

The CDK Mindset: Building Serverless Solutions As Repeatable Blueprints



Lee Gilmore | Serverless Advocate | AWS Serverless Hero

Hi, I'm Lee Gilmore

AWS Serverless Hero

www.serverlessadvocate.com



serverless

HERO

Full code examples can be found using the QR code:

The screenshot shows a GitHub repository page for 'cdk-mindset'. The repository has 1 branch and 0 tags. It contains files like 'docs/images', 'orders-service-cdk-mindset', 'LICENSE', and 'README.md'. The 'About' section describes the repository as best practices for building production-grade, repeatable AWS CDK solutions as a blueprint using TypeScript, with a focus on delivering scalable, observable, and maintainable infrastructure from the outset. The 'README' file contains the title 'The CDK Mindset: Building Serverless Solutions As Repeatable Blueprints' and a brief description of the talk's focus on best practices for building production-grade, repeatable AWS CDK solutions. A green curved arrow points from the text 'Full code examples can be found using the QR code:' to the GitHub repository page.



What are we covering?

From a high-level we are covering:



Making your solutions **modular** and extensible



Environment specific **configuration**



Ephemeral environments and CICD are key for fast flow



Observability & monitoring should not be an after-thought



Add **governance and best practices** using AWS CDK Nag



Aspects and L3 constructs for standards and **reuse**

What is the AWS CDK (**Cloud Development Kit**)?



```
import * as cdk from 'aws-cdk-lib';
import * as sqs from 'aws-cdk-lib/aws-sqs';

import { Construct } from 'constructs';

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

    // Create an SQS queue
    new sqs.Queue(this, `DemoQueue`, {
      queueName: `demo-queue`,
      visibilityTimeout: cdk.Duration.seconds(300),
    });
  }
}
```



- This talk is ideally for people that have some experience of the AWS CDK
- Infrastructure as Code (IaC) using familiar programming languages, like TypeScript
- Based on constructs: Reusable classes at different levels:
 - L1: **CloudFormation** resources
 - L2: AWS constructs with **sensible defaults**
 - L3: Opinionated patterns and best practices (**curated**)

The Problem Statement

Why is it key that we build solutions **the right way from the start?**

The Problem: Starts easy, struggles to scale

- Getting started is **as easy as a cdk init command**, but what it creates as default won't scale
- You will have **missing production-quality gaps** (observability, governance etc)
- Your code as default **will not be modular** if you don't make early amendments
- The **easiest time to change is from the start**, as refactoring infrastructure and code is expensive and risky!



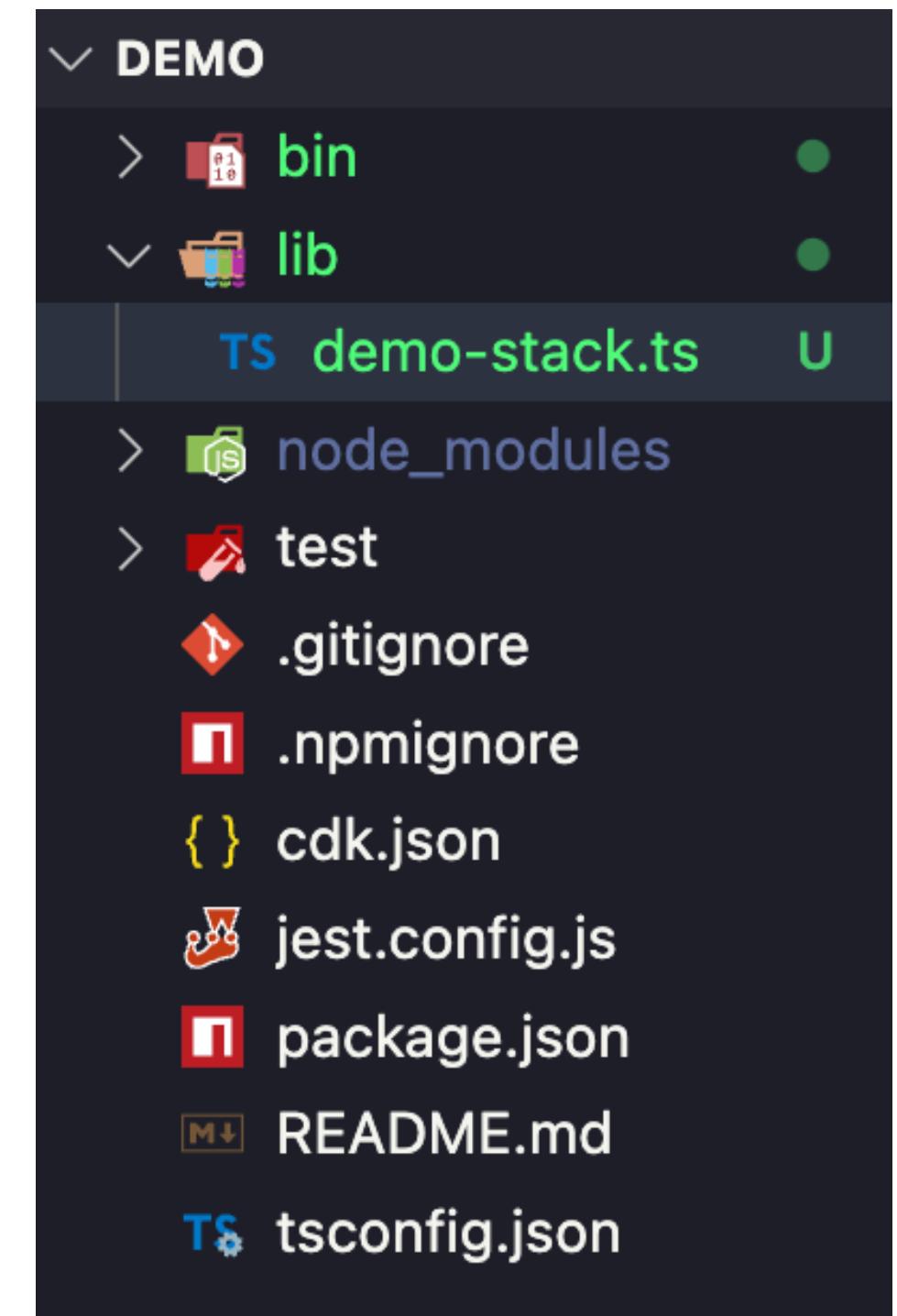
“Making your solutions modular and extensible”

Its really key to make our solutions modular from the outset, or as they grow they can soon bite us!

The Problem: One stack doesn't rule them all

- The CDK init creates **one stack** by default
- Many teams continue to build on one stack
- Eventually, as the app grows, they hit the **500 resource limit** per stack (hard limit)
- This also means that **stateful resources** (databases, etc) can be deleted accidentally when needing to start splitting resources out

`cdk init app --language typescript`



The Problem: One stack doesn't rule them all

- When you are close to hitting this limit, you will see this on synth/deploy:

[Info at /DemoStack] Number of resources: 499 is approaching the maximum of 500

- Once you try to add more, you will get (your synth will fail):

Error: Number of resources in stack 'DemoStack': 501 is greater than allowed maximum of 500

- Conditional creation of resources can hide this until later environments
(e.g. only in production we add a WAF, for example)

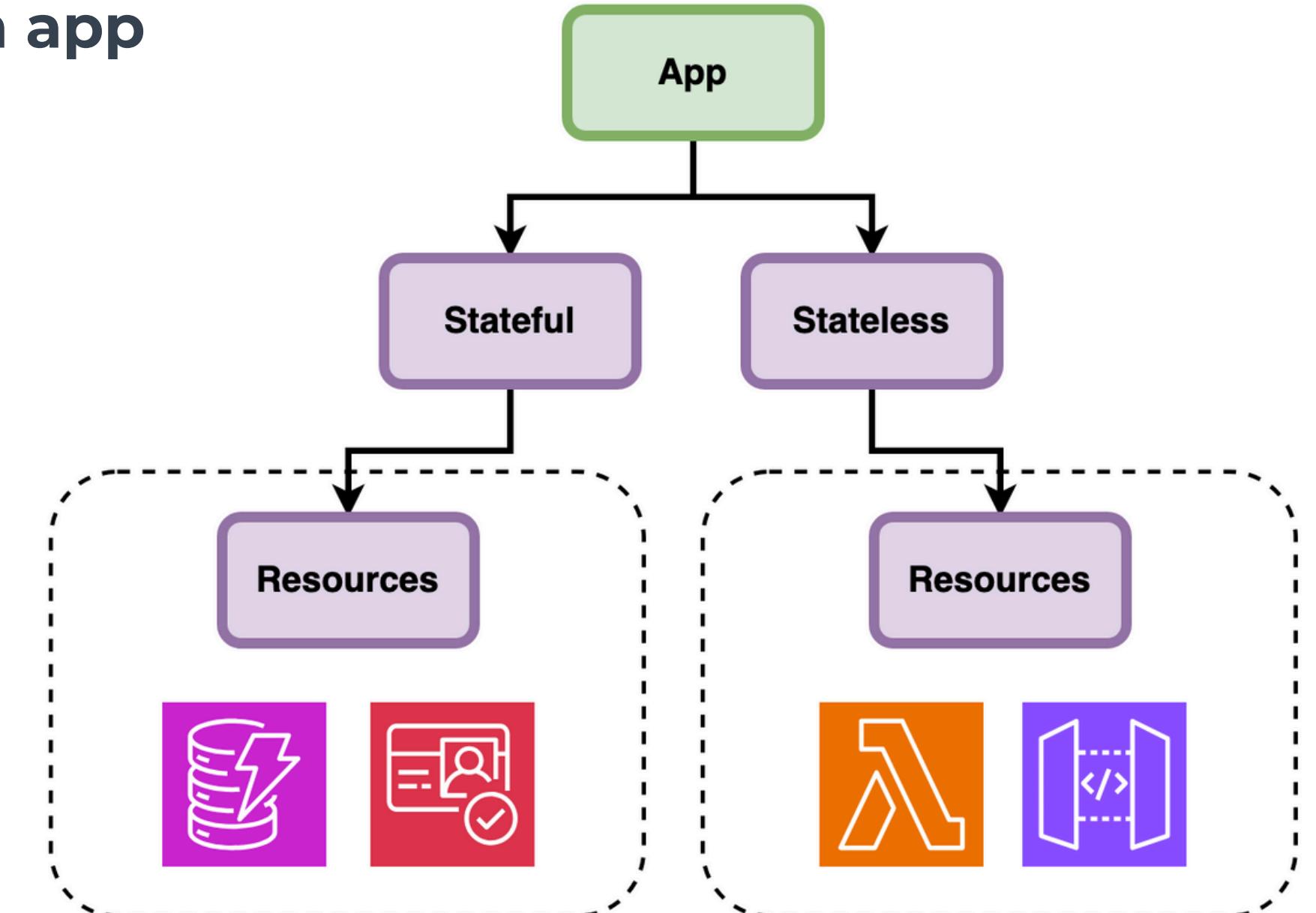
Creating modular stacks: **Stateful** and **Stateless**

