Deploying Containerized Web and Serverless Applications using the Cloud Development Kit (CDK)

November 25, 2019

https://github.com/jvargh/aws-cdk-workshop

Joji Varghese

aws

Quick Facts

Introduction to Infrastructure as Code (IaC)

- What is IaC? Why use it?
  - ***Tool to create Cloud resources on your AWS account with minimal manual effort***
- Where does AWS CDK fit into the IaC space?
  - ***CDK allows you to use IaC in a programming language of your choice***
  - ***CDK allows creation of higher level constructs that create lower level resources on your account***
- What does AWS CDK offer that is unique?
  - ***CDK provides built-in Helper methods that automates manual work***
    - **https://docs.aws.amazon.com/cdk/api/latest/**

What tooling does AWS CDK offer for containerized applications?

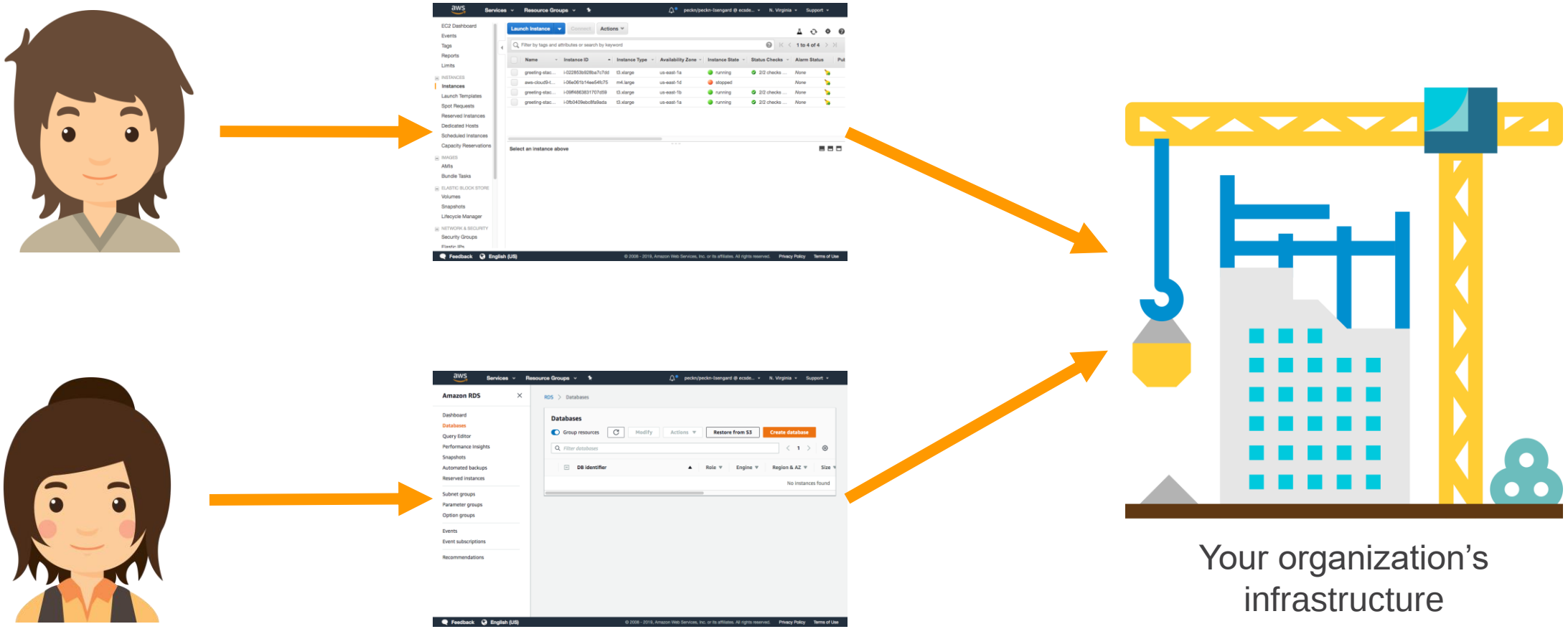
- High level reusable patterns for containers
  - ***aws-ecs-patterns (Beginner, High-level), aws-ecs (Advanced, Low-level) constructs***
- Underlying core constructs for more advanced configurations

Agenda

- IaC levels (ways to implement IaC) and their Pros/Cons
  - Level 0 – IaC by hand
  - Level 1 – Imperative IaC
  - Level 2 – Declarative IaC
  - Level 3 – CDK (Infra is Code, Infra as Class)
- Demos involving containerized applications
  - Deploy a container locally via Docker
  - Deploy Containerized (ECS) Web App with AWS CDK using **aws-ecs-patterns**
  - Deploy Containerized (ECS) Web App using **aws-ecs** construct
  - Deploy Serverless App (Lambda) with integration to SQS / DynamoDB
- Q&A

Ways to Implement Infrastructure as Code

Level 0 – Creating infrastructure by hand



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

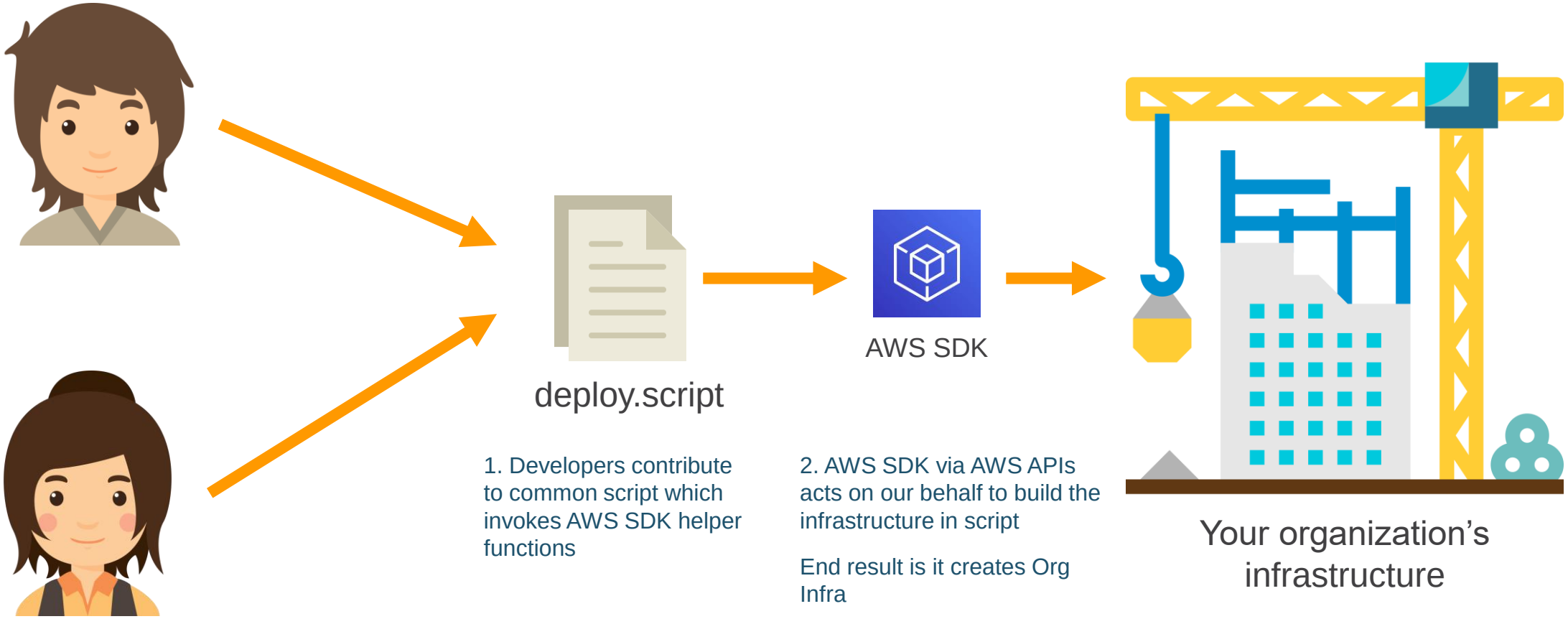
- Clicking things and entering values in the console by hand is slow. Bottleneck for long-term development
- It's hard to reliably reproduce your results when you are doing things manually, by hand.
- People make mistakes in data entry and clicking options.

