

CloudFormation on AWS – EKS Kubernetes

Author: Nho Luong

Skill: DevOps Engineer Lead



Screenshot of the AWS CertMetrics portal (cp.certmetrics.com/amazon/en/credentials/status) showing the "Certification status" page. The page displays four active certifications:

- PROFESSIONAL**: **AWS Certified Solutions Architect - Professional**. Active since 2024-06-21, expires 2027-06-21.
- SPECIALTY**: **AWS Certified Security - Specialty**. Active since 2024-06-19, expires 2027-06-19.
- PROFESSIONAL**: **AWS Certified DevOps Engineer - Professional**. Active since 2024-06-17, expires 2027-06-17.
- ASSOCIATE**: **AWS Certified Solutions Architect - Associate**. Active since 2024-06-15, expires 2027-06-21.

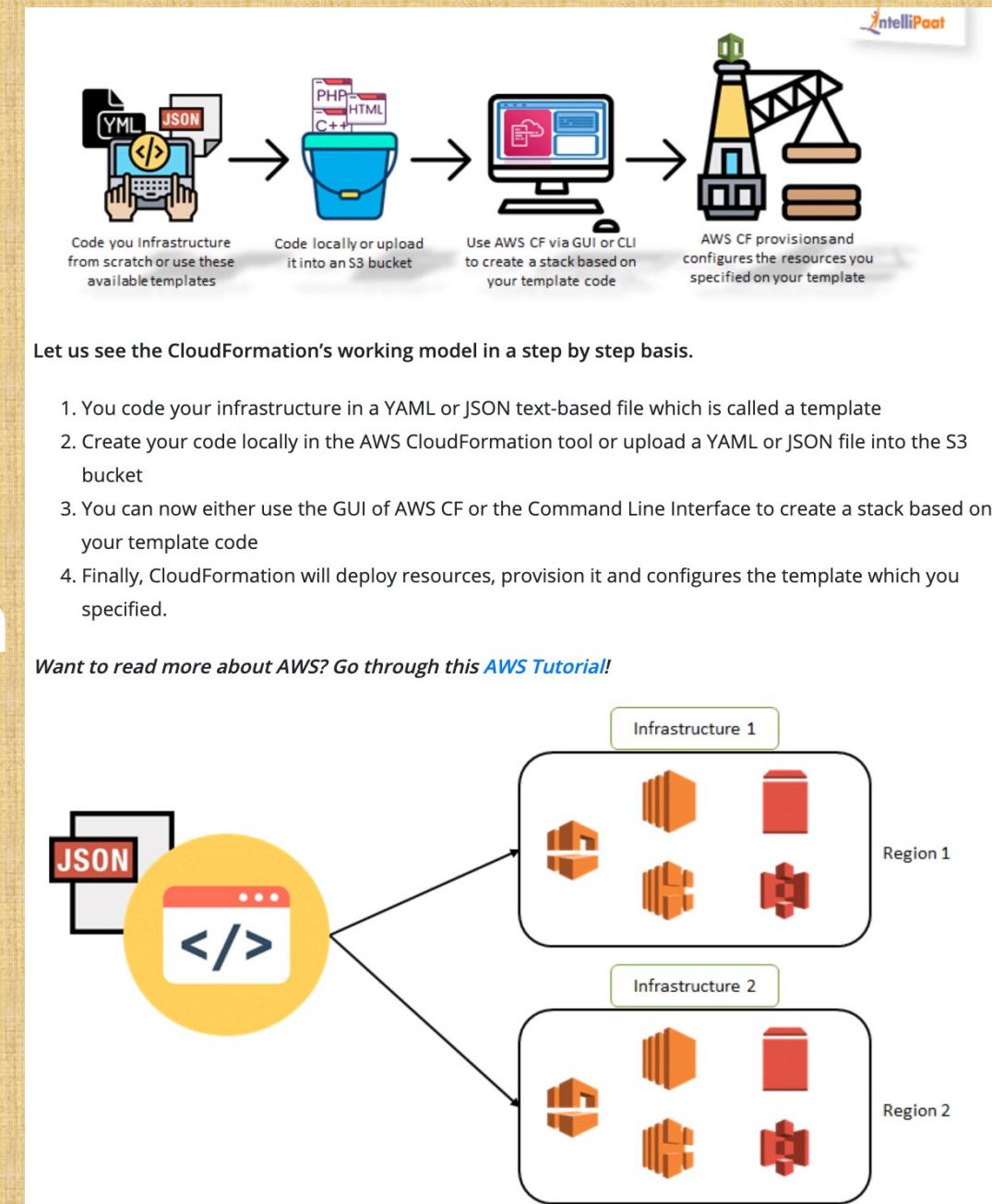
The sidebar on the left shows navigation links for HOME, PROFILE, EXAM REGISTRATION, EXAM HISTORY, CERTIFICATIONS (selected), BENEFITS, DIGITAL BADGES, and SUPPORT AND FAQS. The user profile on the right shows "NHO LUONG" and "AWS04440050".



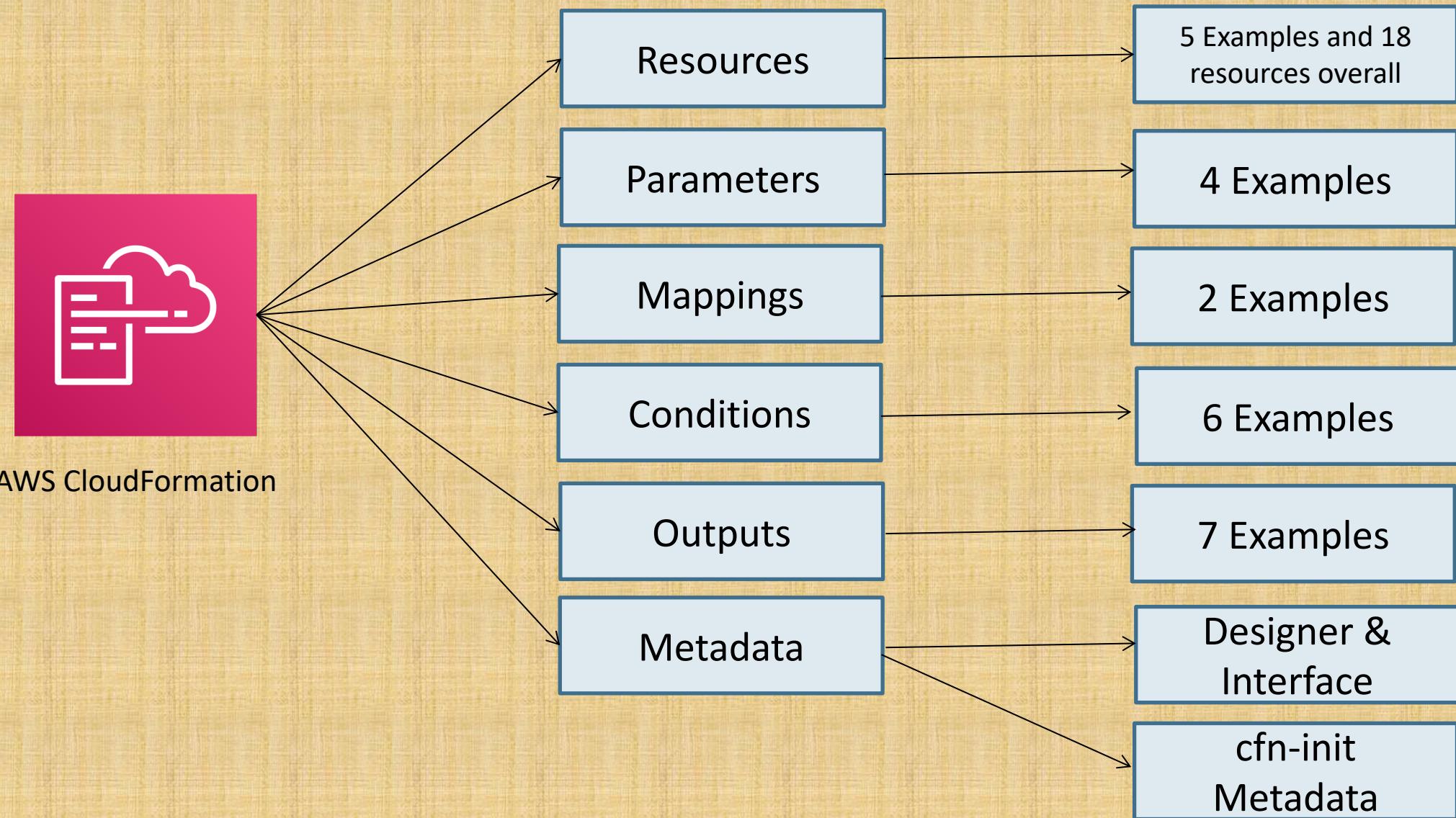
Demos Objectives

- AWS CloudFormation
- Continuous Integration
 - AWS Code Commit
 - AWS Code Build
- Continuous Delivery
 - AWS CodeDeploy
 - AWS CodePipeline
- Infrastructure as Code with CI/CD Tools.

AWS CloudFormation



Template Anatomy



Resources



AWS CloudFormation

Resources

EC2 Instance

CodeBuild

Security Group

CodeBuild IAM Role

Elastic IP

CodeDeploy Application

VPC

CodeDeploy DeploymentGro

Subnet

CodeDeploy Deploymnet

Route Table

CodeDeploy IAM Role

SubnetRoute
TableAssociation

CodePipeline IAM Role

InternetGateway

CodePipeline Pipeline

VPCGatewayAtta
chment

SNS Topic

Templates Written

- ▶ 02-YAML-Basics
 - ! 02-01-YAML-Basics.yml
- ▶ 03-StackConcepts
- ▶ 04-Resources
 - ! 04-01-cfn-ec2-instance.yml
 - ! 04-02-cfn-ec2-add-new-security-group.yml
 - ! 04-03-cfn-ec2-add-new-rule.yml
 - ! 04-04-cfn-ec2-add-elasticIP.yml
 - ! 04-05-cfn-ec2-case-sensitive-test.yml
- ▶ 05-Parameters
 - ! 05-01-cfn-ec2-ParameterType-AWS-KeyName.yml
 - ! 05-02-cfn-ec2-ParameterType-String.yml
 - ! 05-03-cfn-ec2-ParameterType-InstanceType.yml
 - ! 05-04-cfn-ec2-ParameterType-SSM.yml
- ▶ 06-Mappings
 - ! 06-00-Base.yml
 - ! 06-01-cfn-ec2-Mapping-AMI.yml
 - ! 06-02-cfn-ec2-Mapping-AMI-and-Environment.yml
- ▶ 07-Conditions
 - ! 07-00-Base.yml
 - ! 07-01-Conditions-IntrinsicFunction-Fn-Equals.yml
 - ! 07-02-Conditions-IntrinsicFunction-Fn-If-PseudoParameter-AV.yml
 - ! 07-03-Conditions-IntrinsicFunction-Fn-If.yml
 - ! 07-04-Conditions-IntrinsicFunction-Fn-Not.yml
 - ! 07-05-Conditions-IntrinsicFunction-Fn-Or.yml
 - ! 07-06-Conditions-IntrinsicFunction-Fn-And.yml

- ▶ 08-Outputs
 - ! 08-00-Base.yml
 - ! 08-01-cfn-ec2-Outputs-InstanceId.yml
 - ! 08-02-cfn-ec2-Outputs-Intrinsic-Fn-GetAtt.yml
 - ! 08-03-cfn-ec2-Outputs-Export-Intrinsic-Fn-Sub-Pseudo-StackName.yml
 - ! 08-04-cfn-ec2-Outputs-Cross-Reference-Intrinsic-Fn-ImportValue.yml
 - ! 08-05-cfn-ec2-Outputs-Conditions.yml
 - ! 08-06-cfn-ec2-Outputs-Export-Intrinsic-Fn-Join.yml
- ▶ 09-Metadata
 - ! 09-00-Base.yml
 - ! 09-01-cfn-ec2-Metadata-Interface.yml
- ▶ 10-EC2-UserData
 - ! 10-00-Base.yml
 - ! 10-00-Userdata.sh
 - ! 10-01-cfn-ec2-UserData.yml

Metadata - cfn-init

Base Template

Metadata Format

packages

groups

users

sources

files

commands

services

aws-cfn-bootstrap

cfn-init

cfn-signal

outputs

Creation Policy

cfn-hup

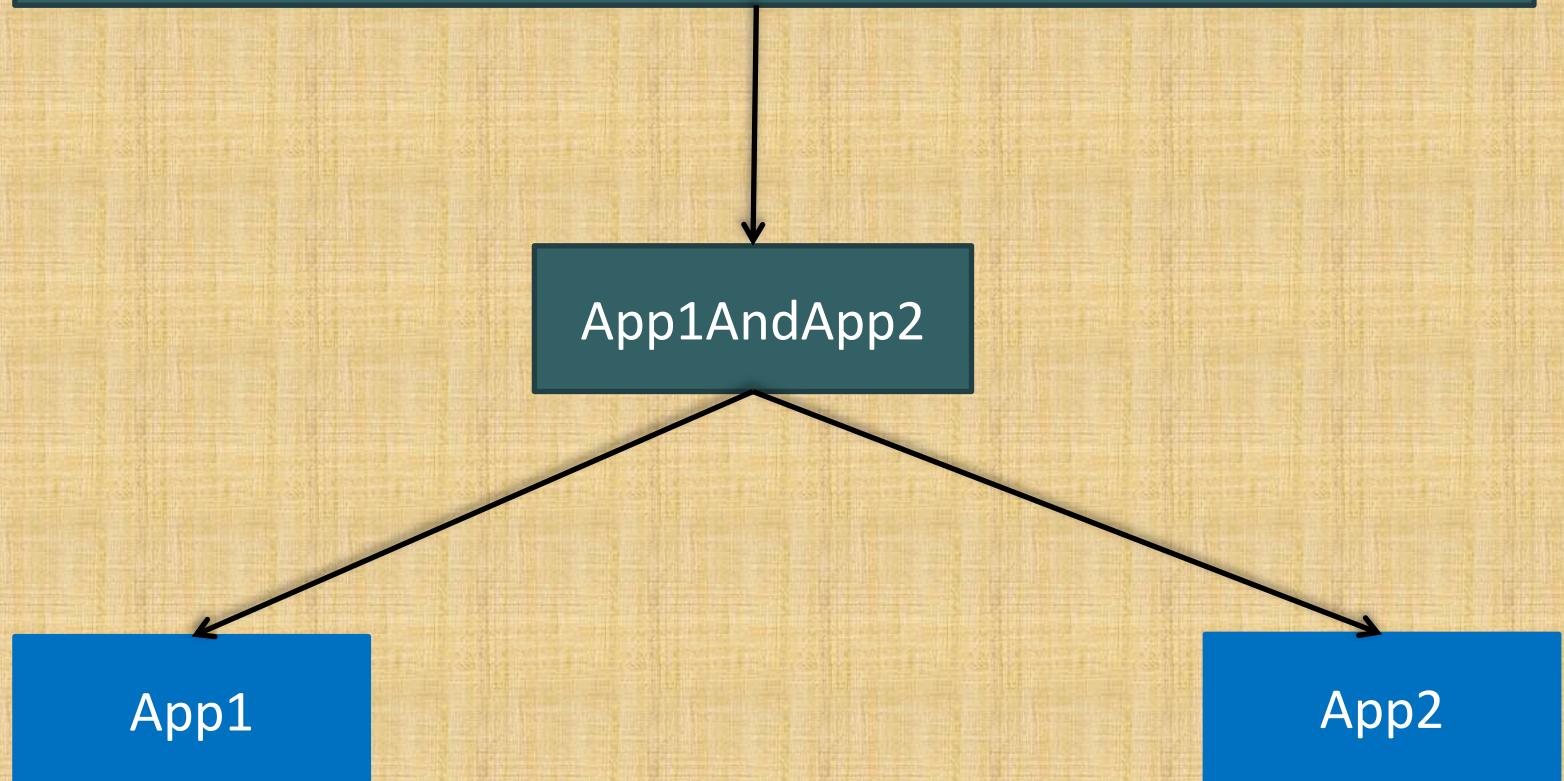
Create Stack & Test

Create Stack & Test

Update Stack – Deploy v2 App

Config Sets

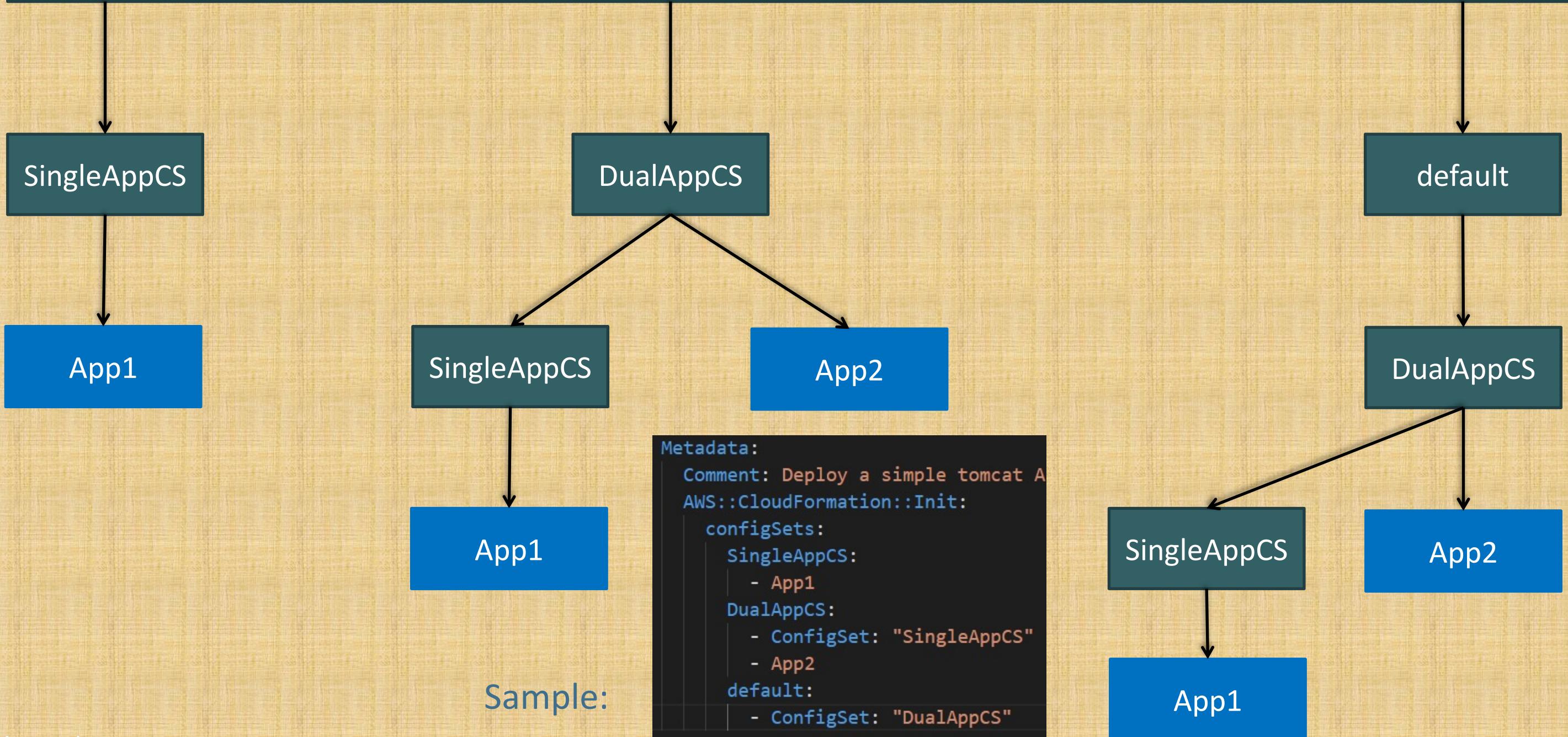
configSets (Single)