- ✓ In the AWS CDK, the **deployable unit is an app** (1 or more stacks)

- ✓ We can split our resources into two stacks:
Stateful and **Stateless**

- ✓ **Stateful** contains resources like UserPools, Databases, etc

- ✓ **Stateless** contains resources like Lambda Functions, API Gateways, etc



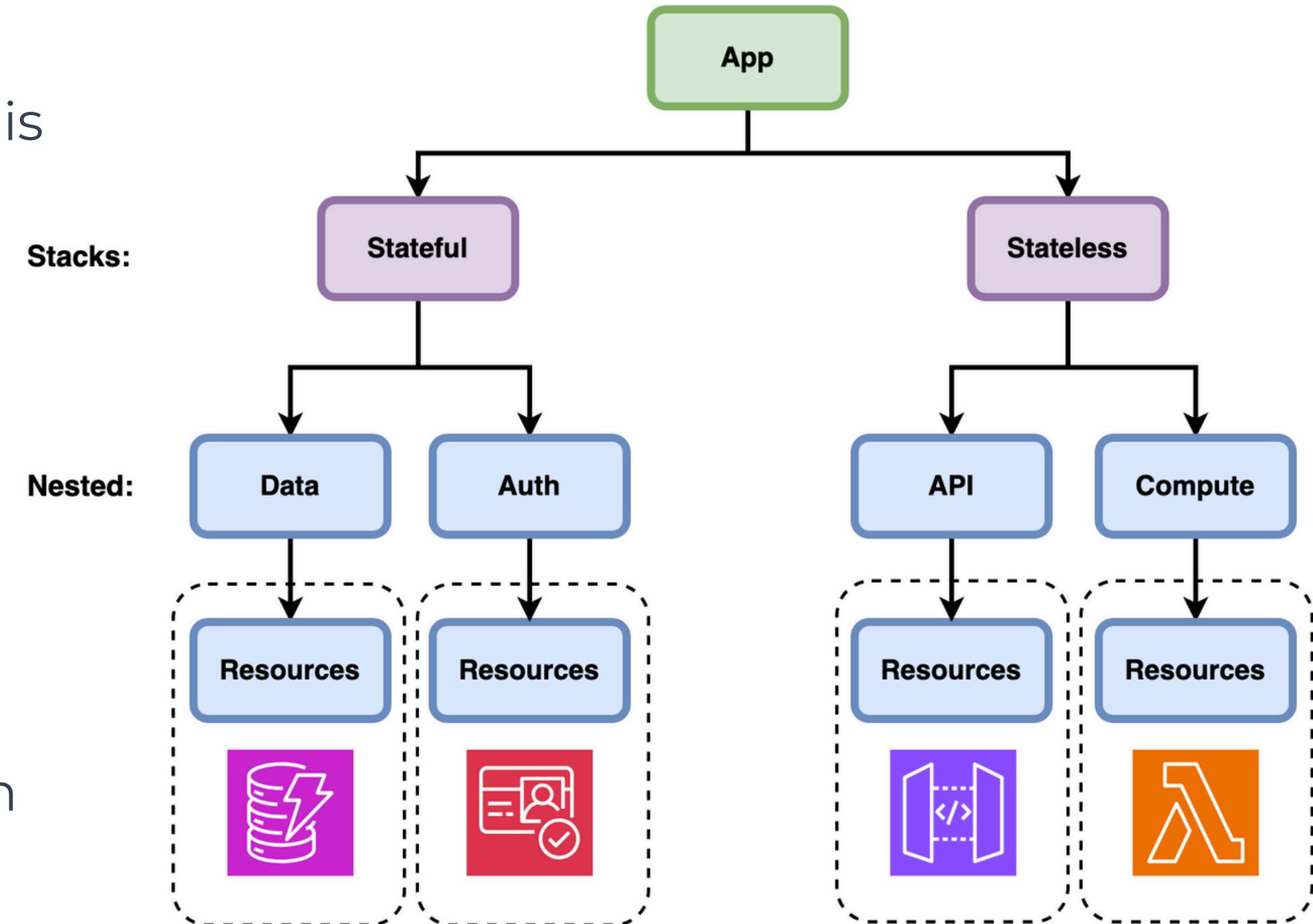
Nested stacks: Further modularising stacks

 We could still hit the stack limits in this approach with Stateful and Stateless

 **Nested stacks** allow us to have child stacks with their own 500 resource limit

 You can split by areas like purpose (Data, Auth, etc), by integration (Partner A, Partner B), or a mix of both

 A nested stack hierarchy can have up to **2500 resources** in it



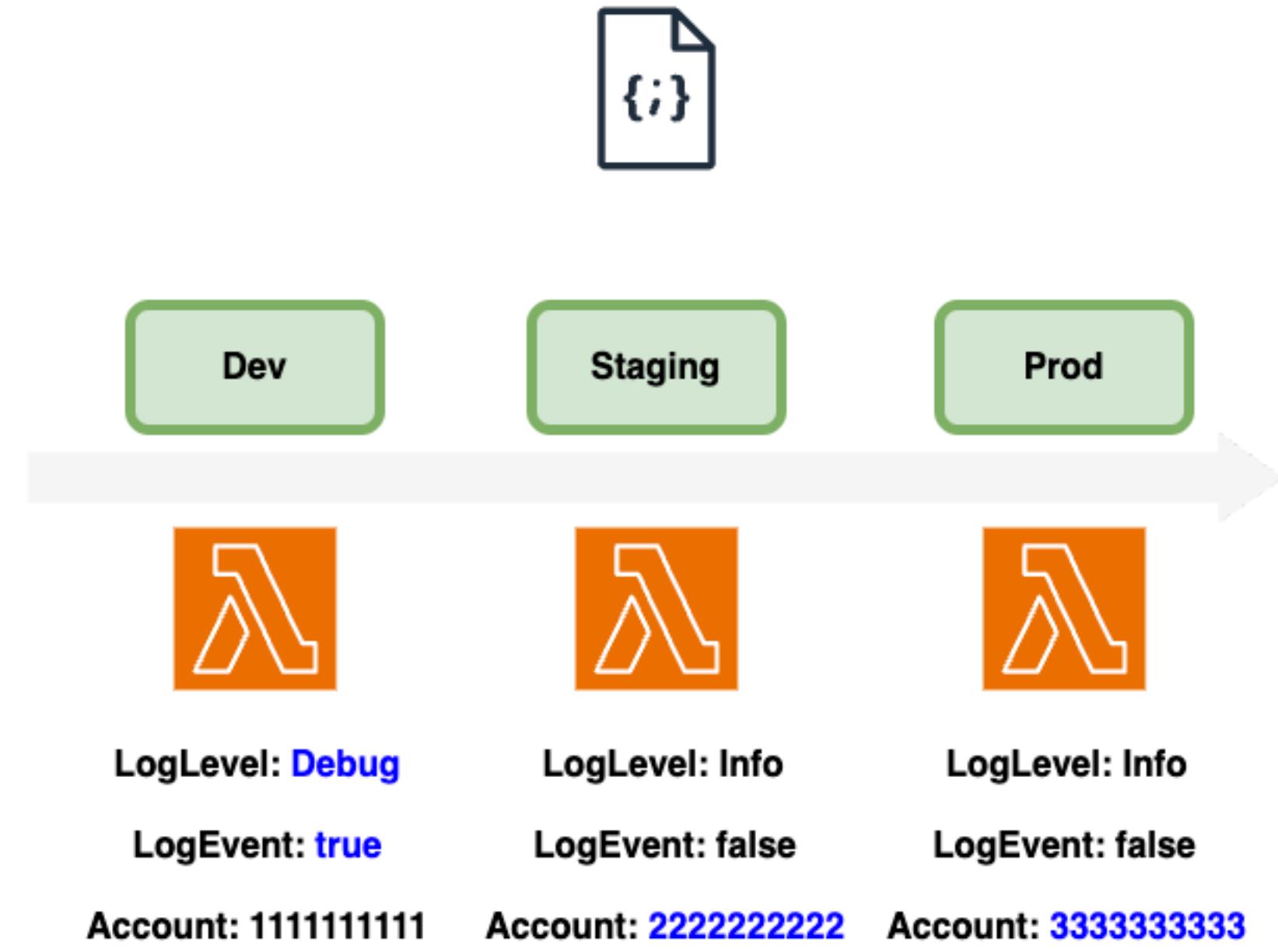


“Environment specific configuration”

Allow your solutions to be deployed with **configuration specific to your environment and stage!**

The Problem: environment specific config

- Many teams start with only one environment in mind - the current '**develop**' one!
- Best practice is that we should deploy the **same code to different stages / accounts**
- Most environments need their **own configuration per stage**
- Examples being API URLs, logging levels, Lambda runtime version, and account IDs