  Can't standardize reliability
- Person A configures things one way, but person B configures things another way

  Can't standardize consistency

Cons

Level 1 – Imperative infrastructure as code



1. Developers contribute to common script which invokes AWS SDK helper functions

2. AWS SDK via AWS APIs acts on our behalf to build the infrastructure in script

End result is it creates Org Infra

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?**
- **Handle edge cases with new requirements. Leads to bulky, unreliable code**

aws

Level 1 – Imperative infrastructure as code

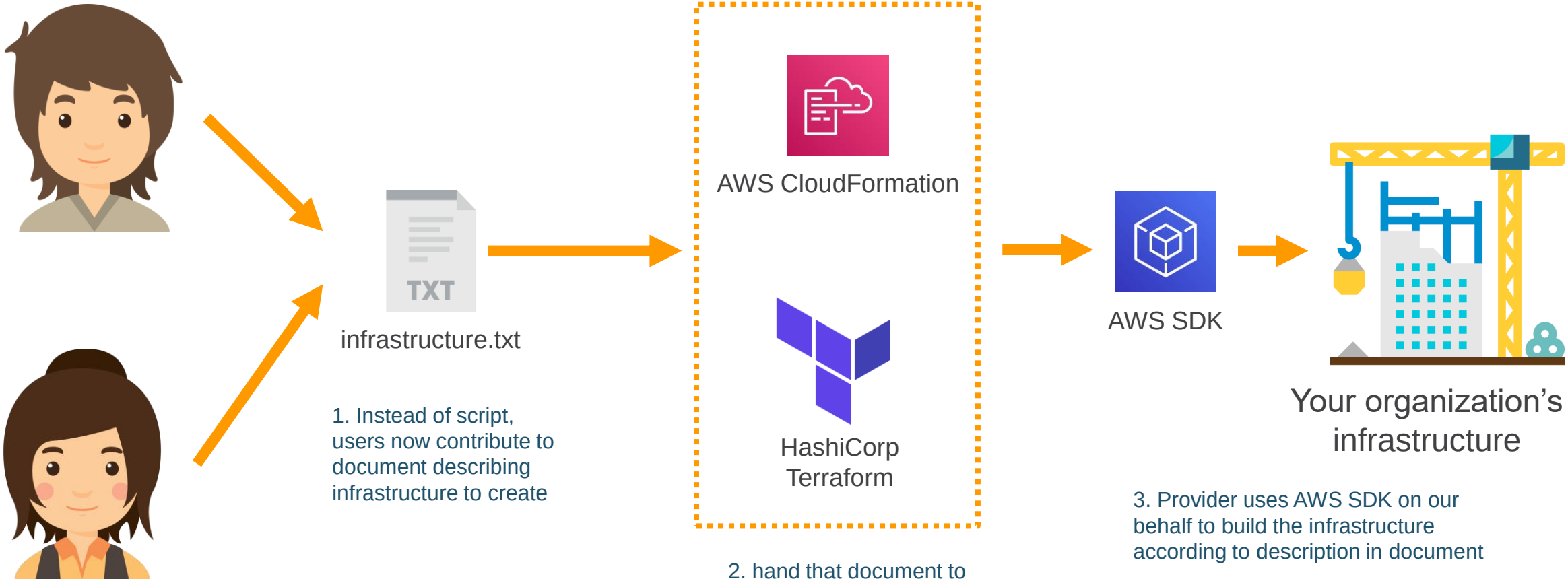
Pros

- 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 edge cases
- Must be careful about multiple people using the script at once

Cons

Level 2 – Declarative infrastructure as code



Level 2 – Declarative infrastructure as code

Narrate your Infrastructure resources into the YAML document in a structured meta-data format ☺

infrastructure.txt

```yaml
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
```

- List every resource to create and its properties, in YAML format in this case
- Helper functions may be built in to aid in fetching values dynamically.
- Not writing code that runs logic but describe what needs to be created and that translates to logic which gets run

