Building a modern cloud application with the AWS Cloud Development Kit

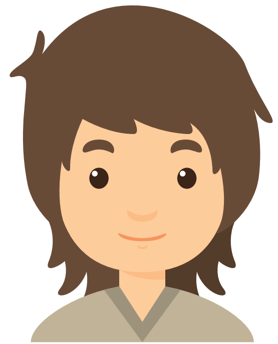
Nathan Peck, Developer Advocate, AWS

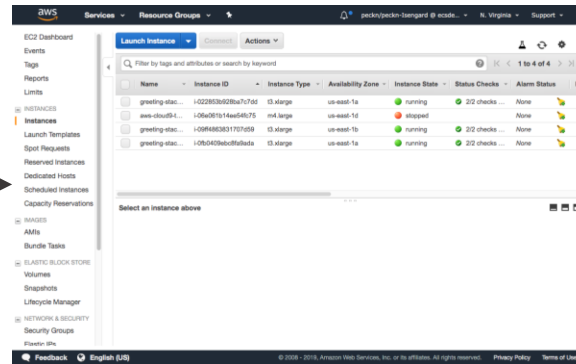
Agenda

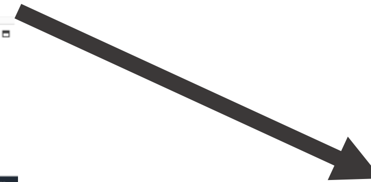
- Introduction to infrastructure as code
  - What is infrastructure as code? Why use it?
  - Where does AWS CDK fit into the infrastructure as code space?
  - What does AWS CDK offer that is unique?
- What tooling does AWS CDK offer for containerized applications?
  - High level reusable patterns for containers
  - Underlying core constructs for more advanced configurations
- Demo deploying a containerized application with AWS CDK

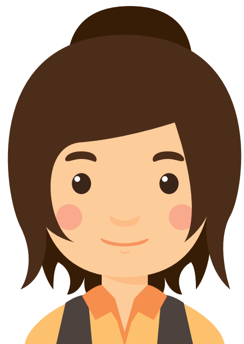
Introduction to infrastructure as code

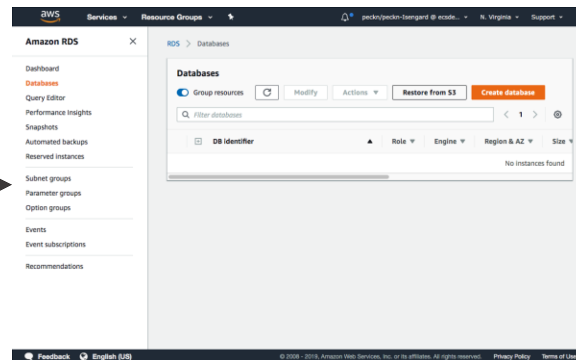
Level 0 – Creating infrastructure by hand

