Reaktor

Migrating hybrid infra architecture from CloudFormation to AWS CDK

Markus Lindqvist
Senior Solutions Architect
AWS User Group Tampere, November Meetup 2019

Reaktor
Yliopistonkatu 4
00100, Helsinki Finland

tel: +358 9 4152 0200
www.reaktor.com
info@reaktor.com

Contents

1. The Cloud is a lie
2. Migration from CloudFormation to AWS CDK
3. Least privilege IAM roles
4. Automated container management
5. Conclusions
6. Discussion, Q&A

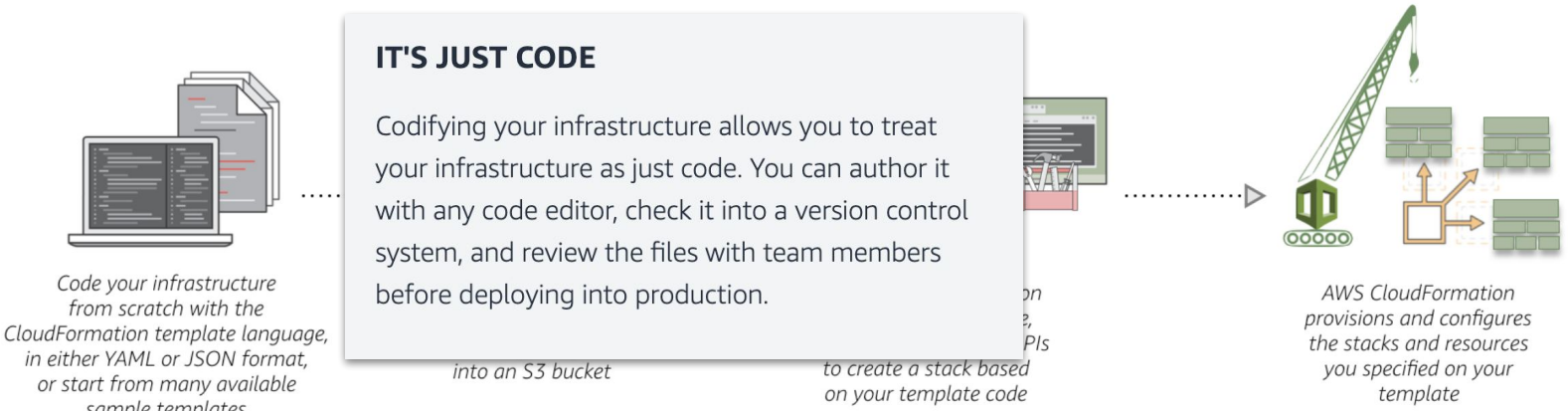
The Cloud is a lie!

```
Resources:
  VirtualMachine:
    Type: 'AWS::EC2::Instance'
    CreationPolicy:
      ResourceSignal:
        Timeout: PT10M
    Properties:
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal --exit-c
          # run some commands
          /opt/aws/bin/cfn-signal --exit-code 0
```

My journey to AWS

What is CloudFormation

“AWS CloudFormation provides a common language for you to describe and provision all the infrastructure resources in your cloud environment”



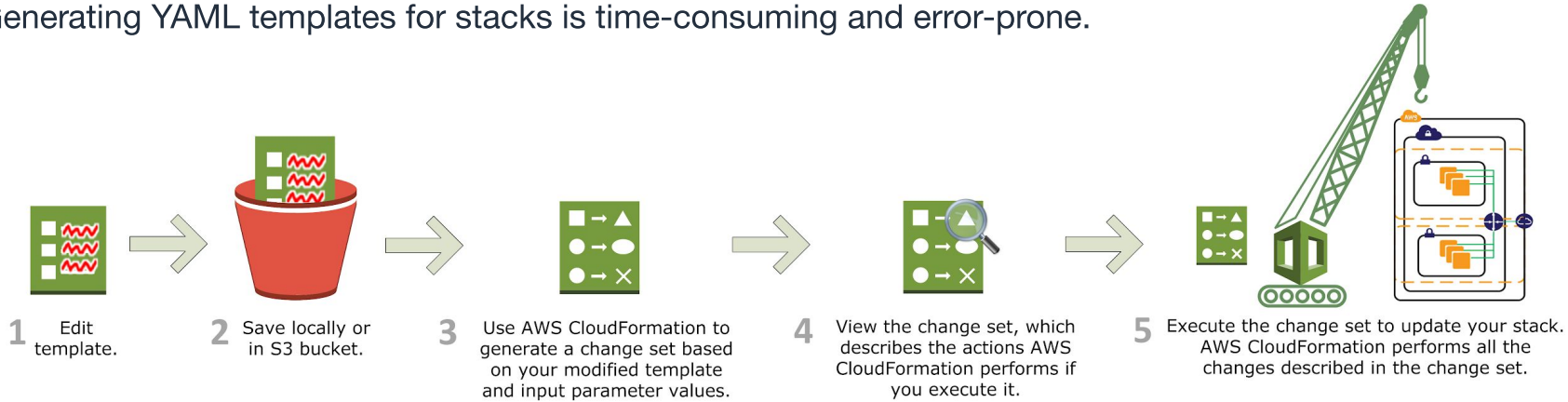
Source: https://aws.amazon.com/cloudformation/

CloudFormation workflow

With CloudFormation you work with **Stacks** that contain resources.