Injecting static config based on stage

 We can create a common interface for our config using TypeScript

 We can now create a typed config per stage through a function

 Our function grabs the correct config for the stage you are building for

 Now each stack has its own config injected into the props per stage



```
● ● ●  
export interface EnvironmentConfig {  
  shared: {  
    stage: Stage;  
    serviceName: string;  
    metricNamespace: string;  
    logging: {  
      logLevel: 'DEBUG' | 'INFO' | 'ERROR';  
      logEvent: 'true' | 'false';  
    };  
  };  
  env: {  
    account: string;  
    region: string;  
  };  
  stateless: {  
    runtimes: lambda.Runtime;  
  };  
  stateful: {  
    tableName: string;  
  };  
}
```

Injecting static config based on stage

- ✓ We can create a common interface for our config using TypeScript
- ✓ **We can now create a typed config per stage through a function**
- ✓ Our function grabs the correct config for the stage you are building for
- ✓ Now each stack has its own config injected into the props per stage



```
export const getEnvironmentConfig = (stage: Stage): EnvironmentConfig => {
  switch (stage) {
    case Stage.staging:
      ...
    case Stage.prod:
      ...
    case Stage.develop:
      return {
        shared: {
          logging: {
            logLevel: 'DEBUG',
            logEvent: 'true',
          },
          serviceName: `my-service-${Stage.develop}`,
          metricNamespace: `my-service-namespace-${Stage.develop}`,
          stage: Stage.develop,
        },
        stateless: {
          runtimes: lambda.Runtime.NODEJS_22_X,
        },
        env: {
          account: '12345678912',
          region: Region.london,
        },
        stateful: {
          tableName: `service-${Stage.develop}`,
        },
      }
  }
};
```

Injecting static config based on stage

- We can create a common interface for our config using TypeScript
- We can now create a typed config per stage through a function
- Our function grabs the correct config for the stage you are building for**
- Now each stack has its own config injected into the props per stage



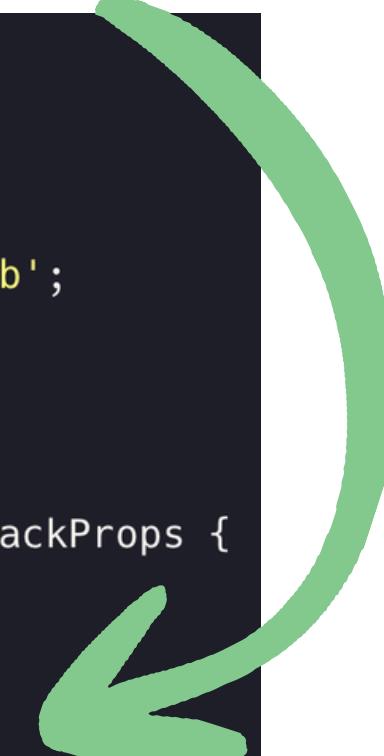
```
const stage = getStage(process.env.STAGE as Stage) as Stage;
const appConfig = getEnvironmentConfig(stage);

const app = new cdk.App();

const statefulStack = new StatefulStack(
  app,
  `StatefulStack-${stage}`,
  {
    env: appConfig.env,
    stateful: appConfig.stateful,
    shared: appConfig.shared,
  },
);
```

Injecting static config based on stage

- ✓ We can create a common interface for our config using TypeScript
- ✓ We can now create a typed config per stage through a function
- ✓ Our function grabs the correct config for the stage you are building for
- Now each stack has its own config injected into the props per stage**



```
import * as cdk from 'aws-cdk-lib';
import * as dsql from 'aws-cdk-lib/aws-dsql';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';

import { Construct } from 'constructs';
import { Stage } from '../types';

export interface StatefulStackProps extends cdk.StackProps {
  stateful: {
    tableName: string;
  };
  shared: {
    stage: Stage;
  };
}

export class StatefulStack extends cdk.Stack {
  constructor(
    scope: Construct,
    id: string,
    props: StatefulStackProps,
  ) {
    super(scope, id, props);

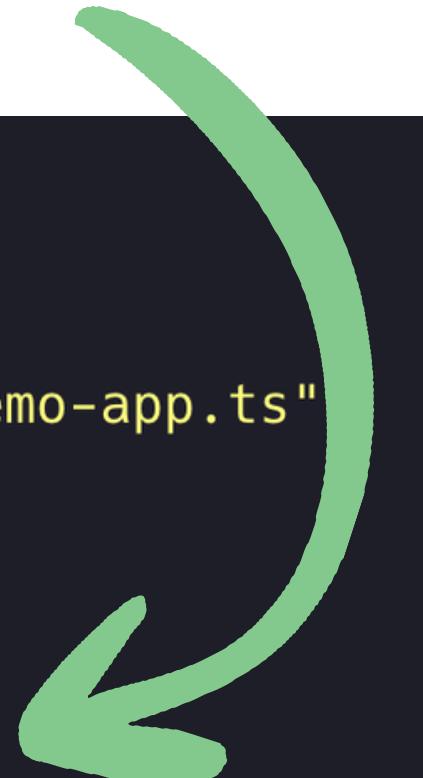
    const {
      stateful: { tableName },
      shared: { stage },
    } = props;
  }
}
```

Why I don't personally use the `cdk.json`

- The **cdk.json file is not typed**, which can lead to mistakes! No intellisense
- We can't pull in additional reusable **functions, types** or **interfaces**
- It's **not flexible!**
- It is **not great for default fallbacks**, i.e. ephemeral environments, which we will talk about in the next section

`app.node.tryGetContext(env).bucketName`

```
● ● ●  
{  
  "app": "npx ts-node --prefer-ts-exts bin/demo-app.ts"  
  ...  
  "context": {  
    "develop": {  
      "bucketName": "my-dev-bucket"  
    },  
    "prod": {  
      "bucketName": "my-prod-bucket"  
    },  
    "@aws-cdk/aws-lambda:recognizeLayerVersion": true,  
    "@aws-cdk/core:checkSecretUsage": true,  
    "@aws-cdk/core:target-partitions": ["aws", "aws-cn"],  
    ...  
  }  
}
```





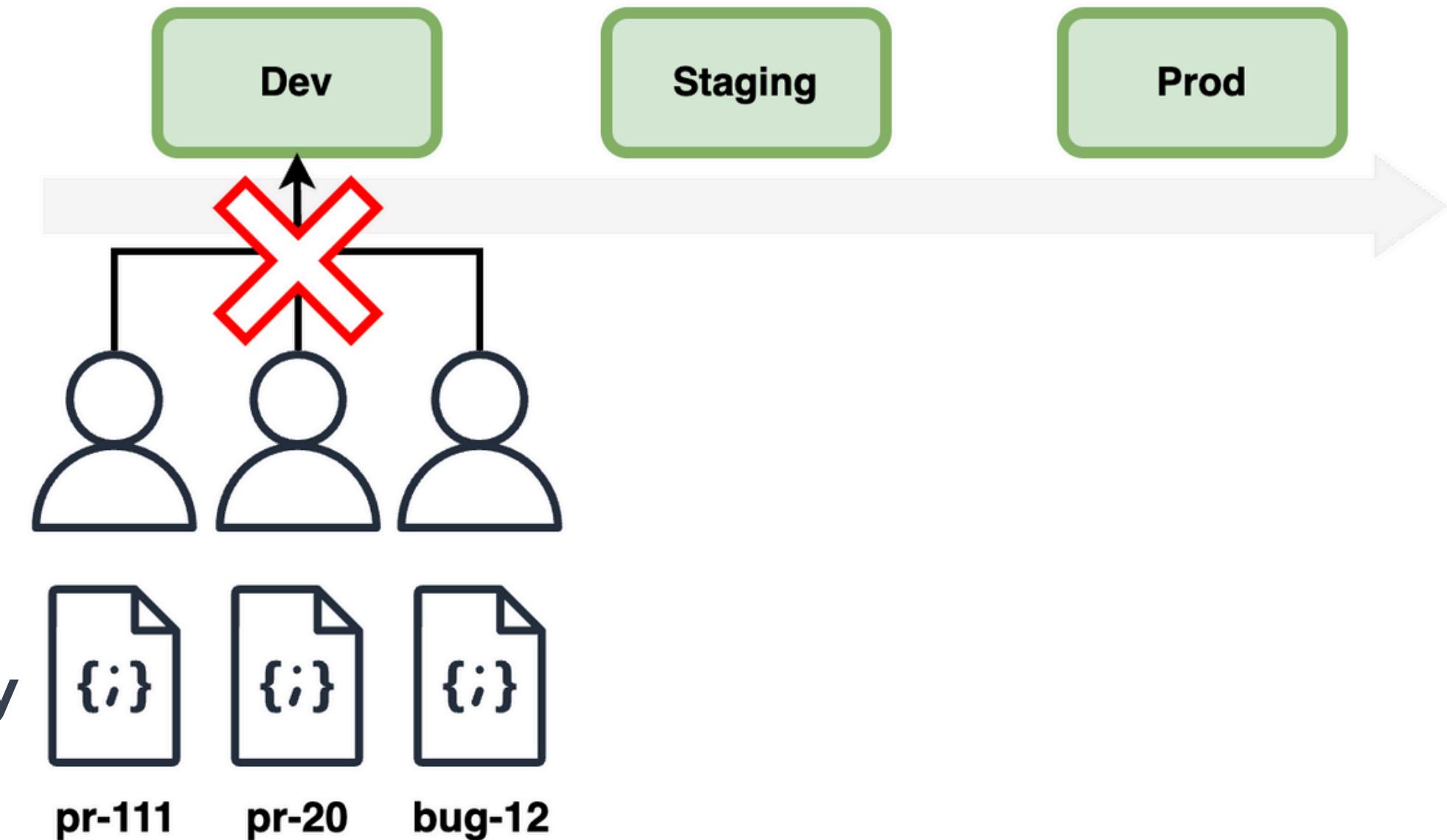
3.0

“Ephemeral environments & CICD are key for flow”