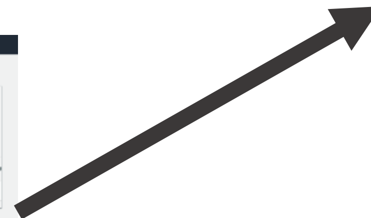


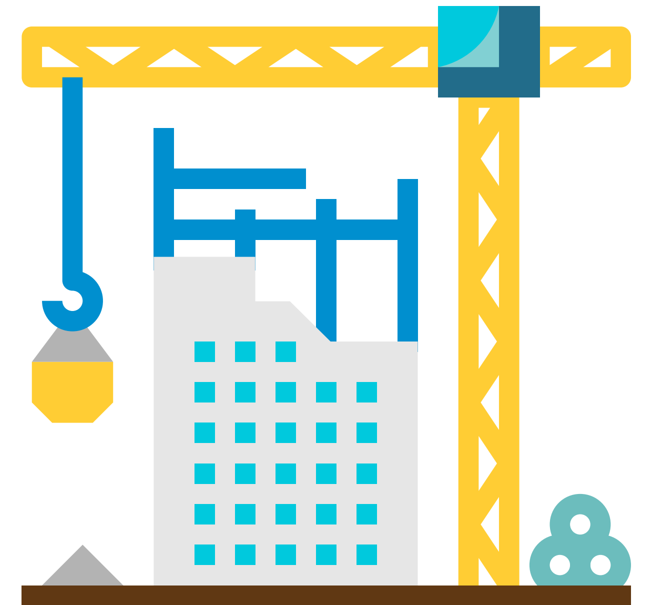












Your organization’s infrastructure

Level 0 – Creating infrastructure by hand

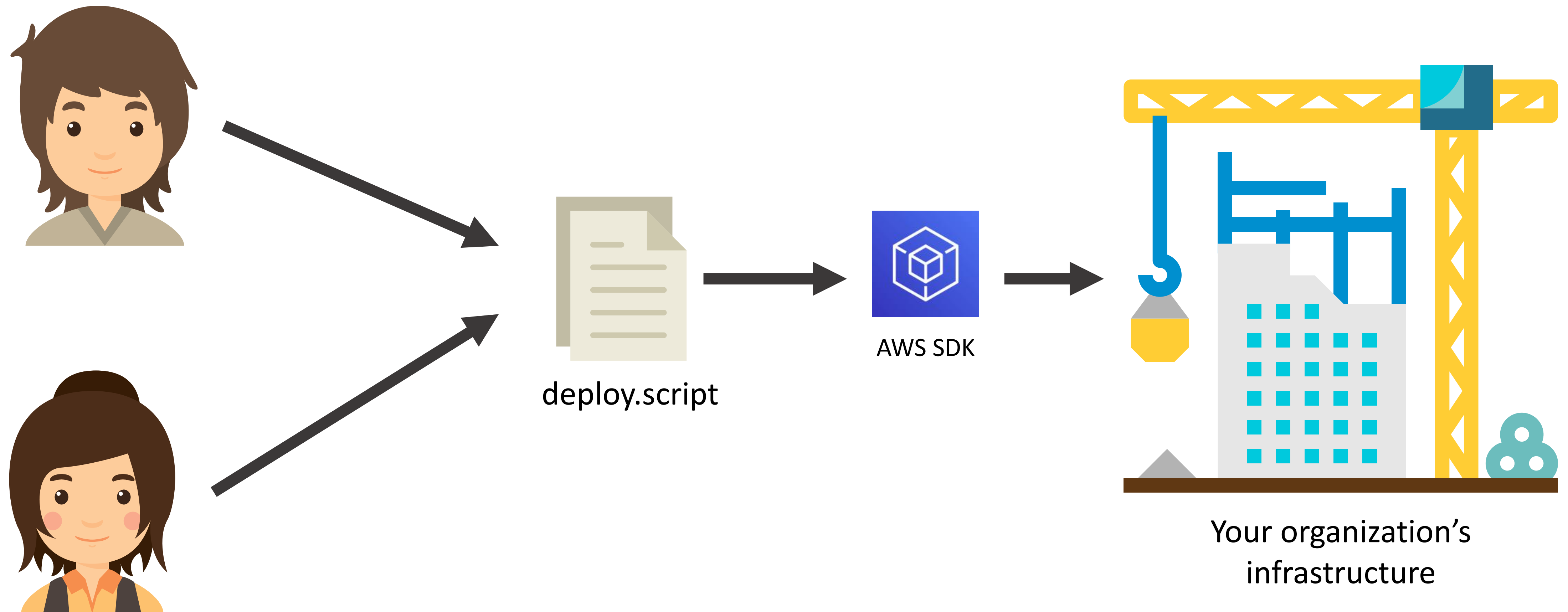
Pros

- Decent for standing up your first exploratory project infrastructure
- Tight interaction with console can help you see errors faster

Cons

- Clicking things and entering values in the console by hand is slow. Long-term bottleneck in development
- It’s hard to reliably reproduce your results when you are doing things by hand.
- People make mistakes in data entry and clicking options.
- Person A configures things one way, but person B configures things another way

Level 1 – Imperative infrastructure as code



Level 1 – Imperative infrastructure as code

deploy.script

```
resource = getResource(xyz)

if (resource == desiredResource) {
  return
} else if (!resource) {
  createResource(desiredResource)
} else {
  updateResource(desiredResource)
}
```

- Lots of boilerplate
- What if something fails and we need to retry?
- What if two people try to run the script at once?
- Race conditions?

Level 1 – Imperative infrastructure as code

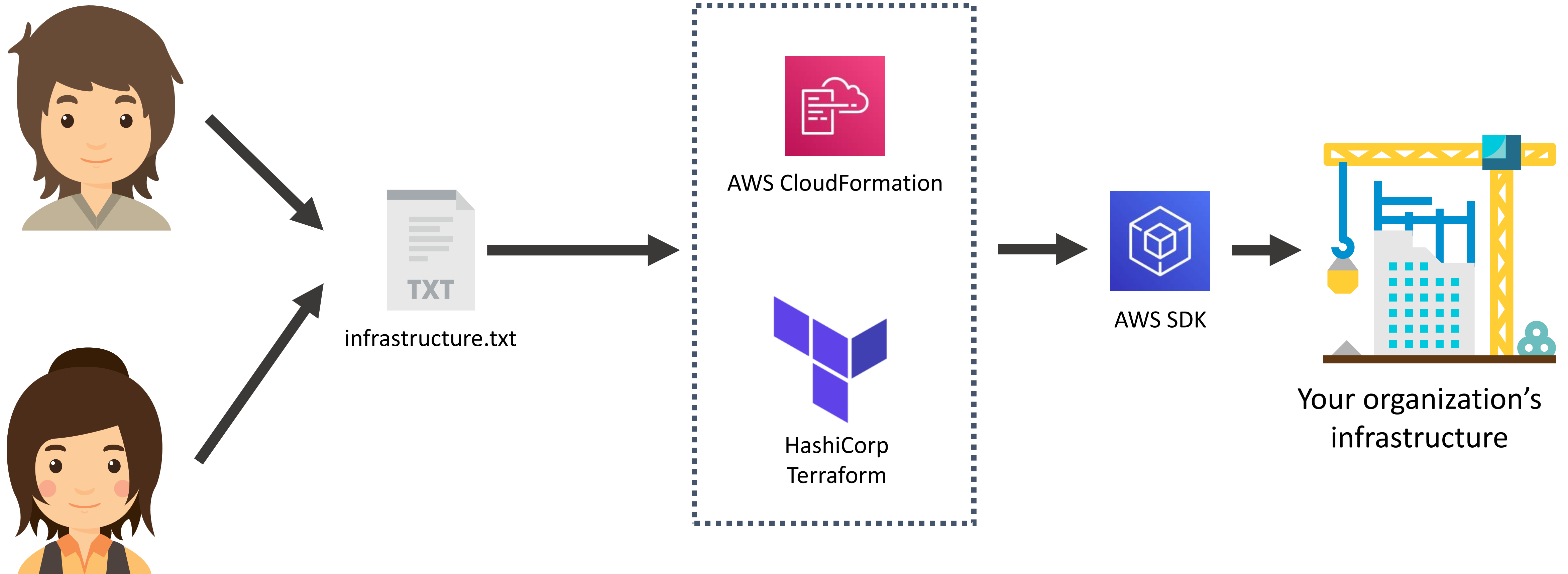
Pros

- If the code is written well it is repeatable and reusable
- Multiple people can work on the script collaboratively, fixing bugs, see all the settings in one place

- Lots of boilerplate code to write, and it can be hard to write reliable code
- Imperative code has to handle all edgecases
- Must be careful about multiple people using the script at once

