Infrastructure as Code

Lloyd Atkinson

With CDK for AWS

Declarative & Consistent Environments via Code

Red Hat

Infrastructure as Code (IaC) is the managing and provisioning of **infrastructure through code instead of through manual processes**. With IaC, configuration files are created that contain your infrastructure specifications, which makes it easier to edit and distribute configurations.

Microsoft

Infrastructure as code (IaC) uses DevOps methodology and **versioning with a descriptive model** to define and deploy infrastructure, such as networks, virtual machines, load balancers, and connection topologies. Just as the same source code always generates the same binary, **an IaC model generates the same environment every time it deploys**.

The problems with manual infrastructure approaches

- Human error. It’s easy to create infrastructure with any method; but it’s not easy to consistently produce the same result when running scripts ad-hoc, “one-off” deployments, and natural changes over time.
- Lack of consistency. Multiple tools, scripts, tools, and methodologies require more time spend on ensuring correctness and consistency.
- Scaling becomes too hard and has limited reproducibility.
- Slower feedback loop. Deployment becomes slower, feedback becomes slower, development becomes slower.

Automated, declarative, maintainable infrastructure as code

- Infrastructure should not be at risk of becoming fragile, resistant to change, or practically impossible to redeploy
- Infrastructure should be able to be version controlled, expressed in written language form, able to be shared, changed, and updated
- Infrastructure should be able to be created and destroyed in minutes – not months
- Infrastructure should be *declared as code* and anyone that needs should be able to make infrastructure changes and have it deployed automatically

Popular IaC solutions

aws Cloud Development Kit

AWS CLOUDFORMATION

Pulumi

HashiCorp Terraform

Microsoft Azure Resource Manager

- In general terms there are two broad categories of IaC: imperative and declarative.
- Imperative IaC is using existing languages and frameworks to create infrastructure step by step by indicating *how* the infrastructure should exist
- Declarative IaC is using schema-based Domain Specific Languages (DSLs) to indicate *what* infrastructure must exist

Popular IaC solutions

Popular IaC tools have different design philosophies; imperative vs declarative, DSLs vs existing language, cloud agnostic vs cloud specific, stateful vs stateless

	Conventions	Declarative	DSL	Cloud Agnostic	Stateless
Nix/NixOS	✓	✓	✓	✓	✓
Terraform	✓	✓	✓	✓	✕
AWS Cloudformation	✓	✓	✓	✕	✓
Azure ARM/Bicep	✓	✓	✓	✕	✓
AWS CDK	✓	✕	✕	✕	✕
Pulumi	✓	✕	✕	✕	✕
PowerShell, Bash, Ansible, Chef, etc	✕ / 🤔	✕	✕	✕	✕