It is key that engineers can **work with ephemeral environments from the outset of a project through CICD pipelines!**

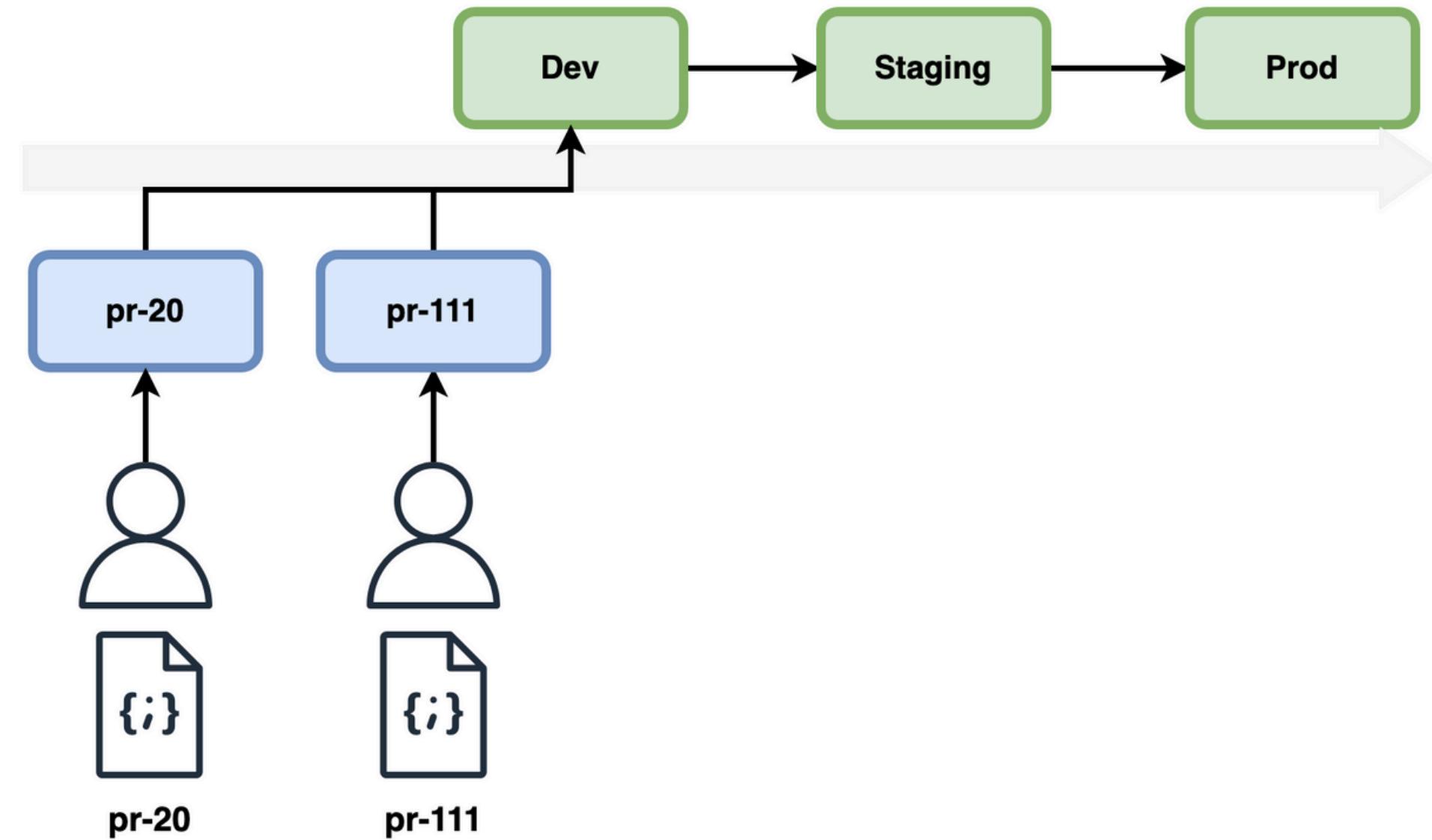
The Problem: Developers standing on toes!

- I have worked with companies where **multiple people are deploying to the same 'development' environment..**
- This gets **messy and frustrating** very quickly for teams!
- It's an old school mentality when **serverless is pay for use and relatively quick to spin up** new environments and tear down when needed



Why are **ephemeral environments** key in Serverless?

- ✓ Ephemeral environments are **short-lived**, deployed environments **based on your current branch and PR**
- ✓ This allows you to **spin up your own environment** for your work, **and tear it down again** once your PR is merged
- ✓ This unblocks teams and allows them to **iterate quickly without handoffs** or delays
- ✓ **Bake these in from the start**



Config + Ephemeral: A match made in heaven!

- Our **config fallback** is to use the ephemeral stage i.e. pr-123, hotfix-22 etc when not develop, staging & prod
- We use the **current Git branch** and **pass it as the stage** using an npm script
- Seamless to the developer!



```
export const getEnvironmentConfig = (stage: Stage): EnvironmentConfig => {
  ...
  default:
    return {
      shared: {
        domainName: 'example.com',
        domainCertificateArn:
          'arn:aws:acm:us-east-1:123456789:certificate/324442ec',
        logging: {
          logLevel: 'DEBUG',
          logEvent: 'true',
        },
        serviceName: `example-service-${stage}`,
        metricNamespace: `example-${stage}`,
        stageName: stage,
      },
      stateless: {},
      env: {
        account: '123456789',
        region: Region.dublin,
      },
      stateful: {},
      client: {
        bucketName: `${stage}-example-demo-bucket`,
      },
    };
};
```

Config + Ephemeral: `npm run deploy`

```
"deploy": "STAGE=$(git rev-parse --abbrev-ref HEAD | sed 's|.*||') cdk deploy --all",
```

`feature/api-123 → api-123`

`hotfix/api-22 → api-22`

`bug/api-77 → api-77`



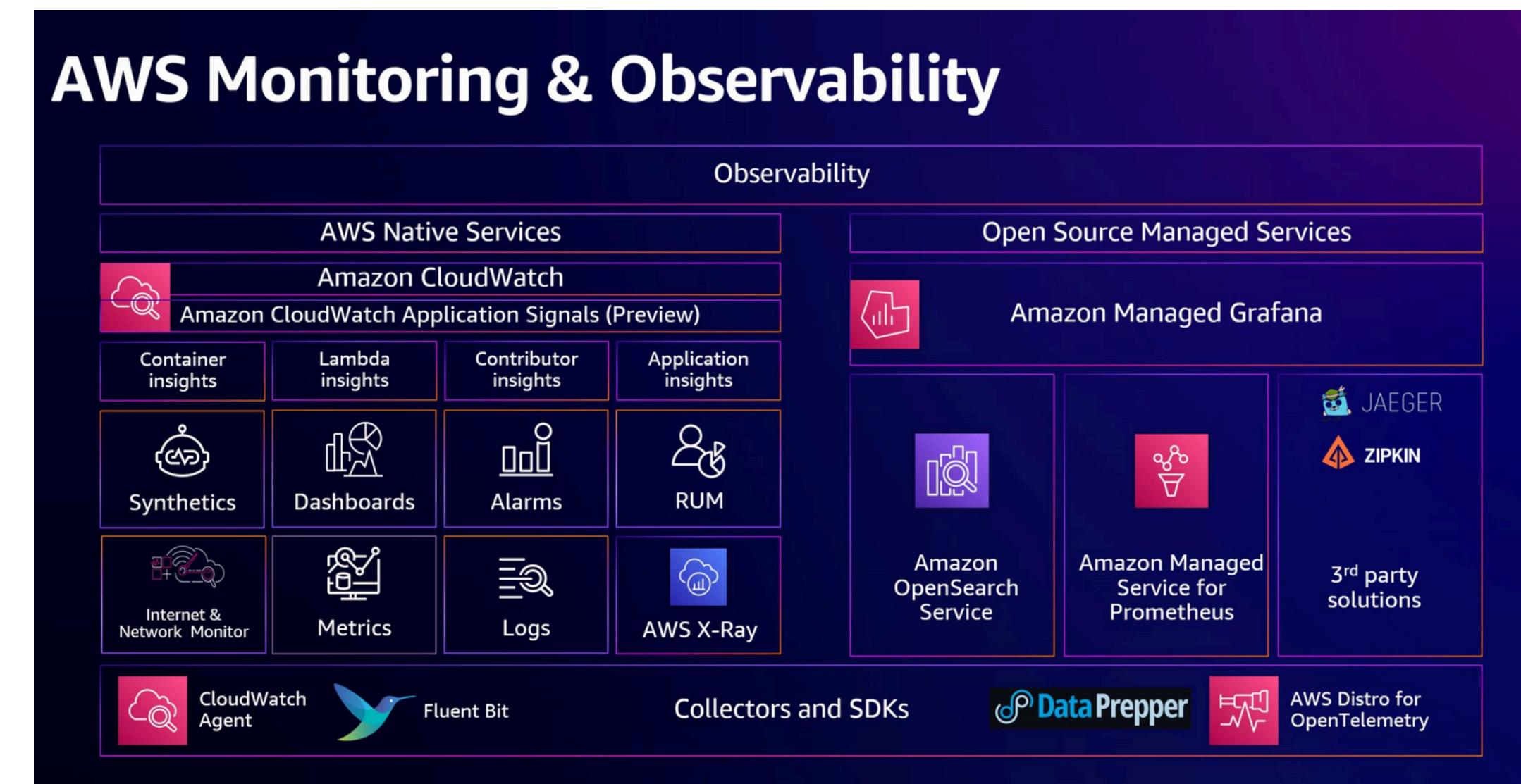


“Monitoring & Observability should not be an after-thought”

Add monitoring & observability from day one using [Lambda Powertools](#) and the [AWS CDK Monitoring constructs](#).