Cons

Level 2 – Declarative infrastructure as code



Level 2 – Declarative infrastructure as code

infrastructure.txt

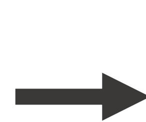
```
Resources:
 # VPC in which containers will be networked.
 # It has two public subnets
 # We distribute the subnets across the first two available subnets
 # for the region, for high availability.
 VPC:
  Type: AWS::EC2::VPC
  Properties:
   EnableDnsSupport: true
   EnableDnsHostnames: true
   CidrBlock: !FindInMap ['SubnetConfig', 'VPC', 'CIDR']

 # Two public subnets, where containers can have public IP addresses
 PublicSubnetOne:
  Type: AWS::EC2::Subnet
  Properties:
   AvailabilityZone:
     Fn::Select:
     - 0
     - Fn::GetAZs: {Ref: 'AWS::Region'}
   VpcId: !Ref 'VPC'
   CidrBlock: !FindInMap ['SubnetConfig', 'PublicOne', 'CIDR']
   MapPublicIpOnLaunch: true
 PublicSubnetTwo:
  Type: AWS::EC2::Subnet
  Properties:
   AvailabilityZone:
     Fn::Select:
     - 1
     - Fn::GetAZs: {Ref: 'AWS::Region'}
   VpcId: !Ref 'VPC'
   CidrBlock: !FindInMap ['SubnetConfig', 'PublicTwo', 'CIDR']
   MapPublicIpOnLaunch: true
```

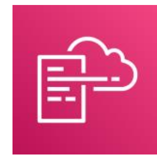
Just a list of each resource to create and its properties, in this case YAML format

Some minor helper functions may be built in to aid in fetching values dynamically.

Level 2 – Declarative infrastructure as code

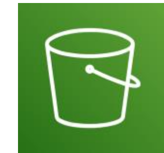
```
Resources:
  BucketOne:
    Type: AWS::S3::Bucket
    Properties:
      AccessControl: PublicRead
```





AWS CloudFormation

AWS SDK

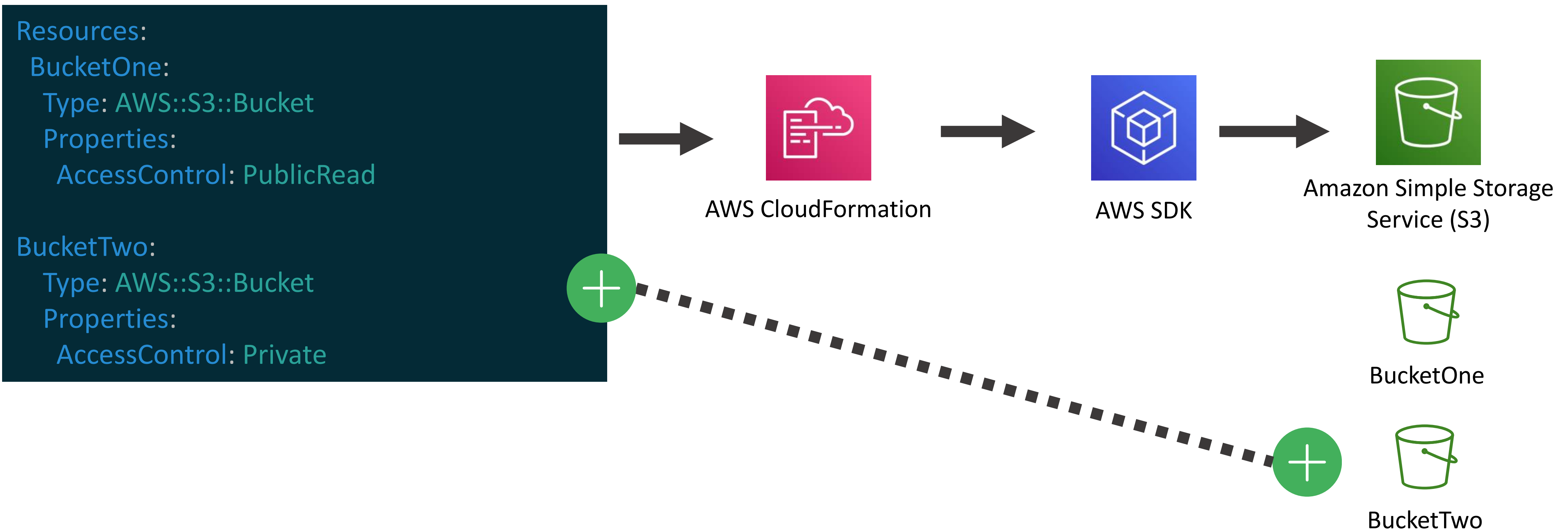


Amazon Simple Storage Service (S3)