aws

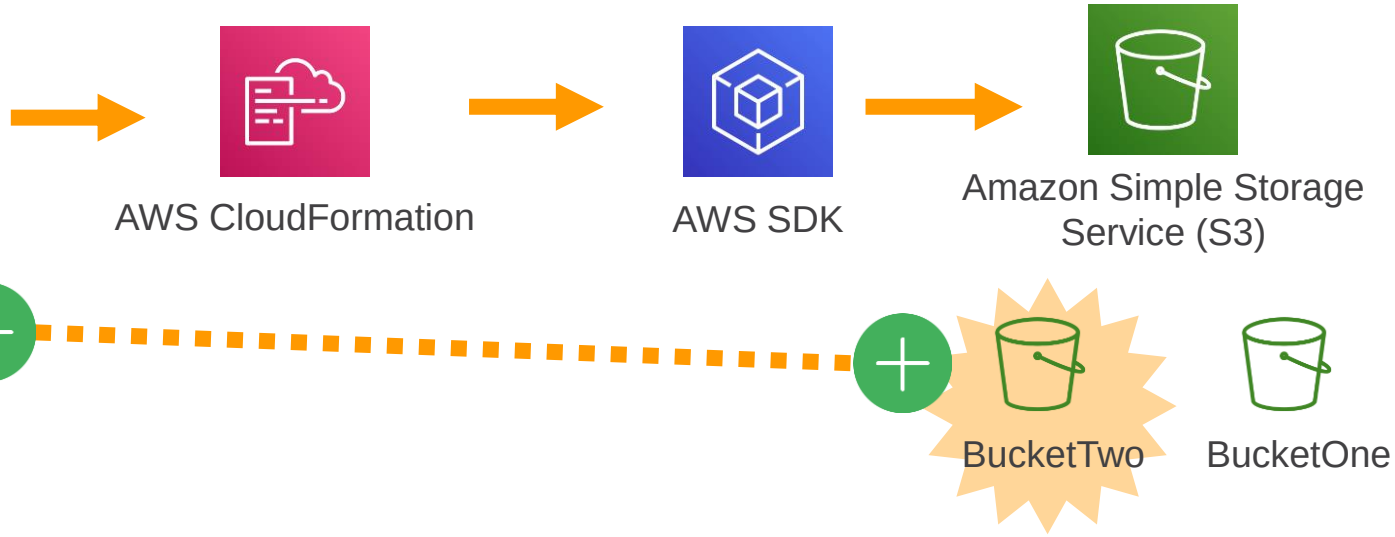
Level 2 – Declarative infrastructure as code

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

S3 bucket described > Pass to CF > SDK creates bucket in S3

Level 2 – Declarative infrastructure as code

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

BucketTwo:
    Type: AWS::S3::Bucket
    Properties:
      AccessControl: Private
```

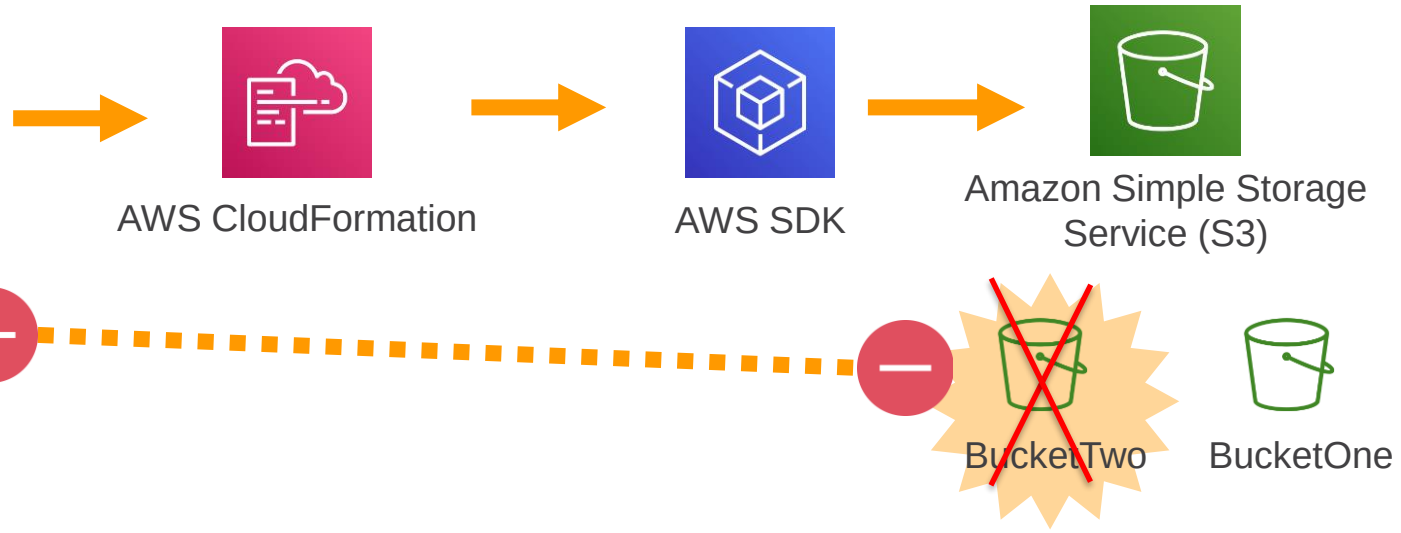


Add or Delete S3 bucket from doc is translated on the infra to add or delete S3 bucket

Level 2 – Declarative infrastructure as code

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