The Problem: How is your system behaving?

- Without it we don't see functions failing silently or DLQs filling up
 - We can't provide **metrics** to our customers on how the system is behaving or create alarms
 - Without standard **logging** it is impossible to correlate and search logs when needed, or build useable dashboards
 - AWS gives us **many monitoring and observability services** we can use to give us key insights



<https://docs.aws.amazon.com/decision-guides/latest/monitoring-on-aws-how-to-choose/monitoring-on-aws-how-to-choose.html>

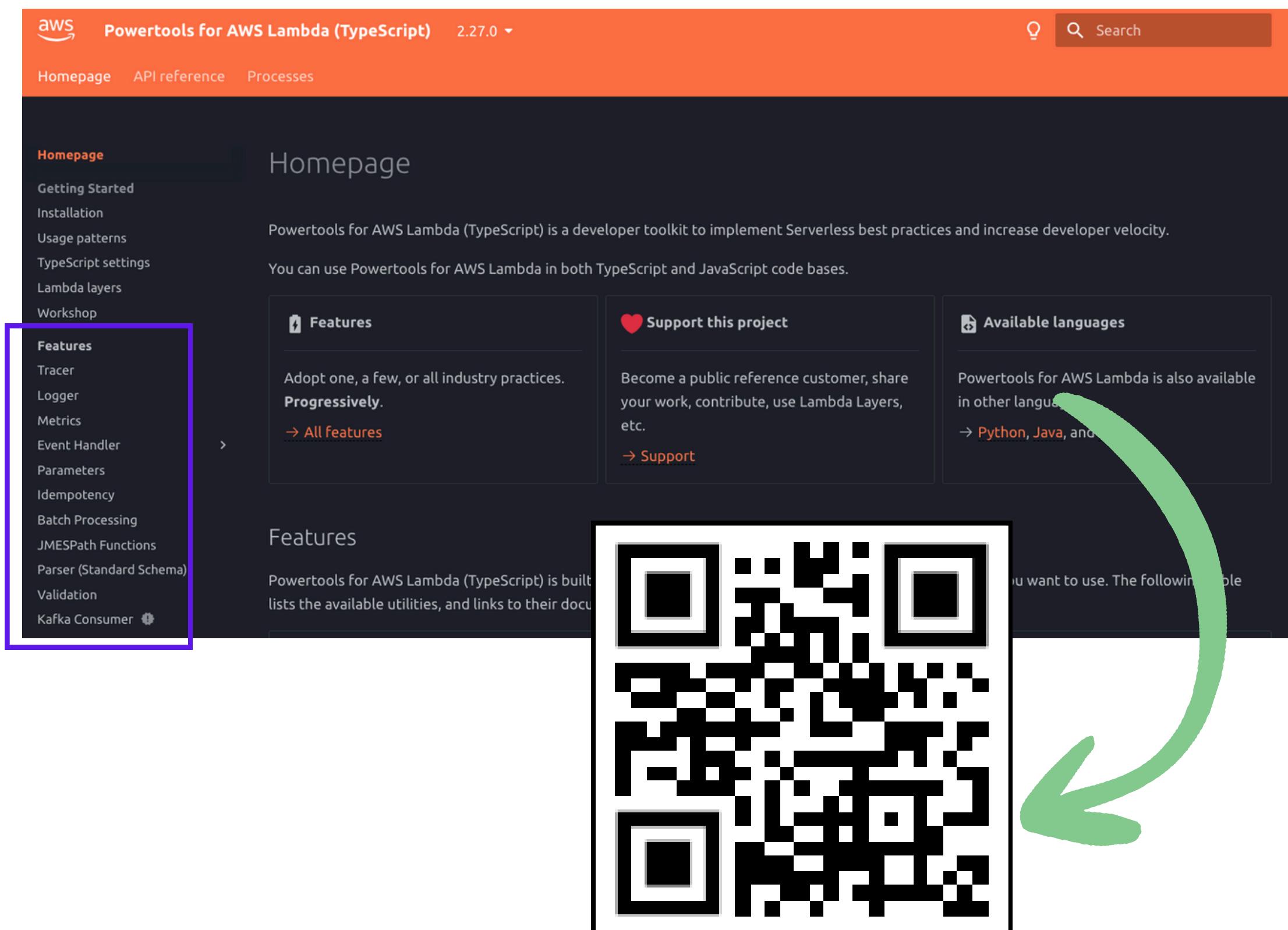
Logging, Metrics and Tracing - Lambda Powertools

✓ We can use **Lambda Powertools** which works with various runtimes (also Java, .Net, Python)

✓ The **Logger** provides us an opinionated logger with output structured as JSON

✓ **Metrics** allows us to instrument our code for both success and failure scenarios

✓ The **Tracer** allows us to add custom annotations to traces



The screenshot shows the Powertools for AWS Lambda (TypeScript) homepage. The top navigation bar includes the AWS logo, the title "Powertools for AWS Lambda (TypeScript)", a version dropdown set to "2.27.0", and search and help icons. Below the navigation is a secondary navigation bar with links to "Homepage", "API reference", and "Processes". The main content area has a dark background with orange highlights. On the left, a sidebar menu is highlighted with a purple box, listing "Homepage", "Getting Started", "Installation", "Usage patterns", "TypeScript settings", "Lambda layers", and "Workshop". Under "Features", there is a list including "Tracer", "Logger", "Metrics", "Event Handler", "Parameters", "Idempotency", "Batch Processing", "JMESPath Functions", "Parser (Standard Schema)", "Validation", and "Kafka Consumer". To the right of the sidebar, the word "Homepage" is displayed. Below it, a paragraph states: "Powertools for AWS Lambda (TypeScript) is a developer toolkit to implement Serverless best practices and increase developer velocity. You can use Powertools for AWS Lambda in both TypeScript and JavaScript code bases." There are three callout boxes: one for "Features" (listing "Adopt one, a few, or all industry practices. Progressively." with a link to "All features"), one for "Support this project" (listing "Become a public reference customer, share your work, contribute, use Lambda Layers, etc." with a link to "Support"), and one for "Available languages" (listing "Powertools for AWS Lambda is also available in other languages" with a link to "Python, Java, and"). A large green arrow points from the bottom right towards a QR code in the bottom right corner.

Basic Code example (TypeScript)

 We start by adding various imports and instantiating the logger, tracer and metrics

 We then add our actual Lambda function code as an async function

 We use the Middy framework that works seamlessly with Lambda Powertools middleware's



```
● ● ●  
  
import { injectLambdaContext } from '@aws-lambda-powertools/logger/middleware';  
import { MetricUnit, Metrics } from '@aws-lambda-powertools/metrics';  
import { logMetrics } from '@aws-lambda-powertools/metrics/middleware';  
import { Tracer } from '@aws-lambda-powertools/tracer';  
import { captureLambdaHandler } from '@aws-lambda-powertools/tracer/middleware';  
import middy from '@middy/core';  
import type { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';  
import httpErrorHandler from '@middy/http-error-handler';  
  
const tracer = new Tracer();  
const metrics = new Metrics();  
const logger = new Logger();
```

Basic Code example (TypeScript)

✓ We start by adding various imports and instantiating the logger, tracer and metrics

✓ We then add our actual Lambda function code as an **async** function

✓ We use the Middy framework that works seamlessly with Lambda Powertools middleware's

```
export const createOrderAdapter = async ({  
  body,  
}: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> => {  
  try {  
    const createdOrder = await createOrderUseCase(body);  
  
    metrics.addMetric('SuccessfulCreateOrder', MetricUnit.Count, 1);  
  
    return {  
      statusCode: 200,  
      body: JSON.stringify(createdOrder),  
    };  
  } catch (error) {  
    let errorMessage = 'Unknown error';  
    if (error instanceof Error) errorMessage = error.message;  
    logger.error(errorMessage);  
  
    metrics.addMetric('CreateOrderError', MetricUnit.Count, 1);  
  }  
};
```

Basic Code example (TypeScript)

✓ We start by adding various imports and instantiating the logger, tracer and metrics

✓ We then add our actual Lambda function code as an async function

✓ **We use the Middy framework that works seamlessly with Lambda Powertools middleware's**

```
export const handler = middy(createOrderAdapter)
  .use(injectLambdaContext(logger))
  .use(captureLambdaHandler(tracer))
  .use(logMetrics(metrics))
  .use(httpErrorHandler());
```

Dashboards & Alarms: Verbose to configure

- We have the metrics and logs in place, but **we need to visualise and alert on them**
- Creating CloudWatch alarms** across all of your services can be verbose to configure! (e.g. dead letter queue length, failing invocations etc)
- CloudWatch dashboards** can also be tedious to build with widgets from various different stacks (and nested stacks!)
- We need to build these in from day one **without having to retrofit or forget about later**

CDK Monitoring Constructs for ease from day one

4.0

We can use the CDK Monitoring Constructs package to add the **most common service alarms** to our constructs

These are curated for adding the alarms in an **easy and non-verbose way**

We can **automatically create dashboards** based on our alarms and constructs

You can consume the library in **multiple supported languages** (TypeScript, Java, Python and .NET)

CDK Monitoring Constructs

npm package 9.15.2 maven central 9.12.0 pypi package 9.15.2 nuget package 9.15.2 Gitpod ready-to-code
custom badge inaccessible

Easy-to-use CDK constructs for monitoring your AWS infrastructure with [Amazon CloudWatch](#).