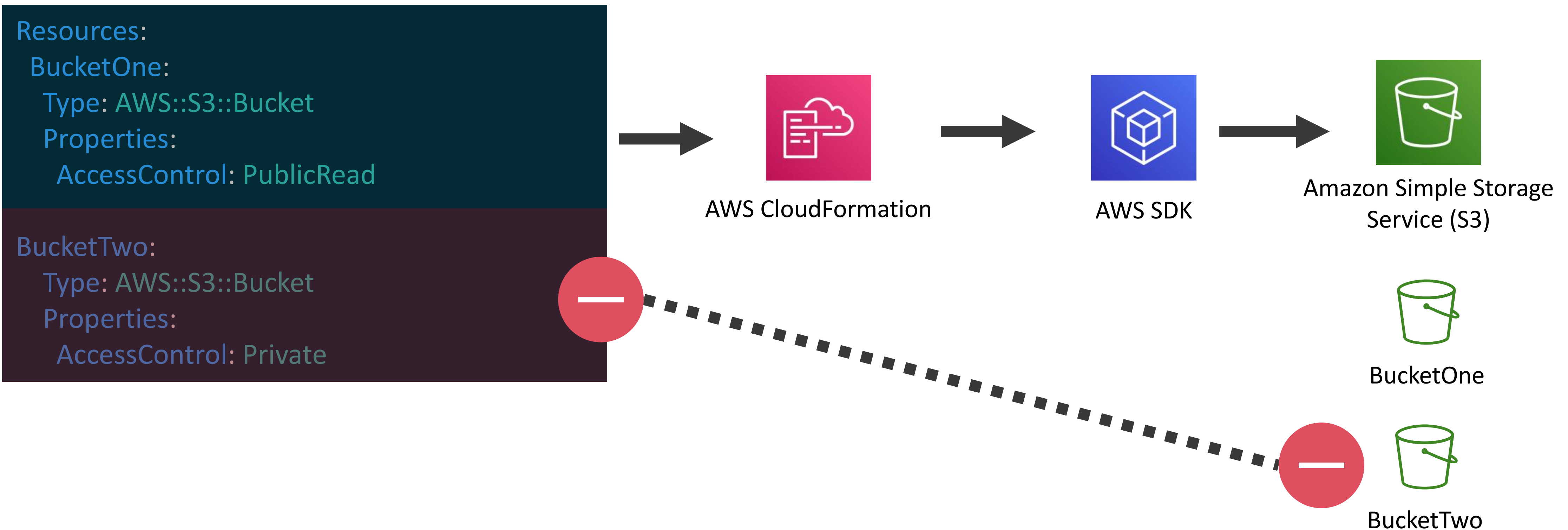
BucketOne

BucketTwo

Level 2 – Declarative infrastructure as code



Level 2 – Declarative infrastructure as code



Level 2 – Declarative infrastructure as code

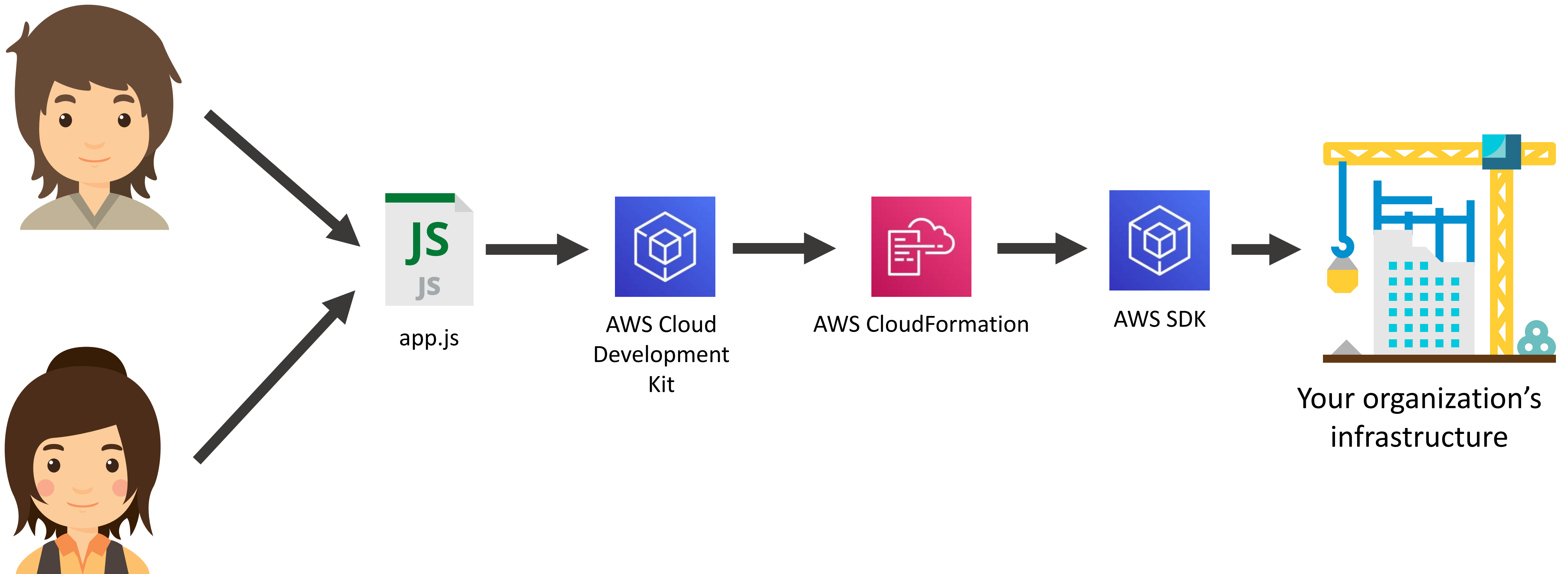
Pros

- No imperative boilerplate to write, creating and updating resources is handled automatically
- Multiple people can work on the template collaboratively
- Conflict resolution, and resource locking can be handled centrally

- 1 to 1 relationship between resources in file and resources on account means lots of boilerplate to write. Templates can be very verbose
- Limited ability to run logic as the file formats are generally things like JSON, YAML, or HCL which have only a few built in functions
- Hard to keep things DRY without loops, functions, etc

Cons

Level 3 – AWS Cloud Development Kit



Level 3 – AWS Cloud Development Kit

app.js

app.py

```
class MyService extends cdk.Stack {
 constructor(scope: cdk.App, id: string) {
  super(scope, id);

  // Network for all the resources
  const vpc = new ec2.Vpc(this, 'MyVpc', { maxAzs: 2 });

  // Cluster to hold all the containers
  const cluster = new ecs.Cluster(this, 'Cluster', { vpc: vpc });

  // Load balancer for the service
  const LB = new elbv2.ApplicationLoadBalancer(this, 'LB', {
   vpc: vpc,
   internetFacing: true
  });
 }
}
```

- Write in a familiar programming language
- Create many underlying AWS resources at once with a single construct
- Each stack is made up of “constructs” which are simple classes in the code
- Still declarative, no need to handle create vs update

One CDK construct expands to many underlying resources

```
// Network for all the resources
const vpc = new ec2.Vpc(stack, 'MyVpc', { maxAzs: 2 });
```

```
cdk deploy
```

