBaseUrl}/${encodeURIComponent(messageId)}`;
  return {
    // The server sends this descriptor. The client middleware turns it into a live
 object.
    toJSON: () => ({ __blocks: 'my-org/delivery' as const, eventsUrl }),
  };
}
```

```
// src/middleware.ts (package export "./middleware")
import { registerMiddleware } from '@aws-cdk-blocks/core/client';

interface DeliveryDescriptor {
  __blocks: 'my-org/delivery';
  eventsUrl: string;
}

function isDeliveryDescriptor(data: unknown): data is DeliveryDescriptor {
  return typeof data === 'object' && data !== null
    && (data as { __blocks?: unknown }).__blocks === 'my-org/delivery';
}

// Runs on import. The generated client imports this module for you.
registerMiddleware({
  onResponse(data) {
    if (!isDeliveryDescriptor(data)) return data;
    return {
      onStatus(handler: (status: string) => void) {
        const events = new EventSource(data.eventsUrl);
        events.onmessage = (event) => handler(event.data);
        return () => events.close();
      },
    };
  },
});
```

Export the middleware module from `package.json`, for example as `./middleware`. The local implementation must return a working Transferable too. The `Realtime` Block that ships with CDK Blocks uses this pattern for its channels.

Use your Block

After building your package, install it in your project and use it like any built-in CDK Block:

```
// cdk-blocks/index.ts
import { ApiNamespace, ApiError, Scope, isBlocksError } from 'aws-cdk-blocks';
import { Notifications, NotificationsErrors } from '@my-org/bb-notifications';

const scope = new Scope('my-app');
const email = new Notifications(scope, 'email', {
  fromAddress: 'noreply@example.com',
});

export const api = new ApiNamespace(scope, 'api', () => ({
  async sendWelcomeEmail(userEmail: string) {
    try {
      return await email.sendEmail({
        to: userEmail,
        subject: 'Welcome!',
        body: 'Thanks for signing up.',
      });
    } catch (e) {
      if (isBlocksError(e, NotificationsErrors.InvalidRecipient)) {
        throw new ApiError('Enter a valid email address.', 400);
      }
      throw e;
    }
  },
}));
```

The same import resolves to the local entry during `npm run dev`, to the CDK entry during synth, and to the runtime entry in deployed compute.

Testing your Block

Test both the local and runtime implementations:

- **Local implementation tests**: The scaffolded `npm test` runs the Node.js test runner on the compiled tests in `dist/`, so run `npm run build` first. Use a real `Scope` from `@aws-cdk-blocks/core` as the parent.
- **Runtime tests**: Deploy to a sandbox and verify against real AWS services.
- **Interface parity**: Ensure every entry accepts the same inputs and returns the same output shapes. `parity.test.ts` checks the error names.

```
// src/index.test.ts
import { test } from 'node:test';
import assert from 'node:assert';
import { Scope, isBlocksError } from '@aws-cdk-blocks/core';
import { Notifications, NotificationsErrors } from './index.mock.js';

const parent = new Scope('test-app');

test('sendEmail returns a messageId', async () => {
  const email = new Notifications(parent, 'email', { fromAddress:
 'noreply@example.com' });
  const result = await email.sendEmail({ to: 'user@example.com', subject: 'Test', body:
 'Hello' });
  assert.ok(result.messageId);
});

test('sendEmail rejects a recipient that is not an email address', async () => {
  const email = new Notifications(parent, 'email-invalid', { fromAddress:
 'noreply@example.com' });
  await assert.rejects(
    email.sendEmail({ to: 'not-an-address', subject: 'Test', body: 'Hello' }),
    (e) => isBlocksError(e, NotificationsErrors.InvalidRecipient),
  );
});
```

Publishing

To share your CDK Block:

- **Within a monorepo**: Reference it as a workspace dependency. No publishing needed.
- **Within your organization**: Publish to a private npm registry (such as AWS CodeArtifact).
- **Publicly**: Publish to the npm public registry. Keep the `aws-blocks` keyword that the scaffolder adds so people can find your CDK Block.

Include a `README.md` with usage examples, API documentation, and information about the AWS resources provisioned. Coding agents use this documentation to help app teams integrate your CDK Block.

Best practices for custom CDK Blocks

- **Keep the interface identical** across entries. Consumers shouldn't need to know which implementation is active. In the CDK entry, every runtime method is a `synthGuard` stub.
- **Document local implementation limitations**: If the local implementation doesn't replicate all production behavior (such as DynamoDB query limits), document the differences.
- **Use `getMockDataDir(this)`** for persistent local data. It follows the `.bb-data/<fullId>/` convention, where `fullId` joins the IDs of the CDK Block and its parent scopes with `-`.
- **Throw named errors** with `blocksError()` so callers can match them by name.
- **Minimize client middleware**: Only add it when a method returns a live object. Most CDK Blocks don't need any.
- **Keep every entry**: Don't drop an entry from the export map. The browser entry can be a stub that exports types and error names.
- **Share types in a separate file**: This ensures type consistency across all implementations.

CDK Blocks for existing apps

You can add CDK Blocks to an application that already exists. Each CDK Block packages a capability's infrastructure as a CDK Construct, the code your app uses to call it, and a local implementation. Your existing app keeps its frontend, its hosting, and the resources it already has, and the CDK Blocks you add deploy to your own AWS account next to them.

This page covers two kinds of existing apps:

- the section called "AWS Amplify Gen 2 apps"
- the section called "Existing AWS CDK apps"

For pre-deployed resources and other infrastructure patterns, see *Existing infrastructure*.

AWS Amplify Gen 2 apps

To add CDK Blocks to an Amplify Gen 2 app, run the project creation command from the root of the app:

```
npm create @aws-cdk-blocks/blocks-app@latest
```

The command looks for `amplify/backend.ts`. When it finds that file, it treats the directory as an Amplify Gen 2 project and selects the `amplify` template. You can't choose this template for a new app. The command lists the changes it will make and asks you to confirm. It doesn't commit anything, so you can review the diff first.

What the command adds

File	Change
`cdk-blocks/`	A new workspace for your CDK Blocks backend. `index.ts` defines a sample API, `index.cdk.ts` creates the CDK Blocks backend, `client.js` is the frontend client, and `cognito-verifier.ts` checks the sign-in tokens of your Amplify users.
`amplify/blocks.ts`	A new file that adds the CDK Blocks backend to your Amplify backend.

File	Change
`amplify/backend.ts`	Exports `backend` and calls `initBlocks(backend)` at the end of the file. If the command can't change the file, it writes the lines to `amplify/blocks-manual-patch.txt` for you to add.
`package.json`	Adds `cdk-blocks` to `workspaces`, adds dependencies, and adds the scripts in the next section. Existing scripts with the same name are kept.
`.gitignore`	Adds the local data directory `.bb-data/` and other CDK Blocks entries.
`amplify.yml`	Adds `export NODE_OPTIONS="--conditions=cdk"` before `npx ampx pipeline-deploy`. If the file doesn't exist, the command creates one. Check its `frontend` settings, such as `baseDirectory`, against your build.

The two lines added to `amplify/backend.ts` are:

```
import { initBlocks } from './blocks.js';
await initBlocks(backend);
```

`initBlocks` does the following:

1. Creates a stack for CDK Blocks with `backend.createStack('blocks')`.
2. Creates a `BlocksBackend` in that stack. It uses `BlocksPresets.sandbox` when `AMPLIFY_SANDBOX` is `true` and `BlocksPresets.production` otherwise.
3. If the app has Amplify auth, passes the user pool ID, the app client ID, and the Region to the CDK Blocks runtime as `COGNITO_USER_POOL_ID`, `COGNITO_CLIENT_ID`, and `COGNITO_REGION`.
4. Adds the CDK Blocks API URL to `amplify_outputs.json` as `custom.blocks_api_url`.

Calling the API with Amplify sign-in

The generated `client.js` reads the API URL from `amplify_outputs.json`. Before each call, it gets the signed-in user's ID token from Amplify Auth and sends it as a bearer token. On the backend, `CognitoVerifier` checks that token. Methods that don't call it stay public.

```
// cdk-blocks/index.ts
const auth = new CognitoVerifier({
  userPoolId: process.env.COGNITO_USER_POOL_ID!,
  clientId: process.env.COGNITO_CLIENT_ID!,
  tokenUse: 'id',
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async putNote(key: string, value: string) {
    const user = await auth.requireAuth(context);
    await store.put(`${user.sub}:${key}`, value);
    return { success: true };
  },
}));
```

`requireAuth` returns the user, or responds with a 401 `NotAuthenticatedException` error when the token is missing or not valid. `requireRole(context, group)` also responds with a 403 `NotAuthorizedException` error when the token's groups don't include the group. These are the error names that the `Auth` Block uses. In your frontend, import the API from the workspace:

```
import { api } from 'cdk-blocks';
const result = await api.greet('World');
```

When you add a namespace or a method, run `npm run blocks:generate-client` to update `client.js`.

Running and deploying

Command	What it does
`npm run blocks:dev`	Serves the CDK Blocks API on `http://localhost:3001` with each CDK Block's local implementation.

Command	What it does
`npm run blocks:stop`	Stops the local server.
`npm run sandbox`	Runs `ampx sandbox`, which deploys your Amplify backend and the CDK Blocks stack to a sandbox.
`npm run sandbox:delete`	Deletes the sandbox.

CDK Blocks doesn't add an `npm run dev` script to an Amplify project. Amplify Hosting deploys production through `amplify.yml`, which runs `npx ampx pipeline-deploy` with the `cdk` condition set. Without that condition, CDK Block imports resolve to their local implementations and create no infrastructure.

To add more CDK Blocks, declare them in `cdk-blocks/index.ts`. For the list of CDK Blocks, see *CDK Blocks reference*.

Existing AWS CDK apps

A CDK Block is a CDK Construct, so you can add CDK Blocks to an existing CDK app. Use the `BlocksBackend` construct to put CDK Blocks inside a stack you already have, next to your other resources. To keep CDK Blocks in its own stack, use `BlocksStack`.