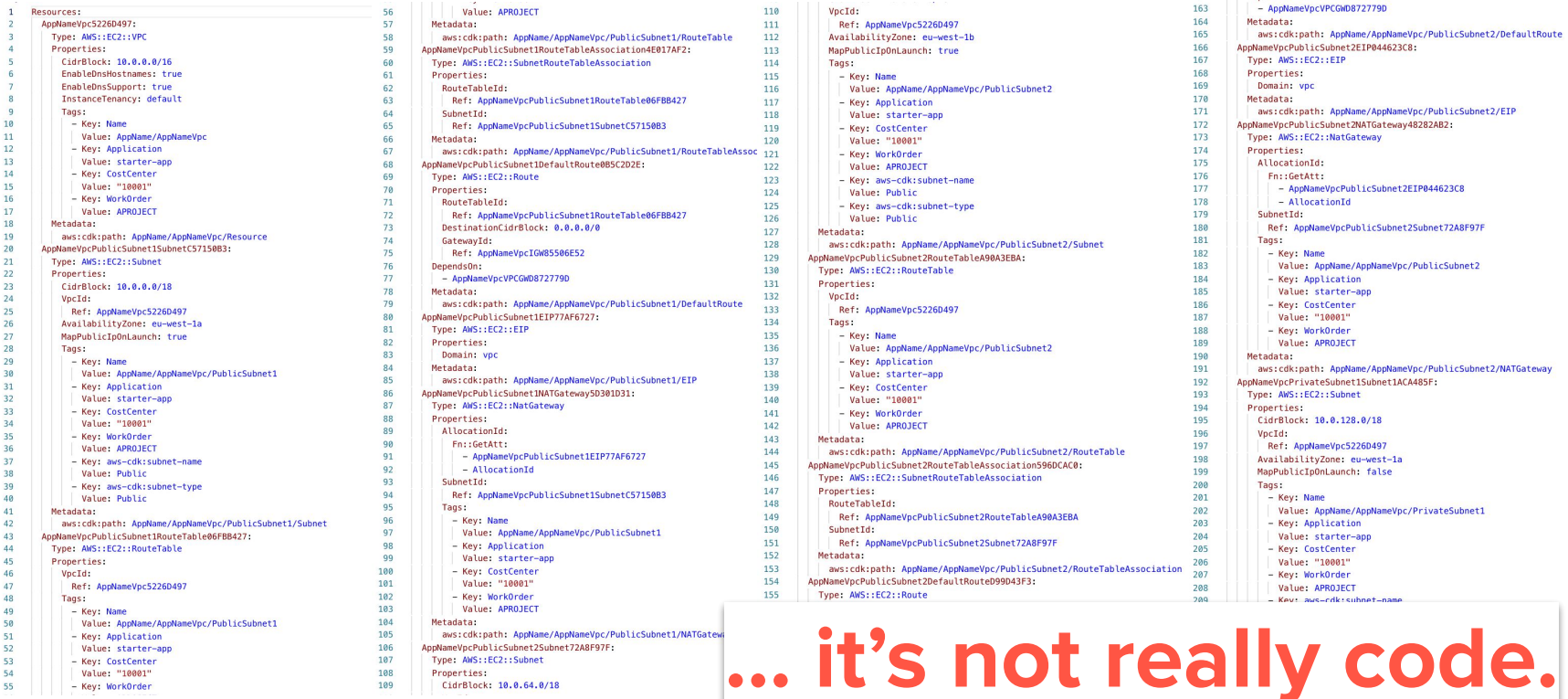
Updating stacks happens via the CLI, Web Console or CodePipeline.

Generating YAML templates for stacks is time-consuming and error-prone.



Source: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-howdoesitwork.html

CloudFormation “code”



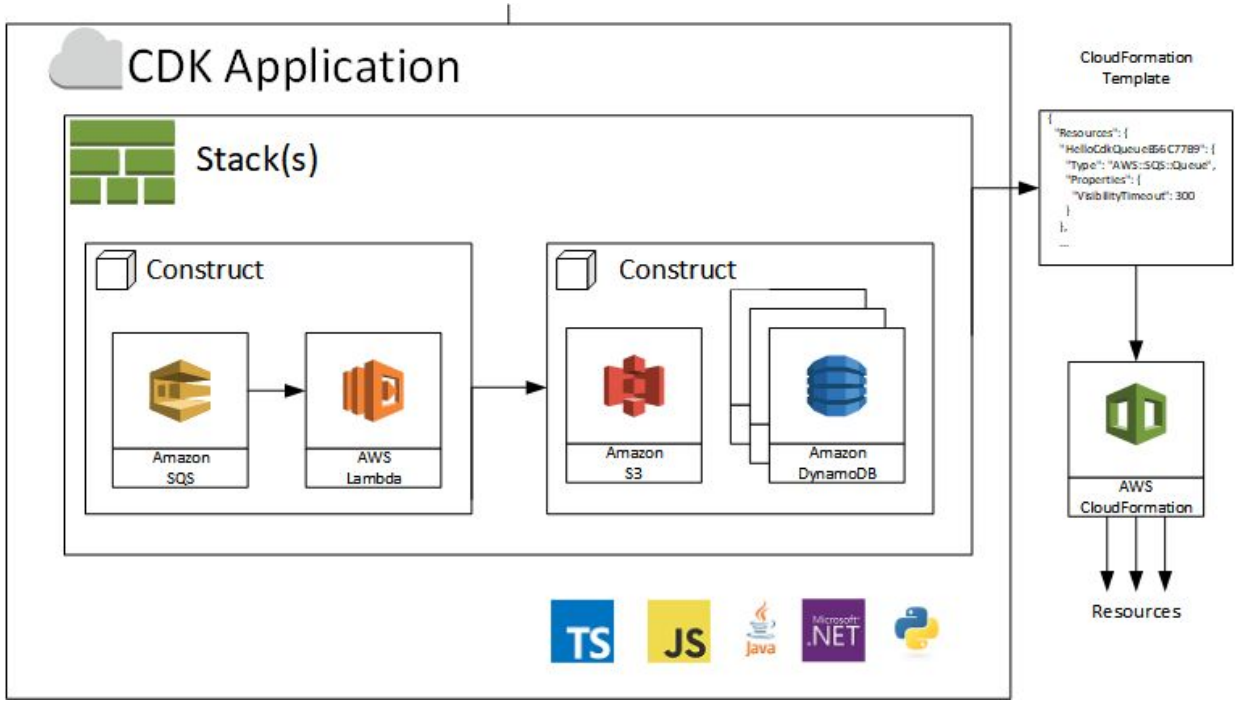
… it’s not really code.

What is AWS CDK?

With AWS CDK you can define CloudFormation resources with multiple programming languages (for example TypeScript).

Concepts of CDK:

- Generate CloudFormation templates
- Deployment tools included
- Reusable constructs



Source: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-howdoesitwork.html

Using TypeScript for producing YAML templates

```
import * as cdk from '@aws-cd
import * as s3 from '@aws-cdk/a

const myApp = new cdk.App();
const stack = new cdk.Stack(myApp, 'BucketStack');
new s3.Bucket(stack, 'ExampleBucket', {
 removalPolicy: cdk.RemovalPolicy.DESTROY,
});
myApp.synth();
```

All CDK apps have a single ‘app’