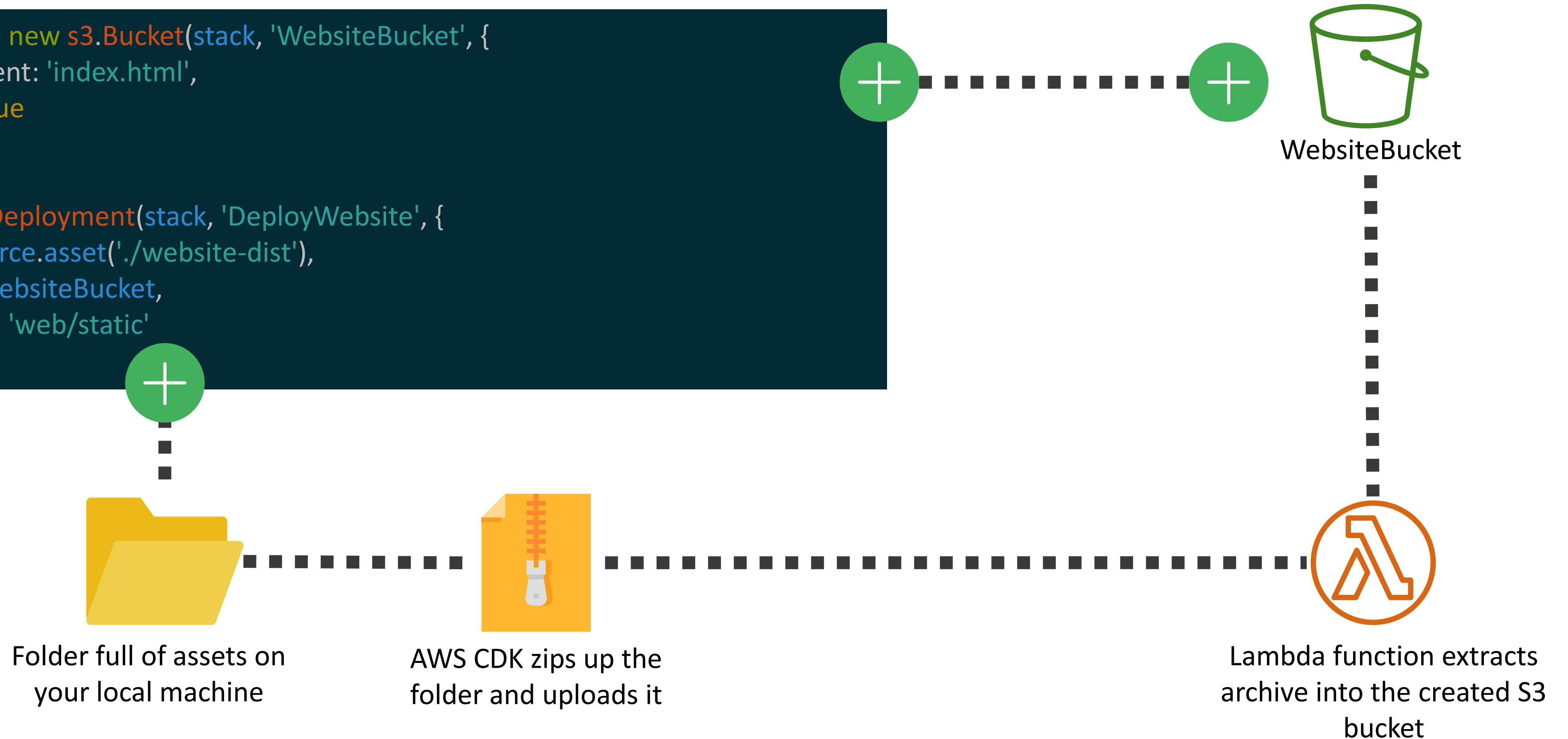
One CDK construct expands to many underlying resources

```
// Network for all the resources
const vpc = new ec2.Vpc(stack, 'MyVpc', { maxAzs: 2 });
```

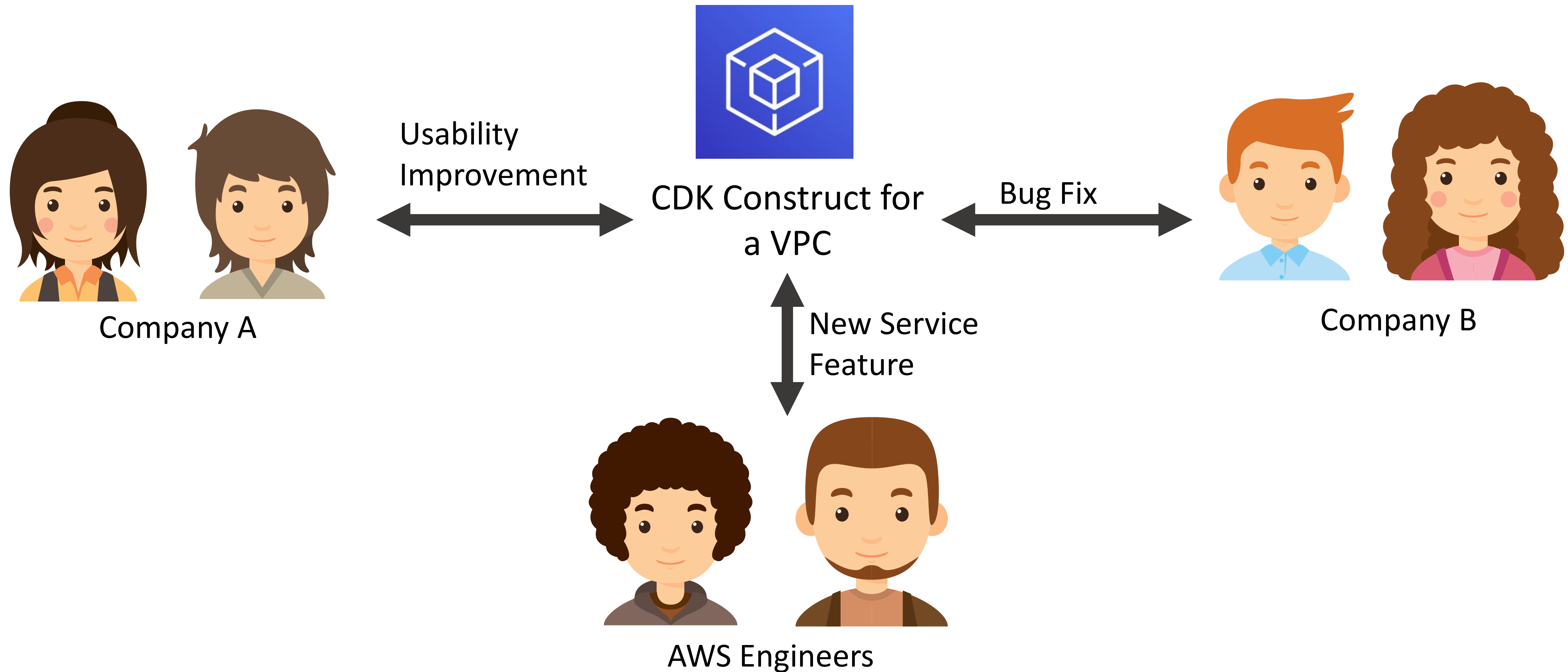
```
cdk synth
```

270 lines of CloudFormation YAML I don’t have to write!