- Easily add commonly-used alarms using predefined properties
- Generate concise CloudWatch dashboards that indicate your alarms
- Extend the library with your own extensions or custom metrics
- Consume the library in multiple supported languages

Installation

- TypeScript
- Java
- Python
- C#

Features

You can browse the documentation at <https://constructs.dev/packages/cdk-monitoring-constructs/>

Item	Monitoring	Alarms	
AWS API Gateway (REST API) (<code>.monitorApiGateway()</code>)	TPS, latency, errors	Latency, error count/rate, low/high TPS	To see metrics, : Advanced Monit
AWS API Gateway V2 (HTTP API) (<code>.monitorApiGatewayV2HttpApi()</code>)	TPS, latency, errors	Latency, error count/rate, low/high TPS	To see route level enable Advanced
AWS AppSync (GraphQL API) (<code>.monitorAppSyncApi()</code>)	TPS, latency, errors	Latency, error count/rate, low/high TPS	
Amazon Aurora (<code>.monitorAuroraCluster()</code>)	Query duration, connections, latency, CPU usage, Serverless	Connections, Serverless Database	



Basic Code example (TypeScript)

4.0

- ✓ We start by adding the monitoring facade construct

- ✓ Example of adding specific API Gateway Rest API alarms

```
// monitoring
const monitoringFacade = new monitoring.MonitoringFacade(
  this,
  `${stage}-orders-monitoring`,
  {
    alarmFactoryDefaults: {
      alarmNamePrefix: `${stage}-orders-dashboard`,
      actionsEnabled: true,
      // default alarm action that can be overridden
      action: new SnsAlarmActionStrategy({
        onAlarmTopic: alertingServiceTopic,
      }),
      evaluationPeriods: 1,
      datapointsToAlarm: 3,
    },
  },
);
```

Basic Code example (TypeScript)

4.0

- We start by adding the monitoring facade construct

Example of adding specific API Gateway Rest API alarms

```
monitoringFacade
  .monitorApiGateway({
    api: this.api,
    add5XXFaultCountAlarm: {
      Critical: {
        maxErrorCount: 1,
        datapointsToAlarm: 1,
        treatMissingDataOverride: TreatMissingData.NOT_BREACHING,
      },
    },
    addLatencyP95Alarm: {
      Critical: {
        // 95% of calls should be less than 1.5 secs
        maxLatency: cdk.Duration.millis(1_500),
        datapointsToAlarm: 1,
        treatMissingDataOverride: TreatMissingData.NOT_BREACHING,
      },
    },
  })
}
```



“Add governance and best practices using CDK Nag”

Add AWS documented best practices and standards into our AWS CDK solutions using CDK Nag

The Problem: Am I building this right?

- With each AWS service resource in the AWS CDK there are **typically between 15-50 different properties to configure**
- Many teams unknowingly start building AWS CDK solutions using constructs that are **not configured against AWS best practices**
- An example could be using an **SQS queue, but not having a dead-letter queue** set. Or perhaps not having PITR set on a DynamoDB table
- How do we get the right guardrails in place to know we are **building the solutions correctly from day one?**

Using CDK Nag for Best Practices

5.0

- ✓ CDK Nag is an open-source AWS project that **checks your solutions against agreed best practices**

- ✓ It contains **a set of rule packs** that you can attach to your stacks (AWS Solutions, HIPPA, NIST, Serverless etc) which **throw warnings or errors for you to resolve**

- ✓ A synth or deploy will show any issues

- ✓ It is far better to **remediate from the start slowly as you build**, than adding at the end and having to fix up all in one go!

cdk-nag

pypi v2.37.40 npm v2.37.40 maven-central v2.37.40 nuget v2.37.40 Go v1.23

[View on Construct Hub](#)

Check CDK applications or [CloudFormation templates](#) for best practices using a combination of available rule packs. Inspired by [cfn_nag](#).

Check out [this blog post](#) for a guided overview!



```
bin > TS cdk-test.ts M X TS cdk-test-stack.ts M
1 #!/usr/bin/env node
2
3 import * as cdk from '@aws-cdk/core';
4 import { CdkTestStack } from '../lib/cdk-test-stack';
5 import { AwsSolutionsChecks } from 'cdk-nag';
6 import { Aspects } from '@aws-cdk/core';
7
8 const app = new cdk.App();
9 new CdkTestStack(app, 'CdkTest');
10 Aspects.of(app).add(new AwsSolutionsChecks({verbose:true}))
```

TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE

```
~/environment/cdk-test $ cdk synth
[Error at /CdkTest/rUserPool/Resource] AwsSolutions-COG1: The Cognito user pool does not have a password policy that minimally specify a password length of at least 8 characters, as well as requiring uppercase, numeric, and special characters.
[Warning at /CdkTest/rUserPool/Resource] AwsSolutions-COG2: The Cognito user pool does not require MFA.
[Error at /CdkTest/rUserPool/Resource] AwsSolutions-COG3: The Cognito user pool does not have AdvancedSecurityMode set to ENFORCED.
Found errors
~/environment/cdk-test $
```

Available Rules and Packs

See [RULES](#) for more information on all the available packs.

1. [AWS Solutions](#)
2. [HIPAA Security](#)
3. [NIST 800-53 rev 4](#)
4. [NIST 800-53 rev 5](#)
5. [PCI DSS 3.2.1](#)
6. [Serverless](#)



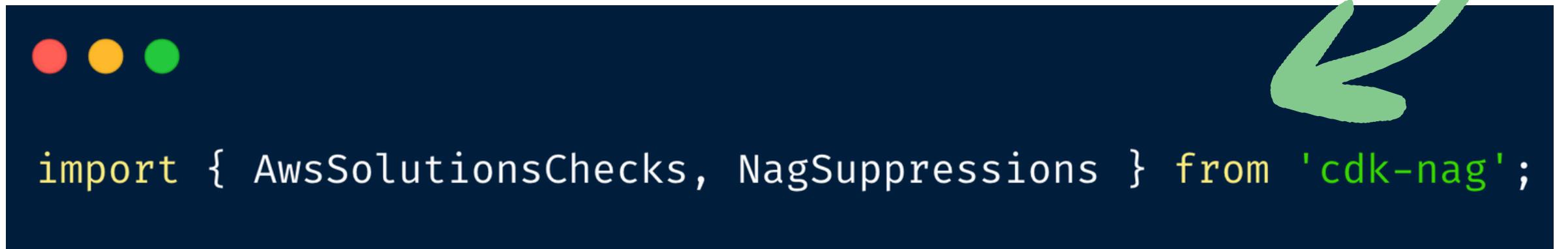
Basic Code example (TypeScript)

5.0

- ✓ We start by adding the relative imports to our stacks

- ✓ We then add the relevant nag pack to the stack that we want to check our resources against

- ✓ We will sometimes get false positives in results, and may want to suppress certain errors or warnings



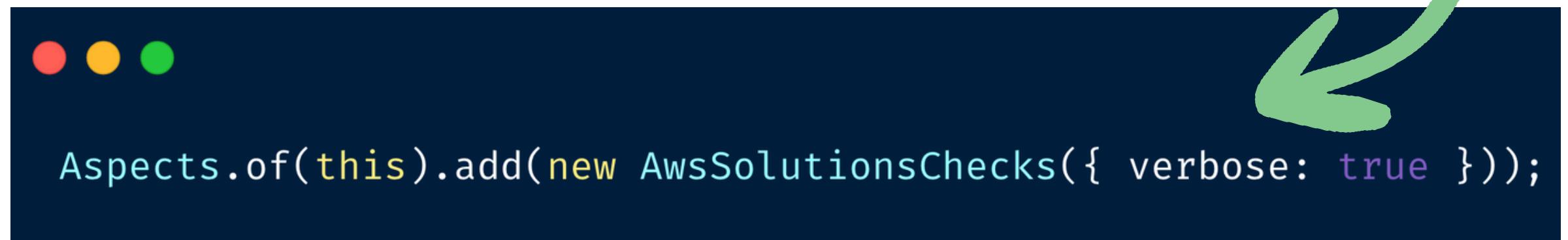
A screenshot of a code editor showing a dark-themed interface with three circular status indicators at the top left (red, yellow, green). Below them, a line of TypeScript code is displayed: `import { AwsSolutionsChecks, NagSuppressions } from 'cdk-nag';`

Basic Code example (TypeScript)



5.0

- We start by adding the relative imports to our stacks
- We then add the relevant nag pack to the stack that we want to check our resources against**



```
Aspects.of(this).add(new AwsSolutionsChecks({ verbose: true }));
```

- 
- We will sometimes get false positives in results, and may want to suppress certain errors or warnings

Basic Code example (TypeScript)

5.0

- ✓ We start by adding the relative imports to our stacks
- ✓ We then add the relevant nag pack to the stack that we want to check our resources against
- ✓ **We will sometimes get false positives in results, and may want to suppress certain errors or warnings**



```
NagSuppressions.addResourceSuppressionsByPath(
  this,
  `/StatefullStack-${stage}/IdempotencyTable/Resource`,
  [
    {
      id: 'AwsSolutions-DDB3',
      reason: 'We dont need PITR on the idempotency table',
    },
  ],
);
```



6.0

“Aspects and L3 constructs for standards”