```
const blocks = await BlocksBackend.create(stack, 'BlocksApi', {
  backendHandlerPath: './cdk-blocks/index.handler.ts',
  backendCDKPath: './cdk-blocks/index.ts',
  defaults: BlocksPresets.production,
});

topic.grantPublish(blocks.executionRole);
registerConfig(blocks, 'ALERTS_TOPIC_ARN', topic.topicArn);
```

`BlocksBackend.create()` is asynchronous, so call it from a static factory method on your stack. Grant your existing resources to `blocks.executionRole`, the IAM role that all CDK Blocks compute in the app shares. Pass values such as ARNs to your code with `registerConfig`. Run synth with the `cdk` condition, for example `NODE_OPTIONS="--conditions=cdk" npx cdk synth`. For the full example, see the section called "Pattern 1: CDK in CDK Blocks".

To create the `cdk-blocks/` workspace, you can run `npm create @aws-cdk-blocks/blocks-app@latest` in the root of the CDK app. In a project that has a `package.json` and no `amplify/backend.ts`, the command adds the workspace, dependencies, and scripts. If `cdk.json` already exists, the command leaves it unchanged.

To use a CDK Block with a resource your CDK app already deploys, such as an existing table or bucket, see the section called "Pattern 2: fromExisting". To compare CDK Blocks with plain CDK Constructs, see *Choosing AWS CDK Blocks*.

Related resources

- *Existing infrastructure*
- *Choosing AWS CDK Blocks*
- *Local development*
- *Deploy to AWS*
- *CLI reference*

Integrating with existing infrastructure

AWS CDK Blocks is designed to work with existing AWS infrastructure. Whether you have an existing CDK application, pre-deployed resources, or a mix of both, you can adopt CDK Blocks incrementally without rewriting your infrastructure.

Choosing a pattern

CDK Blocks provides five patterns for integrating with existing infrastructure. Choose based on your situation:

Pattern	Effort	Use when	Local support
CDK in CDK Blocks	Low	No CDK Block exists for the resource you need	# Manual
`fromExisting`	Lowest	A CDK Block exists and you have a pre-deployed resource	# Automatic
Custom CDK Block	Medium	You're repeating the same resource pattern in multiple places	# Automatic
Vendorize	Low	A CDK Block that AWS provides is almost right but needs CDK changes	# Automatic
Infrastructure outside the AWS CDK	High	Your team has to manage a resource, or the whole backend, with Terraform or another tool it already uses	# Automatic

Pattern 1: CDK in CDK Blocks

Use raw CDK Constructs alongside CDK Blocks. CDK Blocks gives you two shapes for this:

`BlocksStack` : A standalone CloudFormation stack containing the CDK Blocks Lambda and API Gateway. Use for new projects or when you want CDK Blocks isolated in its own deployment unit.

`BlocksBackend` : A CDK Construct you drop into an existing stack. Use when you already have stacks and want CDK Blocks to live alongside your other resources.

Both expose `executionRole`, the one IAM role that all CDK Blocks compute in the app shares. Grant permissions to it. To pass a value such as a resource ARN to your code, call `registerConfig(scope, key, value)` from `aws-cdk-blocks/cdk`. At runtime, the value is available as `process.env[key]`.

Example: BlocksStack with an SNS topic

This example adds an Amazon Simple Notification Service (Amazon SNS) topic, a resource that has no CDK Block.

```
// cdk-blocks/index.cdk.ts
import * as cdk from 'aws-cdk-lib';
import * as sns from 'aws-cdk-lib/aws-sns';
import { dirname, join } from 'node:path';
import { fileURLToPath } from 'node:url';
import { BlocksPresets, BlocksStack, registerConfig } from 'aws-cdk-blocks/cdk';

const __dirname = dirname(fileURLToPath(import.meta.url));

const app = new cdk.App();
const stack = await BlocksStack.create(app, 'my-app', {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: BlocksPresets.production,
});

const topic = new sns.Topic(stack, 'alerts');
topic.grantPublish(stack.executionRole);
registerConfig(stack, 'ALERTS_TOPIC_ARN', topic.topicArn);
```

```
// cdk-blocks/index.ts
import { Scope, ApiNamespace } from 'aws-cdk-blocks';
import { SNSClient, PublishCommand } from '@aws-sdk/client-sns';

const scope = new Scope('my-app');
const sns = new SNSClient({});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async sendAlert(message: string) {
```

```
    await sns.send(new PublishCommand({
      TopicArn: process.env.ALERTS_TOPIC_ARN,
      Message: message,
    }));
    return { ok: true };
  },
}));
```

Like every `ApiNamespace` method, `sendAlert` is a public endpoint until you add an auth check.

For background work, don't wire up a raw queue or schedule. Use AsyncJob or CronJob, which provision the infrastructure and run locally.

Example: BlocksBackend inside an existing stack

Use a static factory method because `BlocksBackend.create()` is asynchronous and constructors cannot use `await`:

```
// my-existing-stack.ts
import * as cdk from 'aws-cdk-lib';
import * as sns from 'aws-cdk-lib/aws-sns';
import { BlocksBackend, BlocksPresets, registerConfig } from 'aws-cdk-blocks/cdk';

export class MyApiStack extends cdk.Stack {
  static async create(scope: cdk.App, id: string, props?: cdk.StackProps) {
    const stack = new MyApiStack(scope, id, props);

    // Your existing resources
    const topic = new sns.Topic(stack, 'alerts');

    // Drop Blocks in as a Construct
    const blocks = await BlocksBackend.create(stack, 'BlocksApi', {
      backendHandlerPath: './cdk-blocks/index.handler.ts',
      backendCDKPath: './cdk-blocks/index.ts',
      defaults: BlocksPresets.production,
    });

    topic.grantPublish(blocks.executionRole);
    registerConfig(blocks, 'ALERTS_TOPIC_ARN', topic.topicArn);
    return stack;
  }
}
```

Relative `backendHandlerPath` and `backendCDKPath` values resolve against the directory you run synth from.

Run synth with the `cdk` condition, for example `"app": "npx tsx -C cdk bin/app.ts"` in `cdk.json`, or `NODE_OPTIONS="--conditions=cdk" npx cdk synth`. The `npm run deploy` and `npm run sandbox` scripts set it for you. Without it, every CDK Block import resolves to its local implementation and produces no infrastructure, so `create()` stops with a `Missing --conditions=cdk` error.

Tradeoffs

- # Full access to any AWS resource with plain CDK and SDK
- # No local implementation. `npm run dev` will call real AWS unless you stub the SDK yourself
- # No type-level guarantee that config values such as `process.env.ALERTS_TOPIC_ARN` exist

If you find yourself repeating this pattern, consider creating a custom CDK Block instead.

Pattern 2: fromExisting

Some CDK Blocks can wrap a pre-deployed AWS resource. The CDK Block provides its typed API and local implementation, but skips provisioning. You keep ownership of the resource.

These CDK Blocks support it. Pass the factory method's result as the option in the Factory method column, such as `table` for `KVStore`. `AppSetting.fromExisting` is different: it returns the setting itself.

Block	Factory method	Wraps
`KVStore`	`table: KVStore.fromExisting(tableName)`	Existing DynamoDB table. Pass a `key` that matches the table's key schema.
`FileBucket`	`bucket: FileBucket.fromExisting(bucketName)`	Existing S3 bucket

Block	Factory method	Wraps
`Database`	`cluster: DatabaseCluster.fromExisting({ name, ca? })`	Existing PostgreSQL database, with its tables and data. `0` stores its connection string. The deploy runs the files in `migrationsPath` on it. For more information, see Existing PostgreSQL databases.
`Auth`	`userPool: Auth.fromExisting(userPoolId, clientId?)`	Existing Cognito user pool. CDK Blocks adds its own app client to the pool. With `allowBearerAuth: true`, it also accepts access tokens from the app client that you pass as `clientId`.
`AppSetting`	`AppSetting.fromExisting(scope, id, { name })`	Existing AWS Systems Manager Parameter Store parameter
`Secret`	`secret: Secret.fromExisting(secretArn)`	Existing Secrets Manager secret
`Metrics`	`metrics: Metrics.fromExisting(namespace)`	Existing CloudWatch namespace

Example

```
// cdk-blocks/index.ts
import { Scope, KVStore, ApiNamespace } from 'aws-cdk-blocks';
import { z } from 'zod';
```

```
const scope = new Scope('my-app');

const sessions = new KVStore(scope, 'sessions', {
  schema: z.object({ sessionId: z.string(), userId: z.string() }),
  key: { partitionKey: 'sessionId' },
  table: KVStore.fromExisting('my-legacy-sessions-table'),
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async getSession(token: string) {
    return sessions.get({ sessionId: token });
  },
}));
```

Tradeoffs

- # Local development still works. `npm run dev` keeps data in files under `.bb-data/`, not in the real resource. A `Database` on an existing database is the exception: it connects to that database
- # IAM permissions are granted automatically to the shared execution role
- # Same typed API as a resource that CDK Blocks manages
- # Limited to the CDK Block's API surface. If you need features the CDK Block doesn't expose, use Pattern 1
- # Cross-account resources require manual IAM (fall back to Pattern 1)

The wrapped resource can come from any tool.

Pattern 3: Custom CDK Block

When you're repeating Pattern 1 for the same resource type, wrap it in a custom CDK Block. This gives you a typed API, a local implementation, and reusability.

See *Custom CDK Blocks* for a complete guide to authoring CDK Blocks.

When to use

- You have ≥2 callers using the same resource pattern
- You want `npm run dev` to work without AWS for this resource
- You want type-safe call sites instead of raw `process.env` + SDK

Pattern 4: Vendorize

If a CDK Block that AWS provides is *almost* right but needs CDK changes you can't get upstream quickly, you can copy its source into your project:

```
npm run vendorize -- @aws-cdk-blocks/bb-kv-store
```

This copies the CDK Block source into your monorepo. You now own it. Modify the CDK, runtime, or local implementation as needed.

Tradeoffs

- # Full control over the CDK Block implementation
- # Local implementations, types, and API surface all still work
- # You're responsible for maintenance. Upstream updates require manual re-sync
- # Only use when a custom CDK Block (Pattern 3) or an upstream pull request won't solve the problem

You can also vendorize a CDK Block without `fromExisting` to point it at a resource that the CDK did not create. Change its `index.cdk.ts` to import the resource instead of creating it.

Pattern 5: Infrastructure outside the AWS CDK

Each CDK Block's CDK code (`index.cdk.ts`) creates its resources and grants the shared execution role access. It then either names each resource from the CDK Block's `fullId`, so the runtime can derive the same name, or passes names and ARNs to the runtime as config. Nothing else in the CDK Block depends on the CDK: the runtime entry (`index.aws.ts`), the local implementation, local development and the typed client work the same whoever created the infrastructure. So a tool other than the CDK can provide the infrastructure, as long as it provides the same contract.

Two shapes work:

Shape	CDK still owns	Your tool owns
Hybrid	The app shell (execution role, Lambda function, API Gateway) and most CDK Blocks	One or more CDK Blocks' resources , through `fromExisting` (Pattern

Shape	CDK still owns	Your tool owns
		2) or a vendorized CDK Block (Pattern 4)
No CDK	Nothing	The app shell and every CDK Block's resources, grants and config

The contract differs by CDK Block. A KVStore needs only a table with the right name and the key schema of its `key` option (or the string partition key `_bkv_pk` without one), which the execution role can read and write. An AsyncJob needs a queue, config, and an event source mapping on the backend function with settings that match its options. A Database needs config and its migrations run.

When to use

- A resource, or the whole backend, has to be managed by Terraform or another tool your team already uses
- You want to keep the CDK Blocks app code, local implementations and typed client without deploying through the CDK

When not to use

- `fromExisting` (Pattern 2) already covers the resource and the CDK can keep deploying everything else
- You are not prepared to keep names, config keys, IAM grants, and CDK Block options in sync by hand

Tradeoffs

- # Your infrastructure tool manages the resources, with its own state, review and drift detection
- # App code, local implementations and the typed client do not change
- # You keep the contract in sync by hand: derived names, config keys, IAM grants, and options such as AsyncJob's `maxRetries`. A mismatch usually shows up at runtime as an error in the function's logs, not at deploy time

- # With no CDK, you replace `npm run deploy`, `npm run sandbox` and `npm run destroy` with your own steps
- # When a CDK Block changes its contract in a new version, you update your infrastructure to match

Raw HTTP routes and webhooks

`ApiNamespace` methods are JSON-RPC calls from your frontend. When another system must call a fixed URL, such as a payment provider sending webhooks, use `RawRoute`. You get the request and set the response yourself.

```
// cdk-blocks/index.ts
import { createHmac, timingSafeEqual } from 'node:crypto';
import { ApiError, RawRoute, Scope, Secret } from 'aws-cdk-blocks';

const scope = new Scope('my-app');
const signingKey = new Secret(scope, 'webhook-signing-key');

new RawRoute(scope, 'payments-webhook', {
  method: 'POST',
  path: '/webhooks/payments',
  handler: async ({ request, response }) => {
    const body = await request.text();
    const key = await signingKey.get();
    if (!key) throw new ApiError('Webhook signing key is not set', 503);

    const expected = Buffer.from(createHmac('sha256', key).update(body).digest('hex'));
    const received = Buffer.from(request.headers.get('x-signature') ?? '');
    if (received.length !== expected.length || !timingSafeEqual(received, expected)) {
      throw new ApiError('Invalid signature', 401);
    }

    const event = JSON.parse(body) as { type: string };
    console.log('Payment event:', event.type);
    response.status = 200;
    response.send({ received: true });
  },
});
```

`RawRoute` takes these options:

- `method`: `GET`, `POST`, `PUT`, `DELETE`, `PATCH`, `HEAD`, or `OPTIONS`.
- `path`: `/health` matches exactly, `/users/{id}` captures one segment into `request.params.id`, and `/files/*` captures the rest of the path into `request.params['*']`. If you omit `path`, it is built from the IDs of any scopes between your top-level scope and the route, plus the route ID. Set `path` explicitly for URLs that other systems depend on.
- `handler`: Receives `request` (`headers`, `text()`, `json()`, `url`, `params`) and `response` (`status`, `headers`, `send()`). The response `Content-Type` defaults to `application/json`.

CDK Blocks serves the route from the same origin as your API, both in `npm run dev` and when deployed. It needs no extra infrastructure. You cannot register `/`, `/aws-blocks`, or a path under `/aws-blocks/api`, and you cannot register the same method and path twice.

A raw route has no auth of its own, so check a signature or token in the handler. If the handler throws an `ApiError`, the response uses its status and the body `{ "error": message }`. An error that you create with `blocksError()` keeps its message and adds its `name` to the body. Any other error becomes an HTTP 500 with the body `{ "error": "Internal error" }`, and the full error goes only to the server log.

CLI reference

AWS CDK Blocks uses standard npm scripts for all development and deployment workflows. After scaffolding a project, all commands run from your project root.

For more information about CLI commands and options, see CDK Blocks CLI documentation on GitHub.

Create a project

```
npm create @aws-cdk-blocks/blocks-app@latest <project-name> [-- --template <template>]
```

Scaffolds a new CDK Blocks project. The command picks its mode from the target directory:

- **Empty or new directory:** creates a standalone app from a template.
- **Existing project (has a `package.json`):** adds the `cdk-blocks/` backend workspace, `cdk.json`, and the CDK Blocks scripts to your project. Works with any frontend framework.
- **Amplify Gen 2 project (has `amplify/backend.ts`):** adds CDK Blocks to your Amplify backend, using the `amplify` template.

Option	Description
`--template <name>`	Template to use (default: `default`). In an existing project, it selects which `cdk-blocks/` workspace to copy, for example `nextjs` for a Next.js dev server.
`--yes`, `-y`	Skip confirmation prompts
`--skip-install`	Skip installing dependencies
`--help`, `-h`	Show usage and the list of templates

Template	Description
`default`	Vite frontend with auth, KVStore, and Realtime (todo app)

Template	Description
`bare`	Vite frontend with a single `greet()` API (smallest starter)
`react`	Vite and React 19 frontend with auth, KVStore, and Realtime (todo app)
`nextjs`	Next.js 16 App Router with Server and Client Components that call CDK Blocks
`auth`	Vite frontend that tours the `Auth` Block: email sign-up with a code, groups, and profile attributes. `--template auth-cognito` still works and creates this template
`auth-oidc`	Vite frontend with OIDC sign-in, using a local stub identity provider for development
`demo`	Fuller example with Auth, KVStore, FileBucket, AppSetting, Secret, AsyncJob, EmailClient, KnowledgeBase, and Database
`backend`	Backend only: a CDK Blocks API and CDK, no frontend
`api-only`	Headless JSON API with a public health check and CRUD routes that require sign-in, no frontend
`sql`	PostgreSQL backend using the `Database` Block with SQL migrations, foreign keys, and transactions (Aurora DSQL)
`agent`	Vite frontend with an AI agent and a tool, using a canned model for development
`cron`	Backend only: scheduled CronJobs that record each run, no frontend
`observability`	Backend only: Logger, Metrics, Tracer, and a CloudWatch dashboard on one API call, no frontend
`amplify`	Overlay for an existing Amplify Gen 2 app. Selected automatic ally; you cannot use it for a new app.

Local development

```
npm run dev
```

Starts the local development server, which runs your app and every CDK Block on your machine, with hot reload. CDK Blocks use local in-memory and filesystem storage. No AWS account required.

- Templates with a frontend: The dev server serves the frontend and API from a single front door at `http://localhost:3000`. The frontend dev server runs on port 3100 behind it. The API lives under `http://localhost:3000/aws-blocks/api/`, with one path for each exported namespace, for example `/aws-blocks/api/api`.
- `backend`, `api-only`, `sql`, `cron`, and `observability` templates: The dev server serves the API on `http://localhost:3001`.
- Amplify projects (the `amplify` overlay): CDK Blocks does not add an `npm run dev` script. Run `npm run blocks:dev` to serve the API on `http://localhost:3001`, and `npm run blocks:stop` to stop it.

`npm run dev` frees its ports at startup by stopping whatever process listens on them. If a dev server for this project is already running, the new one exits instead. For more information, see the section called “Port 3000 or 3001 already in use”. To stop the dev server from another terminal, run `npm run dev:stop`.

Sandbox deployment

```
npm run sandbox
```

Deploys your own sandbox backend to AWS with a full CloudFormation deployment, which can take several minutes the first time. It then runs `cdk watch`, which hotswaps code changes into the Lambda function, and starts the local dev server against the deployed backend. Each developer gets an isolated sandbox.

```
npm run sandbox:deploy
```

Deploys the sandbox once and exits. Use it in scripts and CI.

```
npm run sandbox:preview
```

Deploys the sandbox with your frontend hosted in preview shape, prints the preview URL, and exits. Only templates with a frontend have this script.