CDK helps with your local workflow too



CDK constructs are shareable and reusable



Lots of open source constructs on

alexa-ask
app-delivery
assets
aws-amazonmq
aws-amplify
aws-apigateway
aws-applicationautoscaling
aws-appmesh
aws-appstream
aws-appsync
aws-athena
aws-autoscaling
aws-autoscaling-common
aws-autoscaling-hooktargets
aws-autoscalingplans
aws-backup
aws-batch

aws-budgets
aws-certificatemanager
aws-cloud9
aws-cloudformation
aws-cloudfront
aws-cloudtrail
aws-cloudwatch
aws-cloudwatch-actions
aws-codebuild
aws-codecommit
aws-codedeploy
aws-codepipeline
aws-codepipeline-actions
aws-codestar
aws-cognito
aws-config
aws-datapipeline

aws-dax
aws-directoryservice
aws-dlm
aws-dms
aws-docdb
aws-dynamodb
aws-dynamodb-global
aws-ec2
aws-ecr
aws-ecr-assets
aws-ecs
aws-ecs-patterns
aws-efs
aws-eks
aws-elasticache
aws-elasticbeanstalk
aws-elasticloadbalancing

aws-elasticloadbalancingv2
aws-elasticloadbalancingv2-targets
aws-elasticsearch
aws-emr
aws-events
aws-events-targets
aws-fsx
aws-gamelift
aws-glue
aws-greengrass
aws-guardduty
aws-iam
aws-inspector
aws-iot
aws-iotclick
aws-iotanalytics
aws-iotevents

…. and many more!

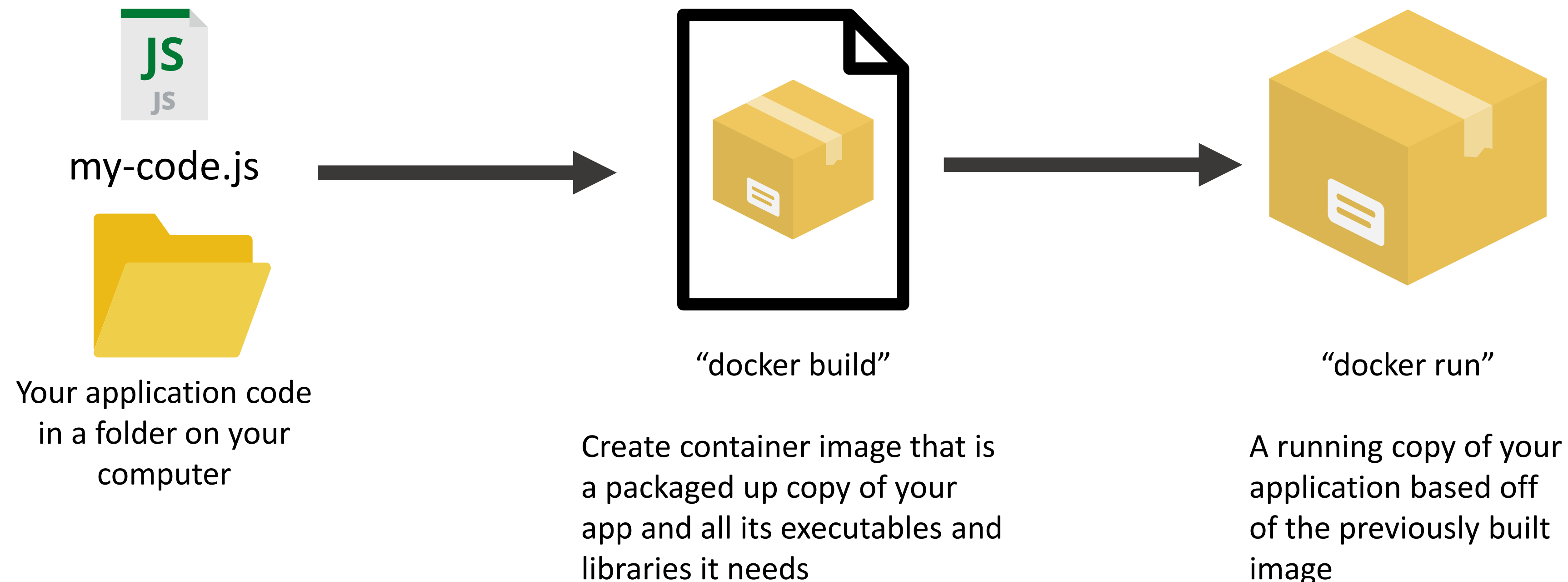
Level 3 – AWS Cloud Development Kit

Pros

- Declarative: creating and updating resources is handled automatically
- Higher level constructs that automatically create many underlying resources
- Multiple people can work on the CDK app collaboratively
- Conflict resolution, and resource locking can be handled centrally
- Use familiar programming languages: Python, JavaScript, TypeScript, .Net, Java
- CDK does more than just create cloud resources, it also helps with your local development workflow
- Easily share and reuse constructs on NPM. Benefit from best practice constructs designed by experts

AWS CDK for containerized applications

First some basic container concepts



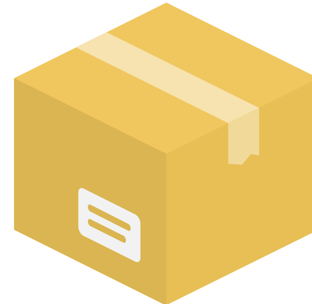
Add some container orchestration concepts