An app can have multiple stacks

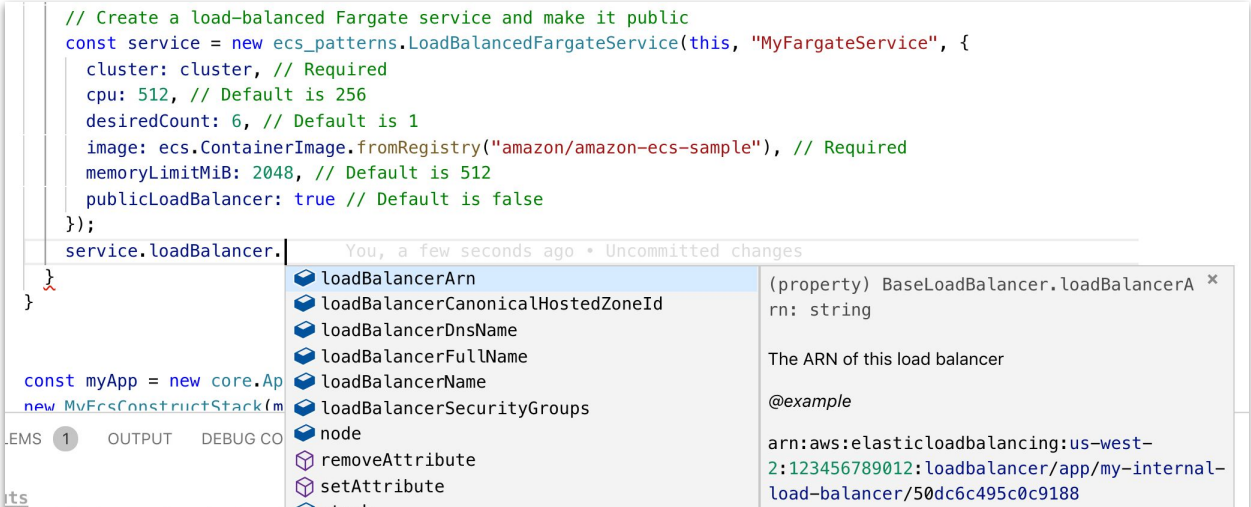
Different resources are instantiated in stacks

```
                                    y: Delete
    DeletionPolicy: Delete
    Metadata:
      aws:cdk:path: BucketStack/ExampleBucket/Resource

                                    etadata

                                    s-cdk=1.0.0,jsii-runtime=node.js/v12.0.0
```

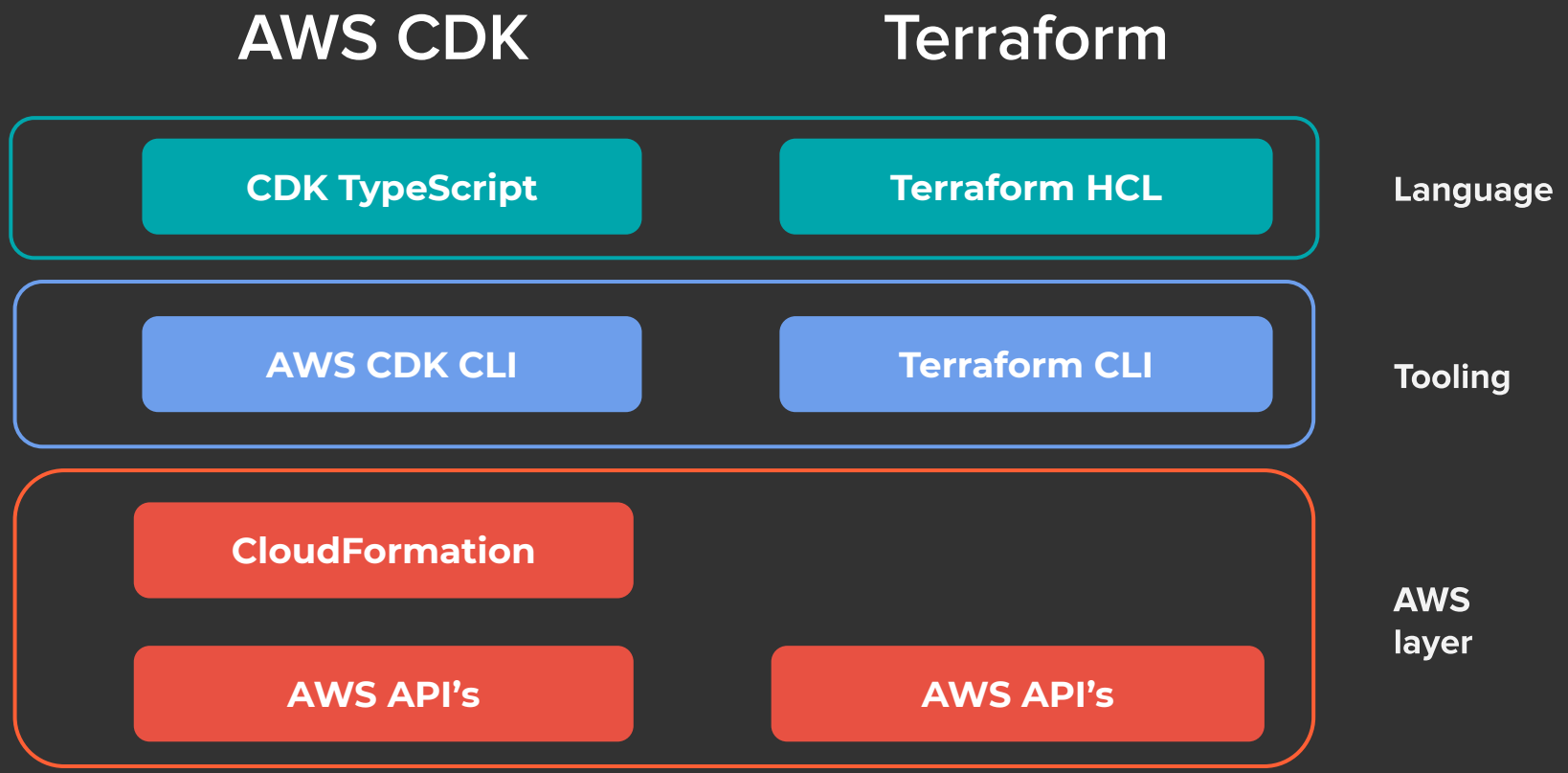
aws-cdk-example.ts → **CDK synth** → Synthesized CloudFormation stack

Main advantages of AWS CDK for developers

- AWS CDK has good IDE-support and tooling
  - This means code completion, inline documentation and smooth navigation!
- AWS CDK is as human readable as code
- Easier to reuse higher-level constructs
  - Use existing infra for sharing components (NPM)
- Deployment methods
- Open source
- AWS official tools