```
BucketTwo:
    Type: AWS::S3::Bucket
    Properties:
      AccessControl: Private
```



Level 2 – Declarative infrastructure as code

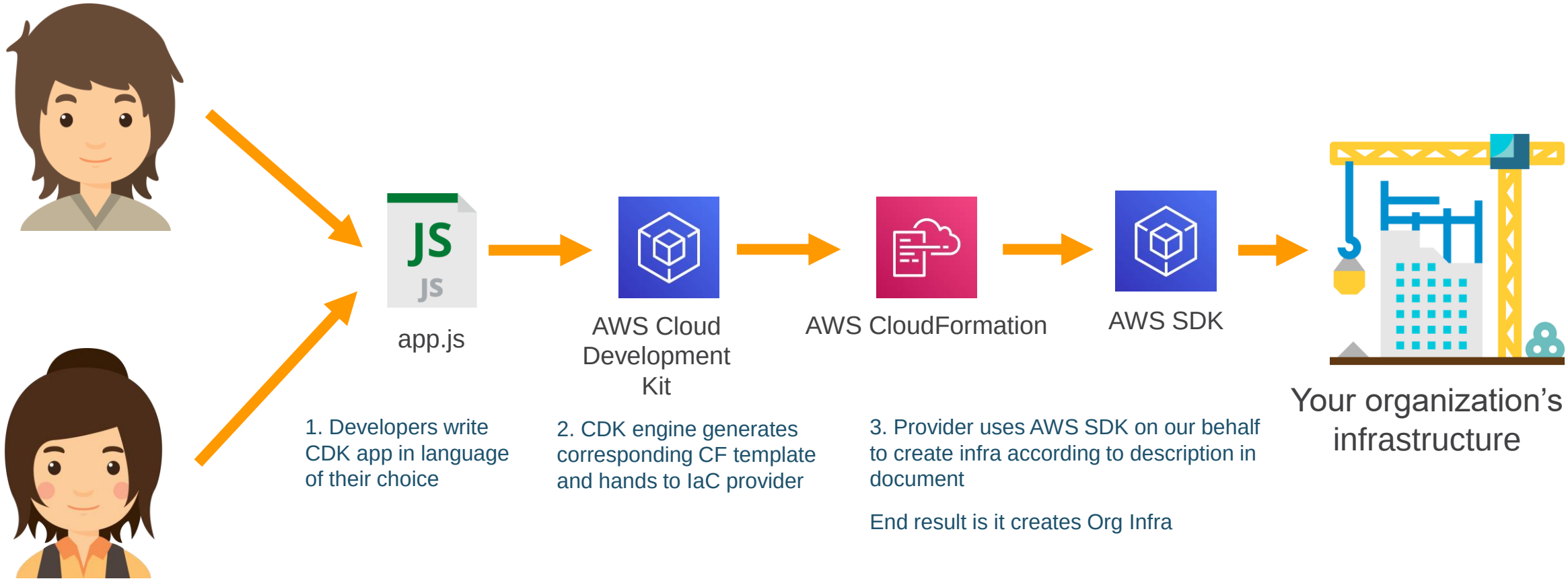
Pros

- No imperative boilerplate to write, creating and updating resources is handled automatically
- Multiple people can work on the template collaboratively. since CF locks stack
- 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 (Don't Repeat Yourself) without loops, functions, etc.

Cons

Level 3 – AWS Cloud Development Kit



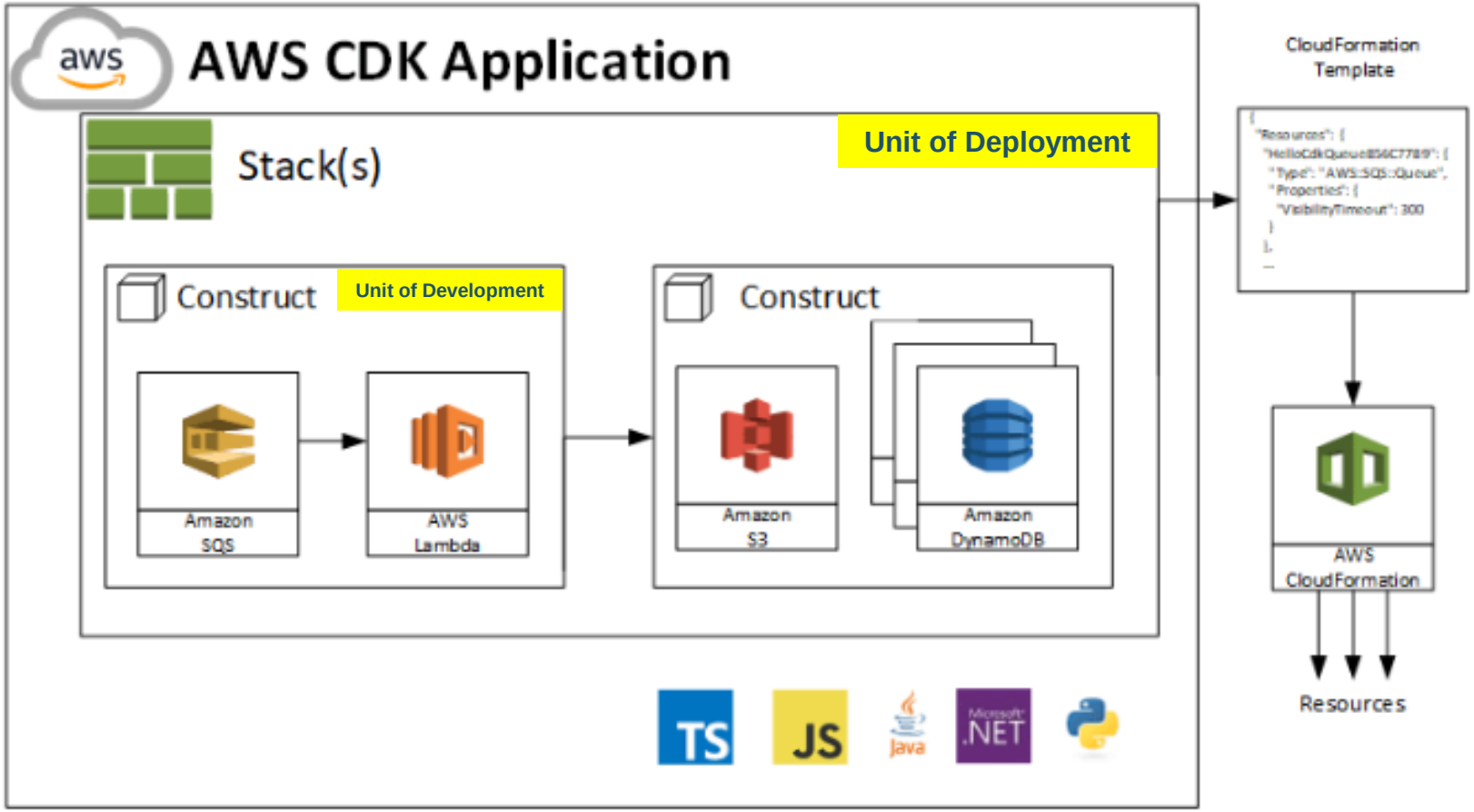
1. Developers write CDK app in language of their choice

2. CDK engine generates corresponding CF template and hands to IaC provider

3. Provider uses AWS SDK on our behalf to create infra according to description in document

End result is it creates Org Infra

Level 3 – AWS Cloud Development Kit



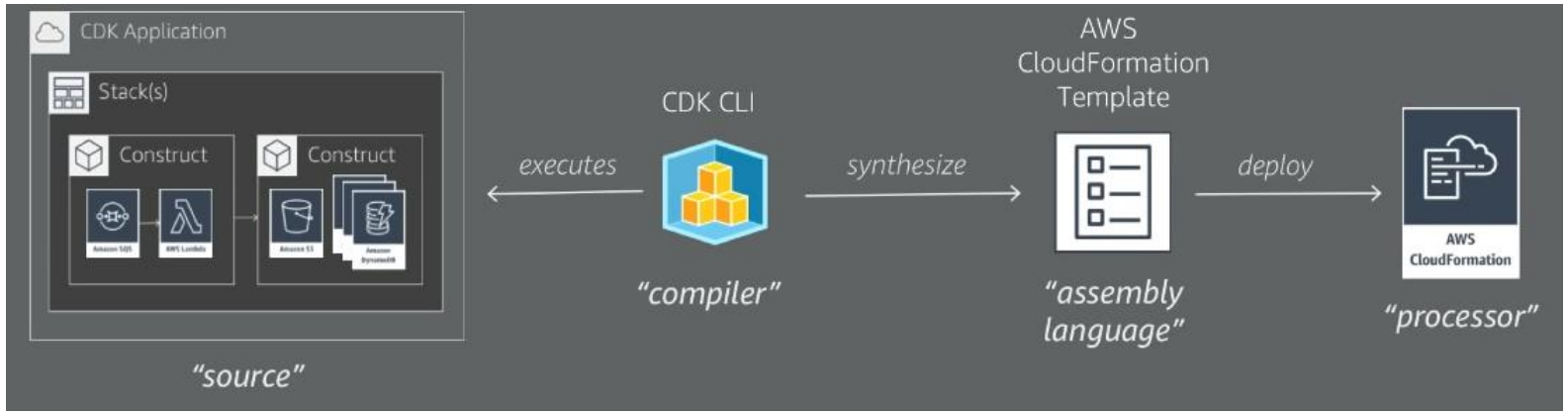
- **App** - A collection of related stacks.
