The CDK Mindset:
Building Serverless Solutions As Repeatable Blueprints

Lee Gilmore | Serverless Advocate | AWS Serverless Hero

aws
serverless
HERO

serverless advocate

aws

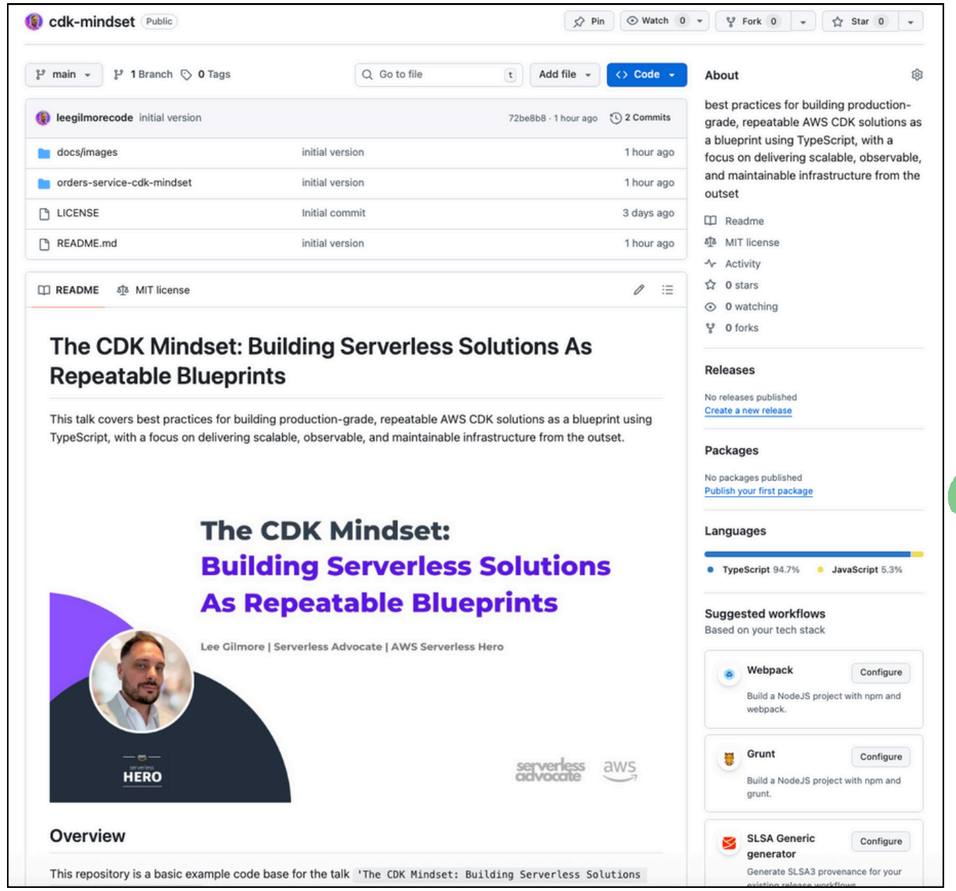
Hi, I'm Lee Gilmore

AWS Serverless Hero

www.serverlessadvocate.com

aws serverless HERO

Full code examples can be found using the QR code:



What are we covering?

From a high-level we are covering:

1.0 Making your solutions **modular** and extensible

2.0 Environment specific **configuration**

3.0 **Ephemeral environments** and CICD are key for fast flow

4.0 **Observability & monitoring** should not be an after-thought

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

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

serverless advocate aws

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.StackPro ) {
    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 wont 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!

serverless advocate aws

“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!

serverless advocate

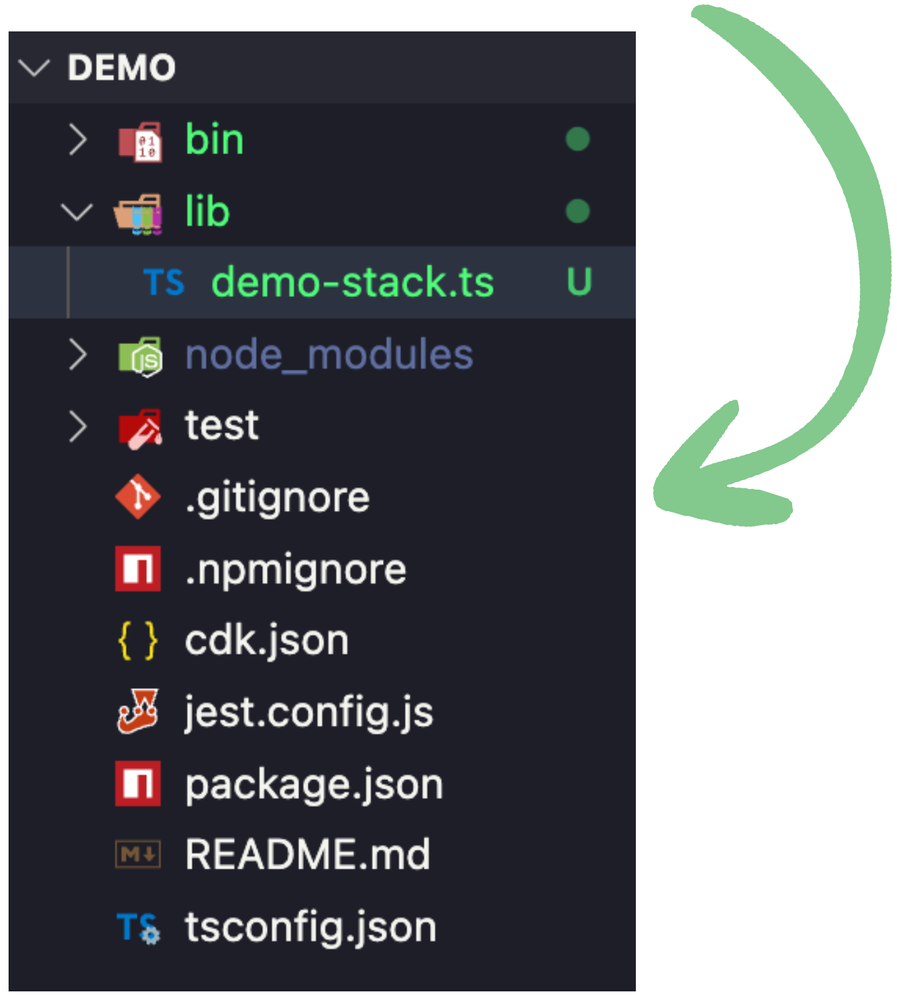
aws

1.0

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):