Differences between AWS CDK and Terraform

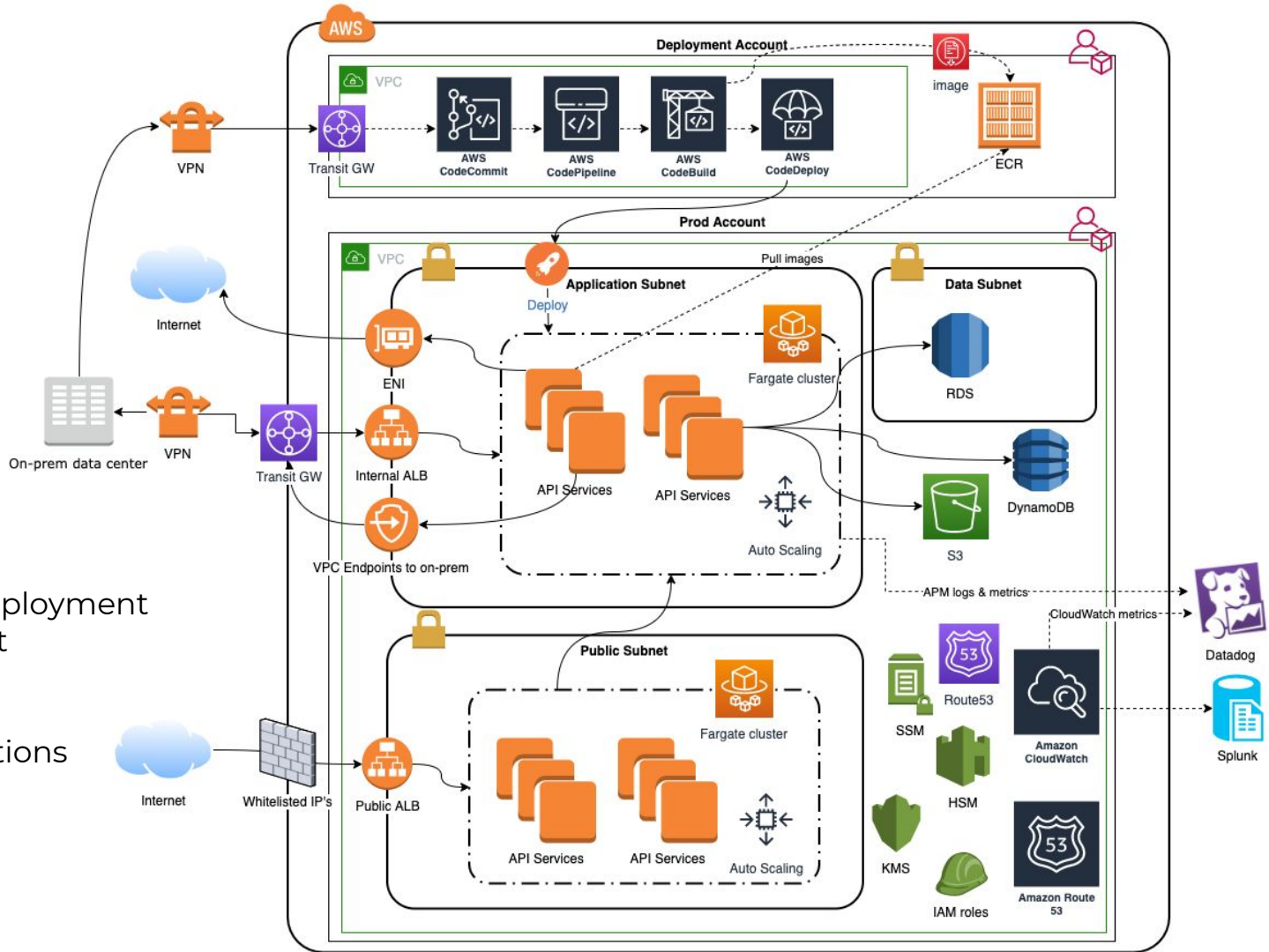


Migration from CloudFormation to AWS CDK

Production grade AWS hybrid infrastructure

Features:

- Fargate cluster
- Cross Account Deployment
- Log management
- Monitoring
- Backups
- On-prem connections
- High security



The infrastructure to be migrated from CF to CDK

Data stacks to be migrated:

- CodeCommit repositories
- ECR repositories
- S3 bucket stacks
- Relational databases
- DynamoDB databases
- SSM parameters
- Route53 stacks

The databases and repositories need migrations before they can be redeployed as AWS CDK stacks.

Functional stacks to be migrated:

- CodePipeline stacks
- Lambda stacks (devops purposes)
- Fargate clusters with public load balancer
- Fargate clusters with internal load balancer

Functional stacks can be fully redeployed (almost) any time with mostly just DNS changes due to load balancer addresses changing.

Redeploying CodeCommit as AWS CDK

Yes, it really is this simple :-)

```
const repositoryNames = [ 'api-service-1', 'api-service-2', 'api-service-3', 'api-service-4', ... ];

export const createCodeCommitStack = (app: cdk.App, props: cdk.StackProps) => {
    const stack = new cdk.Stack(app, 'code-commit-repositories', props);
    const codeCommitRepos = repositoryNames.map((name) =>
        new codecommit.Repository(stack, `${name}`, {
            repositoryName: name,
        }));
    codeCommitRepos.forEach((repo) => applyTags(repo, props.tags as Tags));
};
```

Redeploying ECR's as AWS CDK

```
export const createEcrRepositories = (app: cdk.Construct, props: cdk.StackProps) => {
    const stack = new cdk.Stack(app, 'ecr-repositories', props);
    const repositories = ['repository-1', 'repository-2', 'repository-3'];
    repositories.forEach((r) => createEcrRepository(stack, r));
}

const createEcrRepository = (stack: cdk.Stack, name: string) => {
    const accounts = [DevopsAccountNumber, DevAccountNumber, ProdAccountNumber];
    const ecrRepository = new ecr.Repository(stack, `${name}Ecr`, {
        repositoryName: name,
    });
    ecrRepository.addToResourcePolicy(new iam.PolicyStatement({
        actions: ['ecr:GetDownloadUrlForLayer', 'ecr:BatchGetImage', 'ecr:BatchCheckLayerAvailablity'],
        principals: accounts.map((account) => new iam.AccountPrincipal(account)),
    }));
};
```

Least privilege IAM roles

Integrating Datadog monitoring

```
npm install --save-dev
   aws-cdk-datadog-ecs-integration
```

Adding a Datadog sidecar container couldn't be easier. Using pre-made reusable AWS CDK construct libraries makes extending stacks extremely easy.

```
const environment = {
  DD_APM_IGNORE_RESOURCES: 'GET .*health-check',
  DD_TAGS: `env:${env}`
};
new DatadogFargateIntegration(stack, `${service.name}`,
  taskDefinition,
  {
    datadogApiKey,
    environment,
    logging,
  }
);
```

Log delivery to Kinesis stream

Required for **audit** and **compliance** purposes.