- **Stack** - The set of AWS resources that are created and managed as a single unit when AWS CloudFormation instantiates a template
- **Construct** - Everything defined in the CDK is a **Construct**. It can be thought of as a re-usable "Cloud Component" representing anything from a single AWS resource to architectures of arbitrary complexity.

Level 3 – AWS Cloud Development Kit

app.js

app.py

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

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

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

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

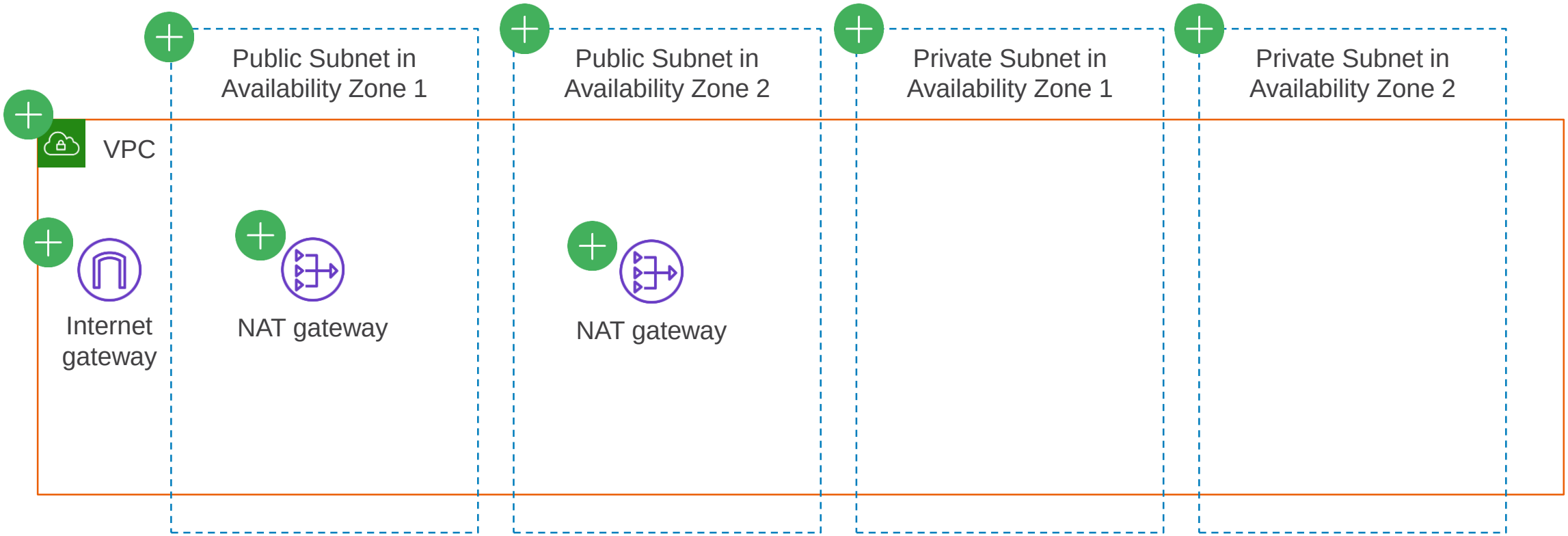


- Write in a familiar programming language
- Each stack is made up of "constructs" which are simple classes in the code
- Create many underlying AWS resources at once with a single construct (ec2.Vpc, ecs.Cluster)
- Resources organized into a Stack within same application
- Still declarative, no need 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 });
```