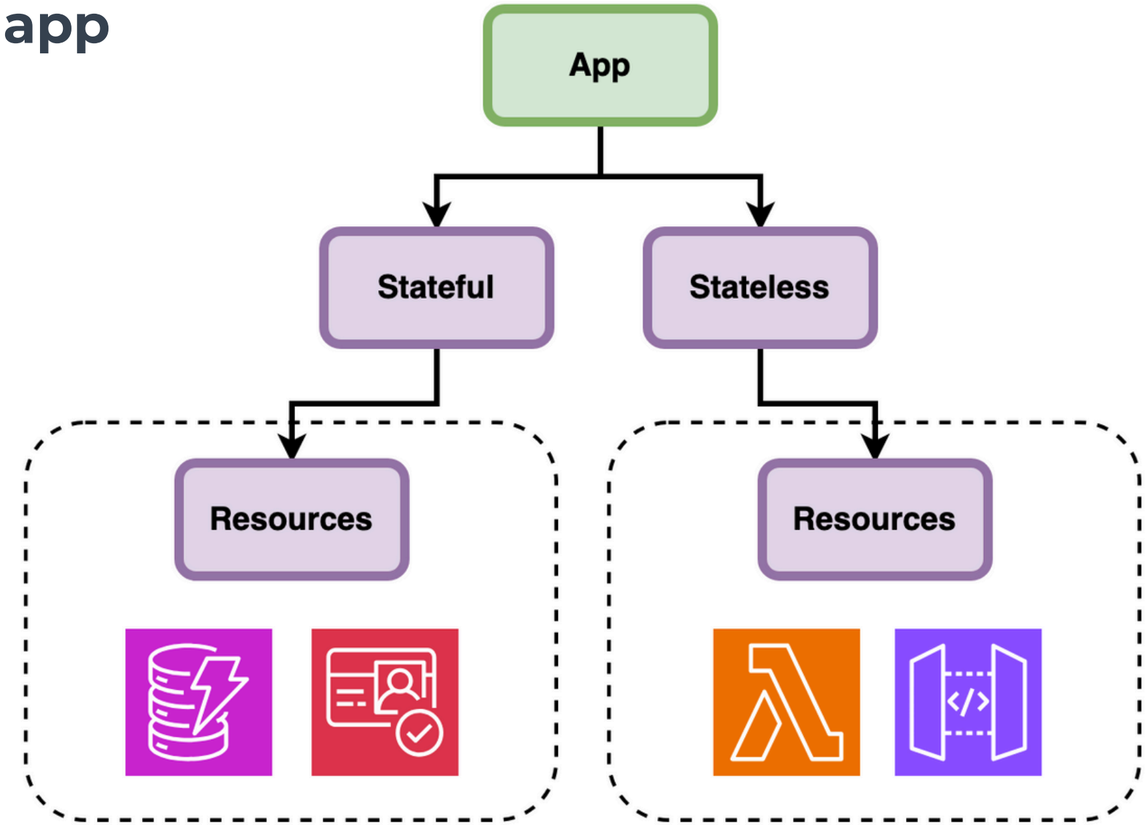
  Error: Number of resources in stack 'DemoStack': 501 is greater than allowed maximum of 500

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

1.0

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

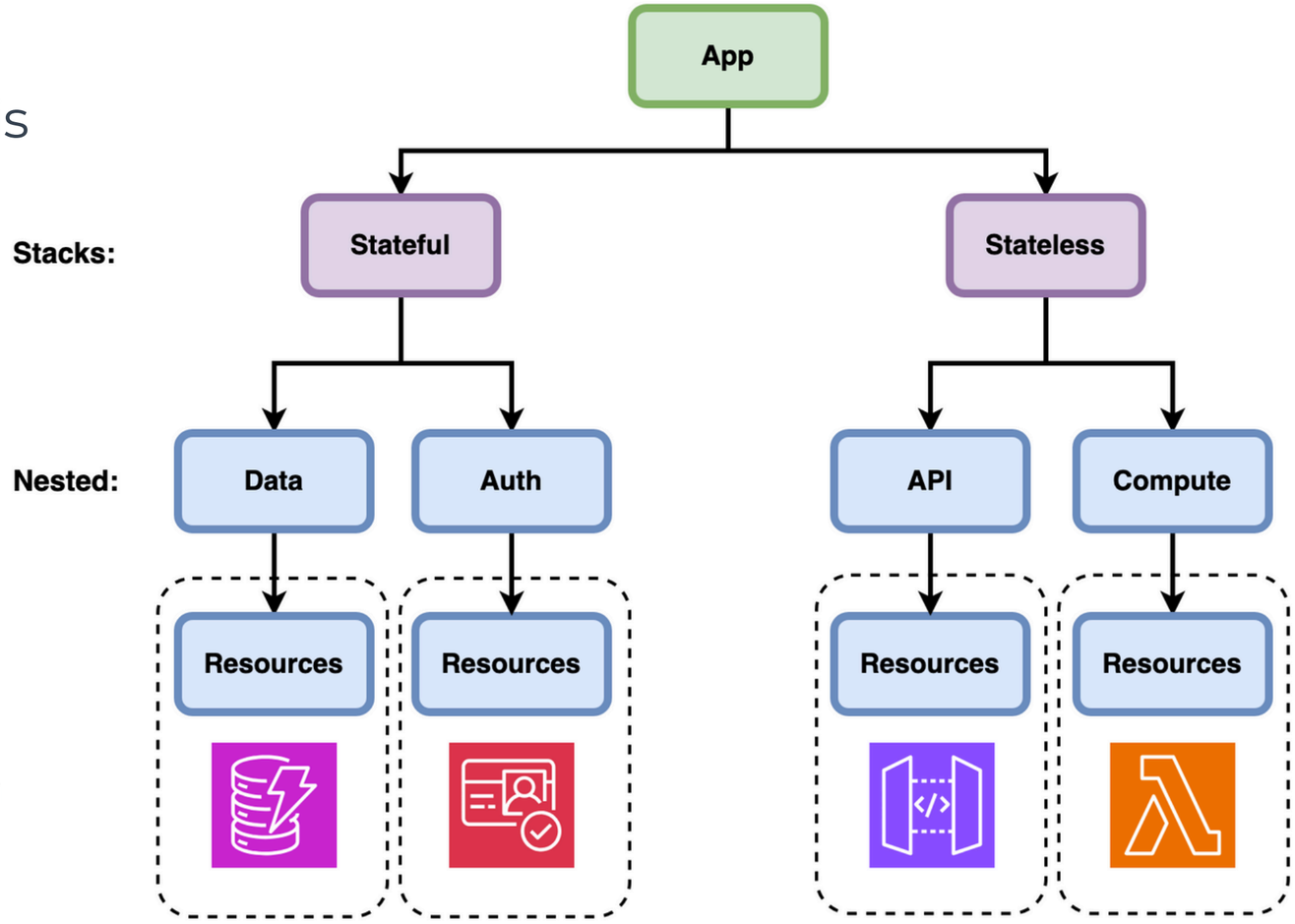


serverless advocate aws

1.0

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



serverless advocate aws

“Environment specific configuration”