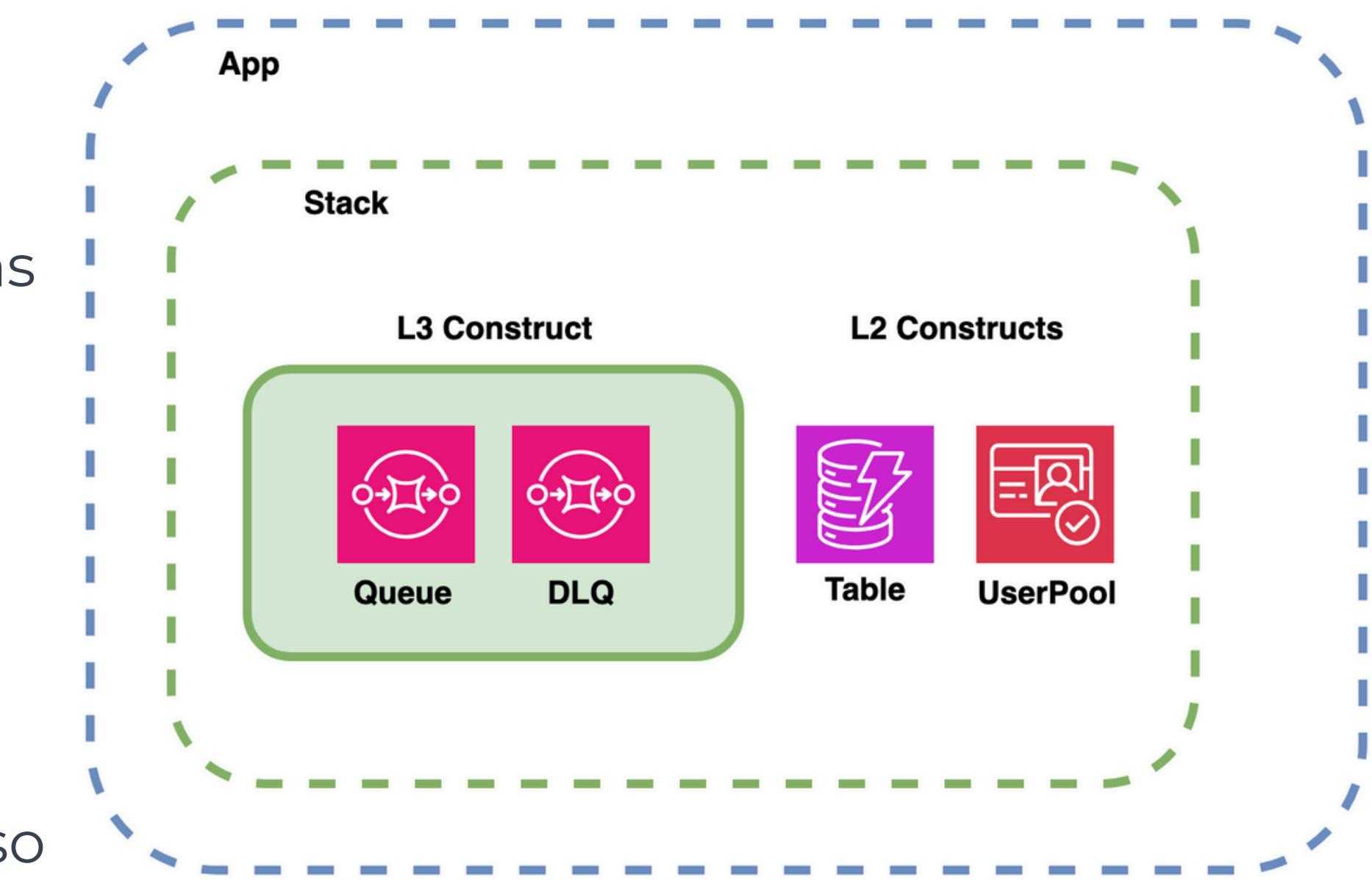
We can use aspects and L3 constructs **for reusability and standards**

The Problem: How do I reuse code and standards?

- What if we need **our own curated constructs**, or multiple constructs formed into a pattern (reusable building block)?
- We are likely to have **our own standards** to adhere to in our company
- How do we **reuse these easily** across code bases?
- How do I **create my own checks**, for example that all constructs have the right level of tagging? (similar to CDK Nag)

Leveraging L3 constructs for reuse

- We can package our best practices as defaults using L3 constructs
- We can create building blocks as patterns by integrating one or more constructs together
- We can embed our own best practices alongside AWS's with sensible defaults
- We can publish to a package repository so consumers can easily install and use



Basic Code example (TypeScript)

6.0



We can start by importing the correct packages and creating our interfaces



We can create our main class that sets out sensible defaults



We can then utilise this L3 construct without having to think about the internals

```
import * as cdk from 'aws-cdk-lib';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';

import { Construct } from 'constructs';

interface DynamoDbTableProps
  extends Pick<
    dynamodb.TableProps,
    'removalPolicy' | 'partitionKey' | 'sortKey' | 'tableName'
  > {
  partitionKey: dynamodb.Attribute;
  sortKey: dynamodb.Attribute;
  removalPolicy: cdk.RemovalPolicy;
  tableName: string;
}

type FixedDynamoDbTableProps = Omit<
  dynamodb.TableProps,
  'removalPolicy' | 'partitionKey' | 'sortKey' | 'tableName'
>;
```

Basic Code example (TypeScript)

6.0



We can start by importing the correct packages and creating our interfaces



We can create our main class that sets out sensible defaults



We can then utilise this L3 construct without having to think about the internals

```
export class DynamoDbTable extends Construct {
  public readonly table: dynamodb.Table;

  constructor(scope: Construct, id: string, props: DynamoDbTableProps) {
    super(scope, id);

    const fixedProps: FixedDynamoDbTableProps = {
      billingMode: dynamodb.BillingMode.PAY_PER_REQUEST,
      pointInTimeRecoverySpecification: {
        pointInTimeRecoveryEnabled: true,
      },
      contributorInsightsEnabled: true,
    };

    this.table = new dynamodb.Table(this, id + 'Table', {
      // fixed props
      ...fixedProps,
      // custom props
      ...props,
    });
  }
}
```

Basic Code example (TypeScript)

6.0



We can start by importing the correct packages and creating our interfaces



We can create our main class that sets out sensible defaults



We can then utilise this L3 construct without having to think about the internals



```
● ● ●  
const usersTable = new DynamoDbTable(this, 'Users', {  
  tableName: 'UsersTable',  
  partitionKey: { name: 'userId', type: dynamodb.AttributeType.STRING },  
  sortKey: { name: 'createdAt', type: dynamodb.AttributeType.STRING },  
  removalPolicy: cdk.RemovalPolicy.DESTROY, // for dev environments  
});
```

Cloud Blocks by Leighton

6.0



Fully open-source and free to use



Has a collection of L3 constructs that we use internally on our projects



We will be adding more over the coming months

Cloud Blocks Construct Reference

Cloud Blocks

license MIT maintained yes Main passing Release Publication passing code style biome

Cloud Blocks is an open-source collection of **AWS CDK constructs** designed to help teams build secure, reusable, and production-ready cloud infrastructure faster.

Purpose

- Provide **reusable building blocks** for AWS CDK projects.
- Promote **best practices** in networking, observability, security, and more.
- Reduce boilerplate by delivering **ready-to-use constructs**.
- Empower teams to adopt CDK more quickly and consistently.

Installation & Usage

Install the `@leighton-digital/cloud-blocks` npm package in the project:

```
pnpm install @leighton-digital/cloud-blocks
```

Use in your CDK app:

```
import * as cdk from "aws-cdk-lib";
import { ApiGatewayCloudFrontDistribution } from "@leighton-digital/cloud-blocks";

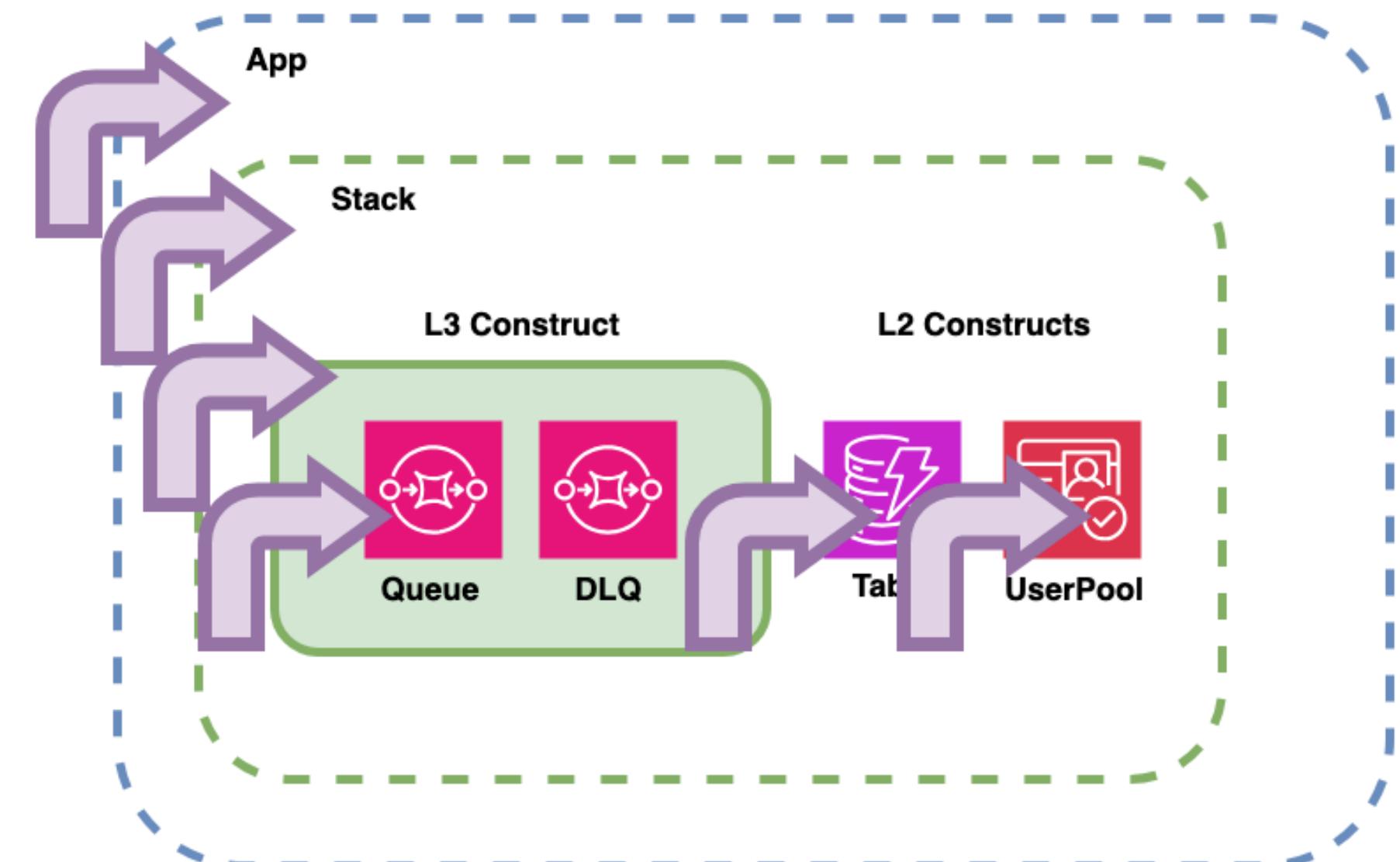
export class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
  }
}
```