With just few lines, any log group can be streamed to a Kinesis stream on a compliance / logging account which is used as the permanent store.

```
new logs.SubscriptionFilter(stack, `${service.name}SubscriptionFilter`, {
  logGroup,
  filterPattern: logs.FilterPattern.allEvents(),
  destination: new logs_destinations.KinesisDestination(
    kinesis.Stream.fromStreamArn(stack, `${service.name}Stream`, service.logDestinationArn)),
});
```

Autoscaling the cluster capacity

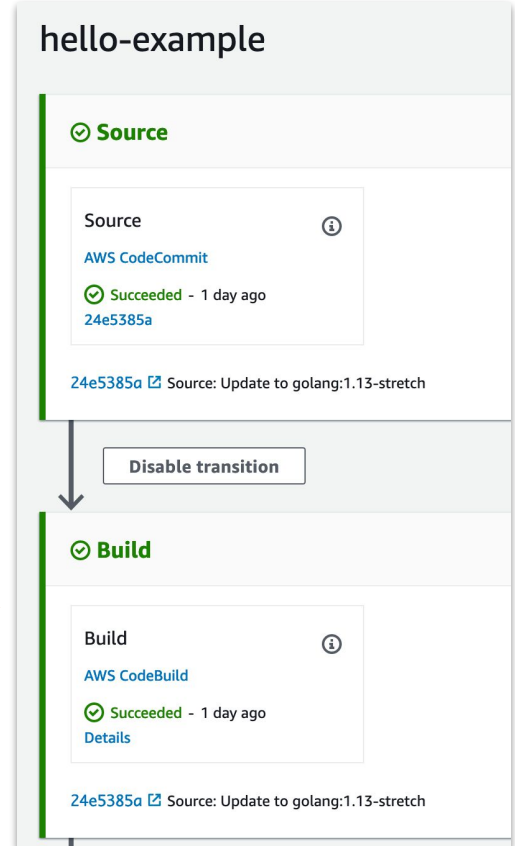
```
const scaling = s.autoScaleTaskCount({
       minCapacity: services[i].desiredCount,
       maxCapacity: services[i].maxCount || 2
});
const cpuProps = services[i].cpuScalingProps;
if (cpuProps) {
  scaling.scaleOnCpuUtilization(`CpuScaling`, cpuProps);
}
const memoryProps = services[i].memoryScalingProps;
if (memoryProps) {
  scaling.scaleOnMemoryUtilization(`MemoryScaling`, memoryProps);
}
```

Automated scaling can be configured for each service based on **CPU** or **memory** usage, or as combination of both.

You could also scale by the number of requests.

Building images in AWS

```
const source = new codepipeline.Artifact('Source');
pipeline.addStage({ stageName: 'Source',
    actions: [ new codepipeline_actions.CodeCommitSourceAction({
            actionName: 'Source',
            repository,
            output: source,
            role: codeCommitRole(pipeline.stack, repository.repositoryName),
        })]});

const buildOutput = new codepipeline.Artifact();
pipeline.addStage({ stageName: 'Build',
    actions: [new codepipeline_actions.CodeBuildAction({
            actionName: 'Build',
            input: source,
            project,
            outputs: [buildOutput],
        })]});
```



Deploying Lambdas with CDK

- Previously we used the Serverless Framework and AWS SAM to deploy Lambda functions to perform simple tasks
- These were refactored to AWS CDK as well ➔ less technologies, less complexity
- The code us slightly more verbose than pure serverless configuration
- Inline code makes deployment painless

```
const updateLambda = new lambda.Function(stack, FunctionName, {
    functionName: FunctionName,
    description: FunctionDescription,
    runtime: lambda.Runtime.NODEJS_8_10,
    handler: 'index.handler',
    role: lambdaRole,
    code: lambda.Code.inline(
        fs.readFileSync('./update-cluster-lambda/index.js').toString()),
    environment: {
        TARGET_CLUSTER: `${env}-cluster`,
        TARGET_IMAGE_TAG: tag,
    }
});
```

Using unsupported CloudFormation constructs

- Example: At the moment there is no redirect support for ALB's in CDK
- By using raw CloudFormation constructs, we can easily achieve the needed functionality

- https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-elasticloadbalancingv2-listener.html
- https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-elasticloadbalancingv2-listener-redirectconfig.html

```
const createHttpsRedirect = (id: string, scope: cdk.Construct, loadBalancer:
elbv2.ApplicationLoadBalancer) => {
 const actionProperty: elbv2.CfnListener.ActionProperty = {
   type: 'redirect',
   redirectConfig: {
     statusCode: 'HTTP_302',
     protocol: 'HTTPS',
     port: '443',
   },
 };
 const redirectProps: elbv2.CfnListenerProps = {
   defaultActions: [actionProperty],
   loadBalancerArn: loadBalancer.loadBalancerArn,
   port: 80,
   protocol: 'HTTP',
 };
 return new elbv2.CfnListener(scope, `${id}HttpRedirect`, redirectProps);
};
```

Unit testing AWS CDK infrastructure

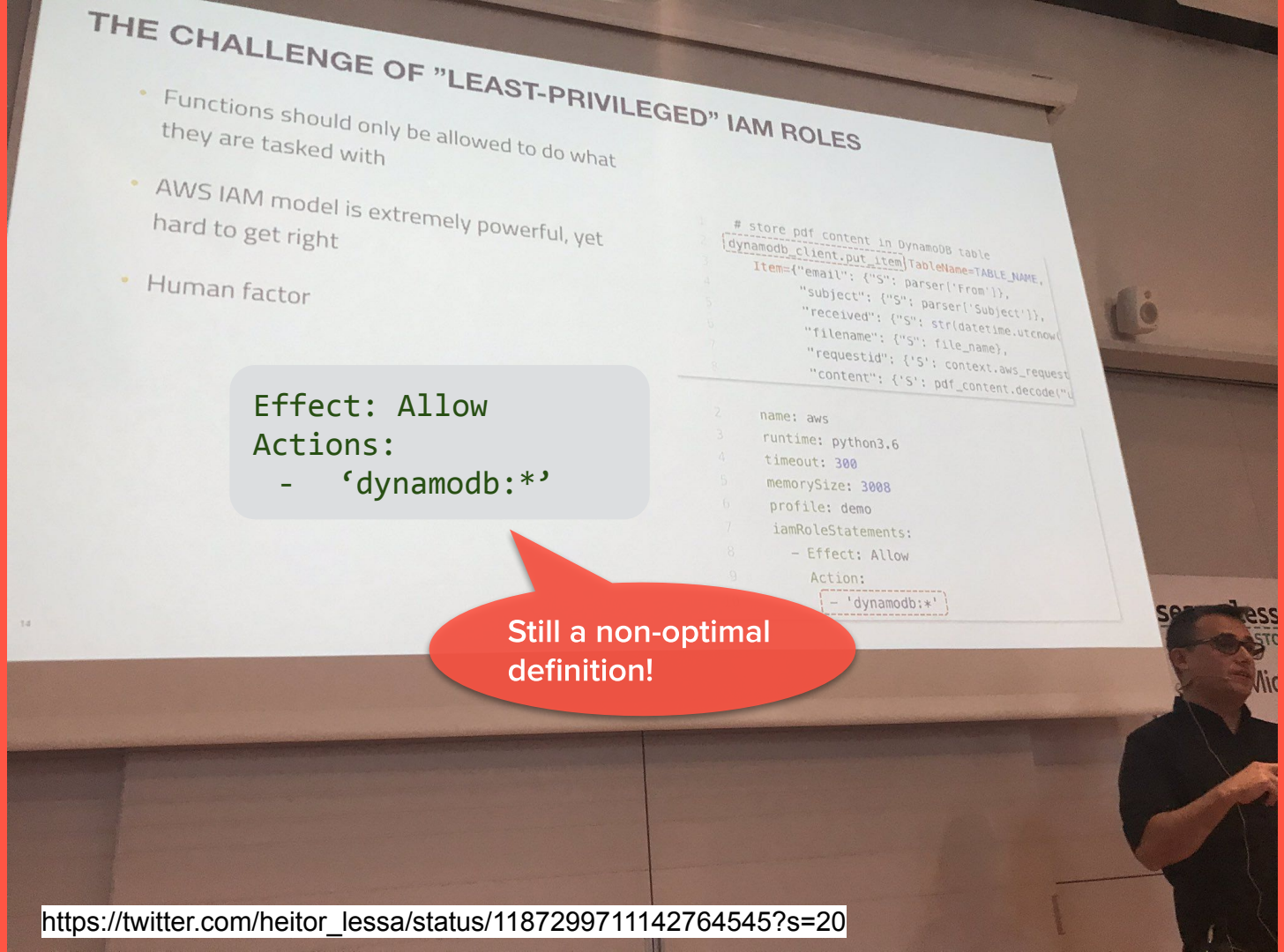
- Still with experimental status in the API, but enables users to write unit tests against their stacks
- Works just like unit testing any other code