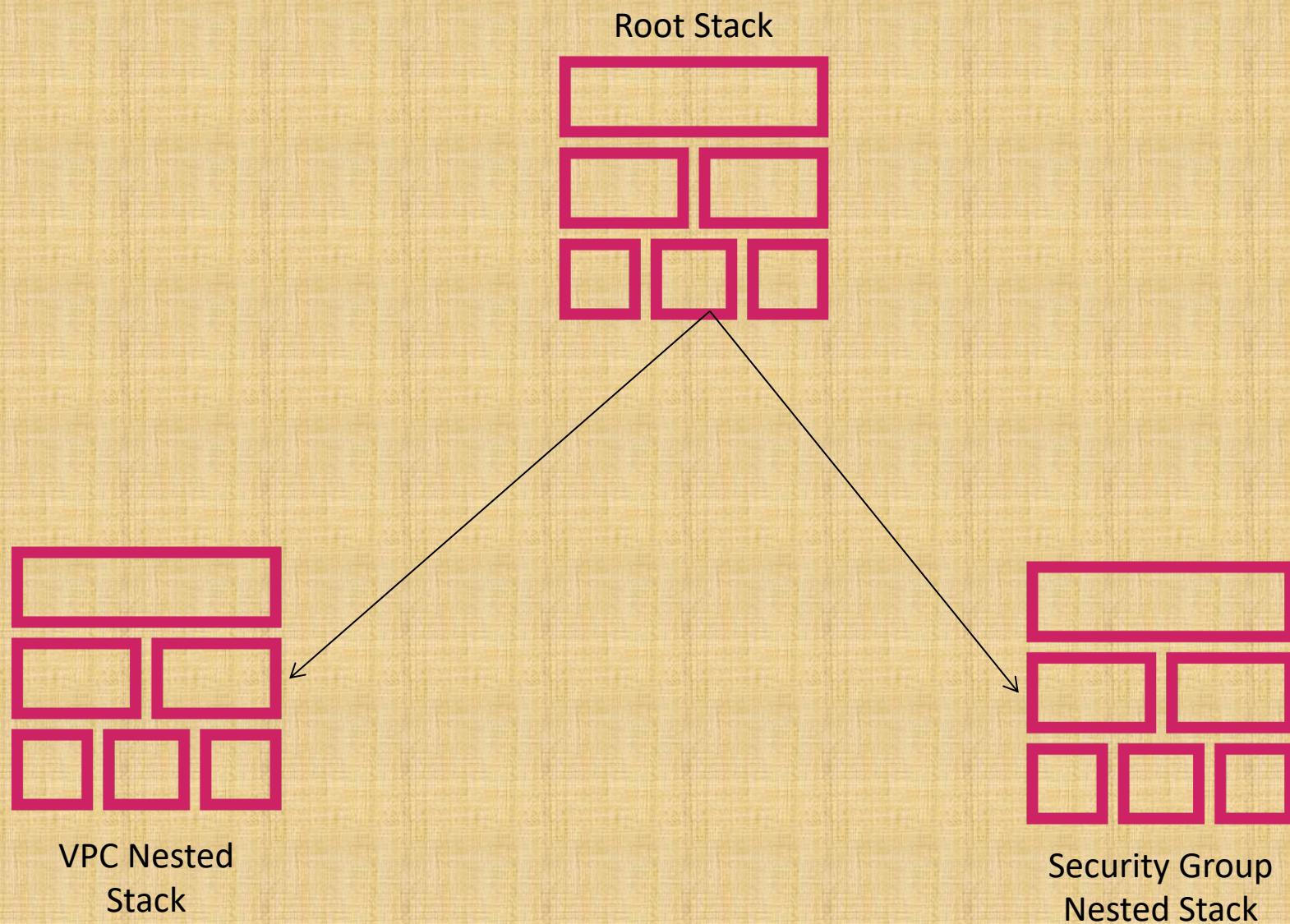
Sample:

```
Metadata:  
Comment: Deploy a simple tomcat Application  
AWS::CloudFormation::Init:  
configSets:  
App1AndApp2:  
- App1  
- App2
```

configSets (Multiple)



Nested Stacks



Templates Written



Author: Nho Luong

Skill: DevOps Engineer Lead

Intrinsic Functions & Pseudo Parameters

Intrinsic Functions

- Fn::Ref
- Fn::Base64
- Fn::FindInMap
- Fn::GetAtt
- Fn::GetAzs
- Fn::ImportValue
- Fn::Join
- Fn::Select
- Fn::Sub

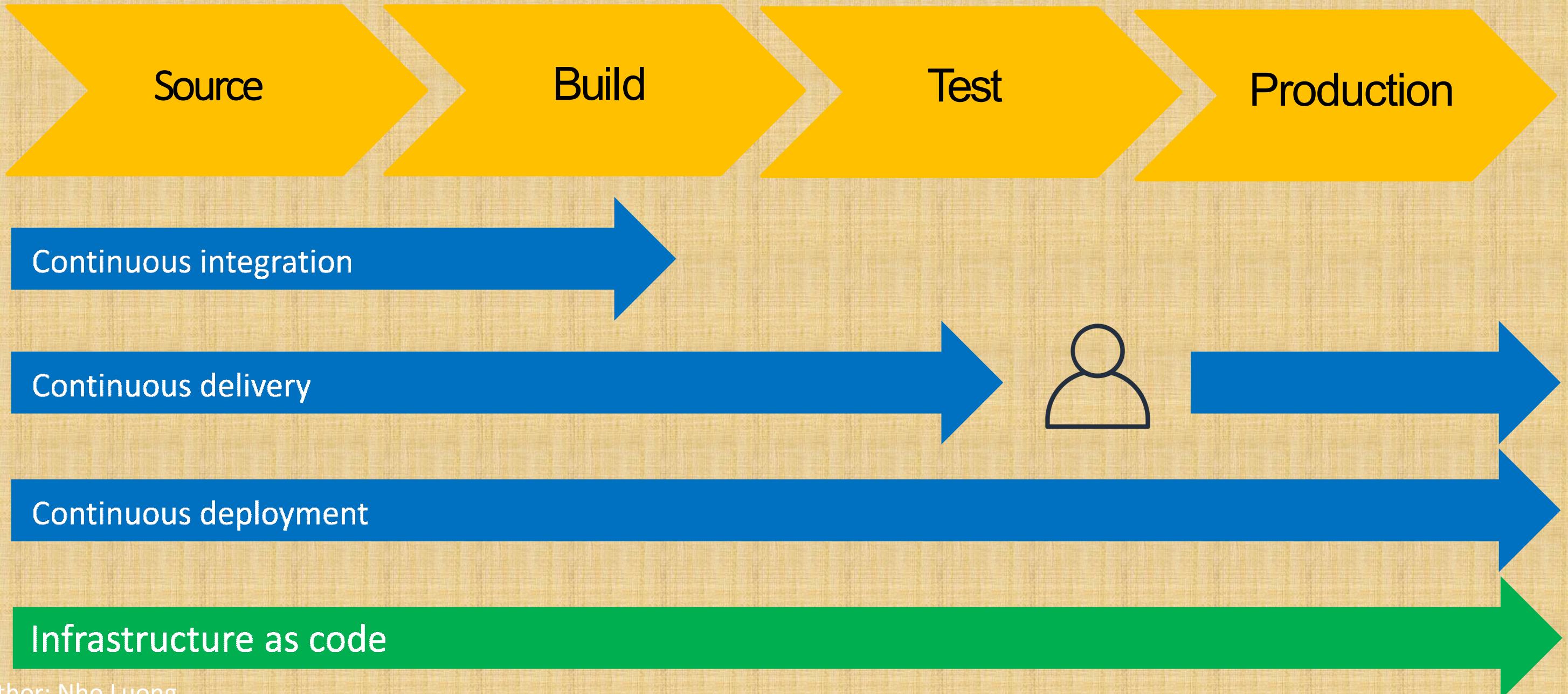
Condition Functions

- Fn::And
- Fn::Equals
- Fn::If
- Fn::Not
- Fn::Or

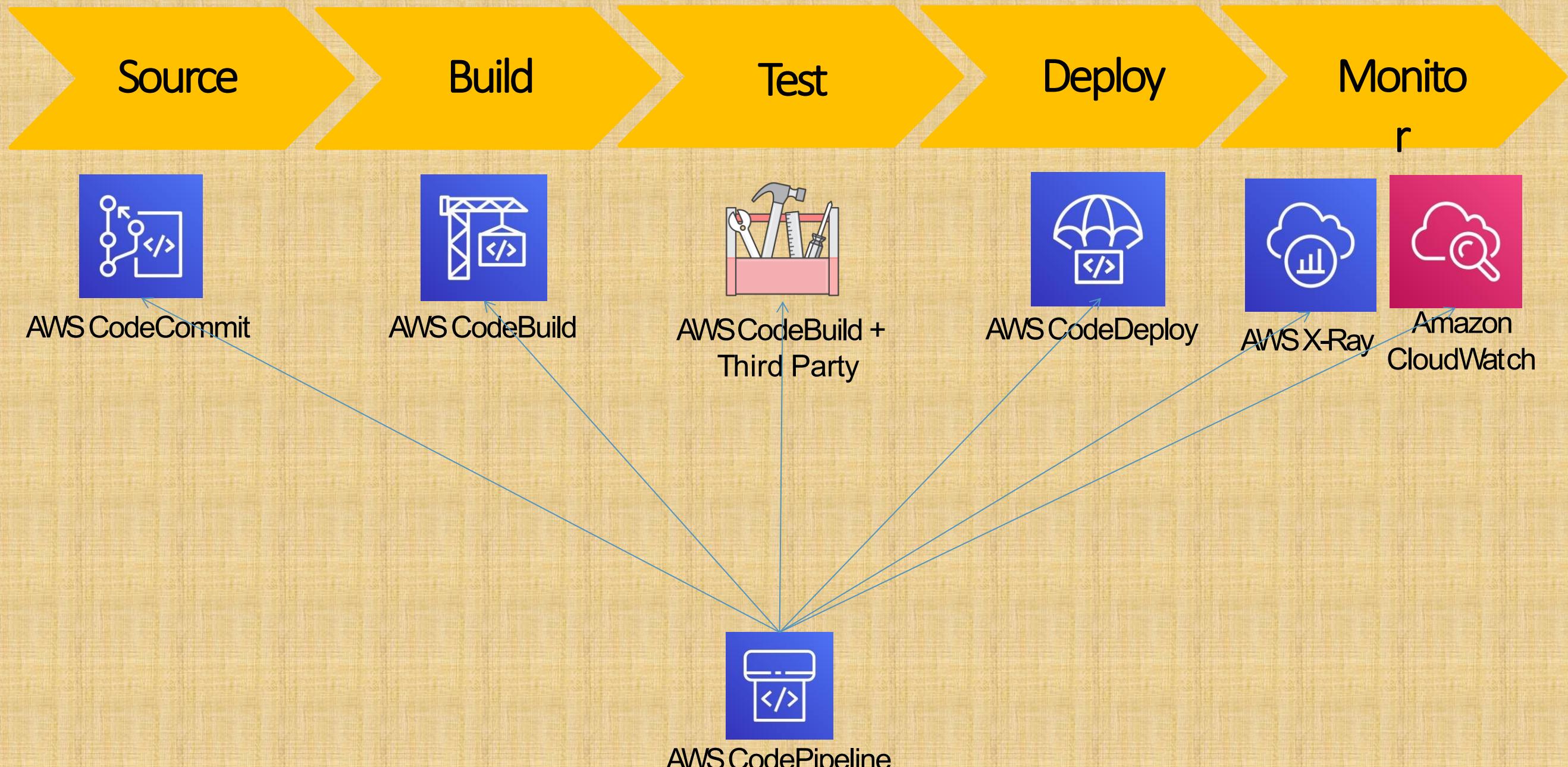
Pseudo Parameters

- AWS::Region
- AWS::AccountId
- AWS::StackName
- AWS::NoValue (very important when using conditions)

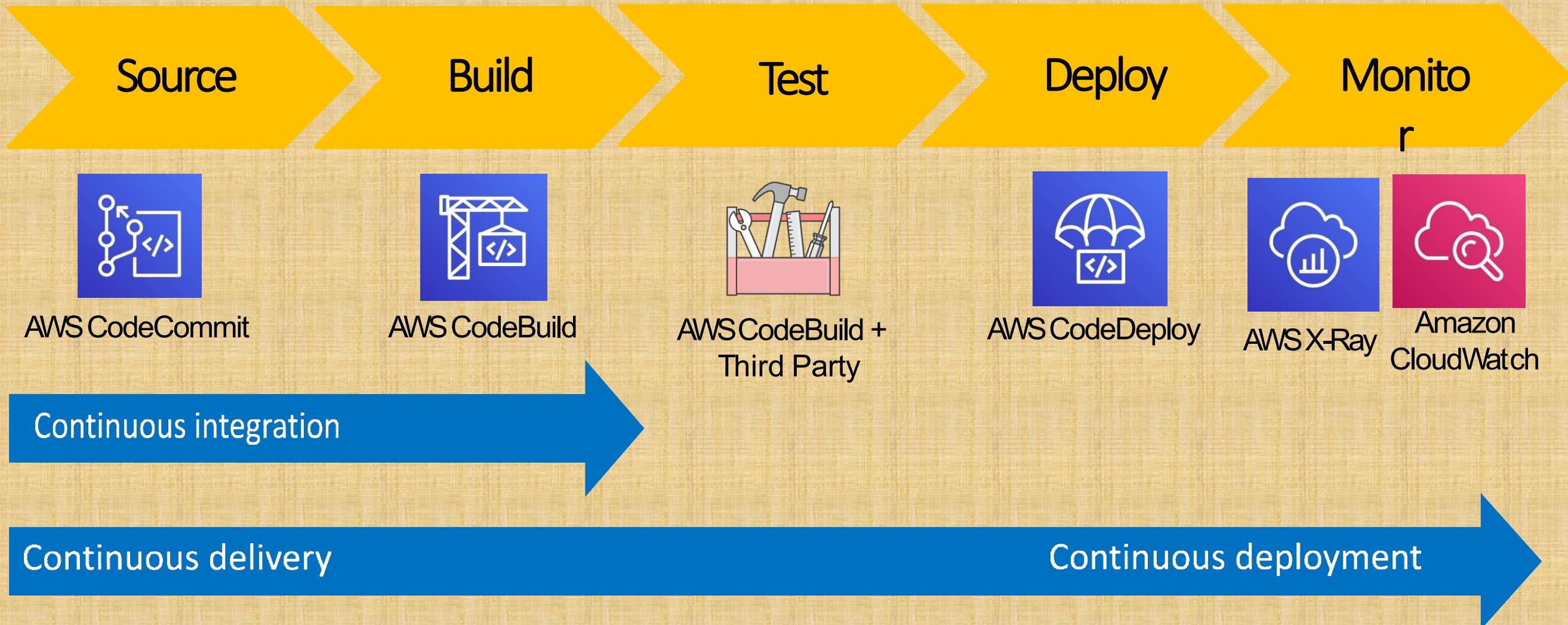
Stages in Release Process



AWS Developer Tools or Code Services



AWS Developer Tools or Code Services



AWS CodePipeline

AWS Developer Tools or AWS Code Services

Source



CodeCommit

Build



CodeBuild

Test



CodeBuild + Third Party

Deploy



CodeDeploy

Monito



AWS X-Ray



CloudWatch

- Version control service
- We can privately store and manage source code
- Secure & highly available

- Fully managed build service, Compiles source code, Runs tests and produces software packages
- Scales continuously and processes multiple builds concurrently.
- No build servers to manage.
- Pay by minute, only for compute resources we use.
- Monitor builds through CloudWatch events.
- Supports following programming language runtimes Ruby, Python, PHP, Node, Java, Golang, .Net Core, Docker and Android

- Automates code deployments to any instance and Lambda
 - Avoids downtime during application deployment
 - Roll back automatically if failure detected
 - Deploy to Amazon EC2, Lambda, or on-premises servers

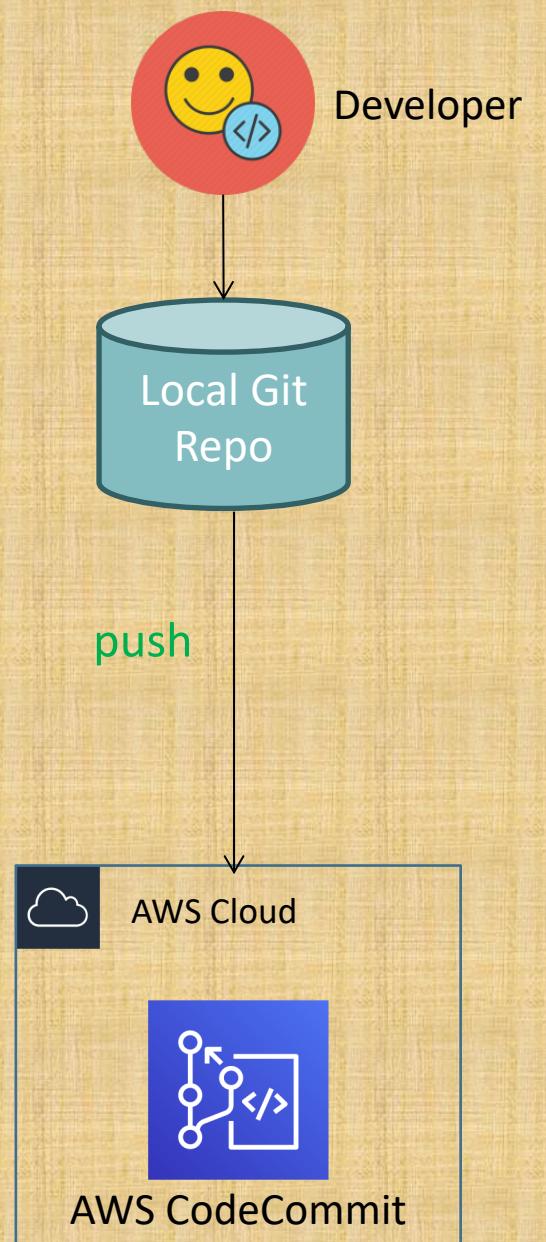


CodePipeline

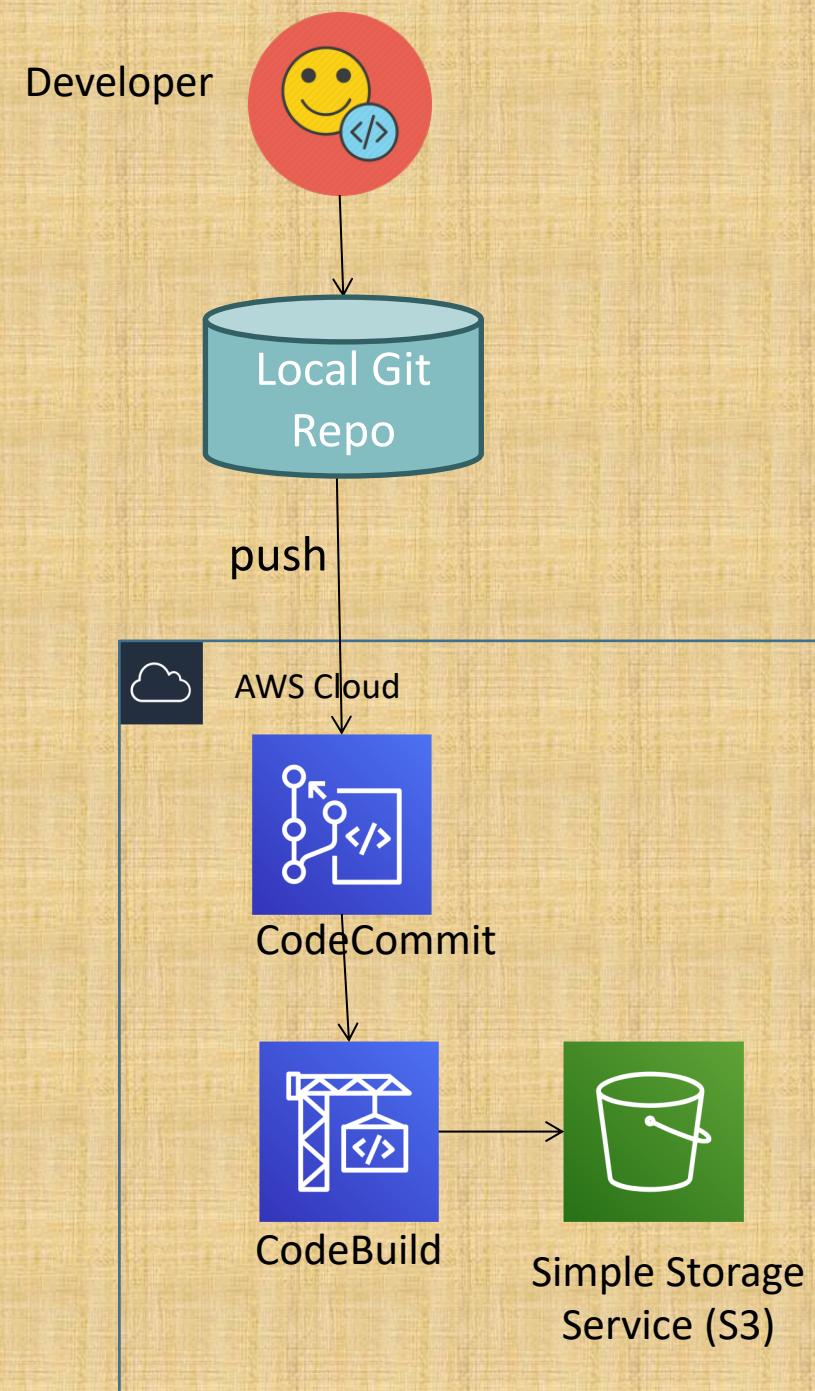
- Continuous delivery service for fast and reliable application updates
- Model and visualize your software release process
- Builds, tests, and deploys your code every time there is a code change
- Integrates with third-party tools and AWS