Register task definition

Description of what app to run and what settings it needs

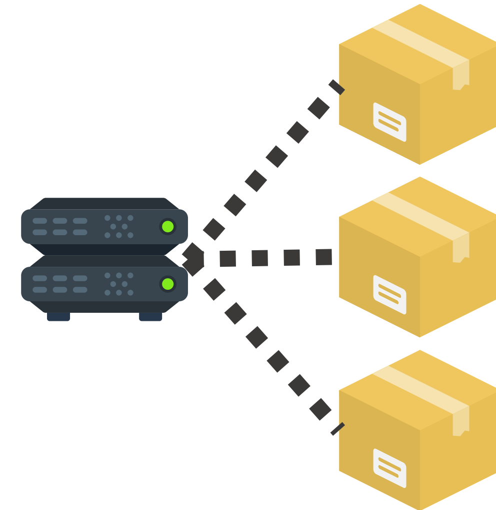
Create cluster

Capacity for running application, either EC2 instances or stay serverless with AWS Fargate



Run single task

Launch a standalone task in a cluster based on a task definition description. Just runs until completion then exits



Create service

Run multiple copies of a task. Hook them up to other resources like a load balancer. Keep running them until I say to stop

Two levels of container abstraction in CDK

@aws-cdk/aws-ecs
1.7.0 • Public • Published 4 days ago
Readme | 23 Dependencies | 34 Dependents

Amazon ECS Construct Library

STABILITY STABLE

- Basic patterns for building Docker images, creating a cluster, task definition, task, or service.
- Stable release

@aws-cdk/aws-ecs-patterns
1.7.0 • Public • Published 4 days ago
Readme | 12 Dependencies | 3 Dependents

CDK Construct library for higher-level ECS Constructs

STABILITY EXPERIMENTAL

- Common architecture patterns built on top of the basic patterns: a load balanced service, a queue consumer, task scheduled to run at a particular time.
- Experimental release, we are still working on this!

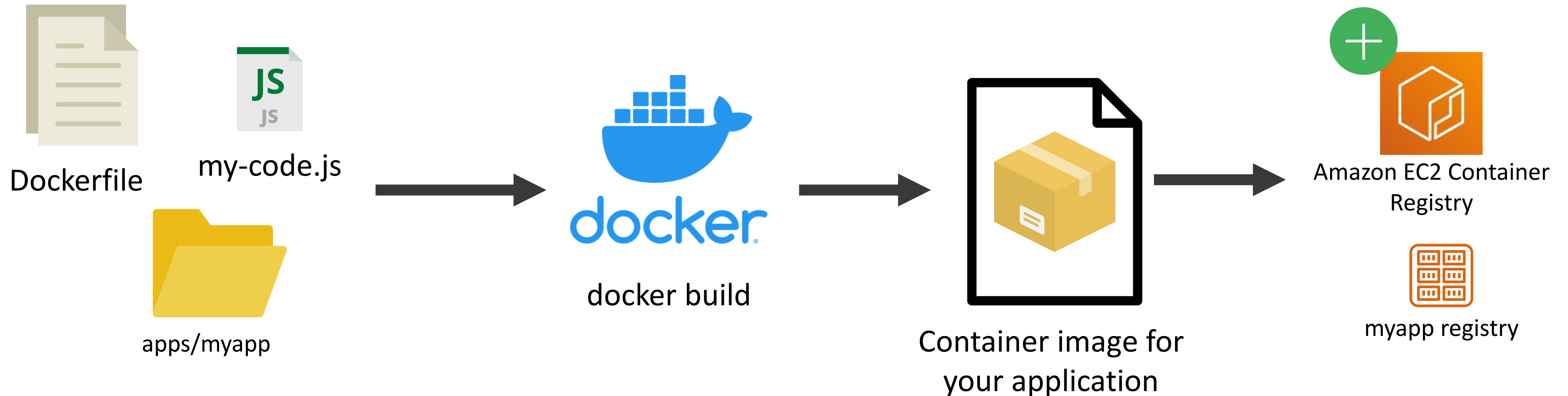
The basics of CDK for containers

aka

“write your first CDK program in 5 mins”

@aws-cdk/aws-ecs: Build a container image

```
import ecs = require('@aws-cdk/aws-ecs');

const image = ecs.ContainerImage.fromAsset("apps/myapp")
```



@aws-cdk/aws-ecs: Create cluster to run application

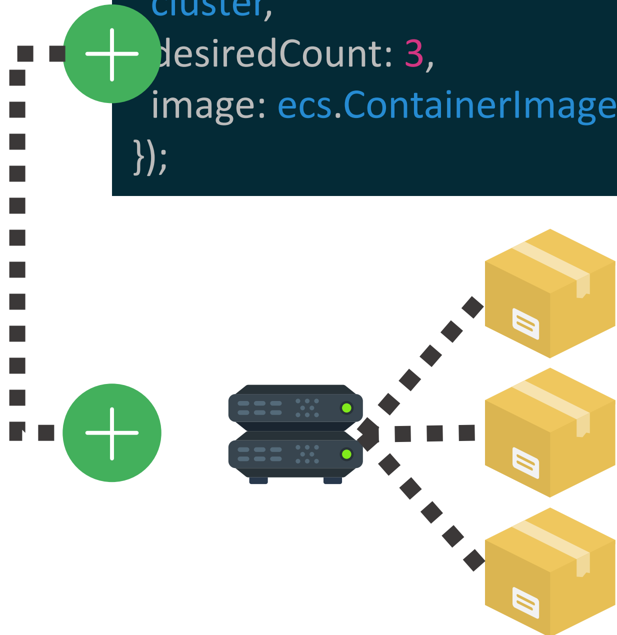
```
import ec2 = require('@aws-cdk/aws-ec2');
import ecs = require('@aws-cdk/aws-ecs');

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

Do you want to stay serverless or do you want to add EC2 instances and run on EC2?