```
npm run sandbox:destroy
```

Deletes every stack the sandbox deployed. A `us-east-1` support stack that holds a Lambda@Edge function cannot be deleted for a few hours after its CloudFront distribution is gone; if the command reports this, run it again later.

Production deployment

```
npm run deploy
```

Deploys every stack in the application to AWS with CDK (CloudFormation), without an approval prompt for IAM changes. It always updates the production stack, `<stackId>-prod`. Add `-- --wait-live` to wait until CloudFront serves the new frontend build. For more information about staged deployments, see the section called “Staged deployments with a pipeline”.

```
npm run destroy
```

Deletes the production stack. It keeps stateful resources such as tables, buckets, and `Auth` user pools. An `Auth` Block with `removalPolicy: 'destroy'` and `deletionProtection: false` loses its user pool. If a delete attempt fails, the command empties the buckets in every stack before it retries. On an app with more than one top-level stack, such as one with a `us-east-1` support stack, the first attempt always fails. Do not use this command there. For more information, see the section called “Full deployment”.

Other project scripts

Projects created from a template also have these scripts. Adding CDK Blocks to an existing project adds only the `dev:server`, `dev:stop`, `sandbox`, `sandbox:destroy`, `sandbox:console`, `deploy`, and `destroy` scripts.

Script	Description
`npm run dev:stop`	Stops the dev server that `npm run dev` started. Add `-- --port <n>` for a port other than the default.

Script	Description
`npm run sandbox:console`	Opens the CloudFormation console's stack list, filtered to the stack from your last deploy, in the Region from your AWS configuration (`us-east-1` if none is set)
`npm run db`	Runs the commands of the `Database` Block. `types --block <id>` writes a `Tables` interface for Kysely from the CDK Block's migration files, and `status --block <id>` lists the applied and pending files in the local database. For an existing database, `connect` stores the connection string and `migrate` applies pending files. Pass the command after a `--` separator, as shown after this table.
`npm run cleanup`	Stops every process with a connection on ports 3000 to 3003 (3001 to 3003 in templates without a frontend), whatever the process is. This includes processes that only connect to those ports, such as a browser.
`npm run secret`	Manages the values for Hosting `secret()` in AWS Secrets Manager with the `set <KEY>`, `list`, and `remove <KEY>` subcommands. These are not `Secret` Block values. `set` prompts for the value, or reads it from standard input with `--value-stdin`. Pass the subcommand after a `--` separator, as shown after this table.
`npm run typegen`	Generates typed keys for `getSecret()` and `getConfig()`. Add `-- --check` in CI to fail when the file is stale.
`npm run spec`	Generates the API spec file, `cdk-blocks/blocks.spec.json`
`npm run typecheck`	Runs the TypeScript compiler without emitting files
`npm run test:e2e`	Runs the template's end-to-end tests

Script	Description
`npm run vendorize`	Copies a CDK Block package's source into your project. Pass the package name after a `--` separator, as shown after this table. For more information, see the section called "Pattern 4: Vendorize".

For example, to set a secret value, to list the migration files of the `Database` Block `main`, and to vendorize the `KVStore` package:

```
npm run secret -- set <KEY>
npm run db -- status --block main
npm run vendorize -- @aws-cdk-blocks/bb-kv-store
```

Related resources

For more information about advanced CLI options, environment variables, and CI/CD configuration, see CDK Blocks CLI reference on GitHub.

Telemetry and identification

The AWS CDK Blocks CLI collects anonymized usage data to help improve the developer experience. This topic explains what data is collected, how it's used, and how to opt out of telemetry collection.

You can review all collected data, understand privacy protections, and choose the opt-out method that works best for your workflow. CDK Blocks CLI telemetry is enabled by default. You can opt out at any time using any of the methods described on this page.

Deployed apps also identify CDK Blocks in two other places: the AWS CloudFormation template metadata and the user agent of requests to AWS services. These are separate from CLI telemetry. For more information, see the section called "CloudFormation template metadata" and the section called "Service request user agent".

CDK Blocks CLI telemetry overview

The CDK Blocks CLI collects anonymized usage data each time you run a command. This data cannot identify you or your organization. No personal information, source code, resource names, or AWS account identifiers are ever collected.

The data helps the CDK Blocks team understand which commands are used most frequently and identify commands that fail often. It also helps prioritize fixes, track adoption of new features and CDK Blocks, optimize performance for common workflows, and understand which environments and tools developers use.

You can review an exhaustive list of the collected data in the following table. This list may change over time as new metrics are added. Any new data collection will follow the same privacy principles.

Data collected

The following table describes all telemetry data that the CDK Blocks CLI collects. It includes the metric name, description, reason for collection, and example values.

Metric	Description	Reason	Example
Telemetry Version	Version of the telemetry event format	This lets us read events correctly when the format changes.	`"1.0.0"`
CDK Blocks Version	Version of the `@aws-cdk-blocks/core` package that ran the command (x.y.z). For `npm create @aws-cdk-blocks/blocks-app`, the version of `@aws-cdk-blocks/create-blocks-app` .	Collecting this helps us bisect our data by version, which can help us identify problematic versions.	`"0.8.0"`
Installation ID	Random UUID stored on local machines executing CLI commands, at	This helps us estimate the distinct number of users who are affected	`"87c785a7-983c-41de-a106-614d5f1d7ac0"`

Metric	Description	Reason	Example
	`~/.blocks/telemetry/installation-id`	by an error. Each unique machine has its own Installation ID. The ID is generated randomly using UUID V4.	
Project ID	Random UUID generated per project, stored as `telemetry.projectId` in `.blocks/config.json` . `npm create @aws-cdk-blocks/blocks-app` sends a new random UUID for each run and does not store it	This helps us understand usage patterns across different projects without identifying the project itself.	`"a5e82e34-c12f-47ee-a507-8244d8f51893"`
Event ID	Random UUID generated for each telemetry event	This creates a unique identifier for each event, helping us deduplicate data.	`"f76343c7-f665-4b83-ad79-01071935f935"`
Timestamp	The time (generated by the client) the event is created	This helps us generate time series graphs and identify when issues occur.	`"2026-06-10T08:08:04.877Z"`
Command Name	The CLI command that was run. Possible values are `create-blocks-app`, `dev`, `sandbox`, `sandbox:destroy` , `deploy`, `destroy`, `cleanup`, `console`, `vendorize`	Along with command state, this information helps us understand feature usage and prioritize improvements.	`"deploy"`, `"dev"`, `"sandbox"`

Metric	Description	Reason	Example
Command State	The result of the command. Possible values are `SUCCESS`, `FAIL`, `ABORTED`	This helps us track error rates on each CLI command and alert us to potential regressions before they are reported by customers.	`"SUCCESS"`
Duration	The length of the command execution in milliseconds	This helps us notice performance patterns and identify bottlenecks in specific commands.	`4500`
Error Code	A categorized error code, if an error occurs	We use this to identify recurring error types and prioritize fixes.	`"CDK_DEPLOY_FAILED"`
Error Phase	Which phase of execution the error occurred in	This helps us narrow down where failures happen in the command lifecycle.	`"deploy"`
Operating System	The OS platform being used	This helps us debug issues that only affect specific operating systems.	`"darwin"`, `"linux"`, `"win32"`
Node Version	The Node.js runtime version used in the environment	This helps us debug any Node.js-specific issues that arise and ensure compatibility.	`"22.12.0"`
CI/CD	Whether or not the CLI command is being invoked from a CI/CD environment	This helps us identify issues specific to CI/CD environments.	`false`

Metric	Description	Reason	Example
Package Manager	The package manager user agent string	This helps us understand the developer toolchain and debug package-manager-specific issues.	`"npm/10.8.2 node/v22.12.0"`
Agent	Our best guess at whether a coding agent is executing the command, and which one	This helps us debug issues that affect coding agent workflows.	`"amazon-q"`
Framework	The frontend framework in use, for `deploy` and `sandbox`. Possible values are `nextjs`, `nitro`, `nuxt`, `astro`, `sveltekit` , `spa`, `static`. Projects without a frontend report none.	This helps us prioritize framework-specific improvements.	`"nextjs"`, `"spa"`, `"static"`
Template	The project template name and version used with `npm create @aws-cdk-blocks/blocks-app`	This helps us understand which starter templates are popular and maintain them accordingly.	`{"name": "nextjs", "version": "0.1.0"}`
Blocks	The names and versions of the CDK Blocks that AWS provides and that the project uses. Names of custom CDK Blocks are never sent.	This helps us understand CDK Block adoption and prioritize development of popular CDK Blocks.	`[{"name": "KVStore", "version": "0.4.0"}, {"name": "Auth", "version": "0.1.0"}]`

Metric	Description	Reason	Example
Block Count	The total number of CDK Blocks in the project	This helps us understand project complexity and plan for scale.	3
Custom Blocks	The number of custom (user-authored) CDK Blocks	This helps us track custom CDK Block adoption and improve the authoring experience.	1

AWS CDK CLI telemetry

Some CDK Blocks commands (such as `sandbox`, `deploy`, and `destroy`) use the AWS CDK CLI under the hood. When this happens, the AWS CDK CLI also collects its own telemetry data on behalf of AWS CDK.

For information about AWS CDK CLI telemetry collection, see Configure AWS CDK CLI telemetry in the *AWS CDK Developer Guide*.

> **Note**
>
> When you disable CDK Blocks CLI telemetry using any of the methods described on this page, telemetry for CDK Blocks-spawned CDK CLI invocations is also disabled automatically. You do not need to separately configure the CDK CLI opt-out.

Data NOT collected

CDK Blocks does **not** collect:

- Your AWS account ID or credentials
- Source code, file contents, or resource names
- File names or directory paths
- IP addresses (not logged server-side)
- Environment variable values

- Any personally identifiable information (PII)

How to opt out of CDK Blocks CLI telemetry

You can disable telemetry using any of the following methods. If any method indicates telemetry is disabled, no data is collected.

An enable setting never overrides a disable from another method. For example, if you disabled telemetry globally, `npx blocks-telemetry --enable` in one project leaves telemetry off. To turn it back on, run `npx blocks-telemetry --enable --global`.

Opt out using the CLI

You can use the CDK Blocks CLI to check and modify your telemetry settings.

To check whether telemetry is currently enabled or disabled, enter the following command:

```
npx blocks-telemetry --status
```

To disable telemetry for the current project, enter the following command:

```
npx blocks-telemetry --disable
```

To disable telemetry globally, enter the following command:

```
npx blocks-telemetry --disable --global
```

To re-enable telemetry for the current project, enter the following command. Add `--global` to re-enable it globally.