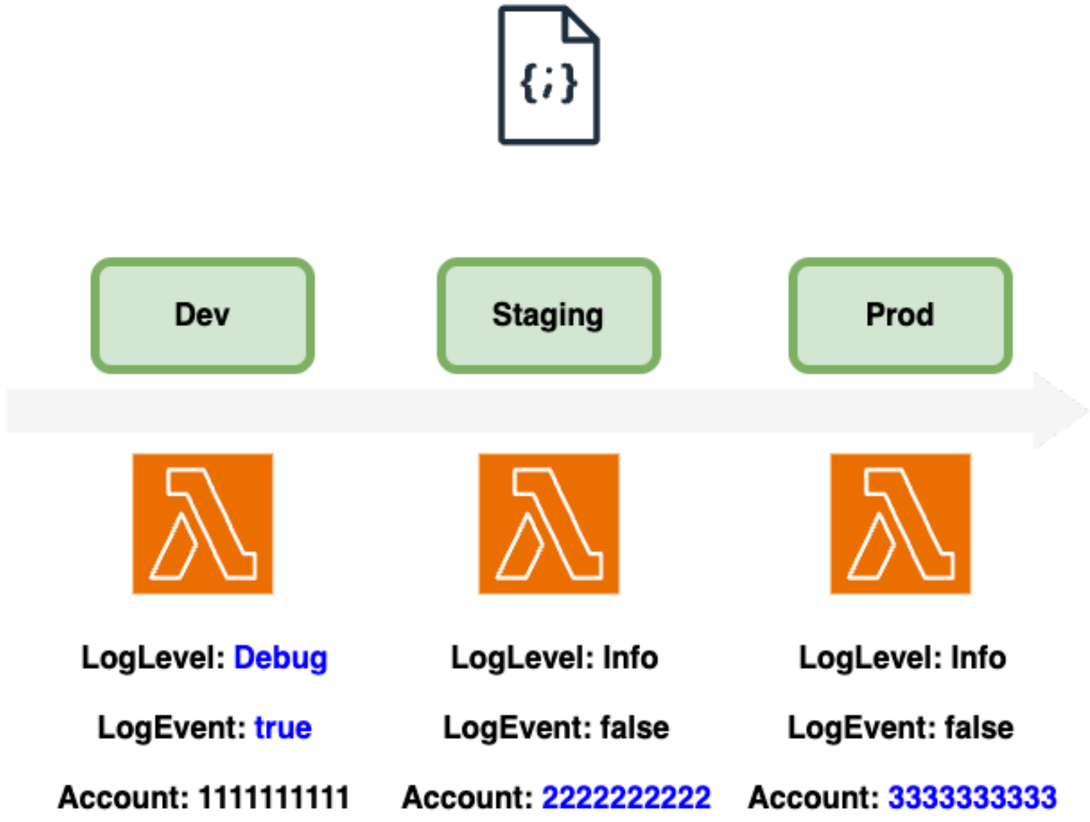
Allow your solutions to be deployed with **configuration specific to your environment and stage!**

serverless advocate

aws

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



2.0

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

2.0

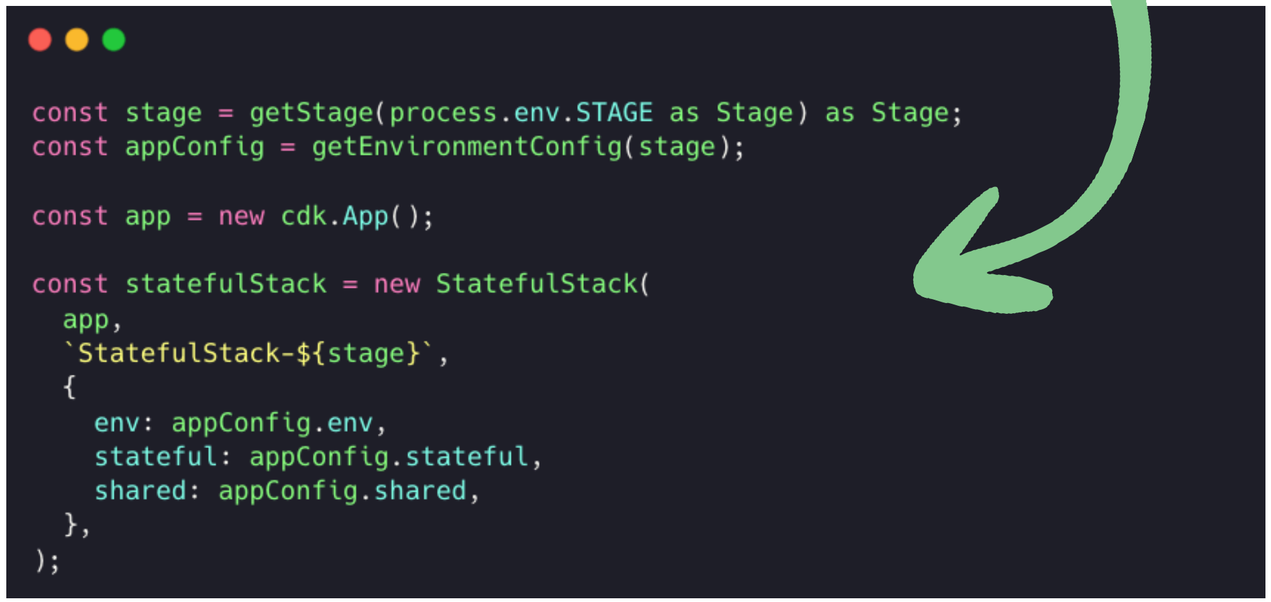
- We can create a common interface for our config using TypeScript
- **We can now create a typed config per stage through a function**
- Our function grabs the correct config for the stage you are building for
- Now each stack has its own config injected into the props per stage

```
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

2.0

- 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,
  },
);
```

2.0

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**

```ts
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;
  }
}
```