Purpose
Installation & Usage
Getting Started (Developers)
Common Commands
Testing & Validation
CDK Constructs
Validation
Project Structure
Creating Cloud Constructs
Contributing

Using Aspects for tagging, policies, and enforcement

- Aspects allow us to **iterate over the tree of constructs** in our CDK application during build to apply some governance
- This is **how the CDK Nag package works**
- We can **create our own Aspects** to check our apps for our own best practices and standards



Basic Code example (TypeScript)

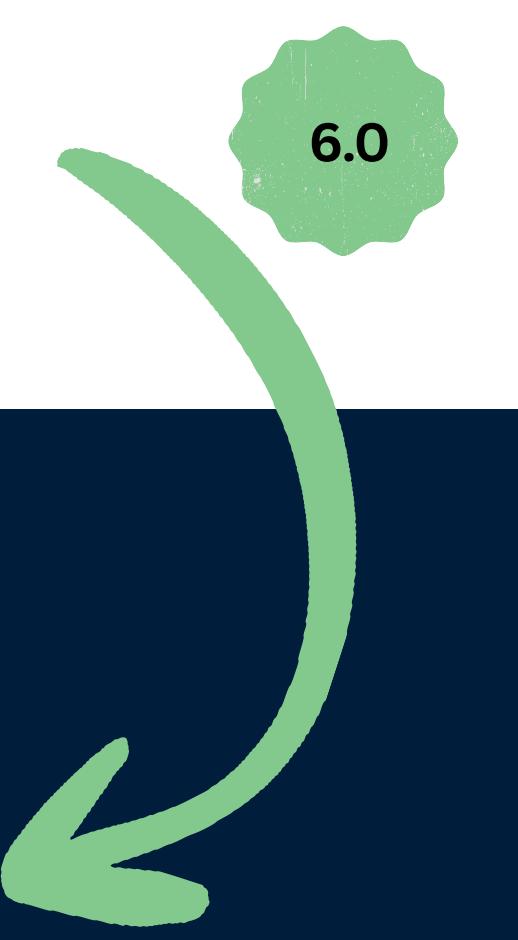
6.0



A quick example would be an aspect that checks all of the constructs in our CDK app to ensure they all have the required tags on them



We then pass in the required tags and associate them to the stack. The build will fail if any resources don't have the required tags



```
import { Annotations, type IAspect, Stack } from 'aws-cdk-lib';
import type { IConstruct } from 'constructs';

export class RequiredTagsChecker implements IAspect {
  constructor(private readonly requiredTags: string[]) {}

  public visit(node: IConstruct): void {
    if (!(node instanceof Stack)) return;

    if (!node.tags.hasTags()) {
      Annotations.of(node).addError(`There are no tags on "${node.stackName}"`);
    }

    for (const tag of this.requiredTags) {
      if (!Object.keys(node.tags.tagValues()).includes(tag)) {
        Annotations.of(node).addError(
          `${tag}` + " is missing from stack with id " + `${node.stackName}`,
        );
      }
    }
  }
}
```

Basic Code example (TypeScript)

6.0



A quick example would be an aspect that checks all of the constructs in our CDK app to ensure they all have the required tags on them



We then pass in the required tags and associate them to the stack. The build will fail if any resources don't have the required tags



```
// we import our required tags
export const requiredTags = [
  'order-service:operations:StackId',
  'order-service:operations:ServiceId',
  'order-service:operations:ApplicationId',
  'order-service:cost-allocation:Owner',
  'order-service:cost-allocation:ApplicationId',
  'order-service:cost-allocation:Environment',
];

// our own aspect to check required tags are applied
Aspects.of(this).add(new RequiredTagsChecker(requiredTags));
```



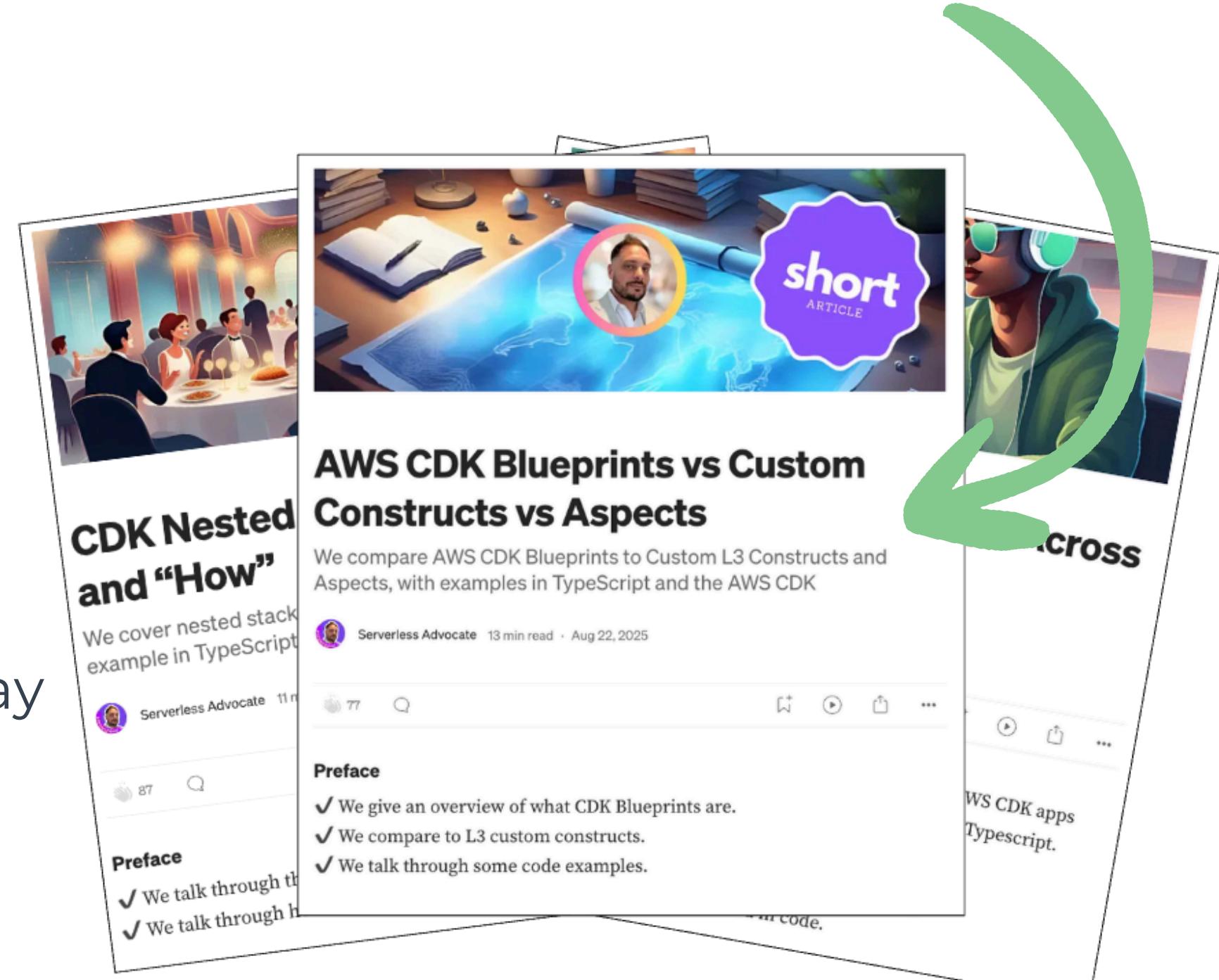
“Wrap up and next steps”

In summary, how can these techniques and frameworks help us with success?

Resources and next steps

www.serverlessadvocate.com

- ✓ Build modular and extensible solutions from the start
- ✓ Ensure they can be easily deployed to different stages with varying configurations
- ✓ Have a key focus on observability from day one
- ✓ Bake in best practices and governance through CDK Nag and a mix of your own curated constructs and aspects

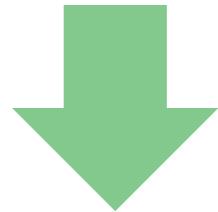


serverless
advocate

aws

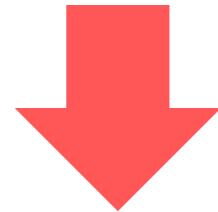
Start your AWS CDK projects in **the right way**

add here



project timeline

not here



Thank You

I challenge you to deep dive into these frameworks and patterns in your own time and educate others in the community with your learnings.

Thank you for attending today, and if you have any questions please feel free to grab me after the session

The screenshot shows a GitHub repository page for 'cdk-mindset'. The repository was created by 'leegilmorecode' and has 2 commits. It contains files such as 'docs/images', 'orders-service-cdk-mindset', 'LICENSE', and 'README.md'. The 'README' file is the active tab, displaying the title 'The CDK Mindset: Building Serverless Solutions As Repeatable Blueprints'. Below the title is a bio for Lee Gilmore, a Serverless Advocate and AWS Serverless Hero, featuring a circular profile picture and a 'HERO' badge. To the right of the repository details, there is a sidebar with sections for 'About', 'Releases', 'Packages', 'Languages', and 'Suggested workflows'. A large green curved arrow originates from the bottom right corner of the slide and points towards the 'About' section of the GitHub page.



serverless
advocate

aws