CodeCommit

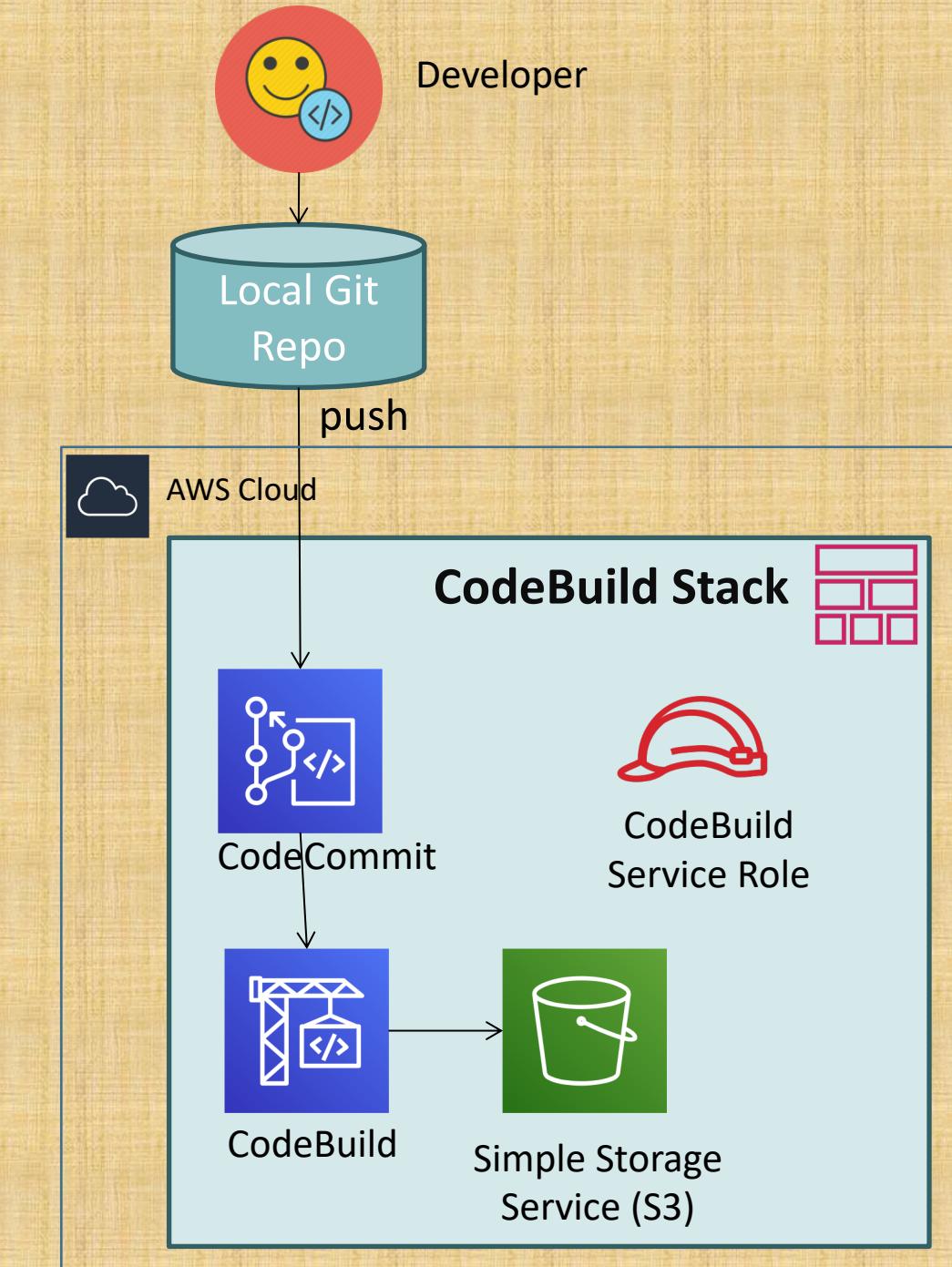
- Build a simple rest service using Java Spring Boot.
- Check-in code to Local Repo and push to CodeCommit.