```
npx blocks-telemetry --enable
```

Opt out using an environment variable

To disable telemetry using an environment variable, set `AWS_BLOCKS_DISABLE_TELEMETRY` to `1`, `true`, or `yes` (not case-sensitive). The following example sets it to `1`:

```
export AWS_BLOCKS_DISABLE_TELEMETRY=1
```

> **Tip**
>
> Add this environment variable to your shell startup file, such as `.bashrc` or `.zshrc`, to persist the setting across sessions.

This method works well for CI/CD pipelines. Use it to disable telemetry without modifying project files.

Opt out using a configuration file

To disable telemetry for a specific project, set `telemetry.enabled` to `false` in the project's `.blocks/config.json` file:

```
{
  "telemetry": {
    "enabled": false
  }
}
```

This disables telemetry for the current project only.

To disable telemetry globally, set the same value in `~/.blocks/config.json`:

```
{
  "telemetry": {
    "enabled": false
  }
}
```

This disables telemetry for all projects on this machine.

Send telemetry data to a local file

If you stay opted in to CDK Blocks CLI telemetry, you can send your telemetry data to a local file. You can then review the file to audit the telemetry data that we are collecting.

Use the `--telemetry-file` option on any CDK Blocks CLI command to send telemetry data to a local file:

```
--telemetry-file='path/to/local/file'
```

As an example, this deploy command sends telemetry data to a local file:

```
npm run deploy -- --telemetry-file='/tmp/blocks-telemetry.json'
```

The file holds one JSON array, with one object for each telemetry event the command sends. Use a path that does not exist yet. If the file already exists, CDK Blocks does not write to it.

The option has no effect when telemetry is disabled. To capture events without sending them, leave telemetry on and point the endpoint at an unused local port:

```
BLOCKS_TELEMETRY_ENDPOINT=http://127.0.0.1:1 npm run deploy -- --telemetry-file='/tmp/
blocks-telemetry.json'
```

CloudFormation template metadata

When you synthesize or deploy an app, CDK Blocks adds an entry to the metadata of the app's AWS CloudFormation template. The entry identifies the CDK Blocks version and the CDK Blocks the app uses. It looks like this:

```
{
  "Metadata": {
    "AWSToolsMetrics": {
      "AWSBlocks": "1;core=0.8.0;mode=production;bb=KVStore,LambdaCompute;custom=0"
    }
  }
}
```

The value has these fields:

Field	Description
1	Version of the value format
core	Version of @aws-cdk-blocks/core
mode	sandbox for a sandbox deployment, production otherwise

Field	Description
bb	Names of the CDK Blocks in the app that ship with CDK Blocks, sorted and comma-separated. `LambdaCompute` is the default compute that every app gets.
`custom`	Number of custom CDK Block instances. Their names are never written.

CDK Blocks writes the entry only to the top-level stack, not to nested stacks. It also adds a `blocks:deployment-type` tag with the value `sandbox` or `production` to the stack.

The CLI telemetry opt-out does not affect the template metadata. To turn off the template metadata, use any of the following:

- Set the environment variable `AWS_BLOCKS_DISABLE_TEMPLATE_METADATA` to `1`, `true`, or `yes` before you synthesize or deploy.
- Set the CDK context value `blocks:template-metadata` to `false`, for example in the `context` section of `cdk.json`.
- Turn off CDK version reporting, for example with `"versionReporting": false` in `cdk.json` or the `--no-version-reporting` option of the CDK CLI.

The `blocks:deployment-type` tag is always added.

Service request user agent

Deployed CDK Blocks that call AWS services through the AWS SDK identify themselves in the user agent of those requests. The user agent includes `aws-blocks/<core version>` and a `bb/<Block>/<version>` entry for the CDK Block making the request. It also includes one entry for each CDK Block that the requesting CDK Block is nested in, counting only CDK Blocks that ship with CDK Blocks. For example, a `KVStore` request carries `aws-blocks/0.8.0 bb/KVStore/0.4.0`, and a request from the session `KVStore` inside an `Auth` Block carries `aws-blocks/0.8.0 bb/Auth/0.1.0 bb/KVStore/0.4.0`. Names of custom CDK Blocks are never included.

The native clients for Kotlin, Swift, and Dart send an `x-blocks-user-agent` header in the form `aws-blocks-<language>/<version>` with each API call. Dart apps running in a web browser do not send it. The backend checks the header and adds it to the AWS SDK user agent as `client/<language>/<version>`. The backend drops a malformed value.

There is no setting to turn off the user agent entries.

Versioning

This topic describes how AWS CDK Blocks handles versioning.

CDK Blocks version numbers have three numeric parts: `MAJOR.MINOR.PATCH`. They follow Semantic Versioning, with the exceptions described in the section called "Exceptions". Breaking changes are limited to major releases.

Minor and patch releases are backward compatible. You can upgrade code written for an earlier version to a newer version with the same major version, and it continues to build and run.

Version numbers

The following list describes each part of a version number.

- `MAJOR` (`1.2.3` to `2.0.0`): A major release can include changes that break existing code. Release notes describe what changed and how to update.
- `MINOR` (`1.2.3` to `1.3.0`): A minor release adds new, backward-compatible features. Existing code continues to work.
- `PATCH` (`1.2.3` to `1.2.4`): A patch release contains backward-compatible bug fixes.

Versions earlier than `1.0.0` are initial development releases. In these versions, a minor release can include breaking changes.

Exceptions

In rare cases, a patch or minor release can include a change that is not backward compatible. This happens only in the following situations:

- **Security fixes:** A fix for a security issue can change behavior that your code relies on.
- **Short-lived bugs:** A bug introduced in a recent release and fixed soon after can be corrected in a way that breaks code that depended on the faulty behavior.

Such changes are described in the release notes.

Upgrade your packages

You can avoid unexpected changes by committing your lock file and upgrading deliberately. For more information about changes in each release, see the releases page on the GitHub website.

Examples and patterns

This topic shows common application patterns you can build with AWS CDK Blocks. Each example demonstrates how CDK Blocks compose together to solve real-world problems.

For complete projects you can start from, see the CDK Blocks project templates on GitHub.

CRUD API with authentication

A typical data-driven application with user authentication and scoped data access. The app stores notes in a `KVStore` with a `key` whose partition field is the user ID, so listing a user's notes reads only that user's items. A `scan()` would read every item in the table.

```
import { ApiNamespace, Scope, KVStore, Auth } from 'aws-cdk-blocks';
import { z } from 'zod';

const scope = new Scope('notes-app');
const auth = new Auth(scope, 'auth');
export const authApi = auth.createApi();

const notes = new KVStore(scope, 'notes', {
  schema: z.object({
    userId: z.string(), // partition key: one partition per user
    id: z.string(),     // sort key
    title: z.string(),
    body: z.string(),
    createdAt: z.number(),
  }),
  key: { partitionKey: 'userId', sortKey: 'id' },
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async create(title: string, body: string) {
    const user = await auth.requireAuth(context);
    const note = { userId: user.userSub, id: crypto.randomUUID(), title, body,
 createdAt: Date.now() };
    await notes.put({ userId: note.userId, id: note.id }, note);
    return note;
  },

  async list() {
```

```
    const user = await auth.requireAuth(context);
    // Reads only this user's partition, not the whole table
    return Array.fromAsync(notes.query({ where: { userId: { equals:
 user.userSub } } }), (entry) => entry.value);
  },

  async delete(id: string) {
    const user = await auth.requireAuth(context);
    await notes.delete({ userId: user.userSub, id });
  },
}));
```

CDK Blocks used: `Auth`, `KVStore`, `ApiNamespace`

Realtime chat

A chat application with WebSocket pub/sub and message history.

```
import { ApiNamespace, Scope, KVStore, Auth, Realtime } from 'aws-cdk-blocks';
import { z } from 'zod';

const scope = new Scope('chat-app');
const auth = new Auth(scope, 'auth');
export const authApi = auth.createApi();

const messageSchema = z.object({ id: z.string(), user: z.string(), text: z.string(),
 ts: z.number() });

// One partition per chat room; the sort key starts with the timestamp, so messages
 sort by time
const messages = new KVStore(scope, 'messages', {
  schema: messageSchema.extend({ room: z.string() }),
  key: { partitionKey: 'room', sortKey: 'id' },
});

const realtime = new Realtime(scope, 'realtime', {
  namespaces: { chat: Realtime.namespace(messageSchema) },
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async sendMessage(text: string) {
    const user = await auth.requireAuth(context);
```

```
    const ts = Date.now();
    const msg = { id: `${ts}-${crypto.randomUUID()}`, user: user.username, text, ts };
    await messages.put({ room: 'general', id: msg.id }, msg);
    await realtime.publish('chat', 'general', msg);
    return msg;
  },

  async getHistory() {
    await auth.requireAuth(context);
    // The 50 newest messages, read from one partition
    const latest = await Array.fromAsync(
      messages.query({ where: { room: { equals: 'general' } }, order: 'desc', limit:
 50 }),
      (entry) => entry.value,
    );
    return latest.reverse();
  },

  async subscribeChat() {
    await auth.requireAuth(context); // Ensure caller is authenticated
    return realtime.getChannel('chat', 'general');
  },
}));
```

Frontend integration

The frontend subscribes to realtime updates:

```
import { api } from 'cdk-blocks';

// Subscribe to messages via API method
const channel = await api.subscribeChat();
channel.subscribe((msg) => {
  console.log(`${msg.user}: ${msg.text}`);
});

// Send a message
await api.sendMessage('Hello, world!');

// Load history
const history = await api.getHistory();
```

CDK Blocks used: `Auth`, `KVStore`, `Realtime`, `ApiNamespace`

File uploads with metadata

An application that lets users upload files from the browser and lists them with their size and upload time.

API methods take and return JSON, so do not pass file contents through them. Instead, an API method returns a file handle. The browser uses the handle to upload or download the file directly with a presigned URL. You can also return a presigned URL yourself with `putUrl()` or `getUrl()`.