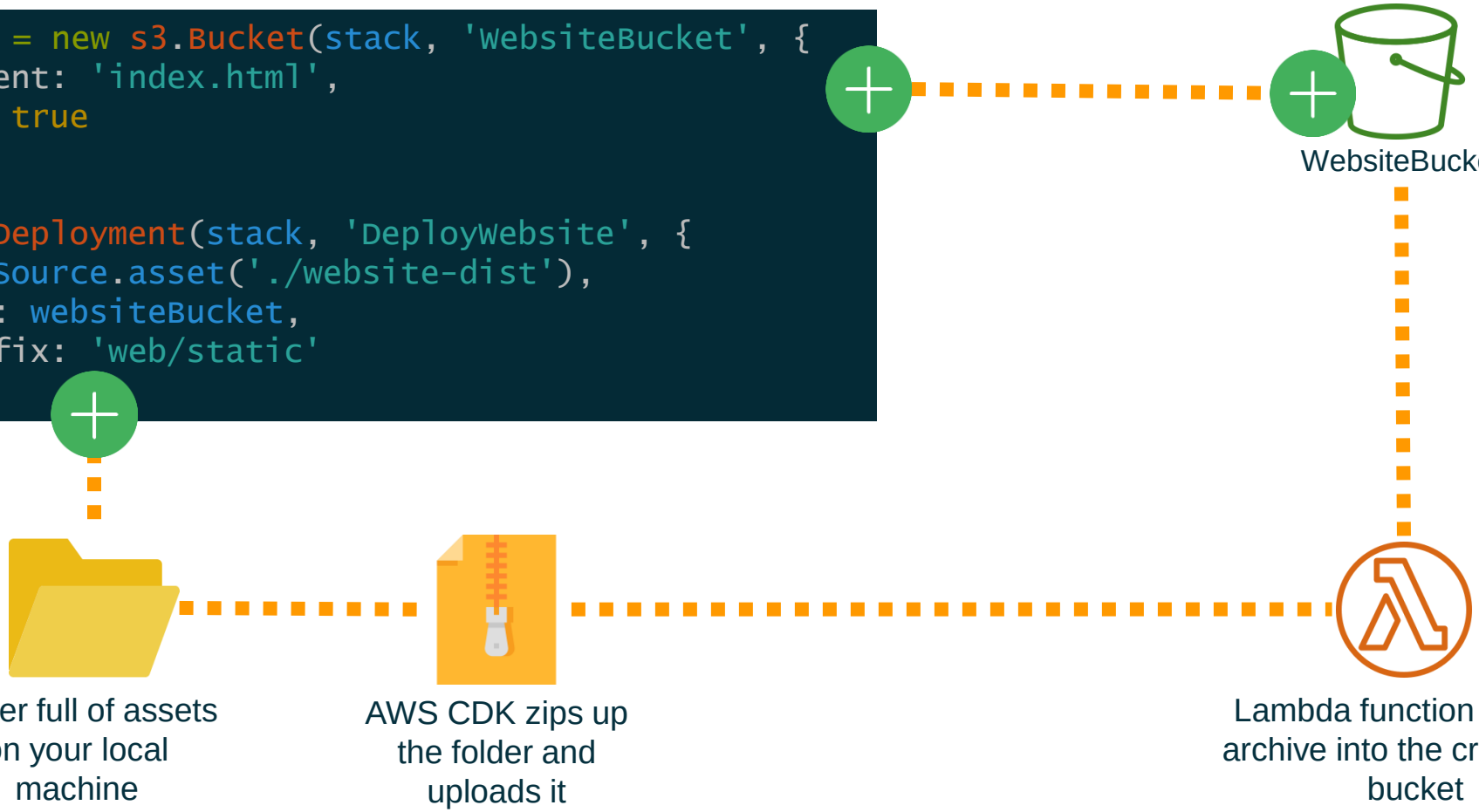
```
cdk synth (-j)
```

CDK Synth – synthesizes your constructs into CF template(s)

270 lines of CloudFormation YAML you don't have to write!

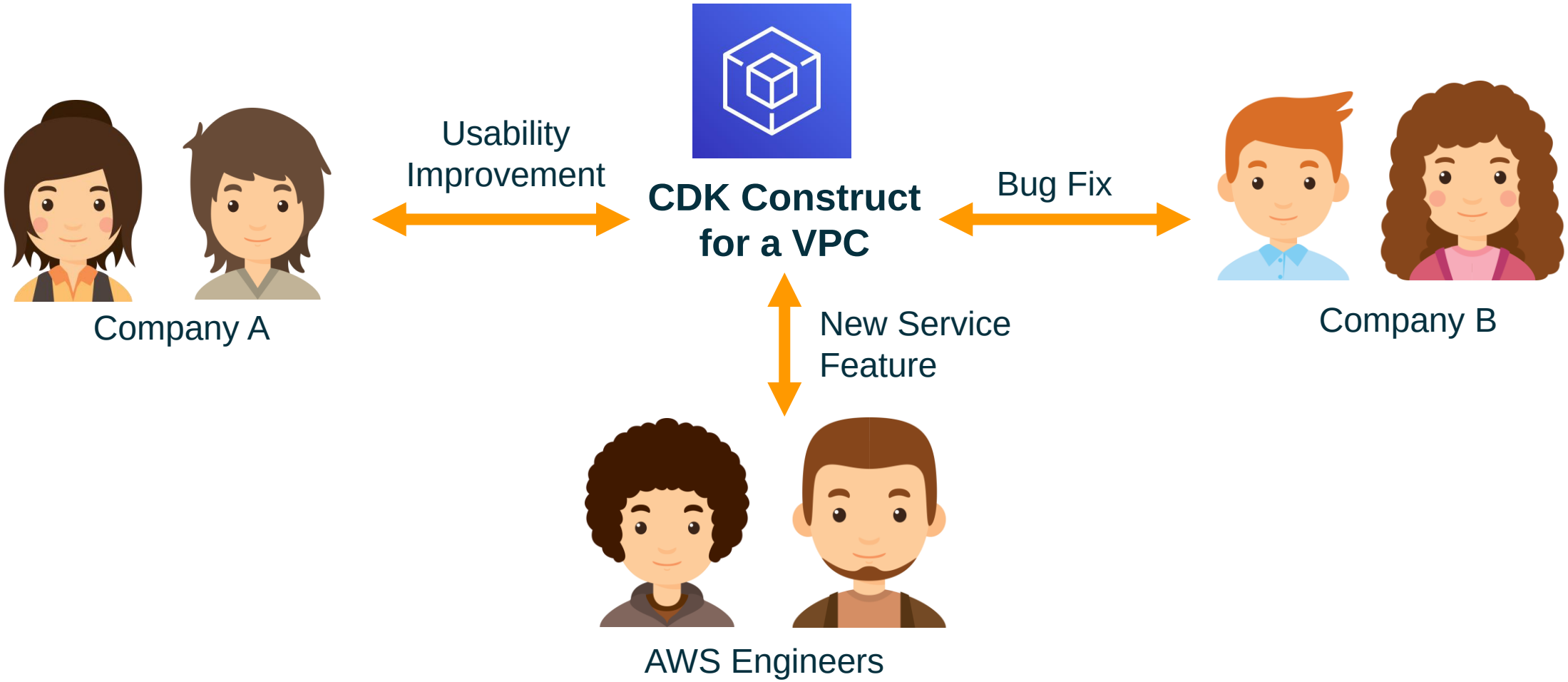
CDK helps with your local workflow too

```
const websiteBucket = new s3.Bucket(stack, 'WebsiteBucket', {
  websiteIndexDocument: 'index.html',
  publicReadAccess: true
});

new s3deploy.BucketDeployment(stack, 'DeployWebsite', {
  source: s3deploy.Source.asset('./website-dist'),
  destinationBucket: websiteBucket,
  destinationKeyPrefix: 'web/static'
});
```