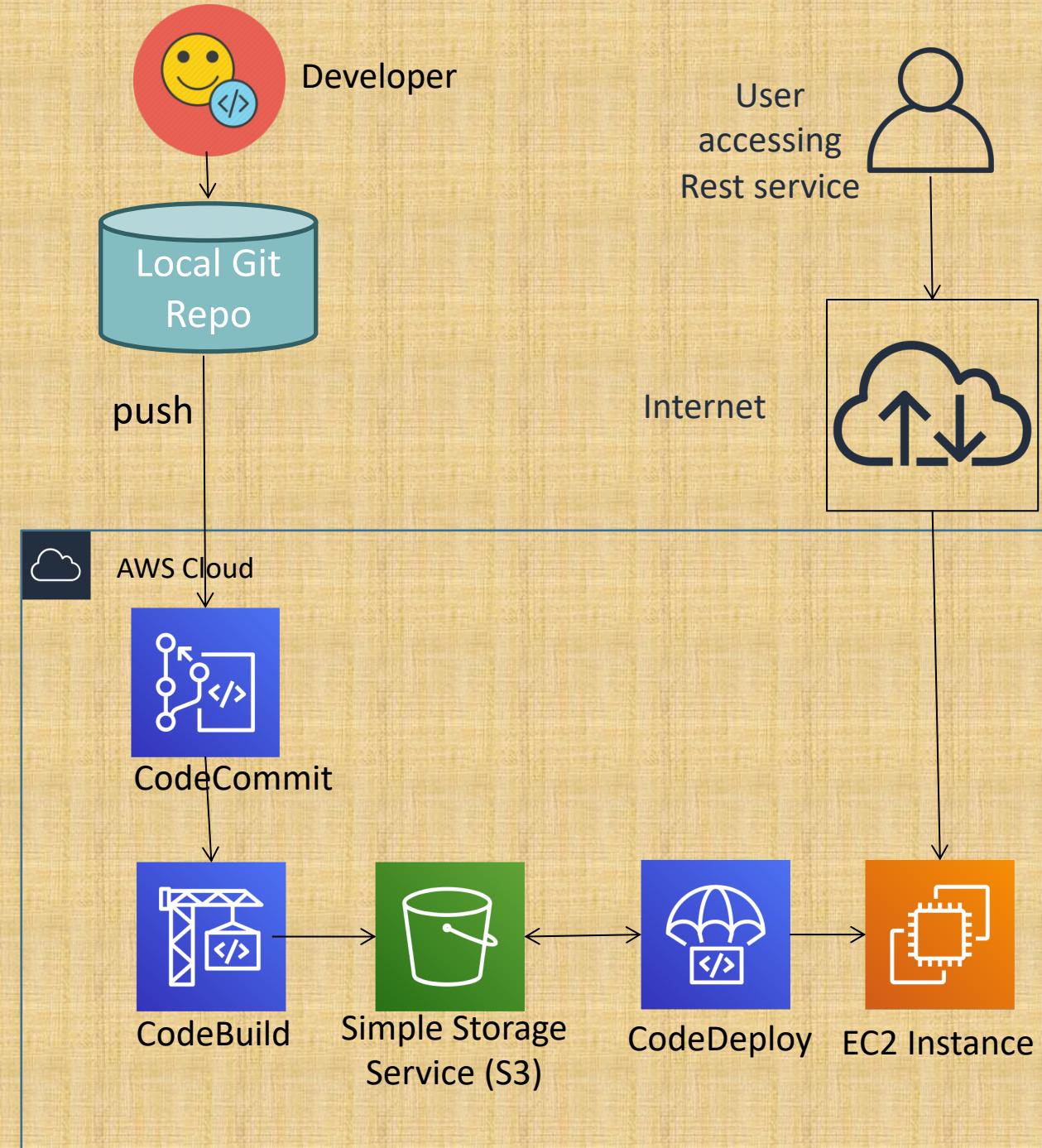
CodeBuild – AWS Web Console



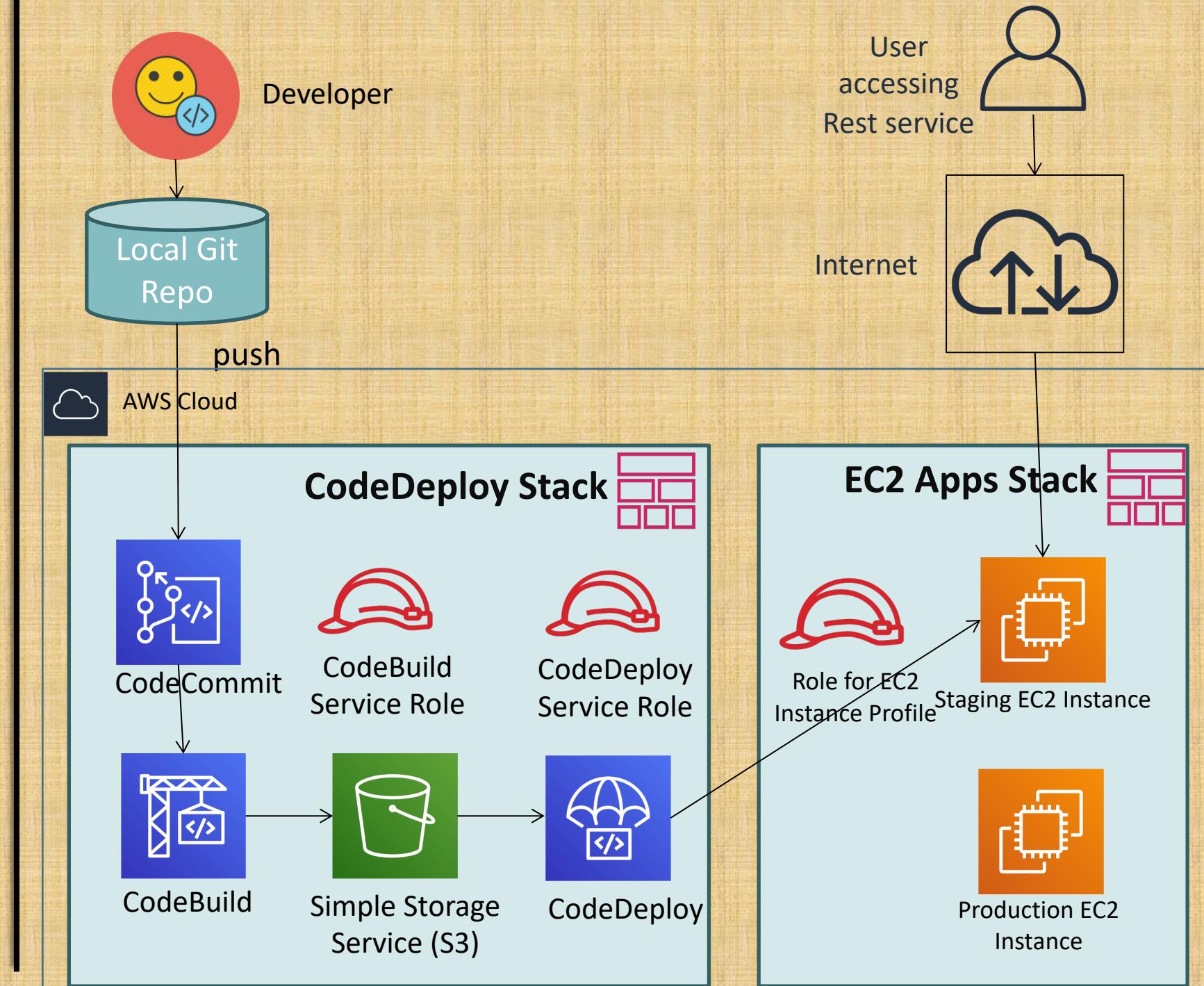
CodeBuild – AWS CloudFormation



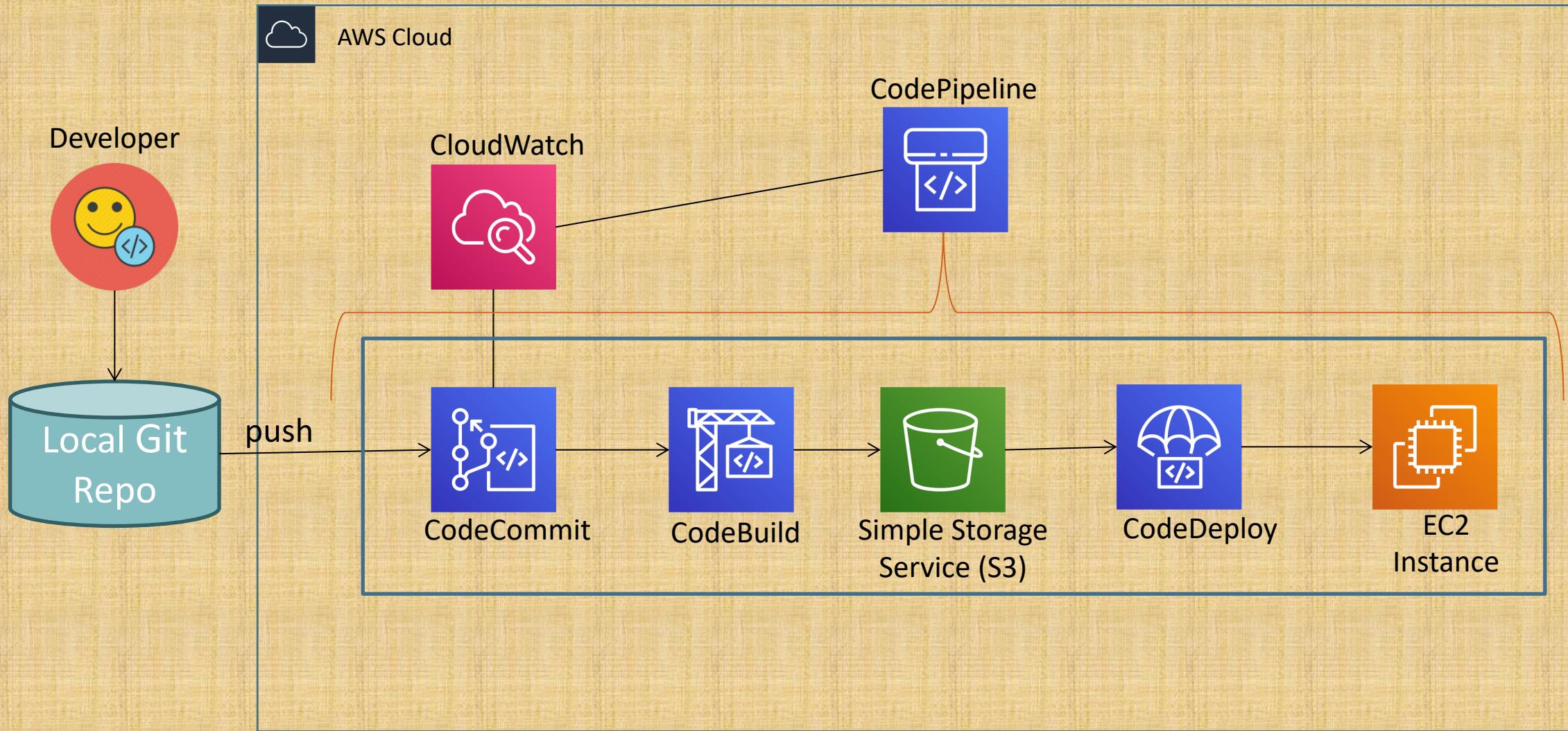
CodeDeploy – AWS Web Console



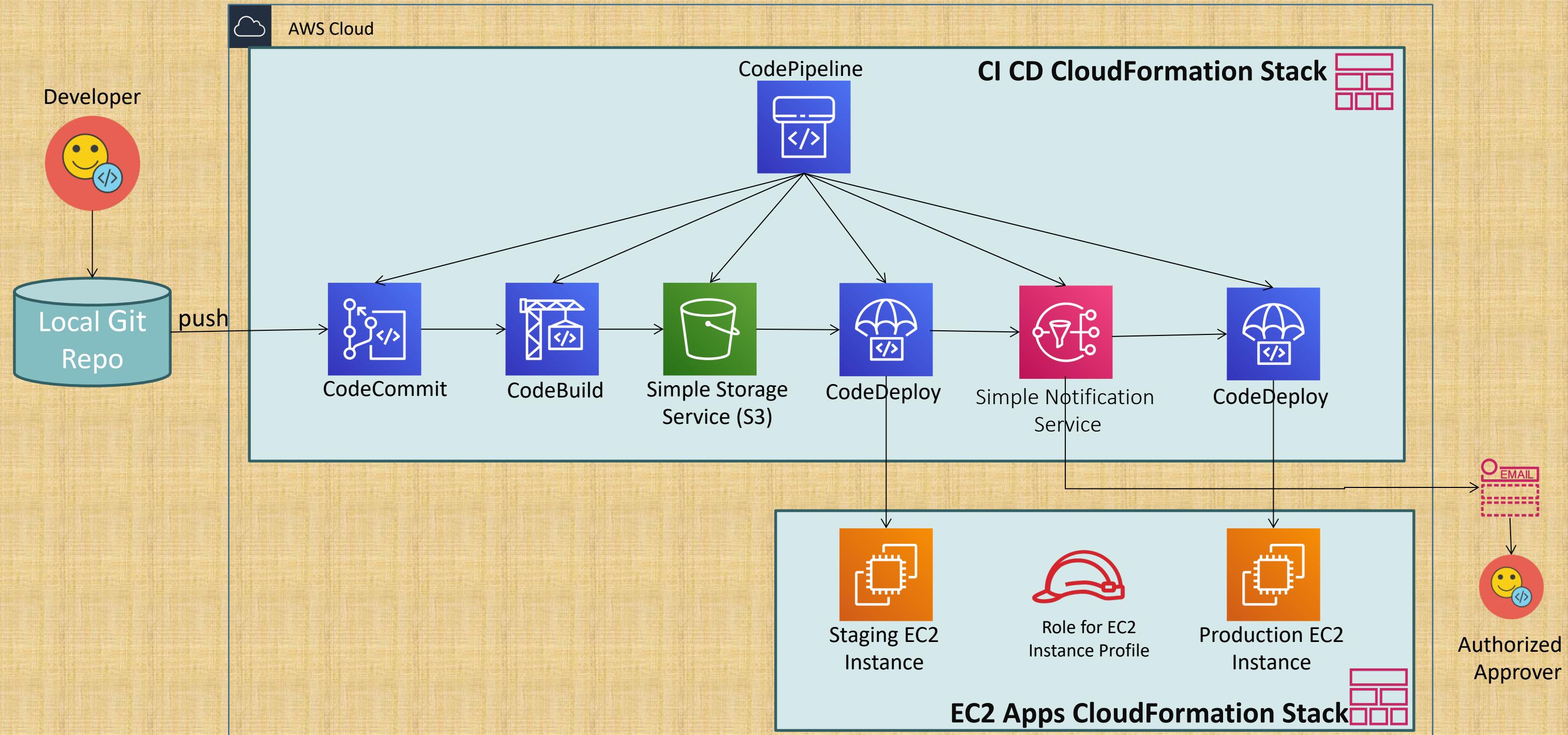
CodeDeploy – AWS CloudFormation



CodePipeline – AWS Web Console



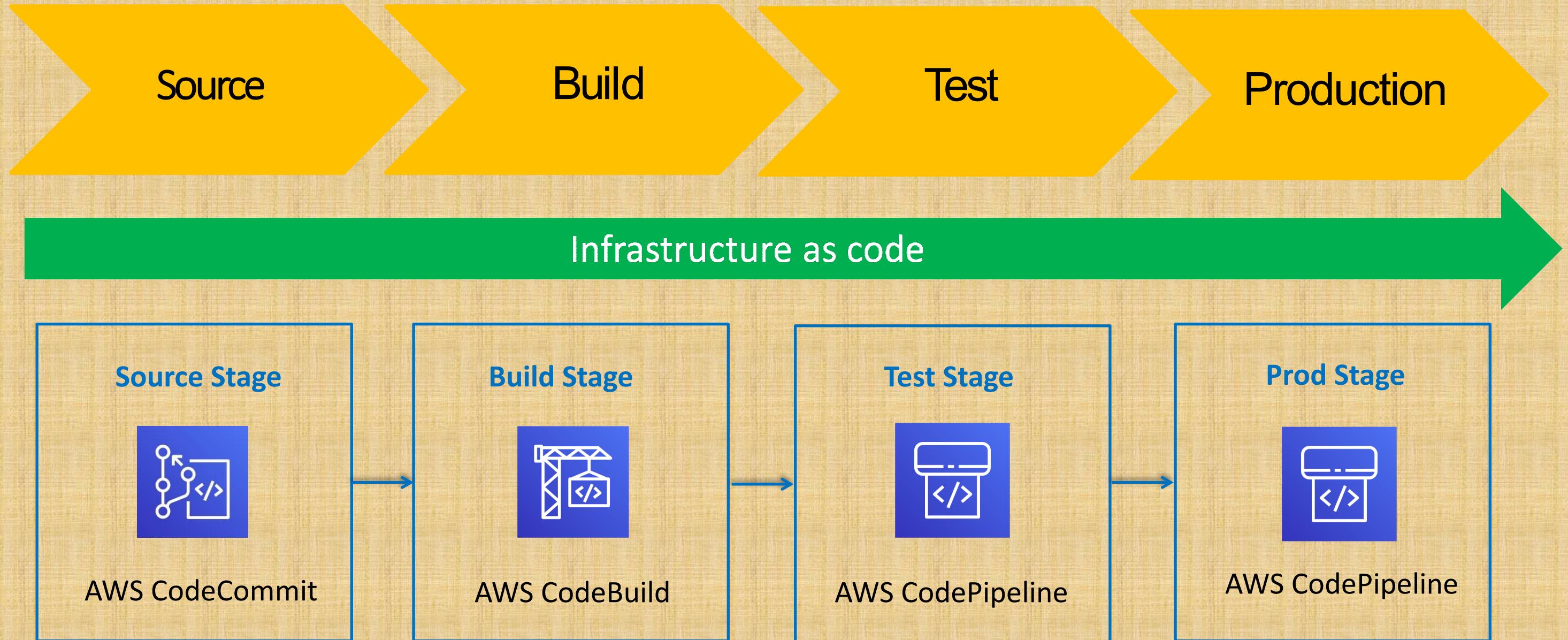
CodePipeline – AWS CloudFormation



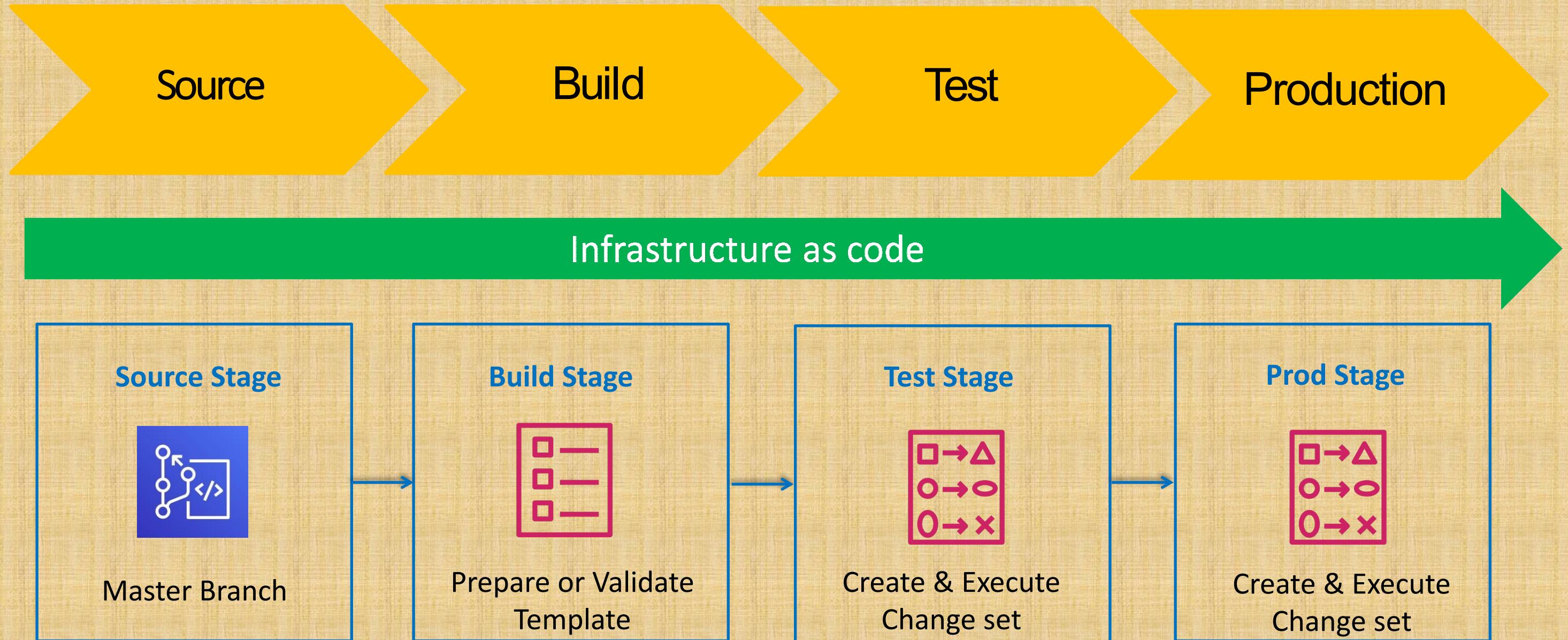
Author: Nho Luong

Skill: DevOps Engineer Lead

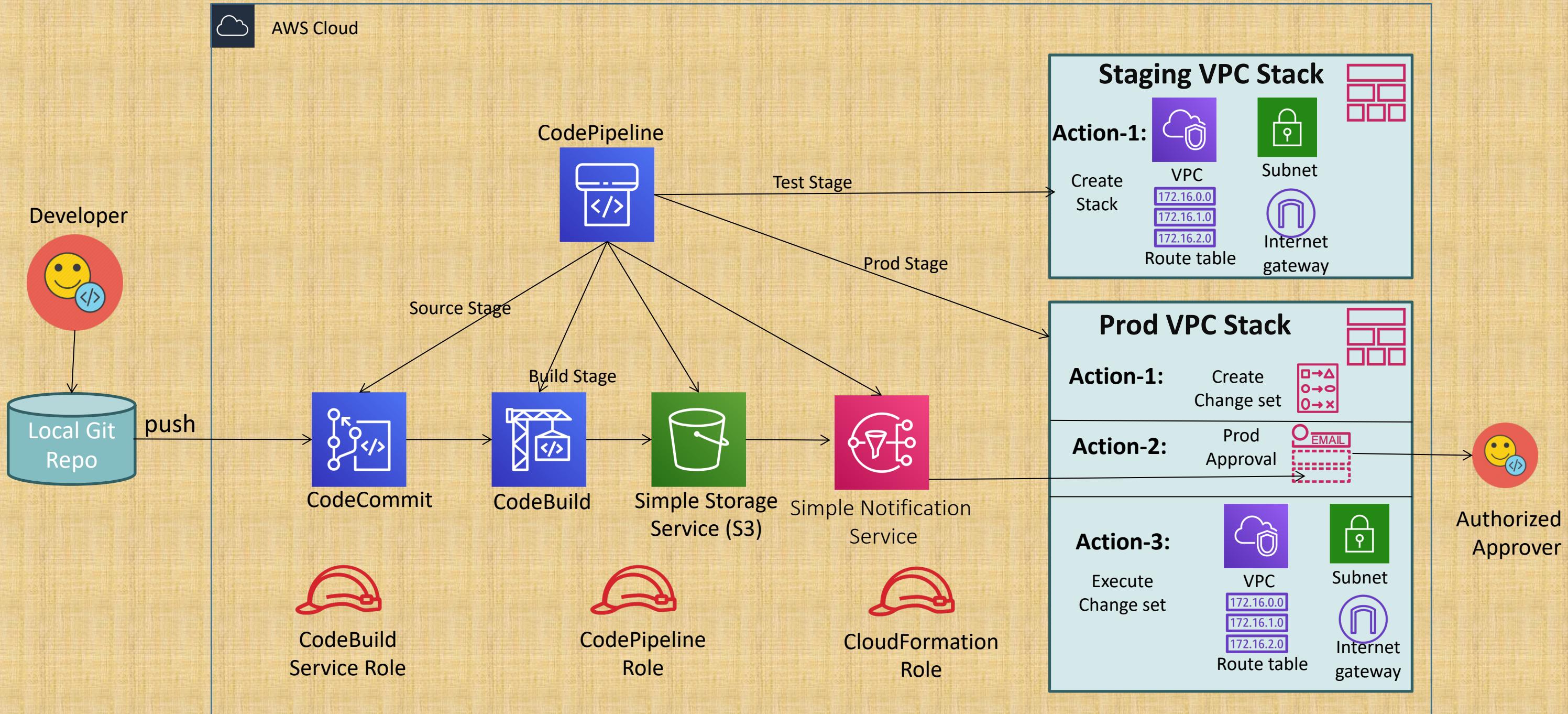
Infrastructure as Code



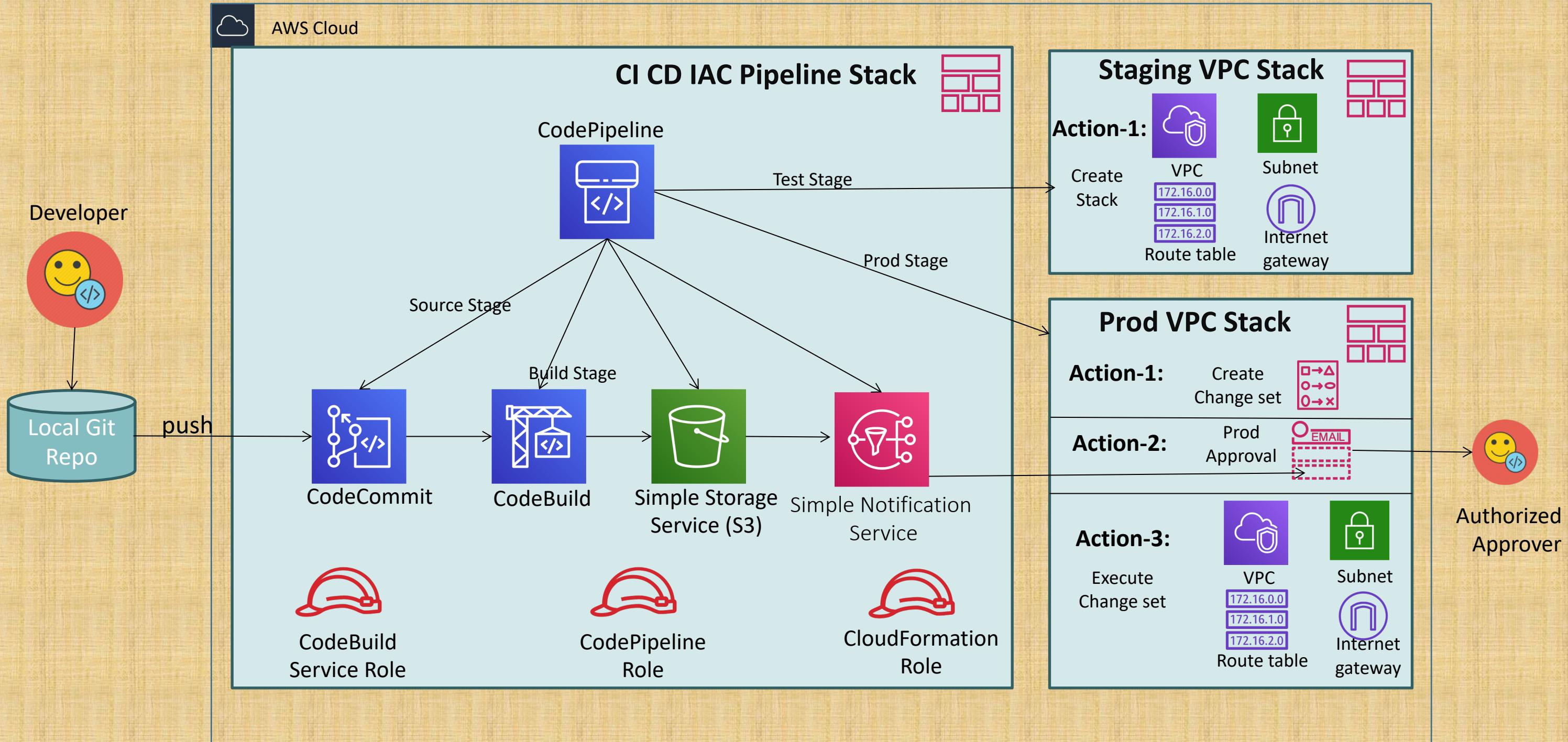
Infrastructure as Code



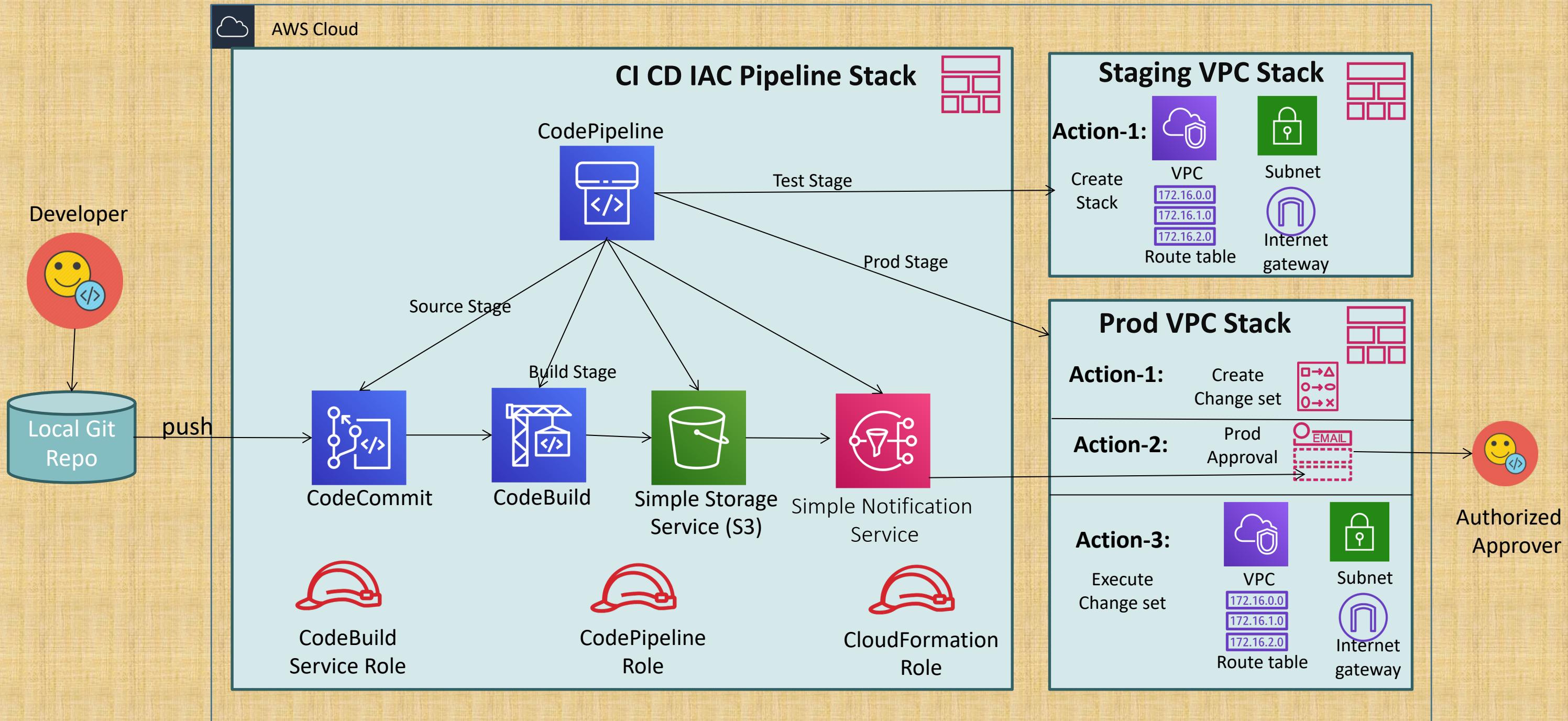
Infrastructure as Code – Manual AWS Web Console



Infrastructure as Code – CFN Template creation Flow



Infrastructure as Code – Execution Flow



Templates Written



Author: Nho Luong

Skill: DevOps Engineer Lead

YAML

Author: Nho Luong

Skill: DevOps Engineer Lead

YAML

- YAML Key Value pairs
- YAML Lists
- YAML Dictionary
- YAML Lists containing Dictionaries
- YAML Lists containing Dictionaries containing Lists
- YAML Pipe
- YAML Greater than Sign
- YAML Comments

YAML – Key Value Pairs

- YAML documents will be full of key value pairs.
- Key and Value are separated by colon.
- We must have a space after colon differentiating the value.
- YAML Supports different data types.
 - Integer
 - Floating point Numbers
 - Strings
 - Boolean
 - Dates - Format: ISO 8601
 - Null values
- **Important Note for Strings:** Quote strings when they have special characters like colons :, braces {}, pipes |, brackets []

Key Value Pairs

```
Name: Nho Luong
Age: 29
Gpa: 4.2
Occupation: Engineer
State: Ho Chi Minh'
AboutMe: "I am a DevOpse engineer Lead"
Male: true
DateOfBith: 1990-09-15T15:53:00
PoliceCases: null
```

YAML – List / Array

- YAML List indented with opening dash.
- Dash indicates that it's a element of an array.
- All members of a list are lines beginning at the same indentation level starting with a “-” (a dash and a space)
- **Block Sequence** indicate each entry with a dash and space
- **Flow Sequence** is written as a comma separated list within square brackets.

List / Array

Block Sequence

Persons:

- nholuong

Flow Sequence

Persons: [nholuong]

YAML Dictionary / Map

- YAML Dictionaries are set of properties grouped together under an item.
- YAML Dictionaries contain key value pairs.

Dictionary

```
Dave:  
Age: 25  
Occupation: Engineer  
State: Ho Chi Minh  
gpa: 4.5  
male: true
```

YAML Pipe

- The pipe notation, also referred to as **literal block**
- All new lines, indentation, extra spaces everything preserved as is.

YAML Greater than Sign

- The greater than sign notation, also referred to as **folded block**.
- Renders the text as a single line.
- All new lines will be replaced with a single space.
- Blank lines are converted to new line character.

YAML Comments

- We can have comments in YAML with
===== > # sign.

Stack Features



Author: Nho Luong

Skill: DevOps Engineer Lead

AWS CloudFormation

- Example Infrastructure Management Demos.
- Quickly replicates our infrastructure.
- Easily controls and tracks changes to our infrastructure.



How does CloudFormation works?



AWS CloudFormation

- **Stack Core Features**

- Create Stack
- Update Stack
- Create Change Set
- Roll back

- **Stack**

- Managing collection of AWS resources as a single unit is called stack.
- We can create, update, delete the collection of AWS resources by creating, updating and deleting stacks.
- To create AWS resources, we create a stack by submitting the template that we created, AWS CloudFormation provisions all those resources automatically for us.



AWS CloudFormation

- **Change Set**

- If we want to make changes to our stack, we can update the stack.
- Before making changes to resources, we can generate a change set, which is summary of proposed changes.
- Change sets allow us to see how our changes might impact current running resources in a stack especially for critical resources, before implementing them we get an idea about the impact.
- **For example:** If we associate a new keypair to ec2 instance, AWS will delete the current ec2 instance and replaces it with new ec2 instance by adding new keypair to it.



Stack Features

- Step 00: Pre-requisites
 - Create Default VPC (if not present)
 - Create Key pairs
 - cfn-key-1
 - cfn-key-2
 - Gather AMI ID
- Step 01: Stack Features
 - Create Stack
 - Update Stack
 - Create Change Sets
 - Rollback



Resources



Author: Nho Luong

Skill: DevOps Engineer Lead

Resources

- Resources are key components of a stack.
- Resources section is a required section that need to be defined in cloud formation template.
- Syntax

Resources:

Logical ID:

Type: *Resource type*

Properties:

Set of properties

Resources:

MyEC2Instance:

Type: "AWS::EC2::Instance"

Properties:

ImageId: "ami-off8a91507f77f867"

- Resources Documentation:

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html>

Resources

- Step 01: Create resource - EC2 Instance
- Step 02: Add Second Resource - New security group and Intrinsic Function Ref
- Step 03: Update Resource Properties - Add new rule to Security group
- Step 04: Add third Resource - Elastic IP
- Step 05: Perform case sensitive test with resource properties



Intrinsic Function: Ref

- The intrinsic function `Ref` returns the value of the specified parameter or resource.
- **Resource Case:** When we specify a resource logical name, it returns a value that we can typically use to refer to that resource.
- **Parameter Case:** When we specify a parameter logical name, it returns the value of that parameter.
- **Syntax:**
 - Long Form
 - `Ref: logicalName`
 - Short Form
 - `!Ref logicalName`

```
MyEIP:  
  Type: "AWS::EC2::EIP"  
  Properties:  
    InstanceId: !Ref MyEC2Instance
```

Parameters



Parameters

- **Parameters:** Parameters enable us to input custom values to our template each time when we create or update stack.
- We can have maximum of **60 parameters** in a cfn template.
- Each parameter must be given a **logical name (logical id)** which must be alphanumeric and unique among all logical names within the template.
- Each parameter must be assigned a **parameter type** that is supported by AWS CloudFormation.
- Each parameter must be assigned a **value at runtime** for AWS CloudFormation to successfully provision the stack. We can optionally specify a **default value** for AWS CloudFormation to use unless another value is provided.