```
import { ApiNamespace, ApiError, Scope, FileBucket, Auth } from 'aws-cdk-blocks';

const scope = new Scope('files-app');
const auth = new Auth(scope, 'auth');
export const authApi = auth.createApi();

const files = new FileBucket(scope, 'uploads', {
  // Lets the deployed frontend upload and download with the presigned URLs
  corsRules: [{
    allowedOrigins: ['http://localhost:3000', 'https://app.example.com'],
    allowedMethods: ['GET', 'PUT'],
    allowedHeaders: ['Content-Type'],
  }],
});

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async startUpload(name: string, contentType: string) {
    const user = await auth.requireAuth(context);
    // The browser sends the file straight to storage with this handle
    return files.createUploadHandle(`${user.userSub}/${name}`, { contentType });
  },

  async listFiles() {
    const user = await auth.requireAuth(context);
    const results = [];
    for await (const file of files.scan({ prefix: `${user.userSub}/` })) {
      results.push({ key: file.path, size: file.size, lastModified:
 file.lastModified.toISOString() });
    }
    return results;
  },

  async download(key: string) {
    const user = await auth.requireAuth(context);
```

```
    if (!key.startsWith(`${user.userSub}/`)) {
      throw new ApiError('Forbidden', 403, { name: 'ForbiddenError' });
    }
    return files.getFileHandle(key);
  },
}));
```

The frontend uploads and downloads with the handles:

```
import { api } from 'cdk-blocks';

// Upload: the file goes from the browser to storage, not through your API
const upload = await api.startUpload(file.name, file.type);
await upload.upload(file);

// Download
const [first] = await api.listFiles();
const download = await api.download(first.key);
const blob = await download.download();
```

A deployed bucket has no cross-origin resource sharing (CORS) rules by default, and it does not inherit the stack's allowed origins. Without `corsRules` that list your frontend origin, browser uploads work locally but fail after you deploy. A rule that combines a wildcard origin (*) with PUT, POST, or DELETE fails synthesis.

CDK Blocks used: `Auth`, `FileBucket`, `ApiNamespace`

SQL-backed application

An application using relational data with joins and transactions. The `cdk-blocks/migrations/db/` folder, named after the CDK Block ID, holds numbered `.sql` files that create the tables. They run once each, before the first query locally and during the deploy in AWS. For a complete project, see the `sql` template.

```
import { ApiNamespace, Scope, Database, Auth, sql } from 'aws-cdk-blocks';

const scope = new Scope('store-app');
const auth = new Auth(scope, 'auth');
export const authApi = auth.createApi();

// Runs the numbered .sql files in cdk-blocks/migrations/db/ once each, in order.
```

```
// The directory is named after the Block ID and is relative to the project root.
const db = new Database(scope, 'db');

interface Order {
  id: number;
  user_id: string;
  status: string;
  created_at: string;
}

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createOrder(items: Array<{ productId: string; qty: number }>) {
    const user = await auth.requireAuth(context);

    return db.transaction(async (trx) => {
      const [order] = await trx.query<Order>(sql`
        INSERT INTO orders (user_id, status)
        VALUES (${user.userSub}, 'pending')
        RETURNING *
      `);

      for (const item of items) {
        await trx.execute(sql`
          INSERT INTO order_items (order_id, product_id, quantity)
          VALUES (${order.id}, ${item.productId}, ${item.qty})
        `);
      }

      return order;
    });
  },

  async getOrders() {
    const user = await auth.requireAuth(context);
    return db.query<Order>(sql`
      SELECT * FROM orders
      WHERE user_id = ${user.userSub}
      ORDER BY created_at DESC
    `);
  },
}));
```

The migration creates both tables:

```
-- cdk-blocks/migrations/db/001_create_orders.sql
CREATE TABLE orders (
  id         BIGINT GENERATED BY DEFAULT AS IDENTITY (CACHE 65536) PRIMARY KEY,
  user_id    TEXT NOT NULL,
  status     TEXT NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE order_items (
  order_id   BIGINT NOT NULL REFERENCES orders (id),
  product_id TEXT NOT NULL,
  quantity   INTEGER NOT NULL
);

CREATE INDEX ASYNC orders_user_id ON orders (user_id);
```

CDK Blocks used: `Auth`, `Database`, `ApiNamespace`

Common patterns

Pattern	Approach
User-scoped data	Use `user.userSub` as the partition key (or key prefix) to isolate data per user
Optimistic UI	Return the created/updated object from API methods so the frontend can update immediately
Background work	Use `AsyncJob` for queued work and `CronJob` for scheduled work
Multi-tenant	Use Scope IDs or key prefixes to partition data by tenant
Feature flags	Store flags in an `AppSetting` and read them in API methods before executing logic

Related resources

For runnable projects:

- CDK Blocks project templates on GitHub used by `npm create @aws-cdk-blocks/blocks-app@latest`
- Test fixtures in the CDK Blocks repository. These are small apps the repository uses for its own tests, and most have no README.

Security in AWS CDK Blocks

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

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

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. AWS CDK Blocks is a framework that you install from npm. It runs in your build and in your AWS account, so compliance coverage comes from the AWS services that it provisions for your app. To learn which compliance programs apply to those services, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using CDK Blocks.

How CDK Blocks manages permissions

CDK Blocks configures IAM permissions for the resources your application uses. When you create a CDK Block, it grants access to its own resource on one IAM role, the execution role.

All compute in a `BlocksStack` or `BlocksBackend` runs as this one role. That includes the default API function, every `Compute` you create, and the handlers for queues, schedules, and WebSocket connections. A `Compute` that runs one `ApiNamespace` or a whole `Scope` gets its own function, memory, and scaling, but it still runs as the shared role. If you move code to its own `Compute`, that code can still reach the same resources.

Because the role is shared, every API method in the stack can reach the resources of every CDK Block in the stack, not only the CDK Blocks it uses. To keep two workloads from reaching each other's data, deploy them as separate stacks, or in separate AWS accounts.

Most grants are scoped to the CDK Block's own resource, but they allow more actions than the CDK Block calls. `EmailClient`, secret `AppSetting`, and existing database `Database` grants are wider, as the table shows. The following table lists the grants for common CDK Blocks.

Block	Actions granted to the execution role	Resource
`KVStore`	`dynamodb:GetItem`, `BatchGetItem`, `Query`, `Scan`, `ConditionCheckItem`, `PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, `DescribeTable`, `GetRecords`, `GetShardIterator`. With a customer managed key, also the KMS actions to use that key.	The table, and `dynamodb:Query` on its indexes
`FileBucket`	`s3:GetObject*`, `s3:PutObject`, `s3:PutObjectTagging`, `s3:PutObjectVersionTagging`, `s3:PutObjectRetention`, `s3:PutObjectLegalHold`, `s3:DeleteObject*`, `s3:List*`, `s3:GetBucket*`, `s3:Abort*`. With a KMS key, also the KMS actions to use that key.	The bucket and its objects
`Database`	`dsql:DbConnect` on a `dsql` cluster, or `rds-db:connect` on an `aurora` cluster. The role connects as a database role that can only read and write data. On an existing database with `DatabaseCluster.fromExisting({ name })`, `ssm:GetParameter` instead, plus `kms:Decrypt` on any key for calls through SSM.	The cluster, the app's database role on an `aurora` cluster, or the parameter that holds the connection string
`Secret`	`secretsmanager:GetSecretValue`, `DescribeSecret`, `ListSecretVersionIds`. With `allowRuntimeWrite: true`, also `PutSecretValue`, `UpdateSecret`, and `UpdateSecretVersionStage`.	The secret

Block	Actions granted to the execution role	Resource
`AppSetting`	`ssm:GetParameter` and `ssm:PutParameter`. With `secret: true`, also `kms:Decrypt` and `kms:Encrypt`.	The parameter. KMS access is limited to calls made through Parameter Store, or to your key if you set `kmsKeyArn`.
`EmailClient`	`ses:SendEmail`, `ses:SendBulkEmail`	All Amazon Simple Email Service (Amazon SES) identities and configuration sets in the account
`AsyncJob`	`sqs:SendMessage`, `ReceiveMessage`, `DeleteMessage`, `ChangeMessageVisibility`, `GetQueueAttributes`, `GetQueueUrl`	The job's queue

The role also has the `AWSLambdaBasicExecutionRole` managed policy and read access to the app's configuration file in Amazon S3. Authentication, `Realtime`, `Agent`, `KnowledgeBase`, and other CDK Blocks add their grants to the same role. `CronJob` invokes your compute through its own scheduler role.

See the full policy

Every `cdk synth` writes a report of the execution role to `cdk.out/<stackName>.blocks-iam.json`, or `cdk.out/<stackName>.<backendId>.blocks-iam.json` for a `BlocksBackend` inside your own stack. The report lists every statement on the role, the CDK Blocks that own the resources each statement references (empty when it names only literal ARNs or *), the managed policies, the permissions boundary, and every IAM role that CDK Blocks created. It is not deployed. The file is sorted, so a CI step can compare it with the previous build and fail when permissions grow.