CDK constructs are shareable and reusable

Subscribe to aws-cdk-interest@amazon.com



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!**

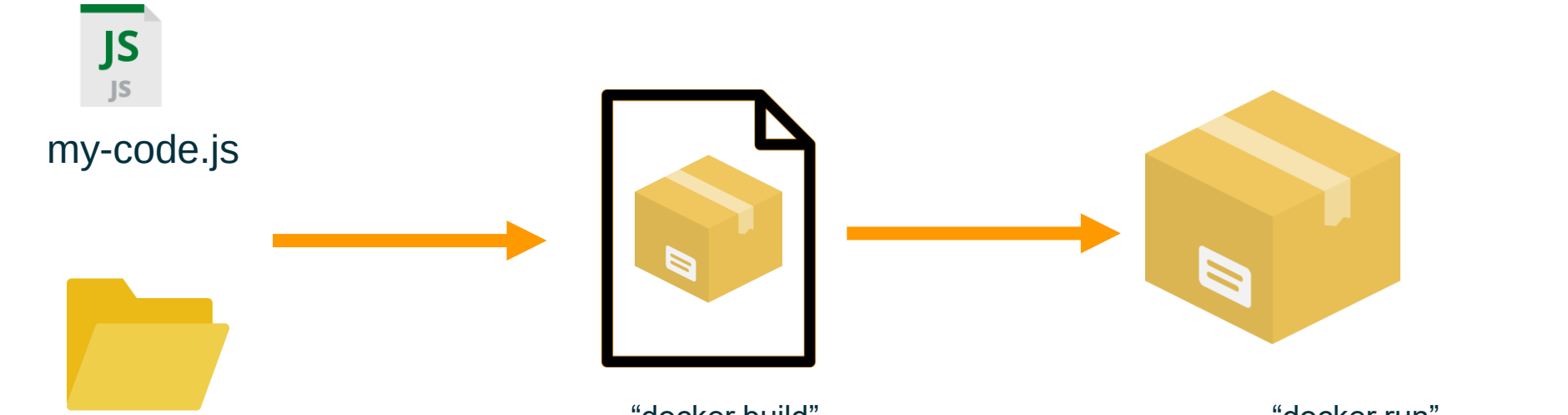
Level 3 – AWS Cloud Development Kit

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

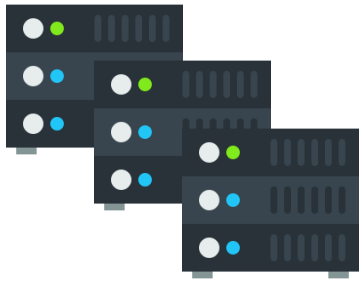


Add some container orchestration concepts

To launch a containerized application in AWS with ECS, follow the below steps

1. Registering task definition

Description of what containerized app to run and what settings it needs e.g. CPU, Memory

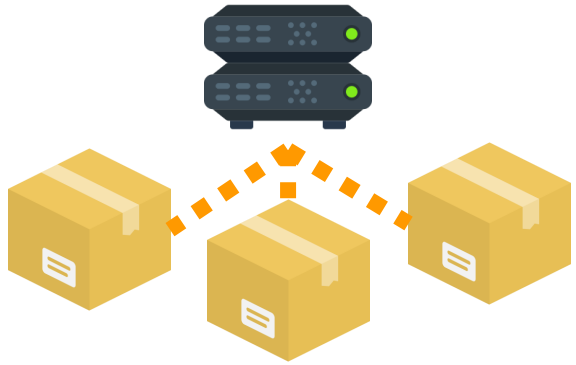


2. Create cluster

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

3. Run single task (Non-HA)

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



4. Create service (HA)

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

Approaches to using containers in CDK

@aws-cdk/aws-ecs-patterns

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

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.

- If you are just starting out with containers 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),
  }
});
```

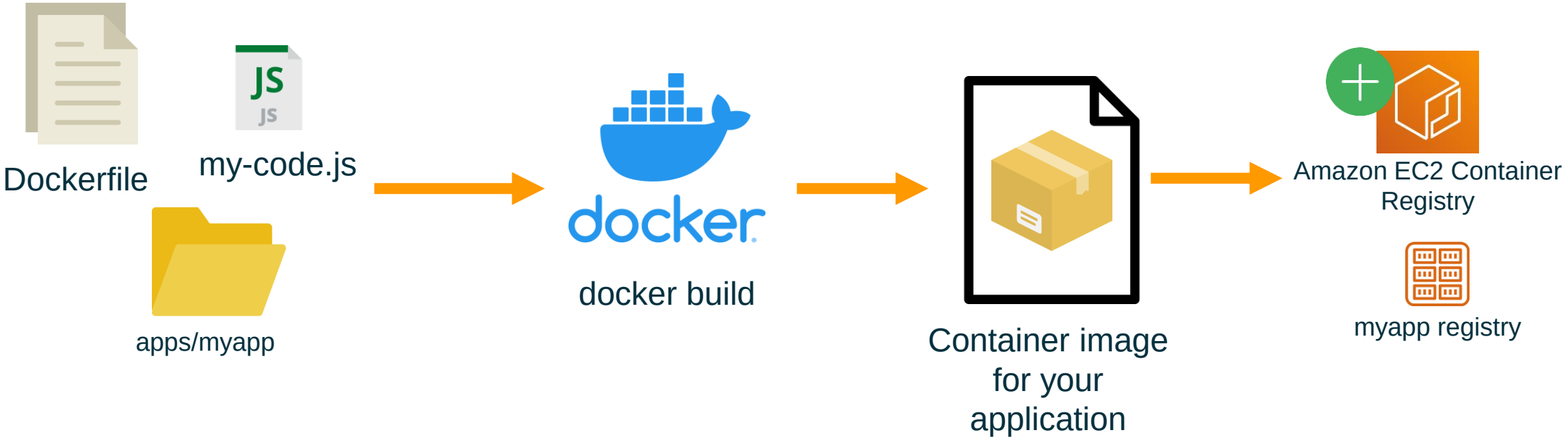
Basic patterns for building Docker images, creating a cluster, task definition, task, or service.

aws

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

This is another example of CDK helping with local workflow, by building and pushing container image to cloud

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

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



Construct 'fromAsset': Builds the docker image using Dockerfile, creates registry in ECR and pushes it to cloud

@aws-cdk/aws-ecs: Create cluster for application

Do you want to stay serverless (Fargate) as below ?

```
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 });
```

Or do you want to add EC2 instances and run on EC2 as below?

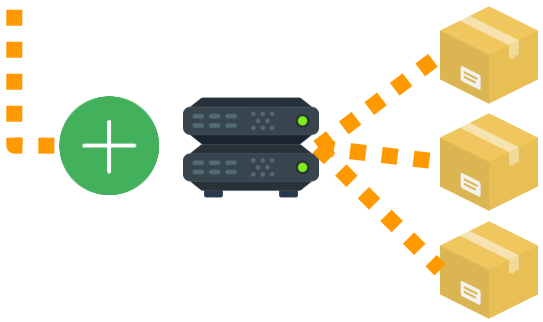
```
cluster.addCapacity('cluster-capacity', {
  instanceType: new ec2.InstanceType("t2.xlarge"),
  desiredCapacity: 3
});
```


aws

@aws-cdk/aws-ecs-patterns: Load balanced service

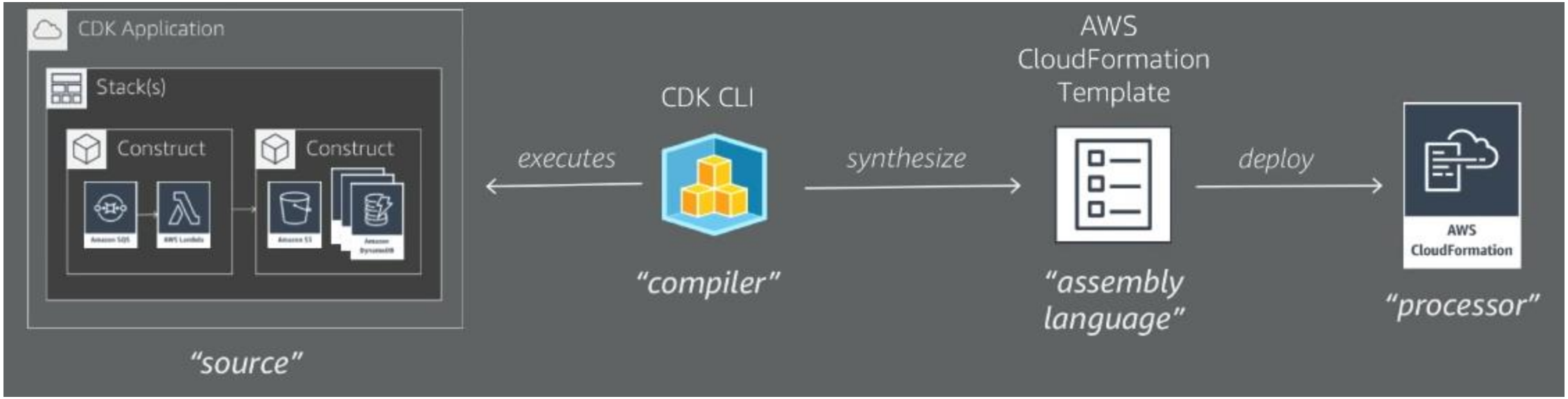
```
import ec2 = require('@aws-cdk/aws-ec2');
import ecs = require('@aws-cdk/aws-ecs');
import ecs_patterns = require('@aws-cdk/aws-ecs-patterns');