Parameters

- Parameters must be declared and referenced within the same template.
- We can reference parameters from the Resources and Outputs sections of the template.
- Syntax

```
Parameters:  
  ParameterLogicalID:  
    Type: DataType  
    ParameterProperty: value
```

```
Parameters:  
  InstanceTypeParameter:  
    Type: String  
    Default: t2.micro  
    AllowedValues:  
      - t2.micro  
      - m1.small  
      - m1.large  
    Description: Enter t2.micro, m1.small, or m1.large. Default is t2.micro.
```

Parameter Properties

- AllowedPattern
- AllowedValues
- ConstraintDescription
- Default
- Description
- MaxLength
- MaxValue
- MinLength
- MinValue
- NoEcho

Parameter Types

- Type (Mandatory)
 - String
 - Number
 - List<Number>
 - CommaDelimitedList
 - AWS Specific
 - AWS::EC2::InstanceId
 - AWS::EC2::VPC::Id
 - List<AWS::EC2::Subnet::Id>
- Type (Mandatory)
 - SSM Parameter Type
 - AWS::SSM::Parameter::Name
 - AWS::SSM::Parameter::Value <String>
 - AWS::SSM::Parameter::Value <List<String>>

Parameters - Practice

- Step 01: Create a parameter type of AWS for **KeyName** property of ec2 instance.
- Step 02: Create a parameter type of **string** for **AvailabilityZone** property of ec2 instance.
- Step 03: Create a parameter type of **string** for **InstanceType** property of ec2 instance.
- Step 04: Create a parameter type of **SSM** for **InstanceType** property of ec2 instance.
 - Pre-requisite: Create a SSM Parameter in parameter store.



Mappings



Author: Nho Luong

Skill: DevOps Engineer Lead

Mappings

- Mappings section matches a **key** to a corresponding set of **named values**.
- For example, if we want to set values based on a region, we can create a mapping that uses **region name as a key** and contains the values we want to specify for each region
- We can use **Fn::FindInMap** intrinsic function to retrieve values in map.

```
Mappings:  
  Mapping01:  
    Key01:  
      Name: Value01  
    Key02:  
      Name: Value02  
    Key03:  
      Name: Value03
```

```
Mappings:  
  RegionMap:  
    us-east-1:  
      "HVM64": "ami-off8a91507f77f867"  
    us-west-1:  
      "HVM64": "ami-0bdb828fd58c52235"  
    eu-west-1:  
      "HVM64": "ami-047bb4163c506cd98"  
    ap-southeast-1:  
      "HVM64": "ami-08569b978cc4dfa10"  
    ap-northeast-1:  
      "HVM64": "ami-06cd52961ce9f0d85"
```

Intrinsic Function: FindInMap

- The intrinsic function **FindInMap** returns the value corresponding to keys in a **two-level map** that is declared in **Mappings** section.
- Parameters**
 - Map Name
 - Top Level Key
 - Second Level Key
 - Return Value

```
Mappings:  
RegionMap:  
us-east-1:  
    HVM64: "ami-off8a91507f77f867"  
    HVMG2: "ami-0a584ac55a7631c0c"  
us-west-1:  
    HVM64: "ami-0bdb828fd58c52235"  
    HVMG2: "ami-066ee5fd4a9ef77f1"  
eu-west-1:  
    HVM64: "ami-047bb4163c506cd98"  
    HVMG2: "ami-31c2f645"  
ap-southeast-1:  
    HVM64: "ami-08569b978cc4dfa10"  
    HVMG2: "ami-0be9df32ae9f92309"  
ap-northeast-1:  
    HVM64: "ami-06cd52961ce9f0d85"  
    HVMG2: "ami-053cdd503598e4a9d"  
Resources:  
myEC2Instance:  
    Type: "AWS::EC2::Instance"  
    Properties:  
        ImageId: !FindInMap  
            - RegionMap  
            - !Ref 'AWS::Region'  
            - HVM64  
        InstanceType: m1.small
```

Mappings - Practice

- Step 01: Create a Mapping to select the AMI ID for ec2 instance property – `ImageId` based on region.
 - Top Level Key: Region (us-east-2, us-west-1)
 - Second Level Key: HVM64
- Step 02: Create a Mapping to select the instance type based on environments (dev or prod) for ec2 instance property - `InstanceType`
 - Top Level Key: Environment (dev, prod)
 - Second Level Key: Instance Type



Pseudo Parameters

- Pseudo parameters are parameters that are **predefined** by AWS CloudFormation.
- We **don't need to declare** them in our template.
- We can use them the same way as we use parameters as an argument for **Ref** function.
- Usage:

```
Outputs:  
  MyStacksRegion:  
    Value: !Ref "AWS::Region"
```

- AWS::AccountId
- AWS::NotificationARNs
- AWS::NoValue
- AWS::Partition
- AWS::Region
- AWS::StackId
- AWS::StackName
- AWS::URLSuffix

Conditions



Conditions

- Conditions section contains statements that define the **circumstances under which entities** are created or configured.
- Example: 1 - We can create a condition and then associate it with a resource or output so that AWS CloudFormation only **creates the resource or output if the condition is true**.
- Example: 2 - We can associate the condition with a **property** so that AWS CloudFormation only **sets the property to a specific value** if the condition is true, if the condition is false, AWS CloudFormation sets the property to a different value that we specify.
- We will use conditions, when we want to re-use the template in different contexts like dev and prod environments.
- **Syntax:**

Conditions:
Logical ID:
Intrinsic function

Conditions:
CreateEIPForProd:
Fn::Equals:
- !Ref EnvironmentName
- prod

Conditions

- Conditions are evaluated based on predefined **Pseudo parameters** or **input parameter** values that we specify when we create or update stack.
- Within each condition we can reference the other condition.
- We can associate these conditions in three places.
 - Resources
 - Resource Properties
 - Outputs
- At stack creation or stack update, AWS CloudFormation evaluates all conditions in our template. During stack update, Resources that are now associated with a **false condition are deleted**.
- **Important Note:** During stack update, **we cannot update conditions by themselves**. We can update conditions only when we include changes that add, modify or delete resources.

Conditions - Intrinsic Functions

- We can use the below listed intrinsic functions to define conditions in cloud formation template.
 - Fn::And
 - Fn::Equals
 - Fn::If
 - Fn::Not
 - Fn::Or
- We will be covering all these functions in our practice exercises.

Conditions - Practice

- Step 01: Create an EIP when environment is prod, use intrinsic function `Fn::Equals`
- Step 02: Create a security group for dev environment when condition is met and demonstrate `Pseudo parameter "AWS::NoValue"` for when environment is prod. Use Intrinsic function `Fn::If`
- Step 03: Create a security group for prod env with prod related condition added. Use Intrinsic function `Fn::If`
- Step 04: Demonstrate Intrinsic function `Fn::Not`
- Step 05: Demonstrate Intrinsic function `Fn::Or`
- Step 06: Demonstrate Intrinsic function `Fn::And`



Outputs



Outputs

- Outputs section declares output values that we can
 - Import in to other stacks (to create **cross-stack references**)
 - When using **Nested stacks**, we can see how outputs of a nested stack are used in Root Stack.
 - We can view outputs on the CloudFormation console
- We can declare maximum of **60 outputs** in a cfn template.
- Syntax:

```
Outputs:  
  Logical ID:  
    Description: Information about the value  
    Value: Value to return  
  Export:  
    Name: Value to export
```

```
Outputs:  
  InstanceId:  
    Description: Instance ID  
    Value: !Ref MyVMInstance  
    Export:  
      Name: !Sub "${AWS::StackName}-InstanceId"  
  MyInstAvailabilityZone:  
    Description: Instance availability zone  
    Value: !GetAtt MyVMInstance.AvailabilityZone  
    Export:  
      Name: !Sub "${AWS::StackName}-InstanceAz"
```

Outputs

- Export (Optional)

- Exports contain **resource output** used for cross-stack reference.
- For each AWS account, **Export** name must be **unique** with in the region. As it should be unique we can use the export name as “**AWS::StackName”-ExportName**
- We can't create cross-stack references **across regions**.
- We can use the intrinsic function **Fn::ImportValue** to import values that have been exported within the same region. We will see this practically.
 - In simple terms, export availability zone in stack1 and use it stack2
- For outputs, the value of the **Name** property of an **Export** can't use **Ref** or **GetAtt** functions that depend on a resource.
- We can't **delete a stack** if another stack references one of its outputs.
- We can't **modify or remove an output** value that is referenced by another stack.
- We can use Outputs in **combination with Conditions**. We will see that in our practice sessions for Outputs.

Outputs Practice

- Step 01: Create a very basic output using intrinsic function `Fn::Ref` - `Instanceld`.
- Step 02: We will use `Fn::GetAtt` intrinsic function to create outputs.
- Step 03: We will use `Fn::Sub` intrinsic function to create outputs and we will use Pseudo Parameter `AWS::StackName`. In addition, we will **export** the Security Group and Availability Zone.
- Step 04: We will create a new stack by referencing the Security Group and Availability Zone export value from previous stack. We will use `Fn::ImportValue` intrinsic function to import those exports.
- Step 05: We will use `Conditions` in Outputs section to demonstrate their combination.
- Step 06: We will demonstrate `Fn::Join` intrinsic function.



Metadata



Author: Nho Luong

Skill: DevOps Engineer Lead

Metadata

- Metadata provides details about the cfn template.
- Syntax:

```
Metadata:  
  Instances:  
    Description: "Information about the instances"  
  Databases:  
    Description: "Information about the databases"
```

```
1 AWSTemplateFormatVersion: 2010-09-09  
2  
3 Metadata:  
4   Instances:  
5     Description: My VM instance  
6
```

- We have three types of metadata keys which are listed below.
- Metadata Keys
 - AWS::CloudFormation::Designer
 - AWS::CloudFormation::Interface
 - AWS::CloudFormation::Init

Metadata Keys

- AWS::CloudFormation::Designer
 - Auto generated during resources drag and drop to canvas.
- AWS::CloudFormation::Interface
 - Used for parameter grouping.
- AWS::CloudFormation::Init
 - Used for application installation and configurations on our aws compute (EC2 instances).
 - This is core and important feature of CloudFormation.
 - We have one complete section outlining the end to end details of init.

Metadata

AWS::CloudFormation::Designer



AWS::CloudFormation::Designer

- Designer, **Visually depicts** how our resources are laid out
- Designer **automatically** add this information when we use it to create view and update templates. Its a system generated metadata.
- It defines the information about our resources such as their **size** and **relative position** in template metadata. All layout information is stored in designer metadata.

```
Metadata:  
'AWS::CloudFormation::Designer':  
  6b56eaae-0bb6-4215-aad6-12345EXAMPLE:  
    size:  
      width: 60  
      height: 60  
    position:  
      x: 340  
      'y': 430  
      z: 2  
    parent: 21ccc9b0-29e9-4a86-9cf2-12345EXAMPLE  
    embeds: []  
    ismemberof:  
      - c3eead73-6a76-4532-9268-12345EXAMPLE  
...  
...
```

AWS::CloudFormation::Designer

- In designer we **drag and drop** the resources.
- When we create templates in Designer, **it enforces some basic relationships** between resources to help us create valid template.
- **Example:** We cannot directly add EC2 instance in a VPC, we must add a subnet in a VPC.
- We can also **validate template** directly in designer.
- We can bring our template which we have written manually and validate in designer using **validate template**.

AWS::CloudFormation::Designer

- **Integrated Editor:**

- We can make all our **template modifications** with this editor.
- It also provides the **auto-complete feature that lists all property names for a resource** so we don't need to memorize all the properties of a resource or refer documentation.
- We can use integrated editor to convert from JSON to YAML and vice versa.

1 Toolbar

2 Resource Type Pane

3 Canvas Pane

4 Fit to Window Pane

5 Full screen and split screen button

6 Integrated JSON and YAML Editor Pane

7 Messages Pane

File: 'template1'

Resource types

- ApplicationAutoScaling
- ApiGateway
- AutoScaling
- CertificateManager
- CloudFormation
- CloudFront
- CloudTrail
- CloudWatch
- CodeBuild
- CodeCommit
- CodeDeploy
- CodePipeline
- Config
- DataPipeline

```
graph TD; LB[LoadBalancer] --> IS[InstanceS]; LB --> WS[WebServer]; ELB[ElasticLoadBalancer] --> LC[LaunchConfig]; IS --> LC
```

template1

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Description: >-
3 AWS CloudFormation Sample Template VPC_AutoScaling_and_ElasticLoadBalancer:
4 Create a load balanced, Auto Scaled sample website in an existing Virtual
5 Private Cloud (VPC). This example creates an Auto Scaling group behind a load
```

Choose template language: **JSON** **YAML** [?](#)

✓ 4/4/2017, 8:10:17 PM - Successful
converted the template to YAML.

Author: Nho Luong

Skill: DevOps Engineer Lead

How I use Designer?

- Firstly, I write my cfn templates manually in editor by referring documentation
 - Which gives me **greater confidence** on that particular resource for which I am writing template.
 - I use **visual studio code** as my editor due to the fact that dealing with YAML spaces is simplified in this editor. I just use tabs and VS code editor takes care of yaml spaces.
- Copy template to **Integrated Editor** and **Validate Template**.
- **Convert** template from JSON to YAML or YAML to JSON.
- Drag resources to canvas and see their properties (some times).
- Copy template to **Integrated Editor** and review template visually on canvas.

CloudFormation Designer - Demo

Author: Nho Luong

Skill: DevOps Engineer Lead

Metadata

AWS::CloudFormation::Interface



AWS::CloudFormation::Interface

- When we create or update stacks in the console, the console lists **input parameters** in alphabetical order by their logical IDs.
- By using this key, we can define our own **parameter grouping and ordering** so that users can efficiently specify parameter values.
- We can also define **labels** for parameters.
- A label is a friendly name or description that the console displays **instead of a parameter's logical ID** which helps users understand the values to specify for each parameter.

Syntax:

```
Metadata:  
AWS::CloudFormation::Interface:  
  ParameterGroups:  
    - ParameterGroup  
  ParameterLabels:  
    ParameterLabel
```

```
Metadata:  
AWS::CloudFormation::Interface:  
  ParameterGroups:  
    - Label:  
      default: "EC2 Instance Configuration"  
    Parameters:  
      - InstanceType  
      - KeyName  
    - Label:  
      default: "Environment Configuration"  
    Parameters:  
      - EnvironmentName  
  ParameterLabels:  
    EnvironmentName:  
      default: "Which environment we are planning to create?"
```

EC2 UserData



Author: Nho Luong

Skill: DevOps Engineer Lead

CloudFormation & UserData

- We can use **UserData** in CloudFormation template for ec2.
- We need to use a intrinsic function **Fn::Base64** with UserData in CFN templates. This function returns the Base64 representation of input string. It passes **encoded data** to ec2 Instance.
- **YAML Pipe (|)**: Any indented text that follows should be interpreted as a **multi-line scalar value** which means value should be interpreted literally in such a way that **preserves newlines**.
- **UserData Cons**
- By default, user data scripts and cloud-init directives run only during the boot cycle when we **first launch** an instance.
- We can update our configuration to ensure that our user data scripts and cloud-init directives run every time we restart our instance. (**Reboot of server required**)

Sample:

```
UserData:  
Fn::Base64: |  
#!/bin/bash  
sudo yum update  
sudo yum -y erase java-1.7.0-openjdk.x86_64  
sudo yum -y install java-1.8.0-openjdk.x86_64  
sudo yum -y install java-1.8.0-openjdk-devel  
sudo yum -y install tomcat8  
service tomcat8 start  
mkdir /usr/share/tomcat8/webapps/ROOT  
touch /usr/share/tomcat8/webapps/ROOT/index.html  
echo "Cloud Formation Tomcat8" > /usr/share/tomcat8/webapps/
```

Helper Scripts

cfn-init, cfn-hup and cfn-signal



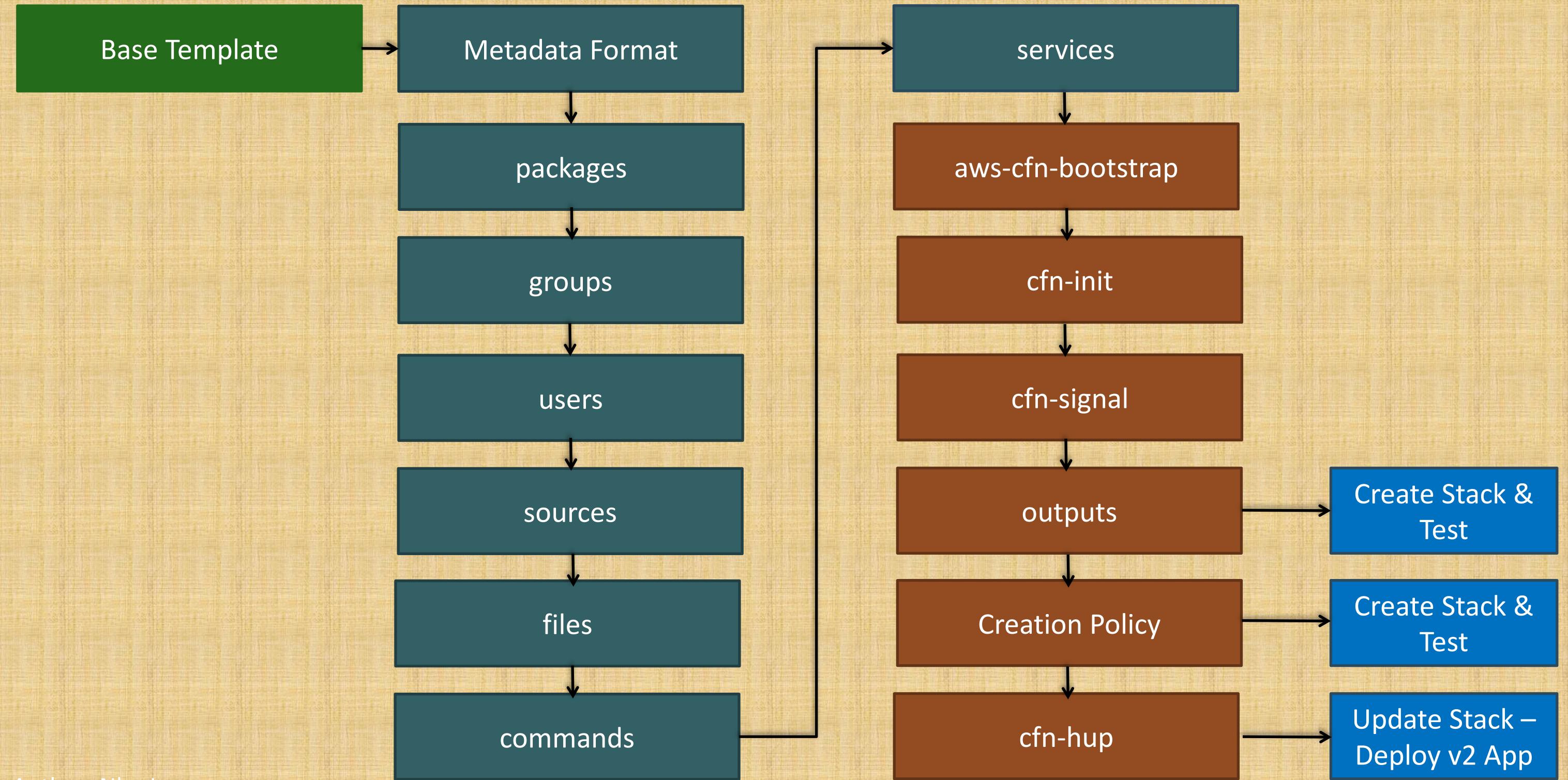
Helper Scripts

- AWS CloudFormation provides the following Python helper scripts that we can use to install software and start services on Amazon EC2 that we create as part of stack.
 - cfn-init
 - cfn-signal
 - cfn-get-metadata
 - cfn-hup

Metadata

AWS::CloudFormation::Init





Author: Nho Luong

Skill: DevOps Engineer Lead

Step 00 – Base Template

- **Resources**

- Security Group
- VM Instances

- **Parameters**

- We will Parameterize **KeyName** parameter

```
6 Parameters:
5   KeyName:
7     Type: AWS::EC2::KeyPair::KeyName
8
9 Resources:
10  MySecurityGroup:
11    Type: AWS::EC2::SecurityGroup
12    Properties:
13      GroupDescription: My SG with port 22, 8080 and 80 inbound
14      SecurityGroupIngress:
15        - IpProtocol: tcp
16          FromPort: '22'
17          ToPort: '22'
18          CidrIp: 0.0.0.0/0
19        - IpProtocol: tcp
20          FromPort: '8080'
21          ToPort: '8080'
22          CidrIp: 0.0.0.0/0
23
24 MyVMInstance:
25   Type: AWS::EC2::Instance
26   Properties:
27     Properties:
28       ImageId: ami-0cd3dfa4e37921605
29       InstanceType: t2.micro
30       KeyName: !Ref KeyName
31       SecurityGroups:
32         - !Ref MySecurityGroup
```

Step-01: Metadata: AWS::CloudFormation::Init

- Type **AWS::CloudFormation::Init** will be used to include metadata section on an ec2 instance for cfn-init helper script.
- Configuration is separated in to **sections**.
- Metadata is organized in to **config keys**, which we can even group in **configsets**.
- By default **cfn-init** calls and **processes the metadata section** when it has single config key (No configsets defined).
- We can even specify **configsets as input to cfn-init script** so that it can process the entire configset with all its configkeys. We will see it in detail in configsets section.
- The cfn-init helper script processes the configuration sections **in the order specified in syntax section**.