```
npx cdk synth --quiet
jq '.executionRole.statements[] | {actions, blocks}' cdk.out/my-app.blocks-iam.json
```

Grant access to your own resources

You don't need to write IAM policies for CDK Block resources. If you add your own CDK resources in the CDK layer, you grant their permissions yourself. Grant them to `stack.executionRole`, where `stack` is the stack that `BlocksStack.create(…)` returns (`blocksStack` in the project templates). Pass the resource's name or ARN to your code with `registerConfig`. Every compute reads it from `process.env` at startup.

```
// cdk-blocks/index.cdk.ts
import * as sns from 'aws-cdk-lib/aws-sns';
import { registerConfig } from 'aws-cdk-blocks/cdk';

const topic = new sns.Topic(blocksStack, 'Alerts');
topic.grantPublish(blocksStack.executionRole);
registerConfig(blocksStack, 'ALERTS_TOPIC_ARN', topic.topicArn);
```

In your backend code, read the ARN with `process.env.ALERTS_TOPIC_ARN`.

Cap every role with a permissions boundary

Set `defaults.permissionsBoundary` to attach an IAM permissions boundary to every role that CDK Blocks creates. This covers the execution role, the helper roles that CDK Blocks create, the CDK custom resource roles, and the roles that `Hosting` creates. A boundary is a ceiling, not a grant. The role's effective permissions are what both its policies and the boundary allow. Pass a policy name, a full policy ARN (needed when the policy has a path), or an `iam.IManagedPolicy`.

The examples on this page keep the template's preset choice, `sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production`, and add settings to it. If you spread `BlocksPresets.production` alone, sandboxes also get production settings, such as no `localhost` CORS origins and retained resources.

```
// cdk-blocks/index.cdk.ts
export const blocksStack = await BlocksStack.create(app, stackName, {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: { ...(sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production),
 permissionsBoundary: 'my-app-boundary' },
});
```

A boundary limits the shared role, but it does not split it. Every handler still runs as the same role.

Authentication

CDK Blocks provides the `Auth` Block to handle user identity. Sessions are kept server side and sent as HTTP-only cookies. Signing out deletes the session, and a password reset ends every session the user had.

Default store

Uses an Amazon Cognito user pool. Supports email and password sign-in, multi-factor authentication (MFA) with SMS, TOTP, or email codes, passkeys, user groups, and account recovery flows. Email codes need an Amazon Simple Email Service (Amazon SES) sender, set with the `email` option. Social, SAML, and OIDC providers sign in through Cognito, and each user becomes a user pool user.

`store: 'builtin'`

Creates no user pool. Your backend verifies each OIDC or OAuth 2.0 sign-in itself and keeps only the session.

For more information, see the section called “Authentication”.

Best practices:

- Define namespaces that hold signed-in methods with `auth.protect`, so a forgotten or unawaited guard can’t leave a method open.
- Every `ApiNamespace` method is a public endpoint until it calls `requireAuth`. The local implementation applies no authentication, so a method that you forget to protect passes every local check. Call `auth.requireAuth(context)` server-side in every API method that accesses user data. Never trust client-side authentication alone.
- Scope data access by `user.userSub` to prevent unauthorized cross-user access. It stays the same for the user’s lifetime.
- Mark attributes that you use for authorization, such as a role or tenant ID, with `userWritable: false`, so end users can’t write them.
- Set `removalPolicy: 'retain'` on any `Auth` Block whose user pool holds real users.

Data protection

CDK Blocks uses the following data protection mechanisms:

Encryption at rest

CDK Blocks encrypt stored data at rest as follows:

- `KVStore`: DynamoDB encryption with the AWS managed key `aws/dynamodb`. Set `encryption: 'customer-managed'` to create a customer managed key for the table, or set `encryption: KVStore.fromKmsKey(keyArn)` to use a key that you already own.
- `FileBucket`: S3 server-side encryption with Amazon S3 managed keys (SSE-S3). The same `encryption` option selects a KMS key instead. `FileBucket` also blocks all public access and rejects requests that do not use TLS. For records that must not change, set `objectLock: { mode, retentionDays }`. To log requests to a bucket, set `accessLogging: true` on the bucket, or `defaults.storageAccessLogging: true` for every bucket in the stack.
- `Database`: Aurora DSQL encryption with an AWS owned key on a `dsql` cluster. An `aurora` cluster also encrypts its storage with an AWS owned key, because the express configuration accepts no customer managed KMS key. An existing database keeps the encryption that you configured for it.
- `Secret`: AWS Secrets Manager encryption with the `aws/secretsmanager` key. `AppSetting` with `secret: true`: Parameter Store SecureString with the `aws/ssm` key, or your key if you set `kmsKeyArn`.

Encryption in transit

All communication between your application and AWS services uses TLS. API Gateway endpoints use HTTPS exclusively.

Local development

During local development, data is stored unencrypted in the `.bb-data/` directory on your machine. This includes the values of `Secret` and secret `AppSetting` Blocks. This directory should be added to `.gitignore` (which CDK Blocks does by default) and should not contain production data.

Network security

By default, CDK Blocks deploys your API behind Amazon API Gateway, which provides:

- HTTPS-only endpoints (HTTP is not supported)
- DDoS protection via AWS Shield Standard
- Request throttling, which CDK Blocks turns on for the backend API stage and the `Realtime` WebSocket stage. For more information, see the section called “API protection”.

Front door

The `apiFrontDoor` option on `BlocksStack.create` or `BlocksBackend.create` sets how clients reach the API. With `'apigateway'` (the default), clients call the API Gateway endpoint directly. With `'cloudfront'`, CDK Blocks puts an Amazon CloudFront distribution in front of the API. Either way, clients should call `publicApiUrl`, which is also the `ApiUrl` stack output. When you pass the backend to `Hosting`, the browser reaches the API on the same origin through the hosting front door, except with `frontDoor: 'none'`, where it calls the API cross-origin. Neither option makes CloudFront the only way in: the API Gateway `execute-api` URL stays reachable. Changing the option later changes the API’s domain and signs out every user, so choose it before you have real users.

Web application firewall

AWS WAF is off by default. Set `apiWaf: { enabled: true }` to create a web ACL with the AWS managed Common and Known Bad Inputs rule groups and a rate limit for each IP address (`rateLimit`, default 1000 requests in 5 minutes). Or pass `apiWaf: { webAclArn }` to attach a web ACL that you already have. Under `apiFrontDoor: 'cloudfront'`, the web ACL must be in `us-east-1`, so outside that Region pass `webAclArn`. `apiWaf` does not cover the `Realtime` WebSocket API. Browser traffic that arrives through `Hosting` is not rate limited by `apiWaf`; turn on the WAF of the hosting front door for that. When `Hosting`’s CloudFront distribution serves the API under `apiFrontDoor: 'cloudfront'`, `apiWaf` creates no web ACL at all, and `cdk synth` warns about it.

Access logs

API Gateway access logging is off in both presets, and `cdk synth` warns about this for the production preset. Turn it on with `defaults.accessLogging: true`. It sets the API Gateway logging role, which the whole account shares in each Region, so turn it on for only one CDK Blocks stack per Region. The same setting also turns on Amazon S3 server access logs for every `FileBucket`, each with its own log bucket. To keep those off, also set `defaults.storageAccessLogging: false`. The following example turns on access logs and the WAF.

```
// cdk-blocks/index.cdk.ts
export const blocksStack = await BlocksStack.create(app, stackName, {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: { ...(sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production),
 accessLogging: true },
  apiWaf: { enabled: true, rateLimit: 2000 },
});
```

VPC

For additional network isolation, place your compute in a virtual private cloud (VPC) with `defaults.vpc`. CDK Blocks puts your compute functions in the VPC's private subnets, creates the VPC endpoints that your CDK Blocks need, and wires the security groups. A VPC adds cost, such as network address translation (NAT) gateways and interface endpoints, so use one only when you need it.

```
// cdk-blocks/index.cdk.ts (the template already imports cdk)
import * as ec2 from 'aws-cdk-lib/aws-ec2';

// A VPC must live in a stack. Create it in a small network stack.
const network = new cdk.Stack(app, `${stackName}-network`);
const vpc = new ec2.Vpc(network, 'AppVpc', { maxAzs: 2, natGateways: 1 });

export const blocksStack = await BlocksStack.create(app, stackName, {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: { ...(sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production),
 vpc: { network: vpc } },
});
```

API protection

Input validation

Always validate inputs in your API methods. CDK Blocks does not perform automatic input validation. Your API methods receive whatever the client sends. To reject bad input, throw `ApiError(message, 400)`. A plain `Error` thrown in an API method reaches the client as a nameless error with status 500 and the message "Internal error", so your message is lost.

To let the frontend check for a specific error, pass a name, as in `new ApiError(message, 404, { name: 'NotFoundError' })`, and check it with `isBlocksError` from `aws-cdk-blocks/client`.