2.0

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"],
    ...
                }
}
```

“Ephemeral environments & CICD are key for flow”

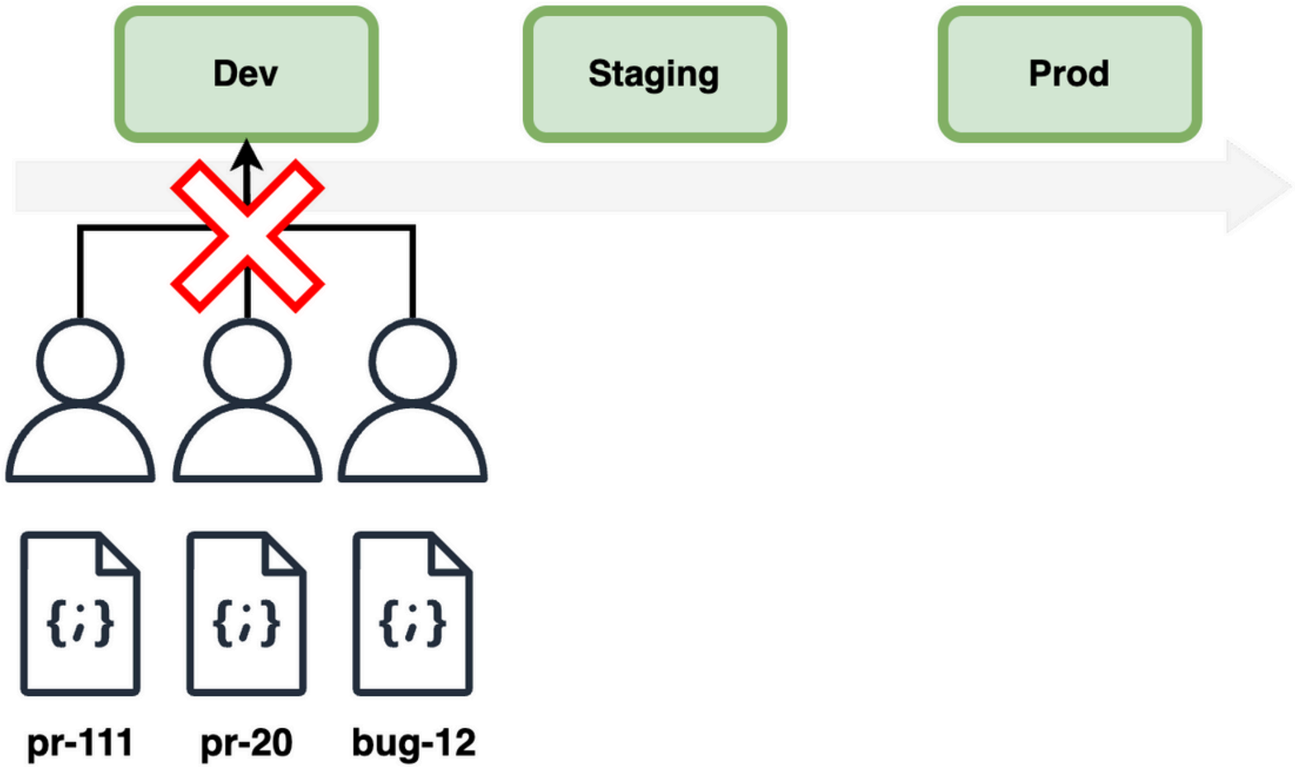
It is key that engineers can **work with ephemeral environments from the outset of a project through CICD pipelines!**

serverless advocate

aws

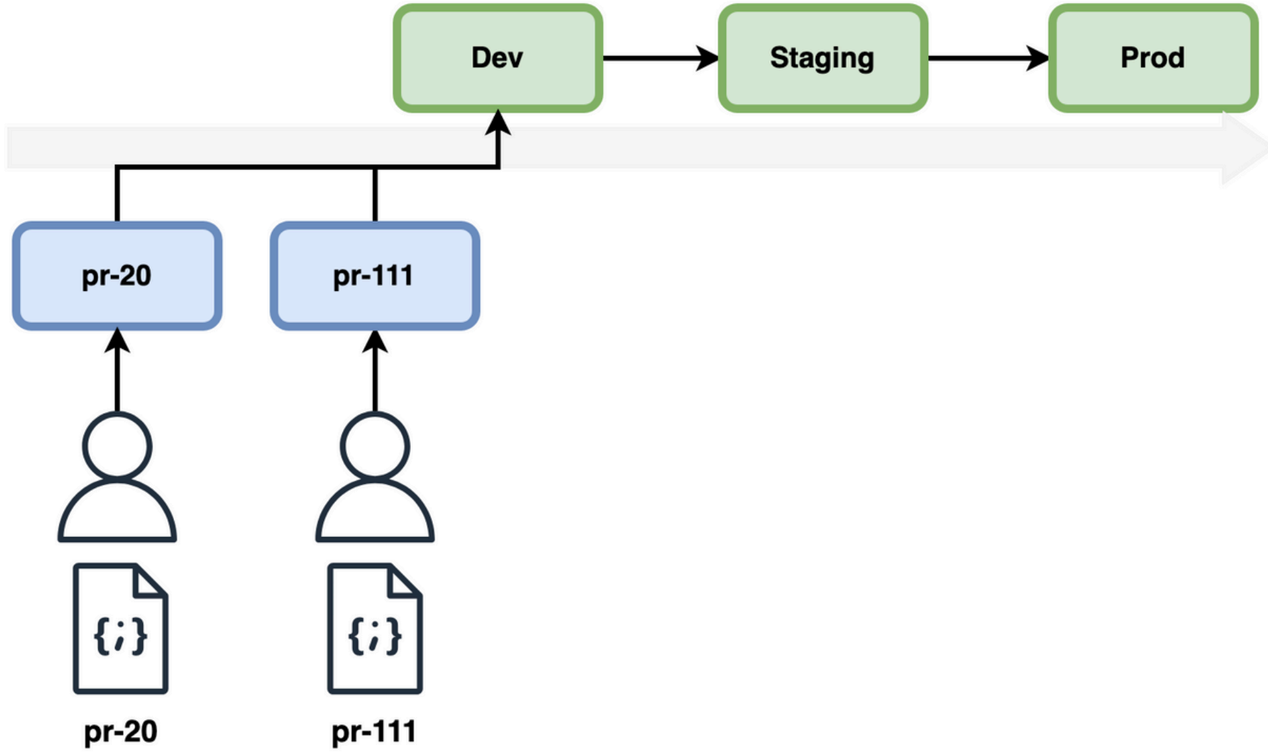
The Problem: Developers standing on toes!

- 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**



3.0

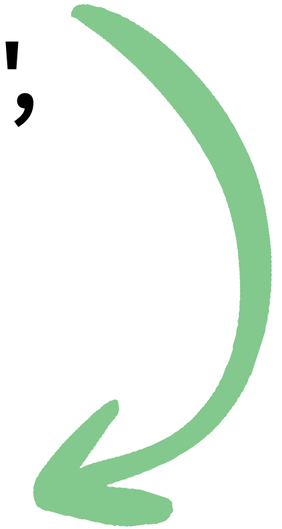
Config + Ephemeral: A match made in heaven!

- 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",
```



(Usually Jira Ticket Numbers)