```
Resources:  
MyVMInstance:  
  Type: AWS::EC2::Instance  
  Metadata:  
    AWS::CloudFormation::Init:  
      config:  
        packages:  
          :  
        groups:  
          :  
        users:  
          :  
        sources:  
          :  
        files:  
          :  
        commands:  
          :  
        services:  
          :  
    Properties:  
      ImageId:  
      InstanceType:  
      KeyName:
```

Step-01: Metadata: Structure

- If we want to process it in different order, we need to separate them into **different config keys** and then use the **order of execution** for config keys in a configset.
- In this step we will just add the **metadata section** with structure.
- We will **incrementally build** the metadata sections in upcoming steps.
- **Metadata Structure:**

```
Metadata:  
Comment: Deploy a simple tomcat Application  
AWS::CloudFormation::Init:  
  config:  
    packages:  
    groups:  
    users:  
    sources:  
    files:  
    commands:  
    services:
```

```
Resources:  
  MyVMInstance:  
    Type: AWS::EC2::Instance  
    Metadata:  
      AWS::CloudFormation::Init:  
        configSets:  
          InstallAndConfigure:  
            - Install  
            - Configure  
        Install:  
          packages:  
          :  
          groups:  
          :  
          users:  
          :  
          sources:  
          :  
          files:  
          :  
          services:  
          :  
        Configure:  
          commands:  
          :  
        Properties:  
          ImageId:  
          InstanceType:  
          KeyName:
```

Step-02: Metadata: packages

- We can use packages key to **download and install** pre-packaged applications.
- On windows systems packages key supports only the **MSI Installer**.
- Supported Package Formats:
 - apt
 - msi
 - python
 - rpm
 - rubygems
 - yum

- Packages with Versions:

```
packages:  
  rpm:  
    epel: "http://download.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rpm"  
  yum:  
    httpd: []  
    php: []  
    wordpress: []  
  rubygems:  
  chef:  
    - "0.10.2"
```

- Our Example:

```
Metadata:  
  Comment: Deploy a simple tomcat Application  
  AWS::CloudFormation::Init:  
    config:  
      packages:  
        yum:  
          java-1.8.0-openjdk.x86_64: []  
          java-1.8.0-openjdk-devel: []  
          tomcat8: []
```

Step-03: Metadata: groups

- We can use groups to create Linux/Unix groups and assign to group id's.
- Groups key is **not supported** for windows systems.
- We can create multiple groups as required.
- We can create **without** group id or create **with a desired group id**.
- Syntax:

```
groups:  
  groupone: {}  
  grouptwo:  
    gid: "501"
```

Step-04: Metadata: users

- We can use the users key to create Linux/Unix users in EC2 Instance.
- Users key is not supported for windows systems.
- The following are the supported keys
 - uid
 - groups
 - homeDir
- Users are created as **non-interactive system users** with a shell of **/sbin/nologin**.
- This is by design and cannot be modified

- Syntax

```
users:  
  user1:  
    groups:  
      - groupone  
      - grouponetwo  
    uid: "501"  
    homeDir: "/tmp"
```

Step-05: Metadata: sources

- We can use the sources key to download an archive file and unpack it in a target directory on EC2 Instance.
- This key is fully supported for both Linux and Windows systems.
- Supported Archive formats
 - tar
 - tar + gzip
 - tar + bz2
 - zip
- Syntax / Example:

```
sources:  
  /tmp: "https://s3.us-east-2.amazonaws.com/codedeploydemobucket0189/cfn/demo2.zip"
```

Step-05: Metadata: sources

- Create S3 bucket
- Disable [block public access to bucket.](#)
- Create [cfn](#) folder
- Upload the zip files [demo1.zip](#), [demo2.zip](#) which contains [demo.war](#) (two versions v1 and v2)
 - Unzip AWS-CloudFormation.zip to local directory
 - Navigate to [11-cfn-init/WAR-Files](#) folder
 - Upload the [demo1.zip](#), [demo2.zip](#) to S3 bucket [cfn](#) folder.
 - Path: [/AWS-CloudFormation/11-cfn-init/WAR-files](#)
 - Make the [demo1.zip](#), [demo2.zip](#) as public file.
 - Copy the S3 http url for both files and perform public access test.
 - Update [demo1.zip](#) url in [sources section](#) of template.

Step-06: Metadata: files

- We can use the `files` key to create files on EC2 Instance.
- The content can be either `inline in the template` or the content can be `pulled from a URL`.
- The files are written to disk in `alphabetical order`.
- `Supported Keys`
 - content
 - source
 - Encoding (plain or base64)
 - group
 - owner
 - mode
 - authentication
 - context

Step-06: Metadata: files

Syntax / Sample:

```
files:
  "/etc/cfn/cfn-hup.conf":
    content: !Sub |
      [main]
      stack=${AWS::StackId}
      region=${AWS::Region}
      mode: "000400"
      owner: "root"
      group: "root"
  "/etc/cfn/hooks.d/cfn-auto-reloader.conf":
    content: !Sub |
      [cfn-auto-reloader-hook]
      triggers=post.update
      path=Resources.MyVMInstance.Metadata.AWS::CloudFormation::Init
      action=/opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --resource MyVMInstance --region ${AWS::Region}
      mode: "000400"
      owner: "root"
      group: "root"
.
```

Step-07: Metadata: commands

- We can use **commands** key to execute commands on EC2 Instance.
- The commands are processed in **Syntax / Example:** alphabetical order by name.

Supported Keys

- command
- env
- cwd
- test
- ignoreErrors
- waitAfterCompletion

```
commands:  
  test1:  
    command: "chmod 755 demo.war"  
    cwd: "/tmp"  
  test2:  
    command: "sudo yum -y erase java-1.7.0-openjdk.x86_64"  
    cwd: "~"  
  test3:  
    command: "rm -rf demo*"  
    cwd: "/var/lib/tomcat8/webapps"  
  test4:  
    command: "cp demo.war /var/lib/tomcat8/webapps"  
    cwd: "/tmp"
```

Step-08: Metadata: services

- We can use services key to define which services should be **enabled or disabled** when the instance is launched.
- On Linux systems this key is supported by using [sysvinit](#).
- On Windows systems, it is supported by using [Windows Service Manager](#).
- Services key also allows us to specify [dependencies on sources](#), packages and files so that if a restart is needed due to files being installed, cfn-init will take care of the service restart.
- Supported Keys
 - ensureRunning
 - enabled
 - files
 - sources
 - packages
 - commands

```
services:  
  sysvinit:  
    tomcat8:  
      enabled: "true"  
      ensureRunning: "true"
```

Step-08: Metadata: services

- The nginx service will be **restarted** if either [/etc/nginx/nginx.conf](#) or [/var/www/html](#) are modified by cfn-init.
- The **php-fastcgi** service will be **restarted** if cfn-init installs or updates php or spawn-fcgi using yum.
- The sendmail service will be **stopped** and **disabled**.

```
services:  
  sysvinit:  
    nginx:  
      enabled: "true"  
      ensureRunning: "true"  
      files:  
        - "/etc/nginx/nginx.conf"  
    sources:  
      - "/var/www/html"  
  php-fastcgi:  
    enabled: "true"  
    ensureRunning: "true"  
    packages:  
      yum:  
        - "php"  
        - "spawn-fcgi"  
  sendmail:  
    enabled: "false"  
    ensureRunning: "false"
```

UserData



Author: Nho Luong

Skill: DevOps Engineer Lead

Step-09: UserData: aws-cfn-bootstrap

- Helper Scripts are updated **periodically**.
- We need to ensure that the below listed command is included in **UserData** of our template **before we call the helper scripts** to ensure that our launched instances get the latest helper scripts.

```
UserData:  
  "Fn::Base64":  
    !Sub |  
      #!/bin/bash -xe  
      # Get latest CloudFormation package - These packages change frequently  
      yum update -y aws-cfn-bootstrap
```

Step-10: UserData: cfn-init

- The cfn-init helper script reads template metadata from the AWS::CloudFormation::Init key and acts accordingly to:
 - Fetch and parse metadata from AWS CloudFormation
 - Install packages
 - Write files to disk
 - Enable/disable and start/stop services
- If we use cfn-init to update an existing file, it creates a backup copy of the original file in the same directory with a .bak extension.
- cfn-init does not require credentials. However, if no credentials are specified, AWS CloudFormation checks for stack membership and limits the scope of the call to the stack that the instance belongs to.

Command Syntax:

```
cfn-init --stack|-s stack.name.or.id \
          --resource|-r logical.resource.id \
          --region region \
          --access-key access.key \
          --secret-key secret.key \
          --role rolename \
          --credential-file|-f credential.file \
          --configsets|-c config.sets \
          --url|-u service.url \
          --http-proxy HTTP.proxy \
          --https-proxy HTTPS.proxy \
          --verbose|-v
```

Command Usage in UserData:

```
# Start cfn-init to Install all Metadata content (pacakges, sources, files, commands and services )
/opt/aws/bin/cfn-init -s ${AWS::StackId} -r MyVMInstance --region ${AWS::Region} || error_exit 'Failed to run cfn-init'
```

Step-11: UserData: cfn-signal

- The cfn-signal helper script [signals AWS CloudFormation](#) to indicate whether Amazon EC2 instances have been successfully created or updated.
- If we install and configure software applications on instances, [we can signal AWS CloudFormation](#) when those software applications are ready.
- We can use the [cfn-signal](#) script in conjunction with a [CreationPolicy](#).

Command Syntax:

```
cfn-signal --success|-s signal.to.send \
    --access-key access.key \
    --credential-file|-f credential.file \
    --exit-code|-e exit.code \
    --http-proxy HTTP.proxy \
    --https-proxy HTTPS.proxy \
    --id|-i unique.id \
    --region AWS.region \
    --resource resource.logical.ID \
    --role IAM.role.name \
    --secret-key secret.key \
    --stack stack.name.or.stack.ID \
    --url AWS CloudFormation.endpoint
```

Step-11: UserData: cfn-hup

- **Important Note:** From here on we will start creating the stack using v12 template file, we will add **cfn-hup** command also to template **UserData** section even though we discuss that section in **step 14**. Reason for doing that is UserData related changes should be included during **instance creation time** only.
- Final Look of UserData:

```
UserData:  
  "Fn::Base64":  
    !Sub |  
      #!/bin/bash -xe  
      # Get latest CloudFormation package - These packages change frequently  
      yum update -y aws-cfn-bootstrap  
      # Start cfn-init to Install all Metadata content (pacakges, sources, files, commands and services )  
      /opt/aws/bin/cfn-init -s ${AWS::StackId} -r MyVMInstance --region ${AWS::Region} || error_exit 'Failed to run  
      # Signal the status from cfn-init  
      /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackId} --resource MyVMInstance --region ${AWS::Region}  
      # Start cfn-hup daemon so that it will keep listening to any changes to EC2 Instance Metadata.  
      /opt/aws/bin/cfn-hup || error_exit 'Failed to start cfn-hup'
```

Step 12 - Outputs

- Add outputs in the template.
- We will add **AppURL** output for easily accessing the application after stack creation.
- Sample

```
Outputs:  
AppURL:  
  Description: Application URL to access  
  Value: !Sub 'http://${MyVMInstance.PublicDnsName}:8080/demo/hello'
```

Step 12: Create Stack using template 11-12-cfn-init-v12-Outputs.yml

- **Observations**
 - CloudFormation gets the signal as soon as VM Instance resource gets created.
 - In other words, we will see that stack status “CREATE_COMPLETE” even though in the back ground application installations are going on in the EC2 Instance.
 - With this approach we have problems like
 - Applications installs fails and we see the stack status as “CREATE_COMPLETE” in green.
 - We will not know what happened to our application installs or configurations until we login to instance.
 - To overcome such type of issues, we need to use “[Creation Policy](#)” which we will see in next step (step 13).

Step-13: Creation Policy

- Associate the **CreationPolicy** attribute with a resource to prevent its status from reaching create complete until AWS CloudFormation receives a specified number of **success signals** or the **timeout period** is exceeded.
- To signal a resource we can use **cfn-signal** helper script.
- The creation policy is **invoked** only when AWS CloudFormation **creates** the associated resource.
- Currently, the only AWS CloudFormation resources that support creation policies are
 - AWS::AutoScaling::AutoScalingGroup
 - AWS::EC2::Instance
 - AWS::CloudFormation::WaitCondition

Step-13: Creation Policy

- Use the **CreationPolicy** attribute when you want to **wait on resource configuration actions before stack creation proceeds**.
- For example, if we **install and configure software applications** on an EC2 instance, we might want those applications to be running before proceeding. In such cases, we can add a **CreationPolicy** attribute to the instance, and then **send a success signal** to the instance after the applications are installed and configured.

- **Syntax:**

```
CreationPolicy:  
  AutoScalingCreationPolicy:  
    MinSuccessfulInstancesPercent: Integer  
  ResourceSignal:  
    Count: Integer  
    Timeout: String
```

```
MyVMInstance:  
  Type: AWS::EC2::Instance  
  CreationPolicy:  
    ResourceSignal:  
      Timeout: PT5M
```

Step 13: Create Stack using template 11-13-cfn-init-v13-CreationPolicy.yml

- Observations

- CloudFormation waits for the status until application installs are completed for that particular resource (in our case its VM Instance).
- Either it waits for success signal and if within specified time (time specified in creation policy) if it didn't get success signal it roll backs the entire stack.

Timestamp	Logical ID	Status	Status reason
18 May 2019 14:28:34	CreationPolicyStack	✓ CREATE_COMPLETE	-
18 May 2019 14:28:32	MyVMInstance	✓ CREATE_COMPLETE	-
18 May 2019 14:28:31	MyVMInstance	⌚ CREATE_IN_PROGRESS	Received SUCCESS signal with UniqueId i-0ceb88feabca7e113
18 May 2019 14:27:18	MyVMInstance	⌚ CREATE_IN_PROGRESS	Resource creation Initiated

Step-14: UserData: cfn-hup

- cfn-hup helper is a **daemon** that detects changes in resource metadata and **runs user-specified actions** when a change is detected.
- This allows us to make **configuration updates on our running EC2 Instance** through the Update Stack feature.
- **cfn-hup.conf**
 - cfn-hup.conf file stores the **name of the stack** and the AWS credentials that the cfn-hup daemon targets.
 - **Format** of cfn-hup.conf
 - We are creating this file using our Metadata Key named **files** in our template.

- Format of cfn-hup.conf

```
[main]
stack=${AWS::StackId}
region=${AWS::Region}
```

```
files:
"/etc/cfn/cfn-hup.conf":
  content: !Sub |
    [main]
    stack=${AWS::StackId}
    region=${AWS::Region}
    interval=3
    mode: "000400"
    owner: "root"
    group: "root"
```

Step-14: UserData: cfn-hup

- **cfn-hup.conf file content**
 - stack
 - credential-file
 - role
 - region
 - umask (default: 022)
 - Interval (default: 15)
 - Verbose
- **hooks.d Directory**
 - To support composition of several applications deploying change notification hooks, cfn-hup supports a directory named hooks.d that is located in the hooks configuration directory.
 - We can place one or more additional hooks configuration files in the hooks.d directory.

Step-14: UserData: cfn-hup - hooks.conf

- User actions that cfn-hup daemon calls periodically are defined in hooks.conf.
- Syntax:

```
[hookname]
triggers=post.add or post.update or post.remove
path=Resources.<logicalResourceId> (.Metadata or .PhysicalResourceId)
(.<optionalMetadatapath>)
action=<arbitrary shell command>
runas=<runas user>
```