IaC maturity rank

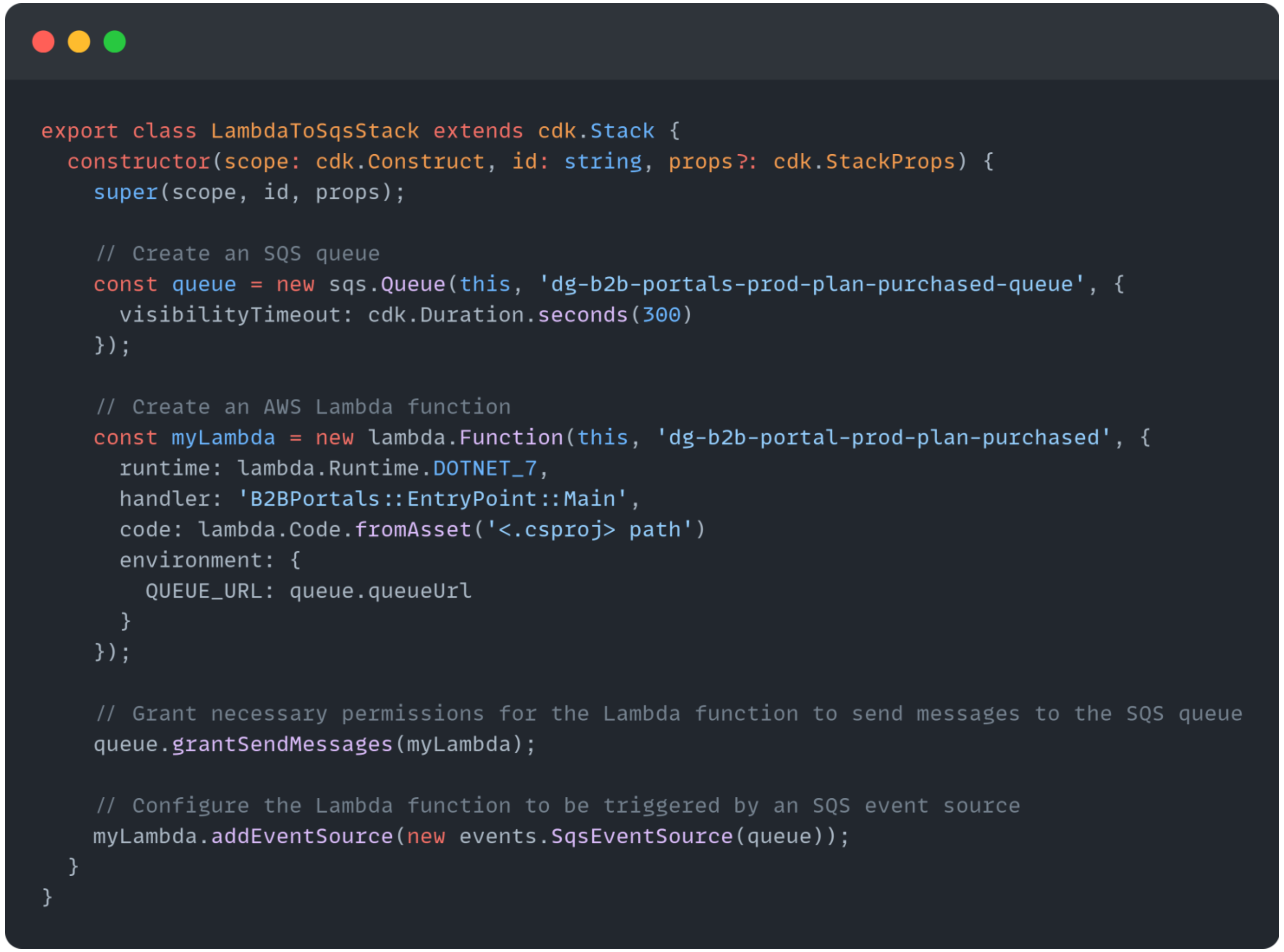
Rank	Solution
A	Automated CI/CD, fast feedback loops, declarative infrastructure as code, unique environments can be rapidly created with PRs in source control, frequent and low ceremony releases, multiple releases daily if desired, unit testing of IaC
B	Automated CI, some manual processes, releases have a degree of ceremony, infrequent releases, some deployment scripts
C	Deployments involve SSH or RDP with servers and manually deploying files, “big bang” releases every months or quarters, slow feedback loop
D	No automation at all, adoption of infrastructure as code requires fully rearchitecting the stack, configuration is not in source control, requires constant usage of GUIs, lack of understanding of how the platform operates

We want to be here!

AWS Cloud Developer Kit (CDK)

- We’ll be using CDK for this project
- We should use existing CDK based projects in the D&G GitHub as a reference to familiarises ourselves with existing conventions and practices
- Following existing usage in other projects in our GitHub, we will be using TypeScript with the CDK library

CDK Example: Lambda Function processing SQS messages



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

    // Create an SQS queue
    const queue = new sqs.Queue(this, 'dg-b2b-portals-prod-plan-purchased-queue', {
      visibilityTimeout: cdk.Duration.seconds(300)
    });

    // Create an AWS Lambda function
    const myLambda = new lambda.Function(this, 'dg-b2b-portal-prod-plan-purchased', {
      runtime: lambda.Runtime.DOTNET_7,
      handler: 'B2BPortals::EntryPoint::Main',
      code: lambda.Code.fromAsset('<.csproj> path')
      environment: {
        QUEUE_URL: queue.queueUrl
      }
    });

    // Grant necessary permissions for the Lambda function to send messages to the SQS queue
    queue.grantSendMessages(myLambda);