feature/api-123 → api-123

hotfix/api-22 → api-22

bug/api-77 → api-77

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

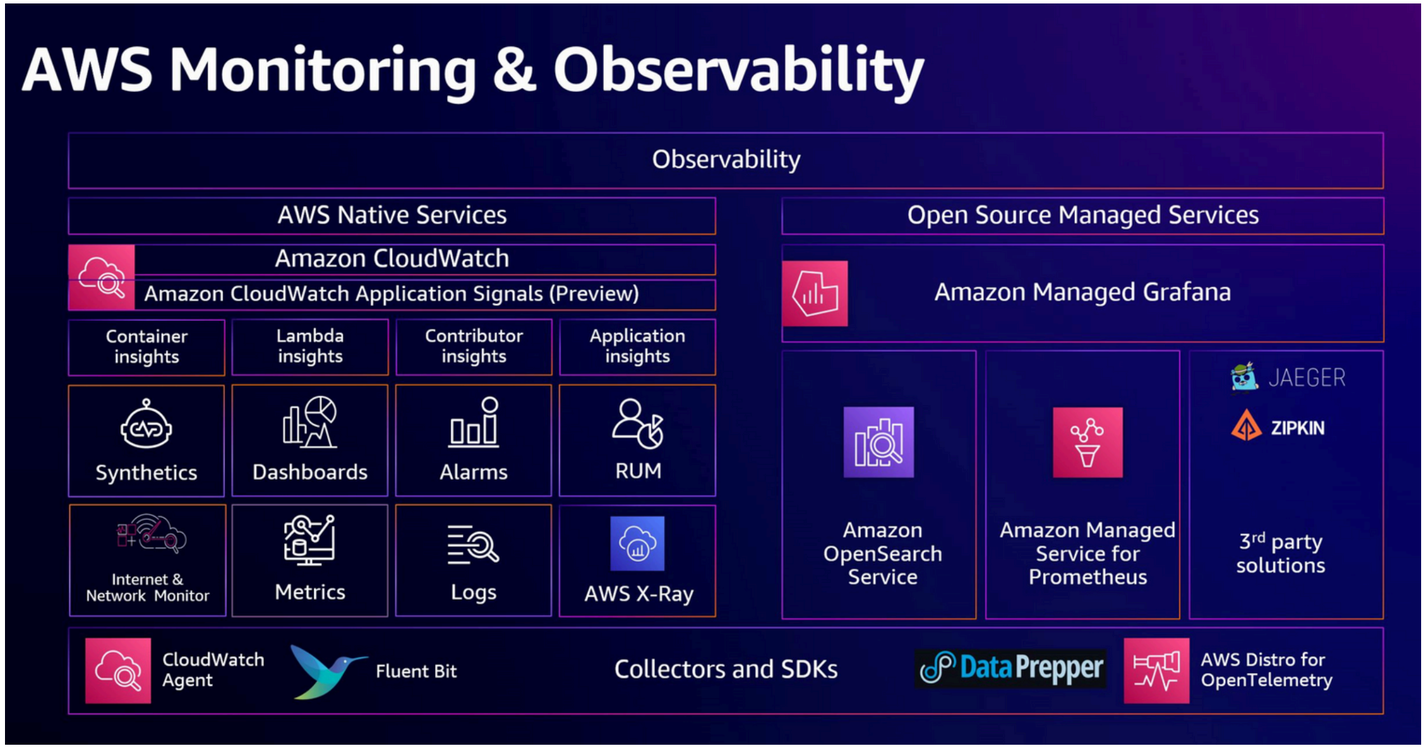
Add monitoring & observability from day one using **Lambda Powertools and the AWS CDK Monitoring constructs.**

serverless advocate

aws

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

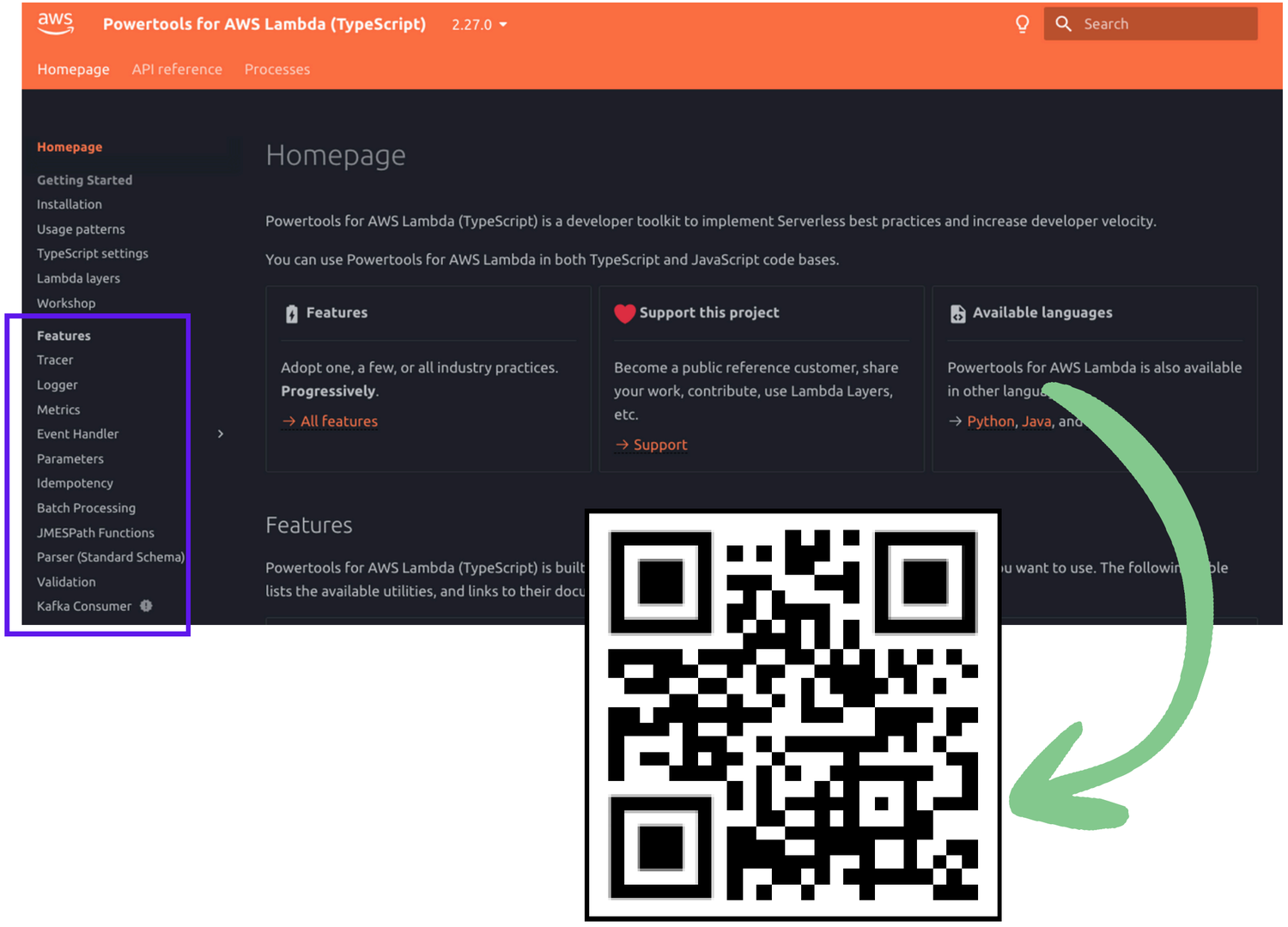
serverless advocate

aws

4.0

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



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

```typescript
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)

4.0

- 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

```typescript
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) (`.monitorApiGateway()`)	TPS, latency, errors	Latency, error count/rate, low/high TPS	To see metrics, Advanced Monit
AWS API Gateway V2 (HTTP API) (`.monitorApiGatewayV2HttpApi()`)	TPS, latency, errors	Latency, error count/rate, low/high TPS	To see route lev enable Advance
AWS AppSync (GraphQL API) (`.monitorAppSyncApi()`)	TPS, latency, errors	Latency, error count/rate, low/high TPS	
Amazon Aurora (`.monitorAuroraCluster()`)	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