- https://github.com/aws/aws-cdk/blob/master/packages/@aws-cdk/assert/README.md

```
'bucket without encryption'(test: Test) {
  const stack = new cdk.Stack();
  new s3.Bucket(stack, 'MyBucket', {
    encryption: s3.BucketEncryption.UNENCRYPTED
  });

  expect(stack).toMatch({
    "Resources": {
      "MyBucketF68F3FF0": {
        "Type": "AWS::S3::Bucket",
        "DeletionPolicy": "Retain",
        "UpdateReplacePolicy": "Retain",
      }
    }
  });

  test.done();
},
```

Least privilege IAM roles

How to pass your security audits

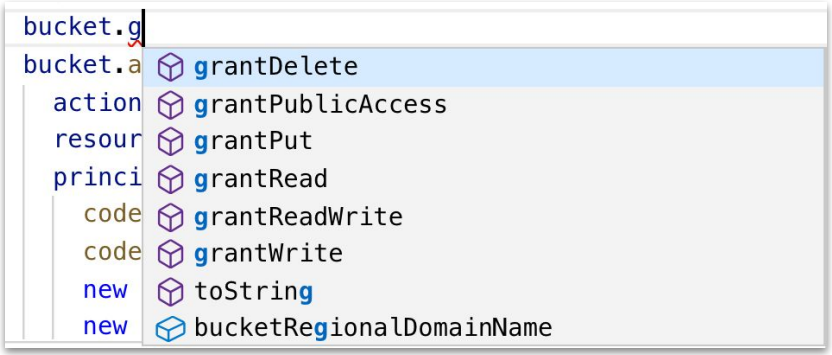


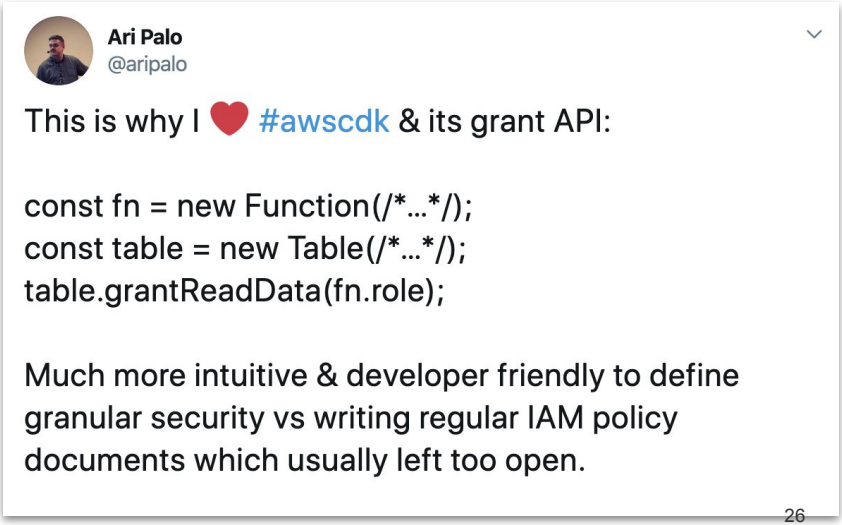
REAKTOR
2019

https://twitter.com/heitor_lessa/status/1187299711142764545?s=20

Least privilege IAM roles

With AWS CDK it's so easy that developers actually do it.





Least privilege IAM roles

```
export const s3ReadWritePolicyForBucket = (bucketName: string) => new iam.PolicyStatement({
  actions: ['s3:PutObject', 's3:ListBucket', 's3:GetObject', 's3:DeleteObject'],
  resources: [`arn:aws:s3:::${bucketName}`, `arn:aws:s3:::${bucketName}/*`],
});
export const dynamodbGetPolicy = (tableName: string) => new iam.PolicyStatement({
  actions: ['dynamodb:GetItem', 'dynamodb:Scan'],
  resources: [`arn:aws:dynamodb:eu-central-1:*:table/${tableName}`]
});
export const sendEmailPolicy = (fromAddress: string) => new iam.PolicyStatement({
 actions: ['ses:SendEmail'],
 resources: ['*'],
 conditions: { 'StringEquals': { 'ses:FromAddress': fromAddress } }
});
```