```
cluster.addCapacity('cluster-capacity', {
  instanceType: new ec2.InstanceType("t2.xlarge"),
  desiredCapacity: 3
});
```

@aws-cdk/aws-ecs-patterns: Launch load balanced service

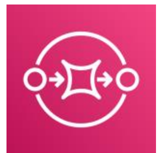
```
import ec2 = require('@aws-cdk/aws-ec2');
import ecs = require('@aws-cdk/aws-ecs’);
import ecs_patterns = require('@aws-cdk/aws-ecs-patterns');

const vpc = new ec2.Vpc(stack, 'MyVpc', { maxAzs: 2 });
const cluster = new ecs.Cluster(stack, 'Cluster', { vpc });

const myService = new ecs_patterns.LoadBalancedFargateService(stack, "my-service", {
  cluster,
  desiredCount: 3,
  image: ecs.ContainerImage.fromAsset("apps/myapp")
});
```



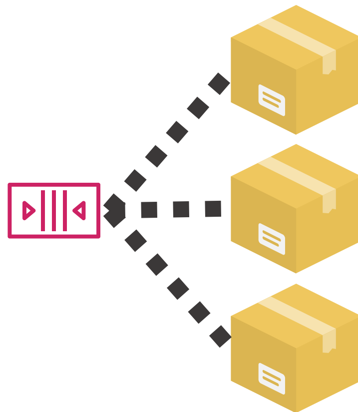
With a few lines we are automatically building a Docker container locally, pushing it up to the cloud in an Amazon Elastic Container Registry, then launching running three copies of it in AWS Fargate, behind a load balancer that distributes traffic across all three.

@aws-cdk/aws-ecs-patterns: Queue consumer

```
const queue = new sqs.Queue(stack);

const consumer = new ecs_patterns.QueueProcessingFargateService(stack, "consumer", {
  cluster,
  queue,
  desiredTaskCount: 3,
  image: ecs.ContainerImage.fromAsset("apps/consumer")
});
```



Amazon Simple Queue Service



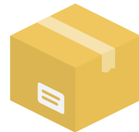
Create an SQS queue, plus a service which autoscales according to how many items are waiting in the queue. If the queue backs up more containers are launched to grab items off the queue.

@aws-cdk/aws-ecs-patterns: Time scheduled container

```
const ecsScheduledTask = new ScheduledFargateTask(stack, 'ScheduledTask', {
  cluster,
  image: ecs.ContainerImage.fromRegistry("apps/my-cron-job"),
  scheduleExpression: 'rate(1 day)',
  environment: [{ name: 'TRIGGER', value: 'CloudWatch Events' }],
  memoryLimitMiB: 256
});
```

Amazon CloudWatch

Every day at 5:00



Execute the container based on a scheduled time or rate. High availability, low cost distributed cron jobs!

Let’s dive a little deeper now

Create a service manually

```
const taskDefinition = new ecs.Ec2TaskDefinition(stack, 'TaskDef');
const container = taskDefinition.addContainer('web', {
  image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"),
  memoryLimitMiB: 256,
});

container.addPortMappings({
  containerPort: 80,
  hostPort: 8080,
  protocol: ecs.Protocol.TCP
});

// Create Service
const service = new ecs.Ec2Service(stack, "Service", {
  cluster,
  taskDefinition,
});
```

Expose service via a load balancer

```
// Create Service
const service = new ecs.Ec2Service(stack, "Service", {
  cluster,
  taskDefinition,
});

// Create ALB
const lb = new elbv2.ApplicationLoadBalancer(stack, 'LB', {
  vpc,
  internetFacing: true
});
const listener = lb.addListener('PublicListener', { port: 80, open: true });

// Attach ALB to ECS Service
listener.addTargets('ECS', {
  port: 80,
  targets: [service],
  // include health check (default is none)
  healthCheck: {
    interval: cdk.Duration.seconds(60),
    path: "/health",
    timeout: cdk.Duration.seconds(5),
  }
});
```

Add access to some other resources

```
const taskDefinition = new ecs.Ec2TaskDefinition(stack, 'TaskDef');
const container = taskDefinition.addContainer('web', {
  image: ecs.ContainerImage.fromRegistry("apps/myapp"),
  memoryLimitMiB: 256,
});

// Grant this task role access to use other resources
myDynamodbTable.grantReadWriteData(taskDefinition.taskRole);
mySnsTopic.grantPublish(taskDefinition.taskRole);
```

- No need to handwrite an IAM policy for your application. CDK already has sensible default access rules built in, and you can grant them to your container applications

Things AWS CDK can automate away for you

- AWS CDK automatically creates security groups and minimal security group rules that allow the load balancer to talk to your tasks

- AWS CDK can automatically build your container image and automatically push it to an automatically created ECR registry

- AWS CDK automatically creates an IAM role for my task. You can then easily add minimal access to other resources on my account

- AWS CDK can automatically create a load balancer and attach it to your service for you

@aws-cdk/aws-ecs-patterns

```
const myService = new ecs_patterns.LoadBalancedFargateService(stack, "my-service", {
  cluster,
  desiredCount: 3,
  image: ecs.ContainerImage.fromAsset("apps/myapp")
});
```

- If you are just starting out with containers I recommend using the ECS patterns as it is easier to get started with.
- If you are an experienced ECS user and want to be able to customize all the settings you normally use then stick to the mid level ECS constructs
- Either way both levels of abstraction remove a lot of boilerplate!

@aws-cdk/aws-ecs

```
const taskDefinition = new ecs.Ec2TaskDefinition(stack, 'TaskDef');
const container = taskDefinition.addContainer('web', {
  image: ecs.ContainerImage.fromRegistry("apps/myapp"),
  memoryLimitMiB: 256,
});

container.addPortMappings({
  containerPort: 80,
  hostPort: 8080,
  protocol: ecs.Protocol.TCP
});

const service = new ecs.Ec2Service(stack, "Service", {
  cluster,
  taskDefinition,
});

const service = new ecs.Ec2Service(stack, "Service", {
  cluster,
  taskDefinition,
});

const lb = new elbv2.ApplicationLoadBalancer(stack, 'LB', {
  vpc,
  internetFacing: true
});
const listener = lb.addListener('PublicListener', { port: 80, open: true });

listener.addTargets('ECS', {
  port: 80,
  targets: [service],
  // include health check (default is none)
  healthCheck: {
    interval: cdk.Duration.seconds(60),
    path: "/health",
    timeout: cdk.Duration.seconds(5),
  }
});
```

We need to see this code in action now…

Demo time!

Thanks a lot!

Q & A