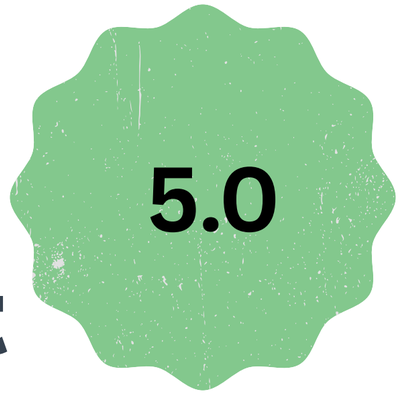
```typescript
// monitoring
const monitoringFacade = new monitoring.MonitoringFacade(
  this,
  `${stage}-orders-monitoring`,
  {
    alarmFactoryDefaults: {
      alarmNamePrefix: `${stage}-orders-dashboard`,
      actionsEnabled: true,
      // default alarm action that can be overridden
      action: 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

serverless advocate

aws

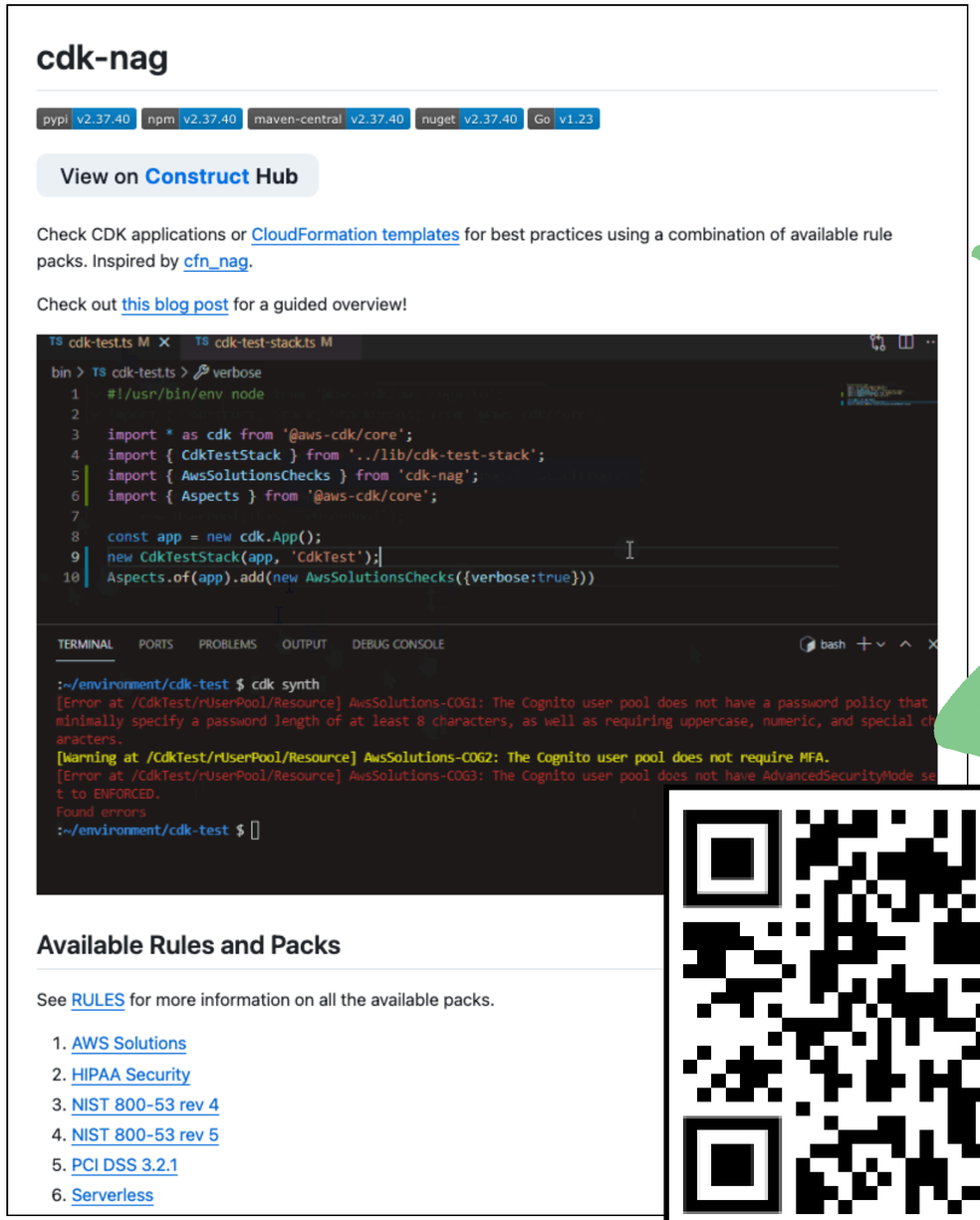
The Problem: Am I building this right?

- 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!



Basic Code example (TypeScript)

- **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

```
import { AwsSolutionsChecks, NagSuppressions } from 'cdk-nag';
```

serverless advocate aws

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

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

serverless advocate

aws

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',
        },
      ],
    );
```

serverless advocate

aws

“Aspects and L3 constructs for standards”

We can use aspects and L3 constructs for reusability and standards

serverless advocate

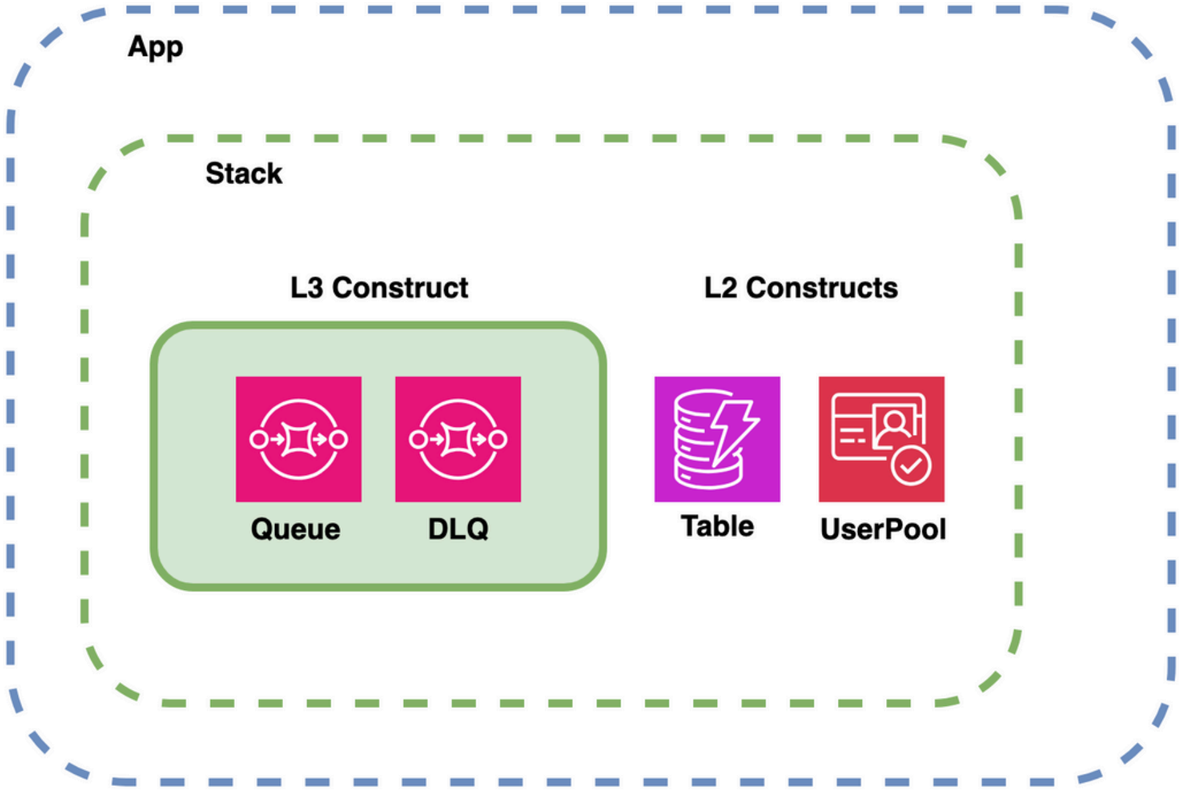
aws

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

```typescript
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