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

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

desiredCount=3 is total containers to be run within ECS cluster



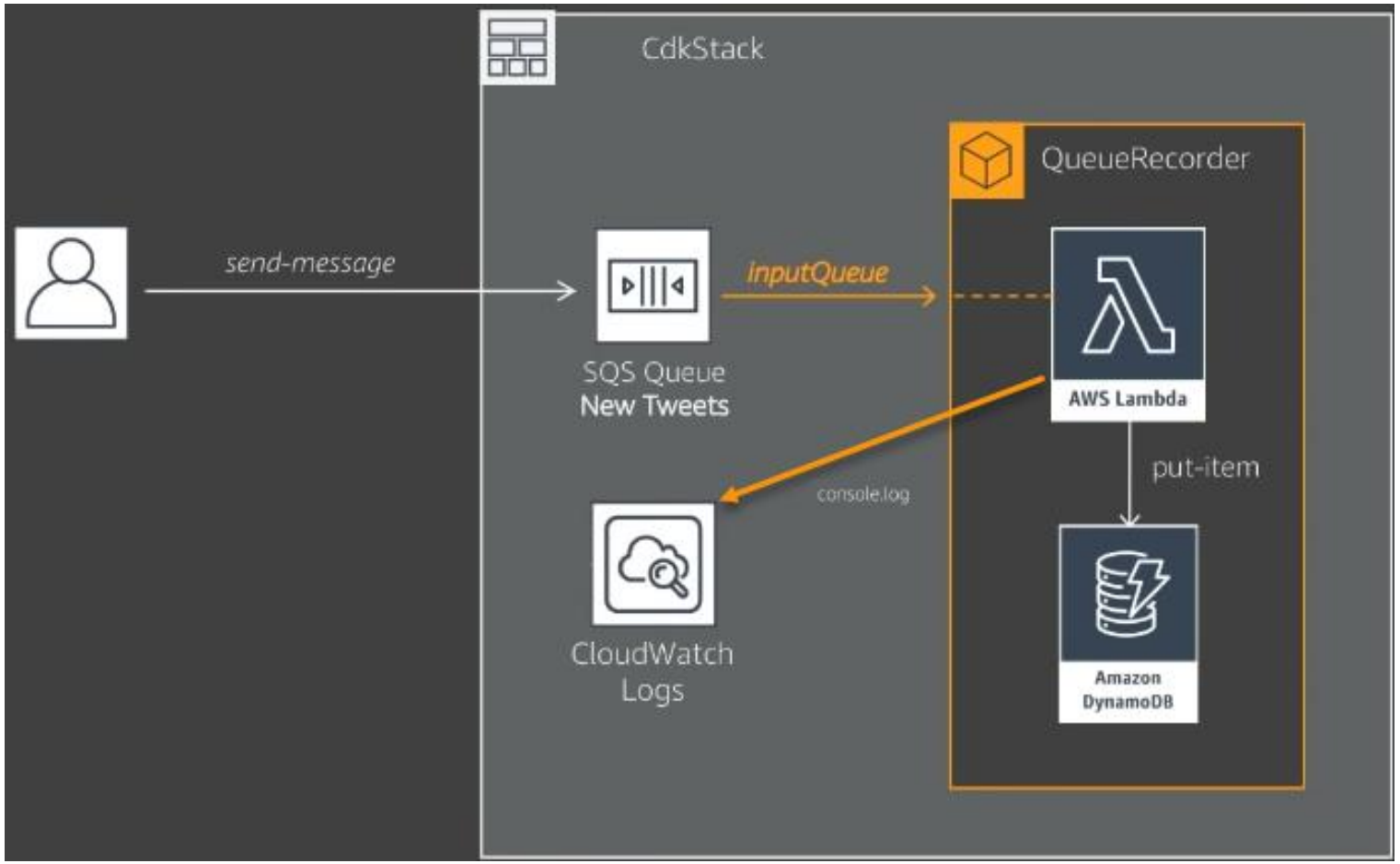
With a few lines we are automatically building a Docker container locally, pushing it up to the cloud in an Amazon Elastic Container Registry, then launching running three copies of it in AWS Fargate, behind a load balancer that distributes traffic across all three.

Deploy containerized (ECS) Web App with AWS CDK



Demo time!

Deploy Serverless App with integration to SQS / DynamoDB



Demo time!

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),
  }
});
```

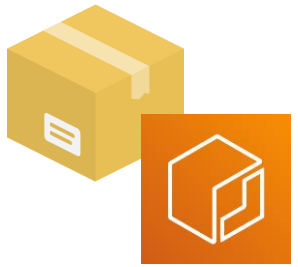
Add access to some other resources

```
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



Amazon Elastic Container Registry

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

AWS Identity and Access Management (IAM)

Role

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

Application Load Balancer

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

“Deploy Web App using aws-ecs construct”

Demo time!

Next steps

- CDK Workshop: https://cdkworkshop.com/
- CDK Documentation: https://docs.aws.amazon.com/cdk/api/latest/
- Github repo (search for CDK): https://aws.github.io/
- Github CDK Samples: https://github.com/aws-samples/aws-cdk-examples
- CDK Wiki: https://w.amazon.com/index.php/AWS/DeveloperResources/AWSSDKsAndTools/CDK
- Use course code to download and test: https://github.com/jvargh/aws-cdk-workshop

Thanks a lot!

Q & A