```
// cdk-blocks/index.ts
import { ApiNamespace, ApiError } from 'aws-cdk-blocks';

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async createItem(title: string) {
    if (!title || title.length > 200) {
      throw new ApiError('Title must be 1-200 characters', 400);
    }
    // ...
    return { title };
  },
}));
```

Rate limiting

CDK Blocks sets throttling on the backend API stage and the `Realtime` WebSocket stage from `defaults.throttling`. The API Gateway front doors that `Hosting` creates do not get these limits. The production preset allows 1,000 requests per second with a burst of 2,000. The sandbox preset allows 200 with a burst of 400. To change the limits, override `throttling`, as shown in the following example. For a limit on each client, use the rate rule in `apiWaf`.

CORS

Your API accepts cross-origin requests only from the origins in `defaults.allowedOrigins`, plus the hosting front door's origin when you use `Hosting`. Each entry in `allowedOrigins` is a regular expression, so escape the dots. The production preset lists no origins. The sandbox preset allows `localhost` and `127.0.0.1`, so a local frontend can call a deployed sandbox. With `Hosting`, browser calls through the hosting front door use the same origin, so you don't need to configure CORS. If you host your frontend somewhere else, add its origin.

```
// cdk-blocks/index.cdk.ts
export const blocksStack = await BlocksStack.create(app, stackName, {
  backendHandlerPath: join(__dirname, 'index.handler.ts'),
  backendCDKPath: join(__dirname, 'index.ts'),
  defaults: {
    ...(sandboxMode ? BlocksPresets.sandbox : BlocksPresets.production),
    throttling: { rateLimit: 2000, burstLimit: 4000 },
```

```
    allowedOrigins: ['https://app\\.example\\.com'],
  },
});
```

Managing secrets

Don't hardcode secrets (API keys, database passwords, third-party tokens) in your source code. Don't put them in Lambda environment variables or in `registerConfig` either. Anyone who can read the function's configuration, or the app's configuration file, can read those values. Store production secrets in the `Secret` Block. Each `Secret` is one AWS Secrets Manager secret, and your code reads it when it needs the value.

```
// cdk-blocks/index.ts
import { ApiNamespace, ApiError, Scope, Secret } from 'aws-cdk-blocks';

const scope = new Scope('my-app');
const paymentKey = new Secret(scope, 'payment-api-key');

export const api = new ApiNamespace(scope, 'api', (context) => ({
  async charge(amount: number) {
    const key = await paymentKey.get();
    if (key === null) {
      throw new ApiError('Payments are not configured', 503);
    }
    // Call the payment provider with key ...
    return { charged: amount };
  },
}));
```

A `Secret` is read-only at runtime by default. The object has no `put()` method, and the execution role can only read the secret. `get()` returns `null` until a value is set. Set the value outside your app: from your stack's settings resource group in the AWS Resource Groups console, which the app's `/aws-blocks/settings` route opens, or with `aws secretsmanager put-secret-value`. The default name comes from the stack and CDK Block IDs, so the CLI works without a `name`. Set `name` if you want one that does not depend on them. Your app reads a new value within 5 minutes, or the time you set with `cacheTtlSeconds`.

If your app sets or rotates the value itself, pass `allowRuntimeWrite: true`. Then `put()` is available and the role can write the secret:

```
const token = new Secret(scope, 'service-token', { allowRuntimeWrite: true });
await token.put('new-value');
```

For a random value that only your app uses, such as a signing key, pass `generate: true`.

`AppSetting` with `secret: true` stores the value as a Parameter Store SecureString. It is a free fallback for a secret of little value in a prototype. For any secret that protects a production app, use `Secret`.

Least privilege for deployment

The IAM user or role that deploys your CDK Blocks application needs permissions to create and manage the underlying resources (Lambda, DynamoDB, API Gateway, CloudFormation, etc.). We recommend:

- Use a dedicated deployment role with scoped permissions.
- Don’t deploy with your personal administrator credentials in production.
- Use CI/CD pipelines with OIDC-based authentication (no long-lived access keys).
- If your deployment role requires a permissions boundary on every role it creates, set `defaults.permissionsBoundary`. For more information, see the section called “How CDK Blocks manages permissions”.

Troubleshooting

Local development issues

Port 3000 or 3001 already in use

Symptom: Another program uses port 3000 or 3001, or a process on that port stops after you run `npm run dev`.

`npm run dev` frees its ports when it starts. It stops whatever process listens on its API port (3000, or 3001 for templates without a frontend) and on its frontend port (3100). It sends `SIGTERM`, then `SIGKILL` if the port is still held, and prints `Reclaimed port <n>`. This happens whatever the process is, so a different app on one of those ports is stopped too. Only a running AWS CDK Blocks dev server for the same project is left alone: a second `npm run dev` exits and tells you to run `npm run dev:stop` first.

Solution: To keep another program on those ports, run the dev server on different ports. `--port` sets the API port, and `--frontend-port` sets the frontend port:

```
npm run dev -- --port 3002 --frontend-port 3102
```

To stop a dev server on a port other than the default, pass the port to `npm run dev:stop`:

```
npm run dev:stop -- --port 3002
```

Local data is stale or corrupted

Symptom: Local development behaves unexpectedly, returns old data, or throws deserialization errors.

Solution: Delete the `.bb-data/` directory to reset all local state:

```
rm -rf .bb-data
npm run dev
```

TypeScript errors after changing the backend

Symptom: Frontend shows type errors after modifying API methods in `cdk-blocks/index.ts`.

Solution: This is expected. This means type safety is working. Update your frontend code to match the new API signature. If your IDE doesn't pick up the changes, restart the TypeScript language server.

Deployment issues

CDK bootstrap required

Symptom: `npm run deploy` or `npm run sandbox` stops before it synthesizes, with a message such as "CDK is not bootstrapped in Region REGION of account ACCOUNT_ID, so `npm run deploy` cannot deploy there."

Solution: Run the bootstrap command that the message prints, from your project directory:

```
npx cdk bootstrap aws://ACCOUNT_ID/REGION --app=
```

You need to do this only one time per account/Region combination. If your app has a support stack in `us-east-1`, bootstrap `us-east-1` too. The check that runs before a deploy does not cover it, so a deploy that reaches that stack can still fail with an error such as "SSM parameter /cdk-bootstrap/hnb659fds/version not found." For more information, see the section called "Bootstrap the AWS CDK".

AWS credentials expired or missing

Symptom: Deployment fails with `ExpiredTokenException` or "Unable to locate credentials."

Solution: Refresh your AWS credentials:

```
# If using IAM Identity Center
aws sso login

# Verify credentials work
aws sts get-caller-identity
```

Sandbox deployment hangs or times out

Symptom: `npm run sandbox` takes more than a few minutes or appears stuck.

Solution: The first sandbox deploy is a full CloudFormation deployment and can take several minutes. If it fails, the sandbox usually leaves the stack in a failed state, because automatic rollback is off, and the next `npm run sandbox` cannot update it. Destroy and recreate:

```
npm run sandbox:destroy
npm run sandbox
```

`sandbox:destroy` handles a stack in `ROLLBACK_COMPLETE`. If it reports `UPDATE_ROLLBACK_FAILED`, first run `aws cloudformation continue-update-rollback --stack-name <your-sandbox-stack>`. To see the stack and its events, run `npm run sandbox:console`. It opens the CloudFormation stack list filtered to the first stack in `.cdk-blocks-sandbox/outputs.json`, which your most recent `npm run sandbox` or `npm run deploy` wrote. It takes the Region from `AWS_REGION`, `AWS_DEFAULT_REGION`, or your AWS CLI configuration, not from `cdk.json`, so check that it opened the Region your stack is in.

Permission denied during deployment

Symptom: Deployment fails with `AccessDenied` or "User is not authorized to perform" errors.

Solution: Your IAM user or role needs permissions to create Lambda functions, DynamoDB tables, API Gateway APIs, IAM roles, and CloudFormation stacks. Use an account with `AdministratorAccess` for initial setup, then scope down for CI/CD.

Runtime issues

Behavior differs between local and deployed

Symptom: Code works locally but fails in production, or vice versa.

Common causes:

Cause	Solution
A `KVStore` indexed field set to `null`	DynamoDB rejects the write on AWS, and the local implement ation accepts it. To keep an item out of an index, leave the field out
DynamoDB query pagination	Handle paginated responses. Local implementations return all results in a single response

Cause	Solution
Aurora connection limits	Reduce concurrent connections or configure connection pooling
Lambda cold starts	First request after idle period is slower. Not a bug
Configuration or environment variables missing	Use the `AppSetting` Block for runtime configuration and the `Secret` Block for secrets. For more information, see the section called "Configuration". To pass a value from your CDK code, such as a resource ARN, call `registerConfig(stack, 'KEY', value)` from `aws-cdk-blocks/cdk`. Every function then reads it as `process.env.KEY`.

API returns 500 Internal Server Error

Symptom: Frontend receives a generic 500 error without a useful message.

Solution: Unhandled exceptions from your own code return as 500s with the message "Internal error" and no name (to avoid leaking internals). Named errors that CDK Blocks throw, such as `ValidationFailedException`, keep their name and message. AWS SDK errors that pass through a CDK Block also return as "Internal error". To send an error the client can see, throw `new ApiError(message, status, { name })`. To see the full error and stack trace, check CloudWatch Logs for the Lambda function.

The handler's log group has a name that CloudFormation generates, not `/aws/lambda/...`. Find it in your stack's resources, then tail it. Your stack name is `<stackId>-prod` for `npm run deploy`, or `<stackId>-<sandboxId>` for a sandbox. It is also a top-level key in `.cdk-blocks-sandbox/outputs.json` after your most recent deploy.

```
aws cloudformation list-stack-resources --stack-name <stack-name> \
  --query "StackResourceSummaries[?contains(LogicalResourceId,
 'HandlerLogGroup')].PhysicalResourceId" \
  --output text
aws logs tail <log-group-name> --follow
```

You can also run `npm run sandbox:console`, check that it opened the Region your stack is in, choose your stack in the filtered list, and choose the log group on the **Resources** tab.

Getting further help

If your issue isn't covered here:

- Check CDK Blocks GitHub Issues for known bugs and workarounds
- File a new issue with your error message, Node.js version, and steps to reproduce