    // Configure the Lambda function to be triggered by an SQS event source
    myLambda.addEventSource(new events.SqsEventSource(queue));
  }
}
```

CDK Constructs and Stacks

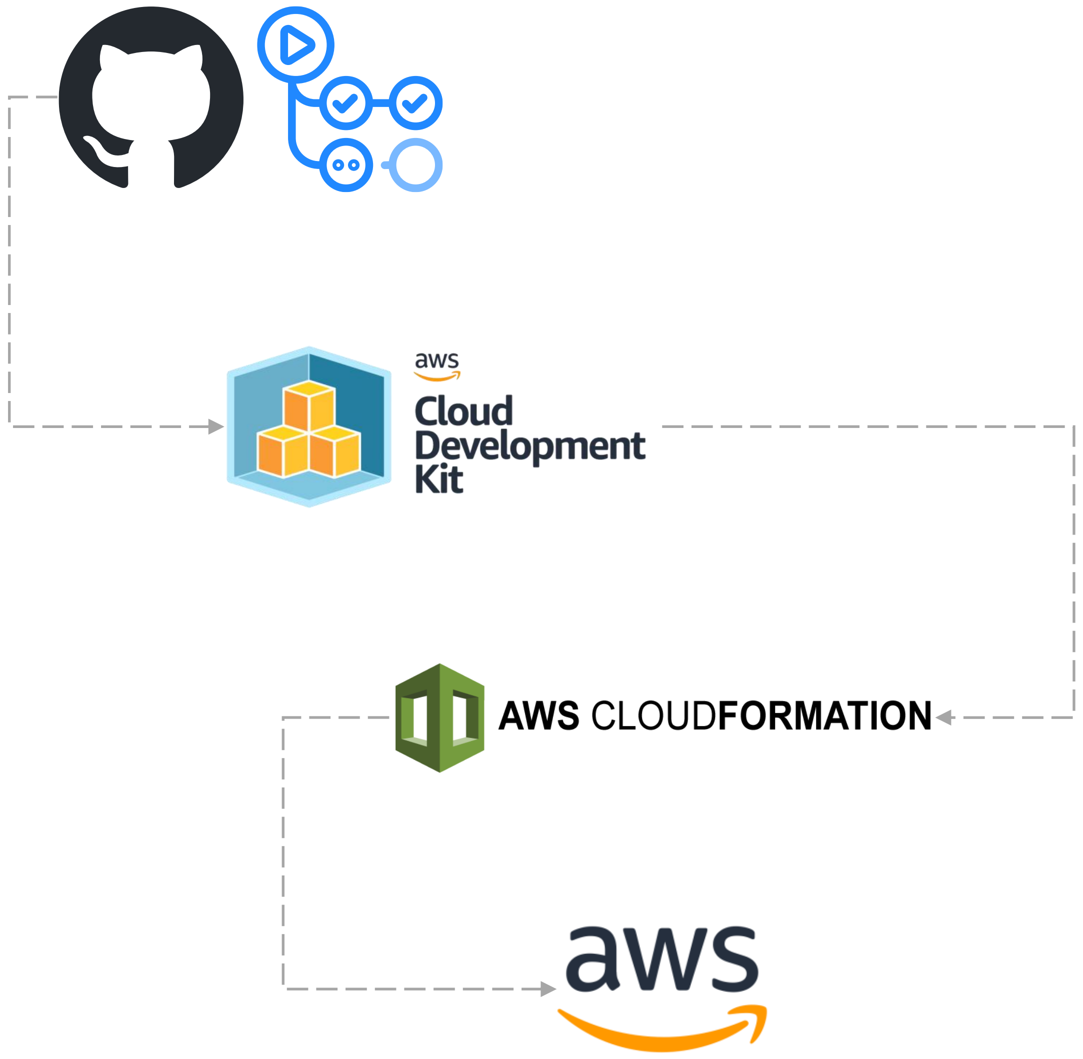
> *The unit of deployment in the AWS CDK is called a stack. All AWS resources defined within the scope of a stack, either directly or indirectly, are provisioned as a single unit.*
>
> AWS Documentation

- Not native AWS terminology, it is CDK specific terminology
- At a high level, constructs are types available in the CDK library for use by developers
  - Lambda, S3 Bucket, API Gateway are examples of constructs
- Stacks are written by developers to define their infrastructure needs
- In our project, some examples of stacks could be:
  - `FrontendStack` – the React SPA
  - `BackendStack` – the .NET Lambdas
  - `ConfigurationStack` – the database for our various portals and their products
  - `DashboardStack` – the database and dashboard for the team to see how many plans are sold daily, how many errors occur, application performance, etc

IaC & CDK Best Practices

- Follow patterns we, as developers, hopefully already follow; modularity, reusability, testing
- AWS Docs provide examples of unit testing CDK
- Ensure appropriate secrets management
- Use IaC to its full capability and don't use environment variables as that is now a redundant step
- Do not be tempted to make "one off" changes to infrastructure through a GUI – if a change is needed then refactor the CDK and deploy it
- Implement the *AWS Well-Architected* guidelines and documentation
- Separate stateless and stateful stacks
    - Have a stack containing databases so that you are free to redeploy your stack containing stateless microservices as often as you like without impact on your database stacks

Workflow combining IaC and CI/CD



Summary

- When planning for new or existing infrastructure always create an architecture diagram
- Try out changes to infrastructure by creating a PR and GitHub Actions will create a unique environment for you – don’t make “one off” changes in AWS UI
- We are using CDK for our IaC needs
- CDK has several best practices documented
- CDK is comprised of stacks, constructs, and many other types
- IaC will allow us to be more agile with tighter feedback loops and faster deployments