Spot two errors

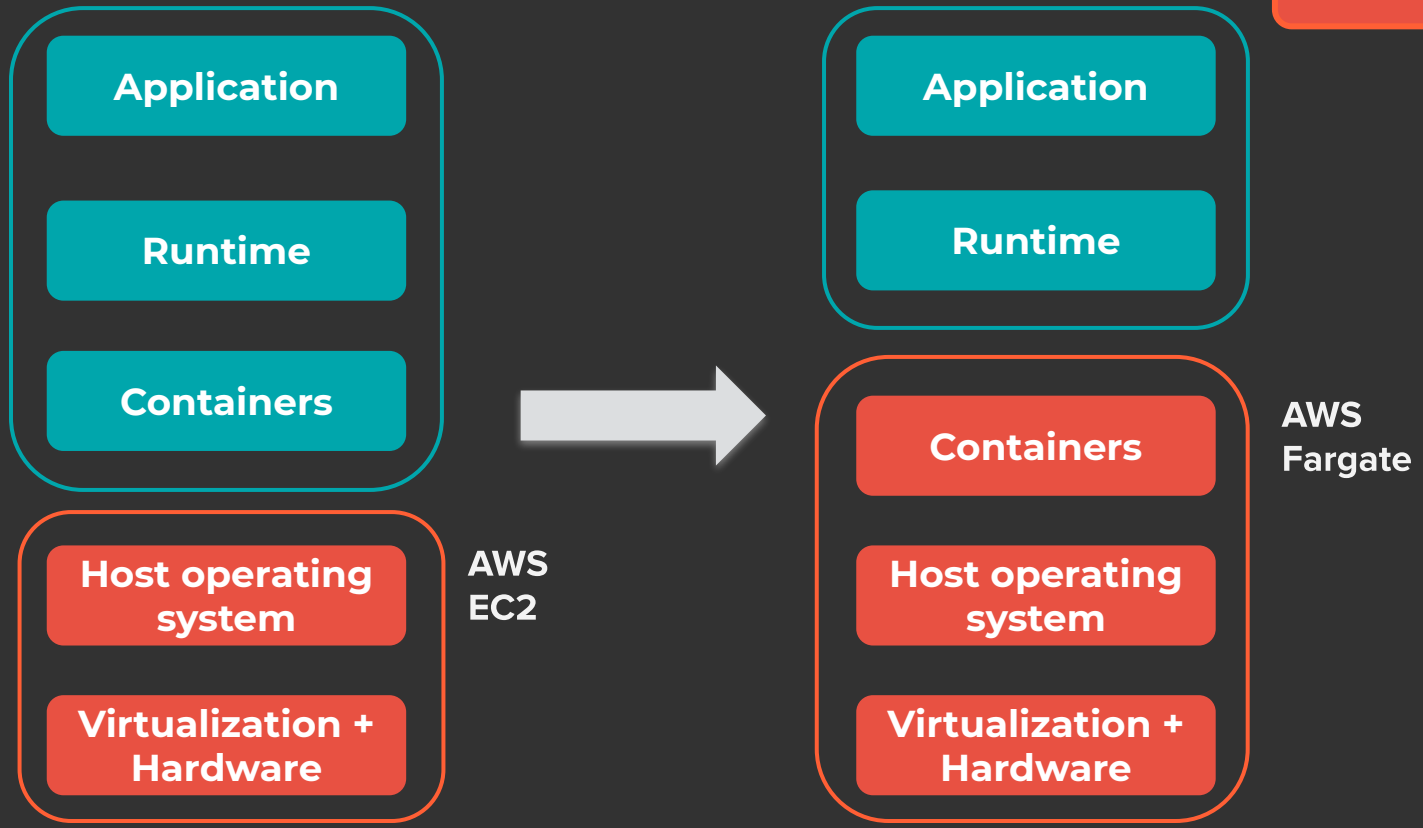
Sometimes errors unfortunately appear only at deployment time. Developers would benefit from compile-time validation whenever possible.

```
const policy = new iam.PolicyStatement({
    actions: ['s3:GetObject'],
    resources: ['*'],
    principals: [new iam.AnyPrincipal()],
    effect: iam.Effect.ALLOW,
    conditions: [
        'IpAddress', {
            'aws:sourceIp': [source]
        }]
});
```

```
const policy = new iam.PolicyStatement({
    actions: ['s3:GetObject'],
    resources: [bucket.arnForObjects('*')],
    principals: [new iam.AnyPrincipal()],
    effect: iam.Effect.ALLOW,
    conditions: {
        'IpAddress': {
            'aws:sourceIp': [source]
        }}
});
```

Automated container management

From “**infrastructure** as a service” to “**containers** as a service”



The **old way** of deploying clusters with never ending CloudFormation templates

```
Resources:
  MasterSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupName: !Join ["-", ["master-sg", !Ref InfrastructureName]]
      GroupDescription: Cluster Master Security Group
      SecurityGroupIngress:
      - IpProtocol: icmp
        FromPort: 0
        ToPort: 0
        CidrIp: !Ref VpcCidr
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: !Ref VpcCidr
      - IpProtocol: tcp
        ToPort: 6443
        FromPort: 6443
        CidrIp: !Ref VpcCidr
      - IpProtocol: tcp
        FromPort: 22623
        ToPort: 22623
        CidrIp: !Ref VpcCidr
      VpcId: !Ref VpcId

  WorkerSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupName: !Join ["-", ["worker-sg", !Ref InfrastructureName]]
      GroupDescription: Cluster Worker Security Group
      SecurityGroupIngress:
      - IpProtocol: icmp
        FromPort: 0
        ToPort: 0
        CidrIp: !Ref VpcCidr
      - IpProtocol: tcp
```

```
  MasterInstanceProfile:
    Type: "AWS::IAM::InstanceProfile"
    Properties:
      Path: "/"
      Roles:
      - Ref: "MasterIamRole"

  WorkerIamRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: !Join ["-", [!Ref InfrastructureName, "worker", "role"]]
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
        - Effect: "Allow"
          Principal:
            Service:
            - "ec2.amazonaws.com"
          Action:
          - "sts:AssumeRole"
      Path: "/"
      Policies:
      - PolicyName: !Join ["-", [!Ref InfrastructureName, "worker", "policy"]]
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
          - Effect: "Allow"
            Action: "ec2:Describe*"
            Resource: "*"
```

Source: https://github.com/Jackrokdj1/openshift/ (Apache 2.0)

The new way of setting up clusters

Using simplified AWS CDK constructs you get same functionality and security with less code.

Services are specified in terms of **CPU** and **memory** that are needed for each container.

Cluster management happens automatically by Fargate.