```typescript
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**

```typescript
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
});
```

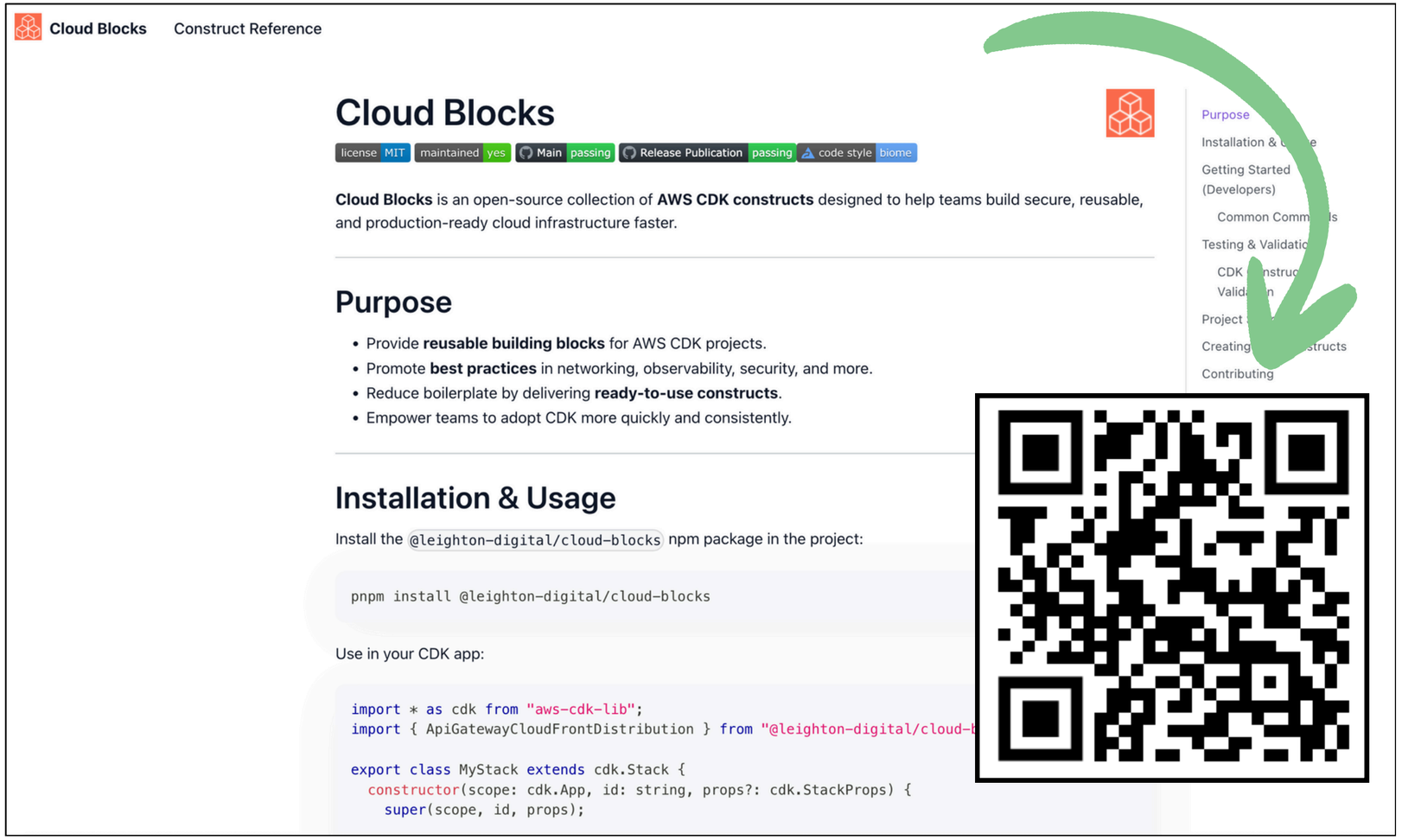
serverless advocate

aws

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



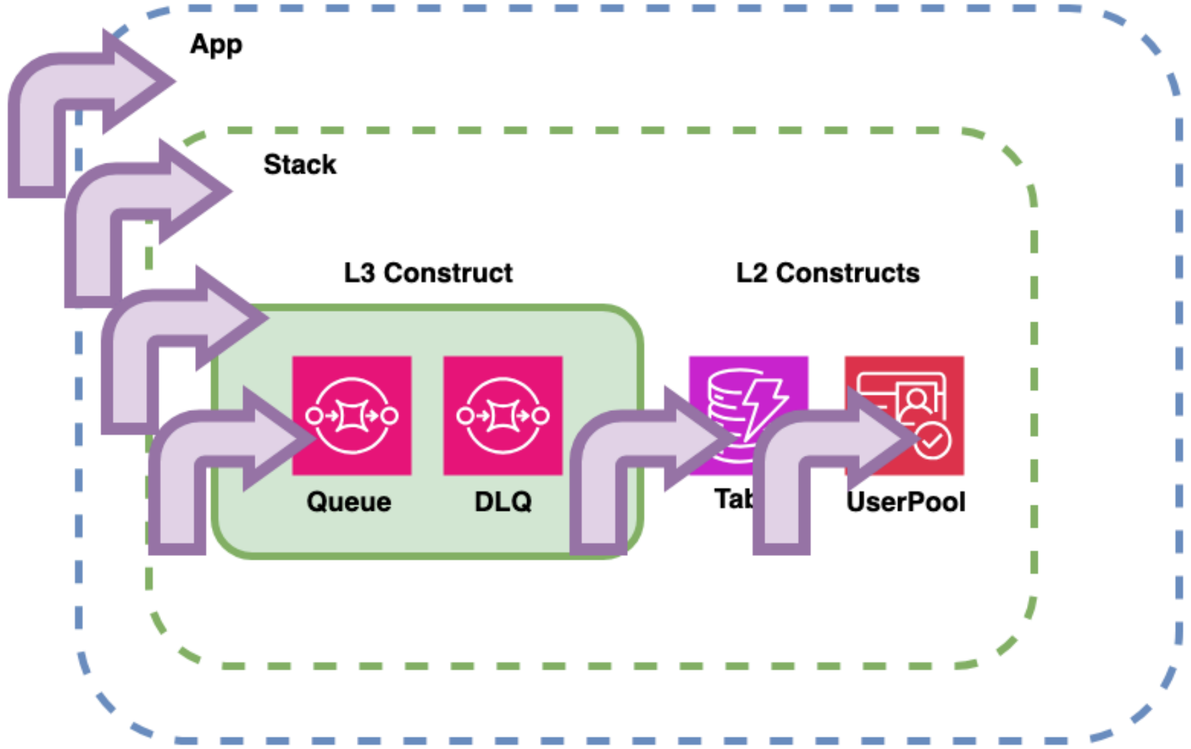
leighton

serverless advocate

aws

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

```typescript
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**

```typescript
// 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?

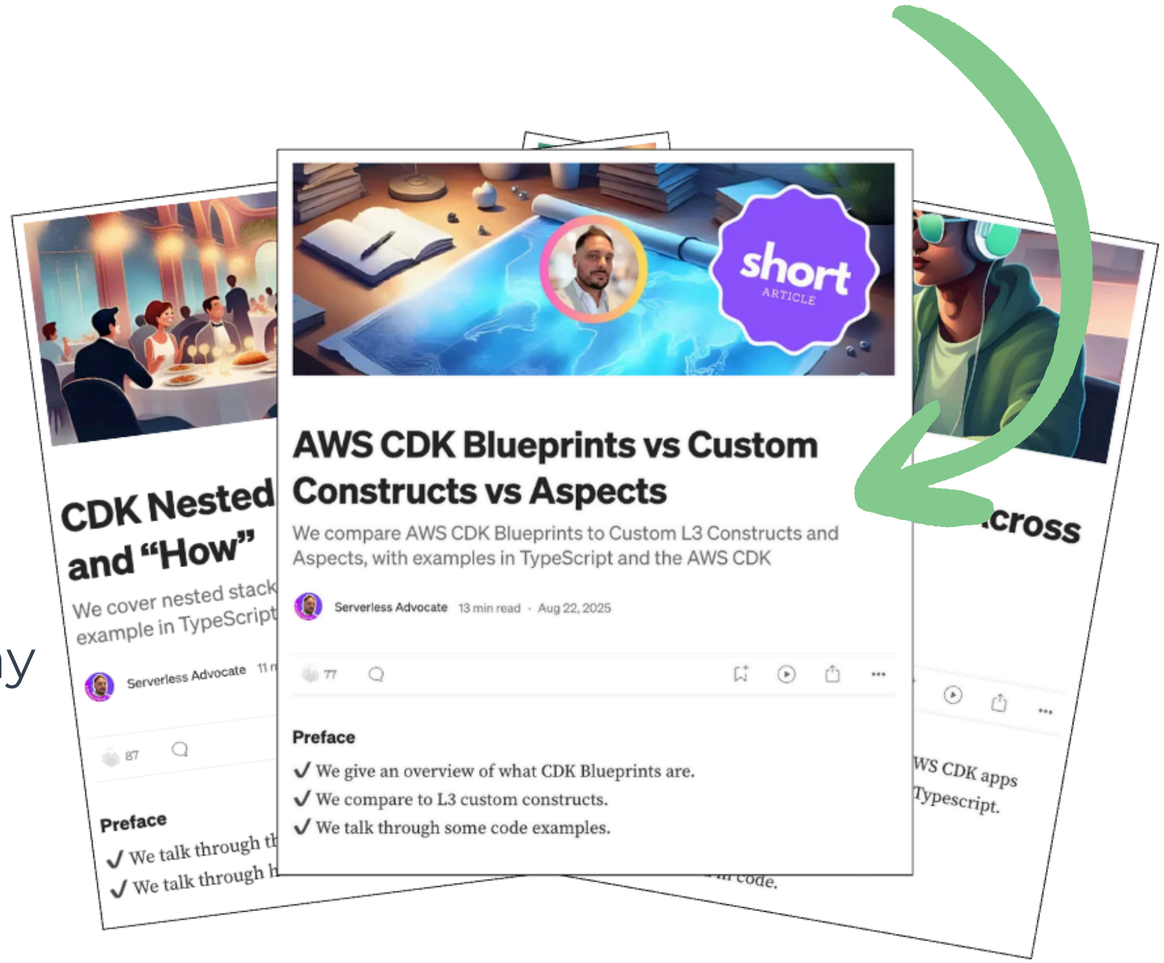
serverless advocate

aws

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

not here

project timeline

---

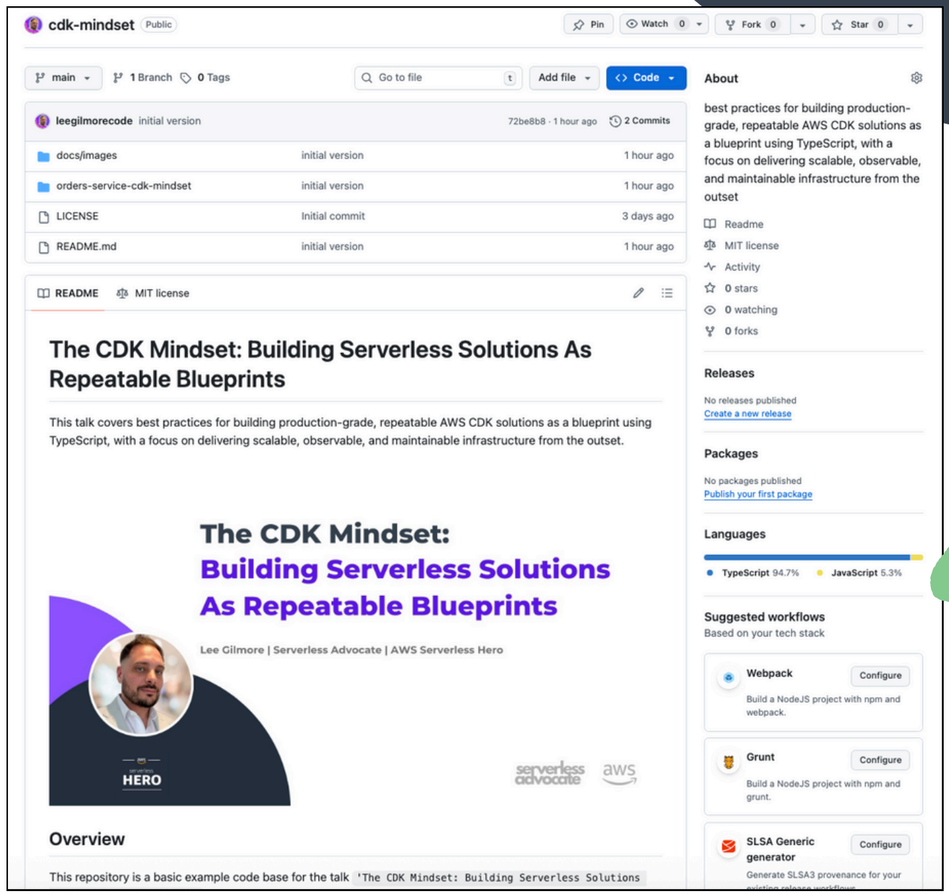
serverless advocate

aws

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



serverless advocate

aws