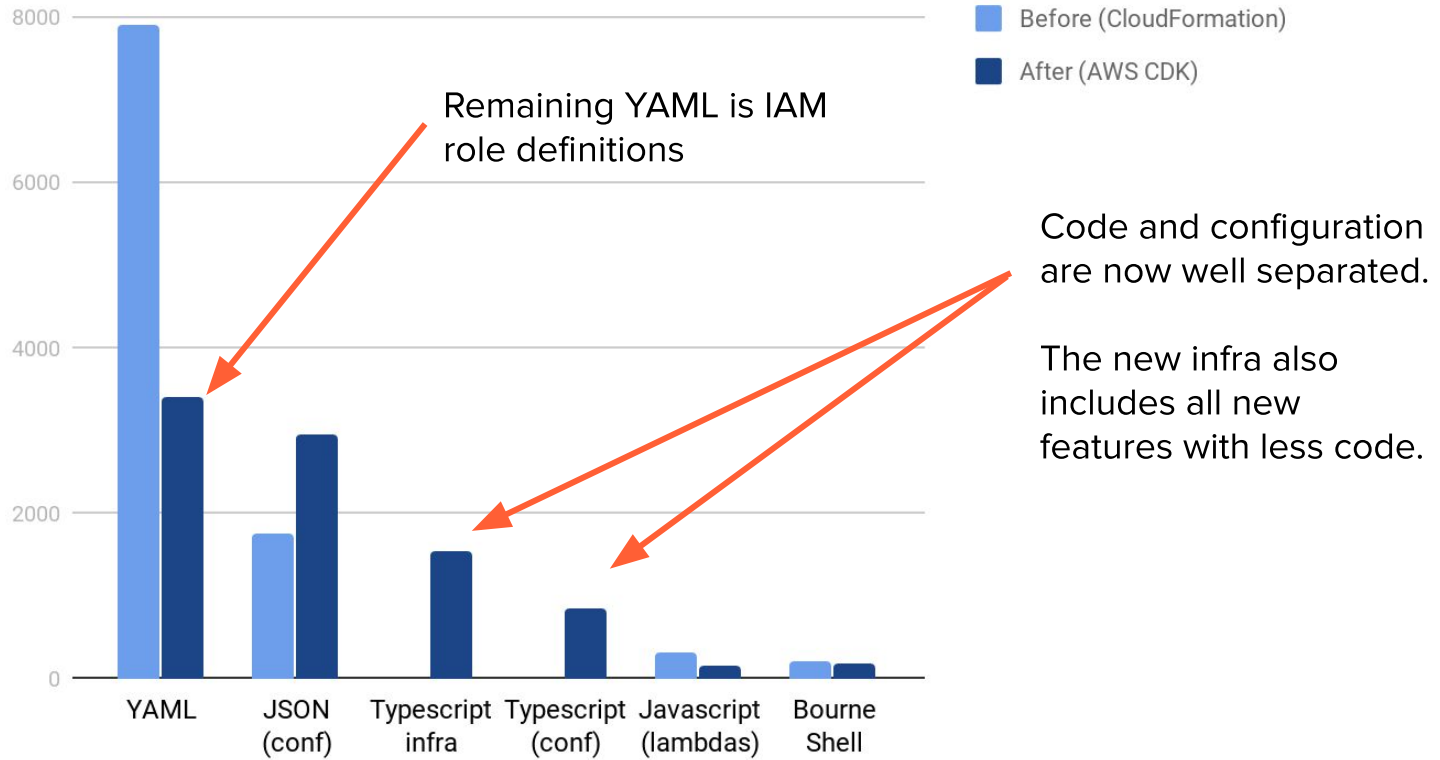
```
const vpc = new ec2.Vpc(this, 'ExampleVpc', { maxAzs: 2 });
const cluster = new ecs.Cluster(this, 'ExampleCluster', { vpc });

// Import existing image and route traffic with a public Network Load Balancer
new ecs_patterns.ApplicationLoadBalancedFargateService(this,
  "ExampleService", {
  cluster,
  cpu: 512,
  memoryLimitMiB: 2048,
  desiredCount: 2,
  image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"),
  publicLoadBalancer: true
});
```

AWS CDK is already becoming a norm

https://twitter.com/rchrdbyd/status/1187824437544050688

Conclusions

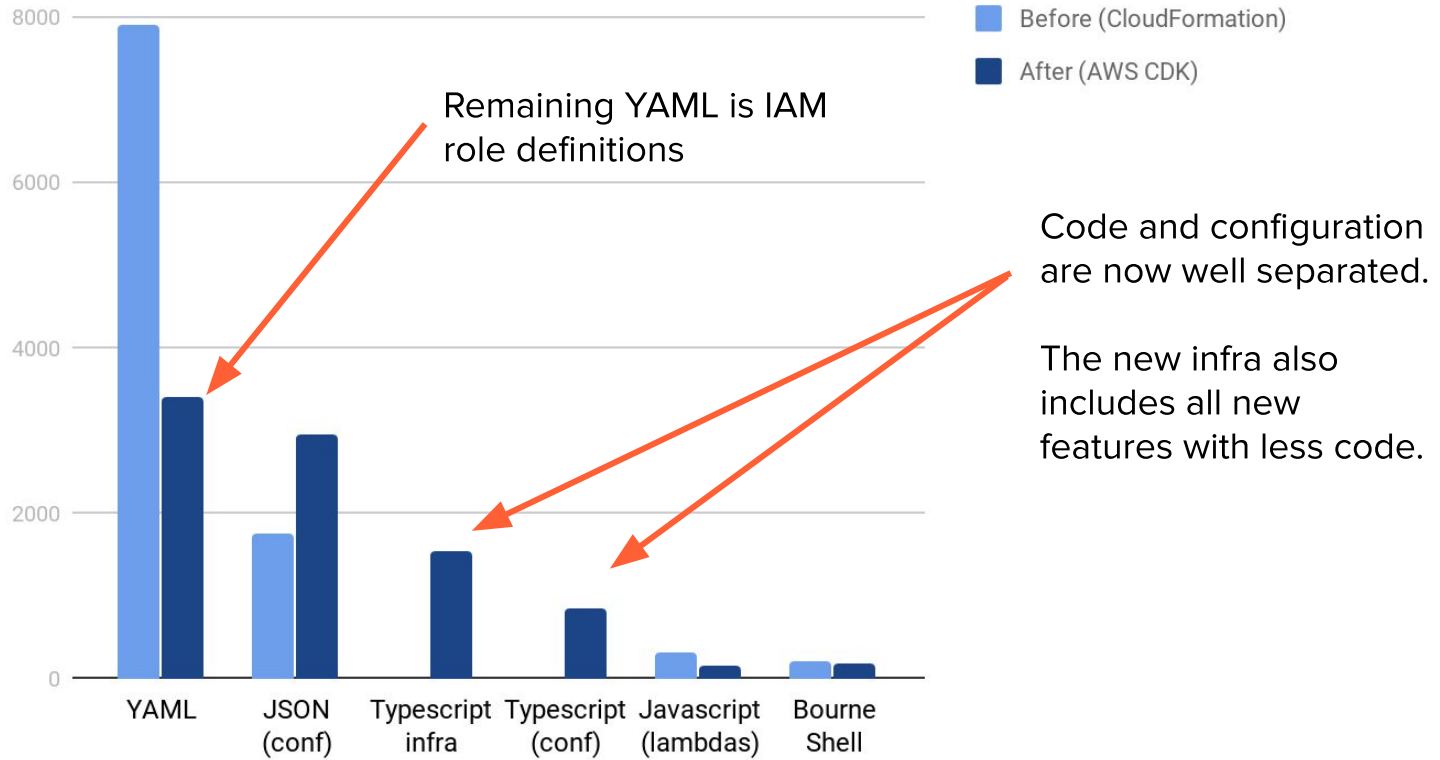
Infrastructure code before and after migration



Code and configuration are now well separated.

The new infra also includes all new features with less code.

Infrastructure code before and after migration



Code and configuration are now well separated.

The new infra also includes all new features with less code.

Executive summary

Experiences from AWS CDK migration in a hybrid architecture.

1. **Cost savings** by using industry best practices
2. **Cloud First**: Outsource container and database management
3. **Faster** development cycle from development to production
4. Increased **information security** and **compliance**
5. Using managed services: **Pay for only what you use**
6. **43% less code** to maintain!

Discussion

Q&A

Thank you.

markus.lindqvist@reaktor.com