```
"/etc/cfn/hooks.d/cfn-auto-reloader.conf":
content: !Sub |
[cfn-auto-reloader-hook]
triggers=post.update
path=Resources.MyVMInstance.Metadata.AWS::CloudFormation::Init
action=/opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --resource MyVMInstance --region ${AWS::Region}
mode: "000400"
owner: "root"
group: "root"
```

Step-14: UserData: cfn-hup - hooks.conf

- When the **action is run**, it is run in a copy of the current environment (that cfn-hup is in), with **CFN_OLD_METADATA** set to the previous value of path, and **CFN_NEW_METADATA** set to the current value.
- The hooks configuration file is loaded at cfn-hup daemon **startup** only, so **new hooks** will require the daemon to be **restarted**.
- A **cache of previous metadata** values is stored at **/var/lib/cfn-hup/data/metadata_db**
- We can delete this cache to **force cfn-hup** to run all **post.add** actions again.

Step 14: Create Stack using template 11-14-cfn-init-v14-Update-App.yml

- **Observations**

- Old war file will be removed
- New war file will be deployed successfully.
- When we access the app new version of application content will be displayed.

Configsets



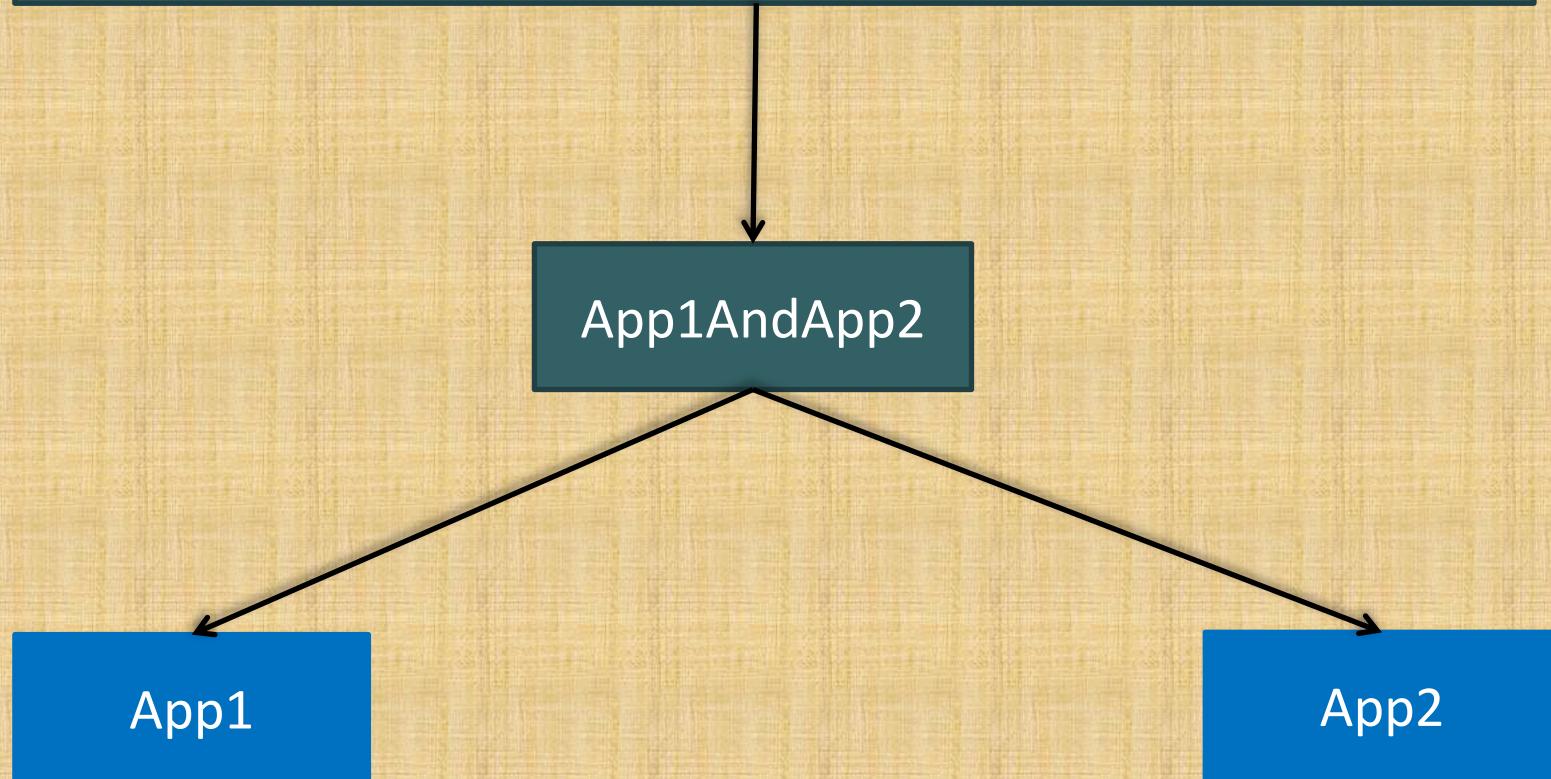
Author: Nho Luong

Skill: DevOps Engineer Lead

Configsets

- We can create more than one config key and have cfn-init process them in a specific order.
 - Single Configset
 - Multiple Configset

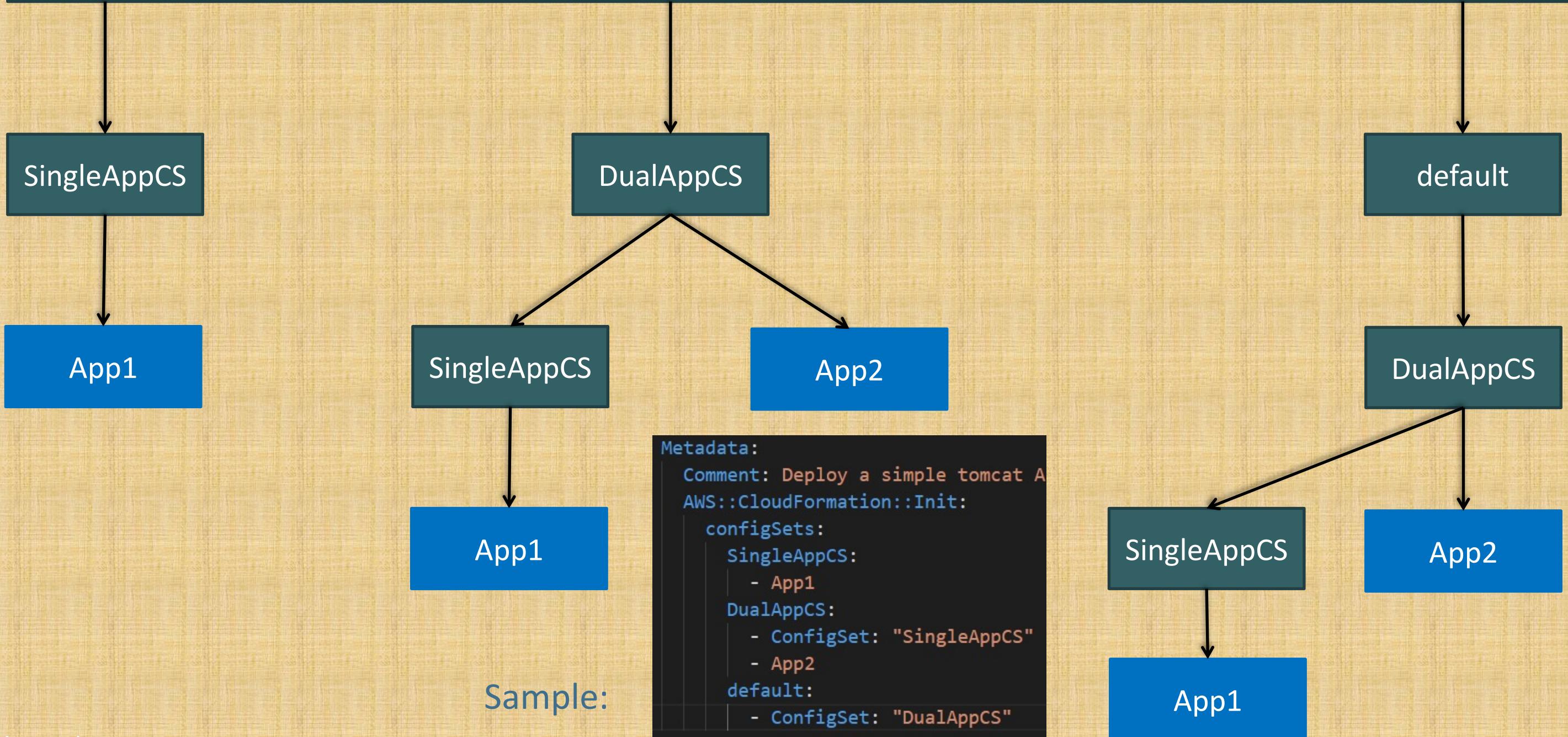
configSets (Single)



Sample:

```
Metadata:  
Comment: Deploy a simple tomcat Application  
AWS::CloudFormation::Init:  
configSets:  
App1AndApp2:  
- App1  
- App2
```

configSets (Multiple)



Step#1: Single Configset

- We will define two config keys App1 and App2
- We will create a configSet with name as App1AndApp2.
- First App1 config key will get executed.
- Next App2 config key will get executed.
- Order of execution will be based on how we define them in configSets.
- Observation
 - Both applications should be accessible

Sample:

```
Metadata:  
Comment: Deploy a simple tomcat Application  
AWS::CloudFormation::Init:  
  configSets:  
    App1AndApp2:  
      - App1  
      - App2  
    App1:  
      packages:  
        ...
```

Step#2: Multiple configSets

- We have created 3 configSets
 - SingleAppCS
 - DualAppCS
 - default
- SingleAppCS: Only App1 should be deployed.

Sample:

```
Metadata:  
Comment: Deploy a simple tomcat Application  
AWS::CloudFormation::Init:  
  configSets:  
    SingleAppCS:  
      - App1  
    DualAppCS:  
      - ConfigSet: "SingleAppCS"  
      - App2  
    default:  
      - ConfigSet: "DualAppCS"
```

Step#3: Multiple configSets

- We have created 3 configSets
 - SingleAppCS
 - DualAppCS
 - default
- DualAppCS: Both App1 and App2 should be deployed

Sample:

Metadata:

```
Comment: Deploy a simple tomcat Application
AWS::CloudFormation::Init:
  configSets:
    SingleAppCS:
      - App1
    DualAppCS:
      - ConfigSet: "SingleAppCS"
      - App2
    default:
      - ConfigSet: "DualAppCS"
```

Step#4: Multiple configSets

- We have created 3 configSets
 - SingleAppCS
 - DualAppCS
 - default
- default: default contains ConfigSet DualAppCS so both apps should be deployed. For default we don't need to specify “--configSets default” it will pick automatically.

Sample:

```
Metadata:  
Comment: Deploy a simple tomcat Application  
AWS::CloudFormation::Init:  
  configSets:  
    SingleAppCS:  
      - App1  
    DualAppCS:  
      - ConfigSet: "SingleAppCS"  
      - App2  
    default:  
      - ConfigSet: "DualAppCS"
```

Nested Stacks



Author: Nho Luong

Skill: DevOps Engineer Lead

Nested Stacks

- The `AWS::CloudFormation::Stack` type nests a stack as a resource in a top-level template.
- We can add output values from a nested stack **within** the root stack.
- We use `Fn::GetAtt` function with nested stacks logical name and the name of the output value in nested stack
- Syntax:

```
VpcId: !GetAtt NestedStackName.Outputs.NestedStackOutputName
```

```
        "NetworkInterfaces":  
        - AssociatePublicIpAddress: "true"  
          DeviceIndex: "0"  
          SubnetId: !GetAtt VPCStack.Outputs.Subnet01Id  
        GroupSet:  
        - !GetAtt SecurityGroupStack.Outputs.DevSGGroupId
```

Nested Stacks – Practice – Create Templates

Step#0:



S3 Bucket

- Create S3 bucket
- This is required for uploading the Nested stack templates to S3

Step#1:



VPC Nested Stack Template

- Create Parameters
- Create Metadata
- Create Resources
 - Create VPC
 - Create Subnets
 - Create Route Table
 - Associate Subnet & Route Table
 - Create IGW
 - Associate IGW to VPC
 - Create Route
 - Create Outputs
 - Test Template
 - Upload to S3

Step#2:



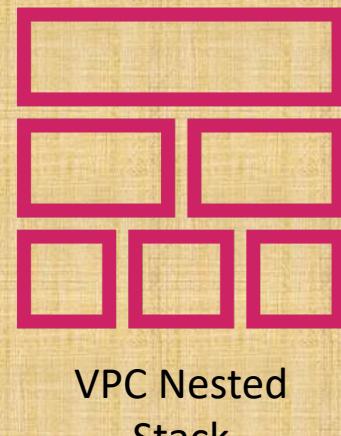
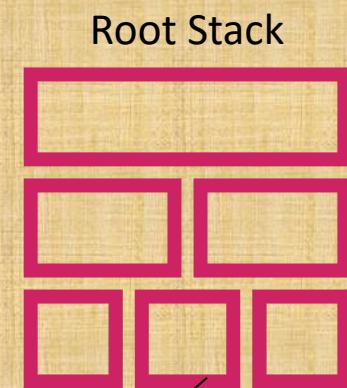
Root Stack Template

- Create Parameters
- Create Resources
 - Create VPC Stack
 - Create EC2 Instance
- Create Outputs

Nested Stacks – Practice – Create Stack

Step#3:

- For VPC Nested Stack
- Create Root Stack
- It automatically creates the vpc nested stack



Nested Stacks – Practice – Create Templates

Step#4



Security Group Nested Stack Template

- Create Parameters
- Create Resources
 - Create Security Group
- Create Outputs

Step#5



Root Stack Template

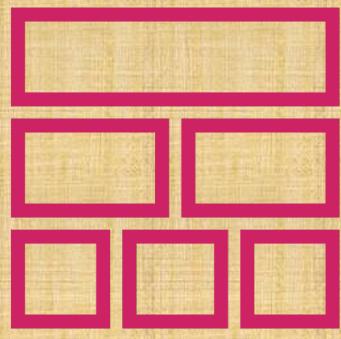
- Create Parameters
- Create Resources
 - Create VPC Stack
 - Create EC2 Instance
- Create Outputs
- Create Resource
 - Create Security Group Stack
 - Update VM Instance resource with security group

Nested Stacks – Practice – Update Stack

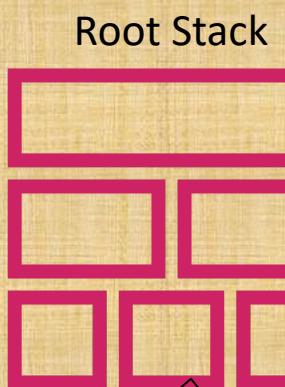
Step#6:

For VPC Nested Stack

- Create Root Stack
- It automatically creates the vpc nested stack

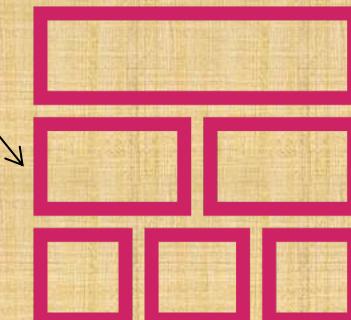


VPC Nested Stack



For Security Group Nested Stack

- Update Root Stack with new template.
- It automatically creates the security group nested stack

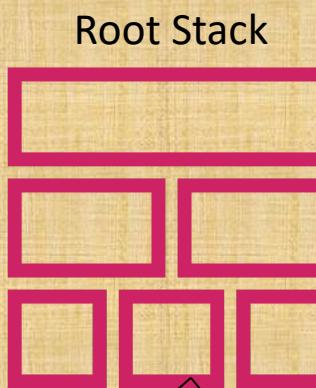
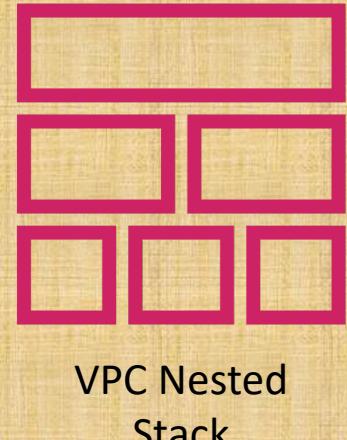


Security Group Nested Stack

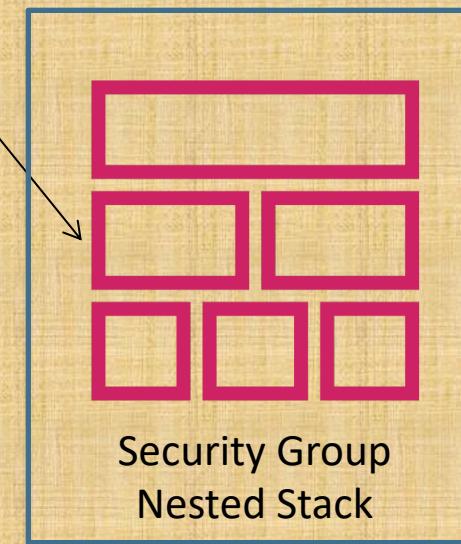
Nested Stacks – Practice – Update Stack 2

Step#7:

- For VPC Nested Stack
 - Create Root Stack
 - It automatically creates the vpc nested stack
- Highly Recommended Approach
 - Always perform updates from Root Stack
 - Never update nested stacks directly.



- For Security Group Nested Stack
 - Update Root Stack with new template.
 - It automatically creates the security group nested stack



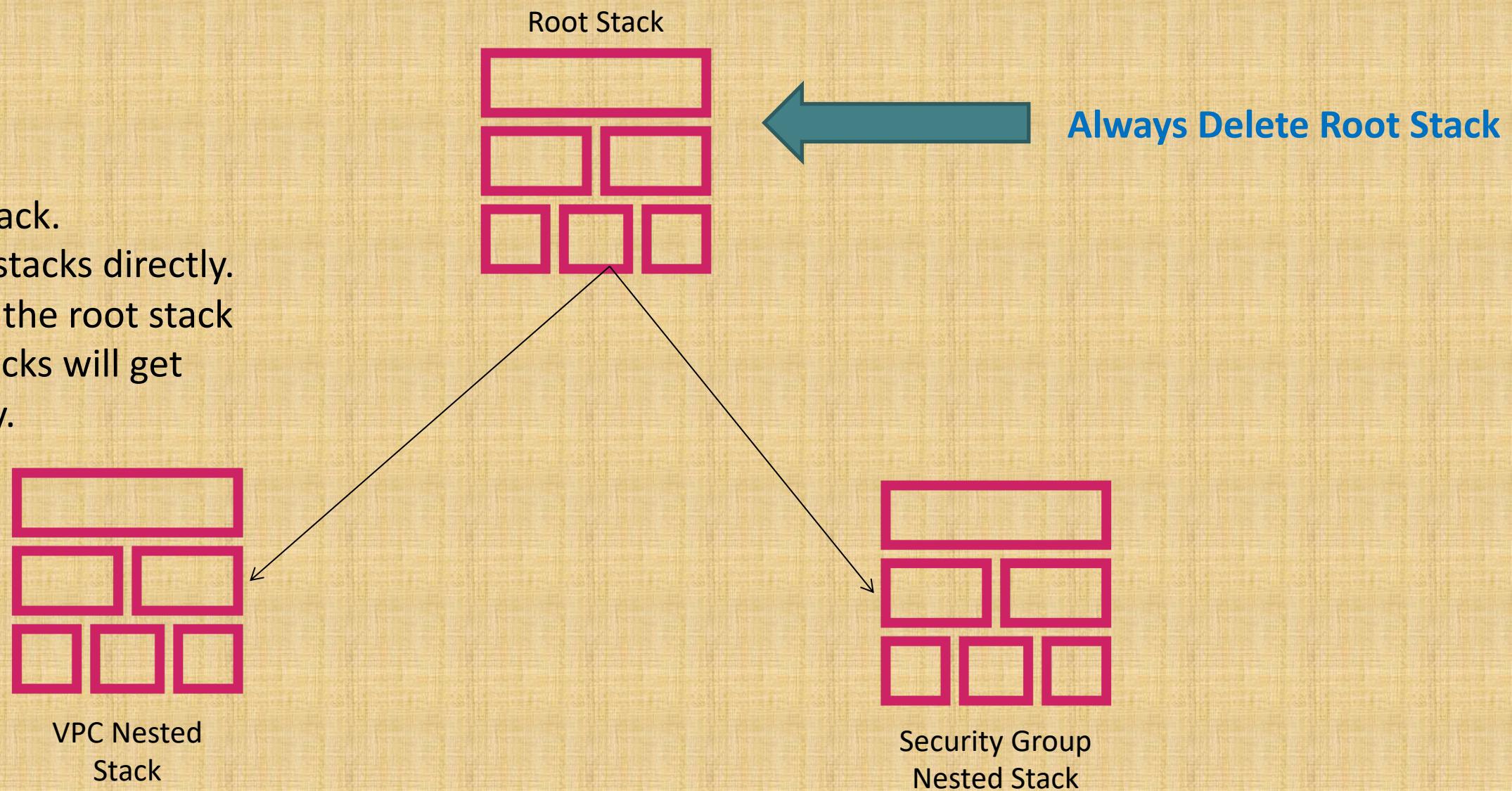
Update Stack

- For Nested Stack Updates
 - Update SG nested stack with new security rule.
 - Upload the new template to S3
 - Update Root Stack with existing template.

Nested Stacks – Practice – Delete Stack

Step#8:

- Recommendations
 - Always delete root stack.
 - Never delete nested stacks directly.
 - Whenever we delete the root stack associated nested stacks will get deleted automatically.



Nested Stacks vs Outputs - Pending

- A nested stack is a stack that you create within another stack by using the AWS::CloudFormation::Stack (p. 954) resource. With nested stacks, you deploy and manage all resources from a single stack.
- You can use outputs from one stack in the nested stack group as inputs to another stack in the group. This differs from exporting values.
- If you want to isolate information sharing to within a nested stack group, we suggest that you use nested stacks. To share information with other stacks (not just within the group of nested stacks), export values.
- For example, you can create a single stack with a subnet and then export its ID. Other stacks can use that subnet by importing its ID; each stack doesn't need to create its own subnet. Note that as long as stacks are importing the subnet ID, you can't change or delete it.



CodeCommit



CodeBuild



CodeDeploy



CodePipeline



CloudWatch



Simple Notification Service

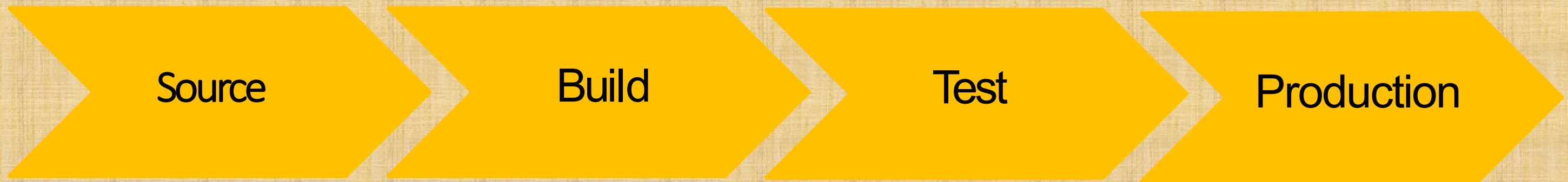


Amazon EC2

Continuous Integration & Continuous Delivery

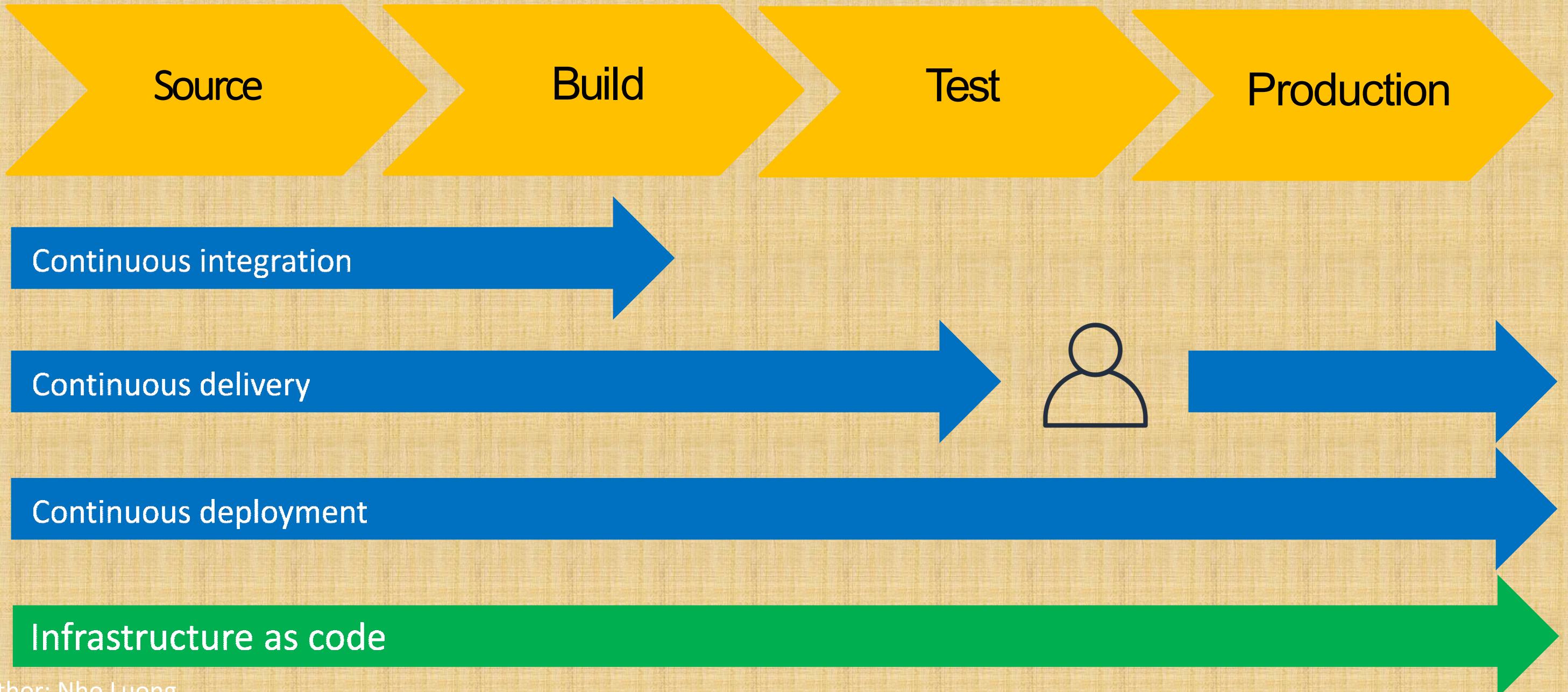


Stages in Release Process

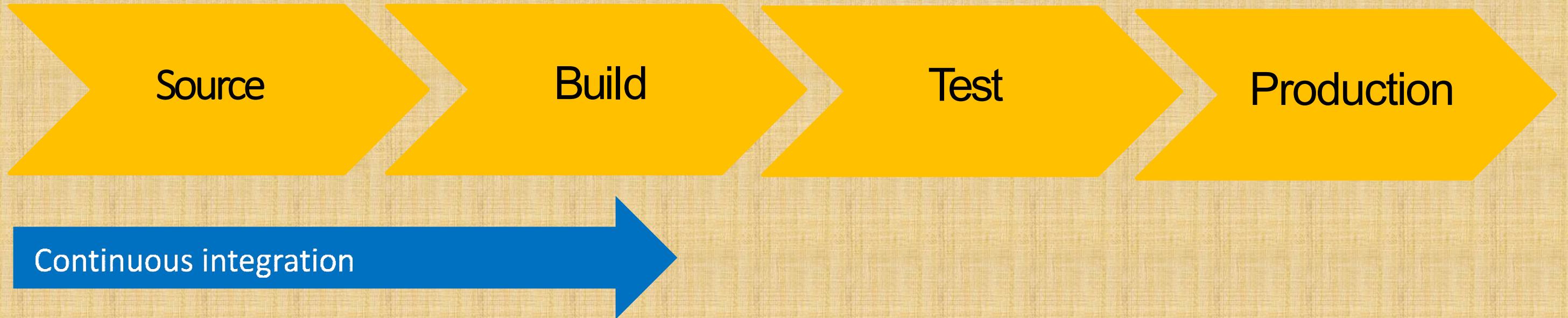


- | | | | |
|---|---|--|---|
| <ul style="list-style-type: none">• Check-in source code• Peer review new code | <ul style="list-style-type: none">• Compile Code & build artifacts (war files)• Unit Tests | <ul style="list-style-type: none">• Integration tests with other systems.• Load Testing• UI Tests• Security Tests | <ul style="list-style-type: none">• Deployment to production environments• Monitor code in production to quickly detect errors |
|---|---|--|---|

Stages in Release Process

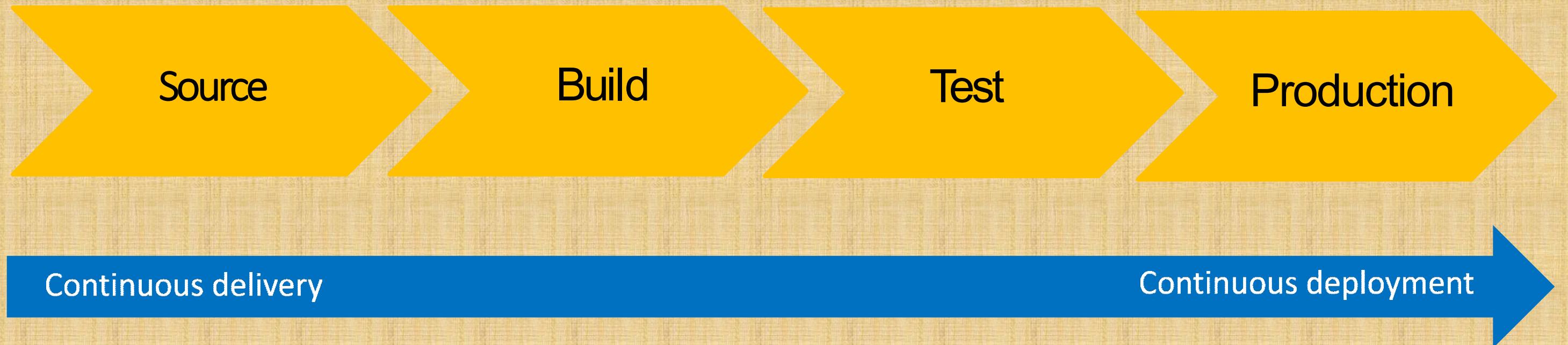


Continuous Integration



- Automatically kick off a new release when new code is checked-in
- Build and test code in a consistent, repeatable environment
- Continually have an artifact ready for deployment

Continuous Delivery

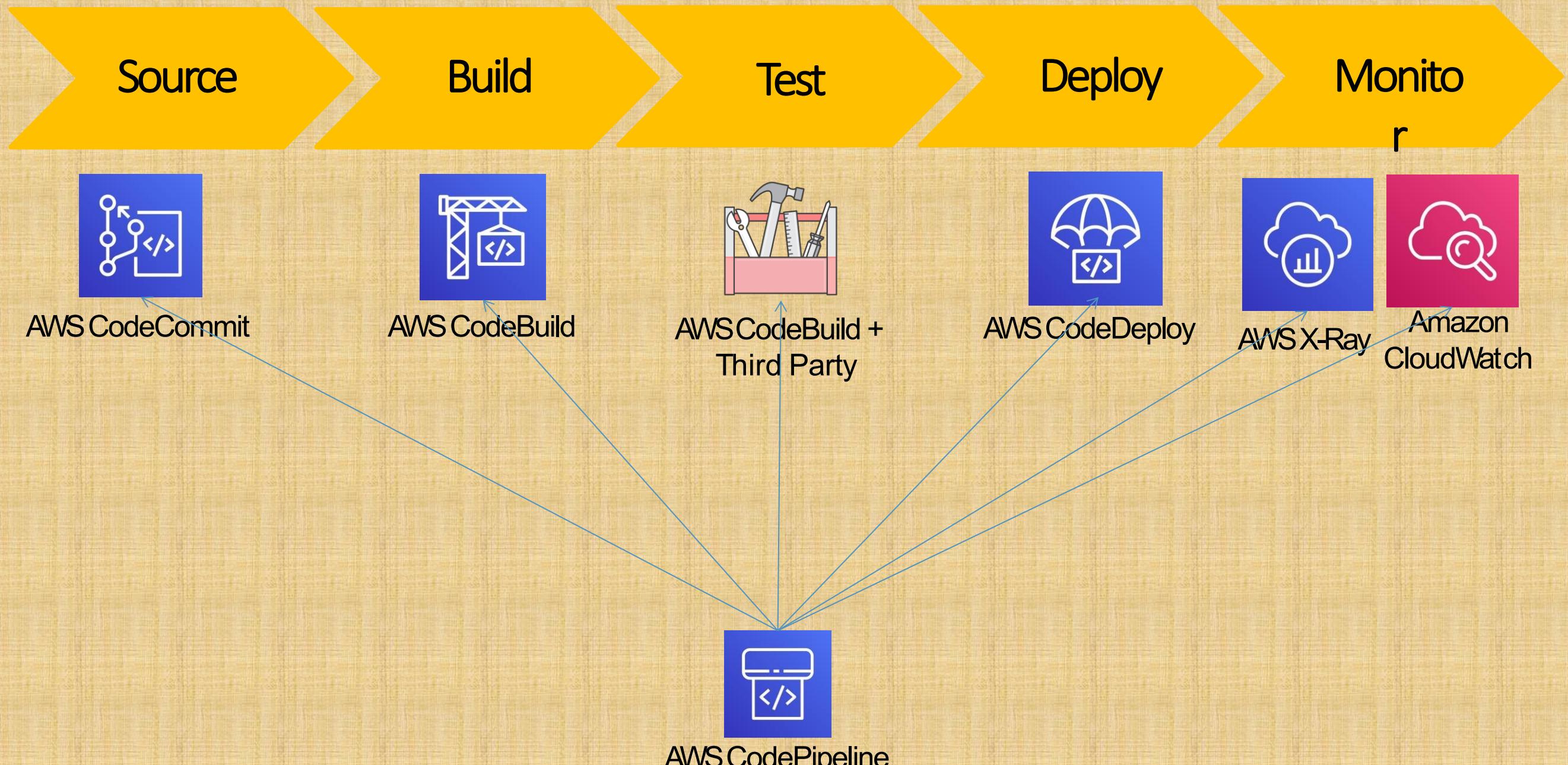


- Automatically deploy new changes to staging environments for testing
- Deploy to production safely without affecting customers
- Deliver to customers faster
- Increase deployment frequency, and reduce change lead time and change failure rate

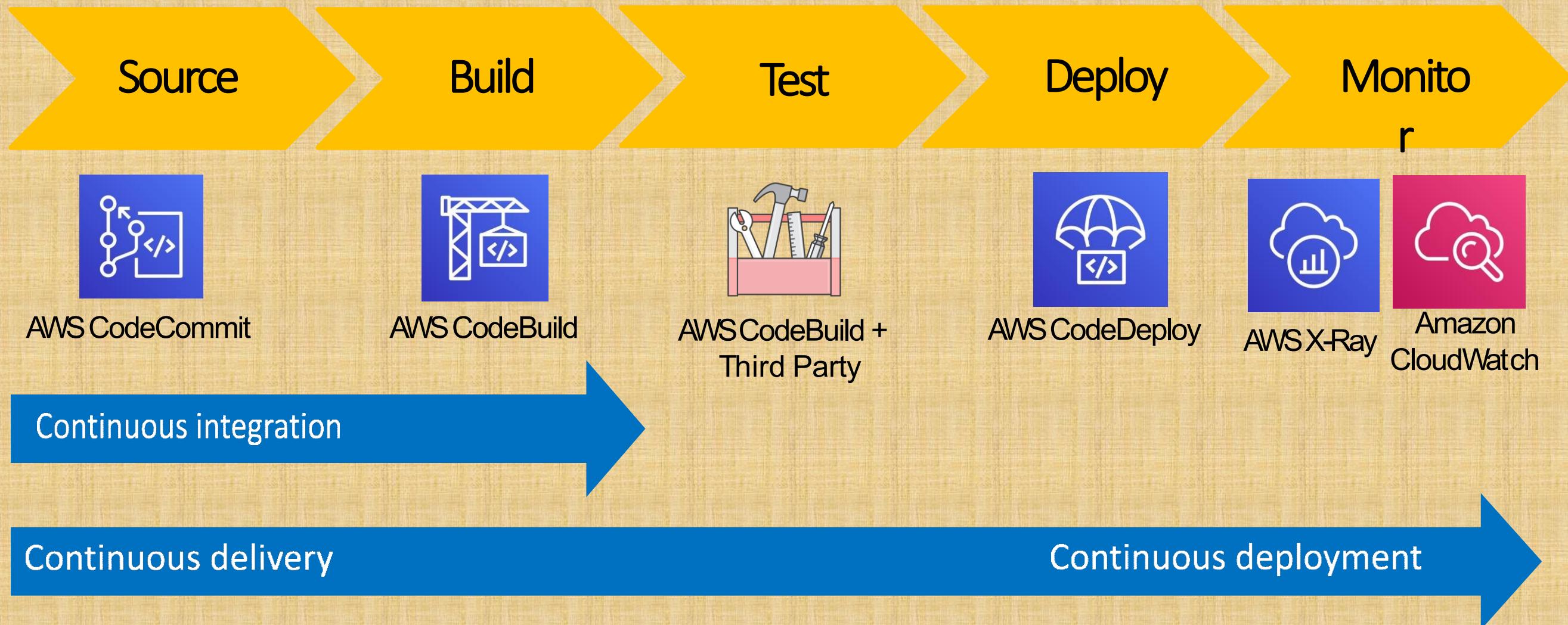
Author: Nho Luong

Skill: DevOps Engineer Lead

AWS Developer Tools or Code Services



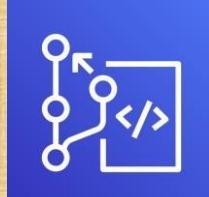
AWS Developer Tools or Code Services



AWS CodePipeline

AWS Developer Tools or AWS Code Services

Source



CodeCommit

Build



CodeBuild

Test



CodeBuild + Third Party

Deploy



CodeDeploy

Monito



AWS X-Ray



CloudWatch

- Version control service
- We can privately store and manage source code
- Secure & highly available

- Fully managed build service, Compiles source code, Runs tests and produces software packages
- Scales continuously and processes multiple builds concurrently.
- No build servers to manage.
- Pay by minute, only for compute resources we use.
- Monitor builds through CloudWatch events.
- Supports following programming language runtimes Ruby, Python, PHP, Node, Java, Golang, .Net Core, Docker and Android

- Automates code deployments to any instance and Lambda
 - Avoids downtime during application deployment
 - Roll back automatically if failure detected
 - Deploy to Amazon EC2, Lambda, or on-premises servers

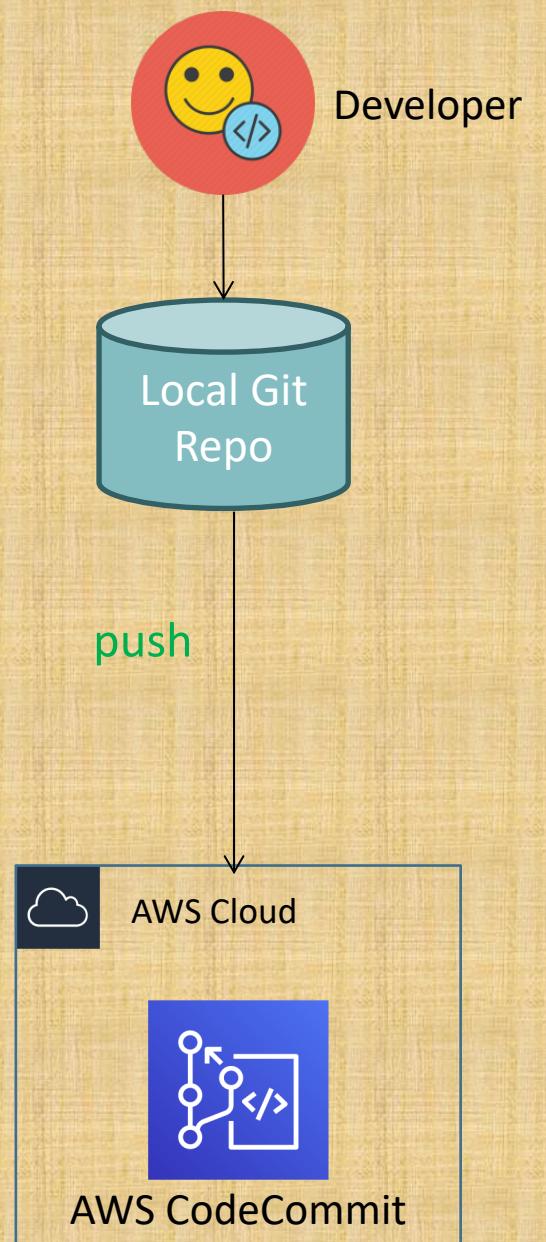


CodePipeline

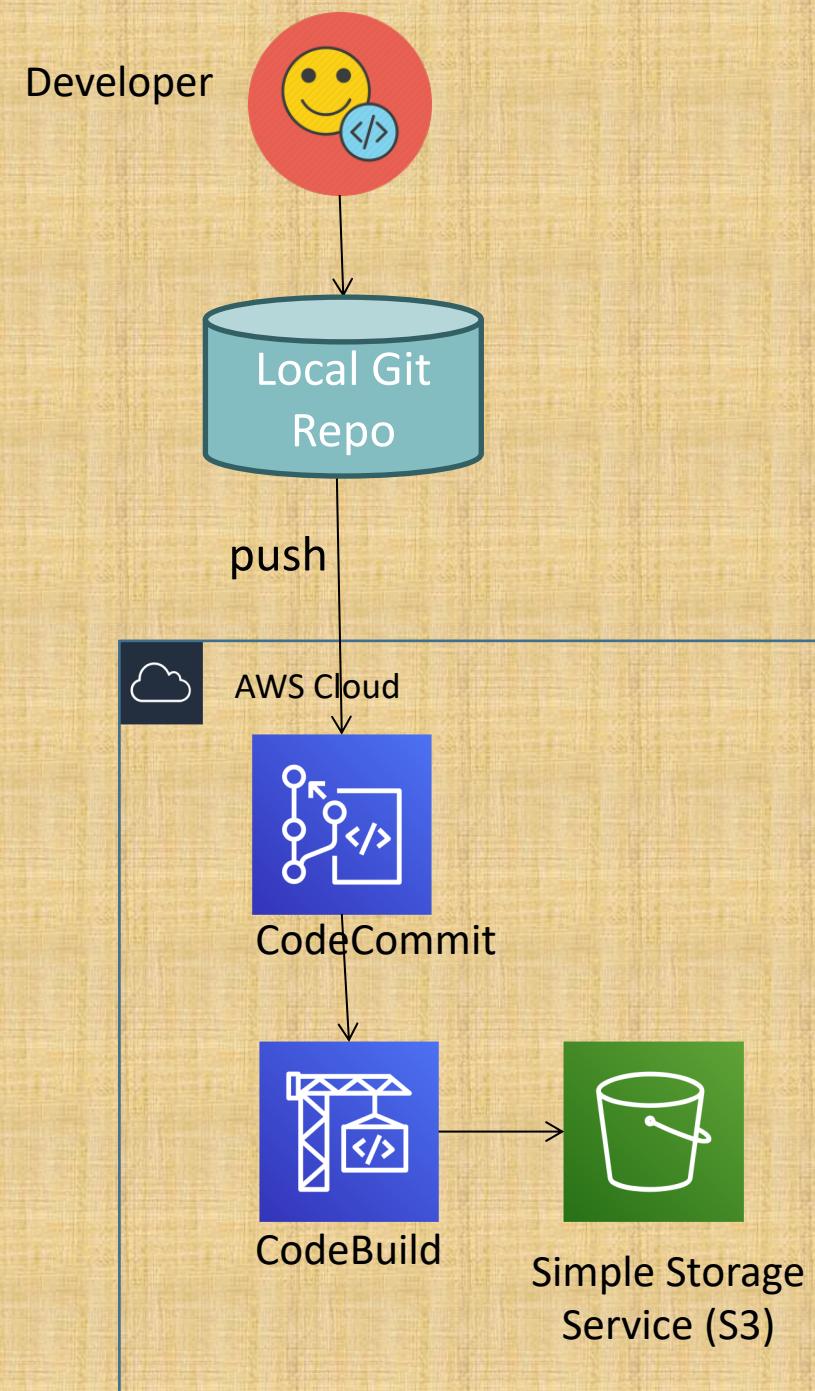
- Continuous delivery service for fast and reliable application updates
- Model and visualize your software release process
- Builds, tests, and deploys your code every time there is a code change
- Integrates with third-party tools and AWS

CodeCommit

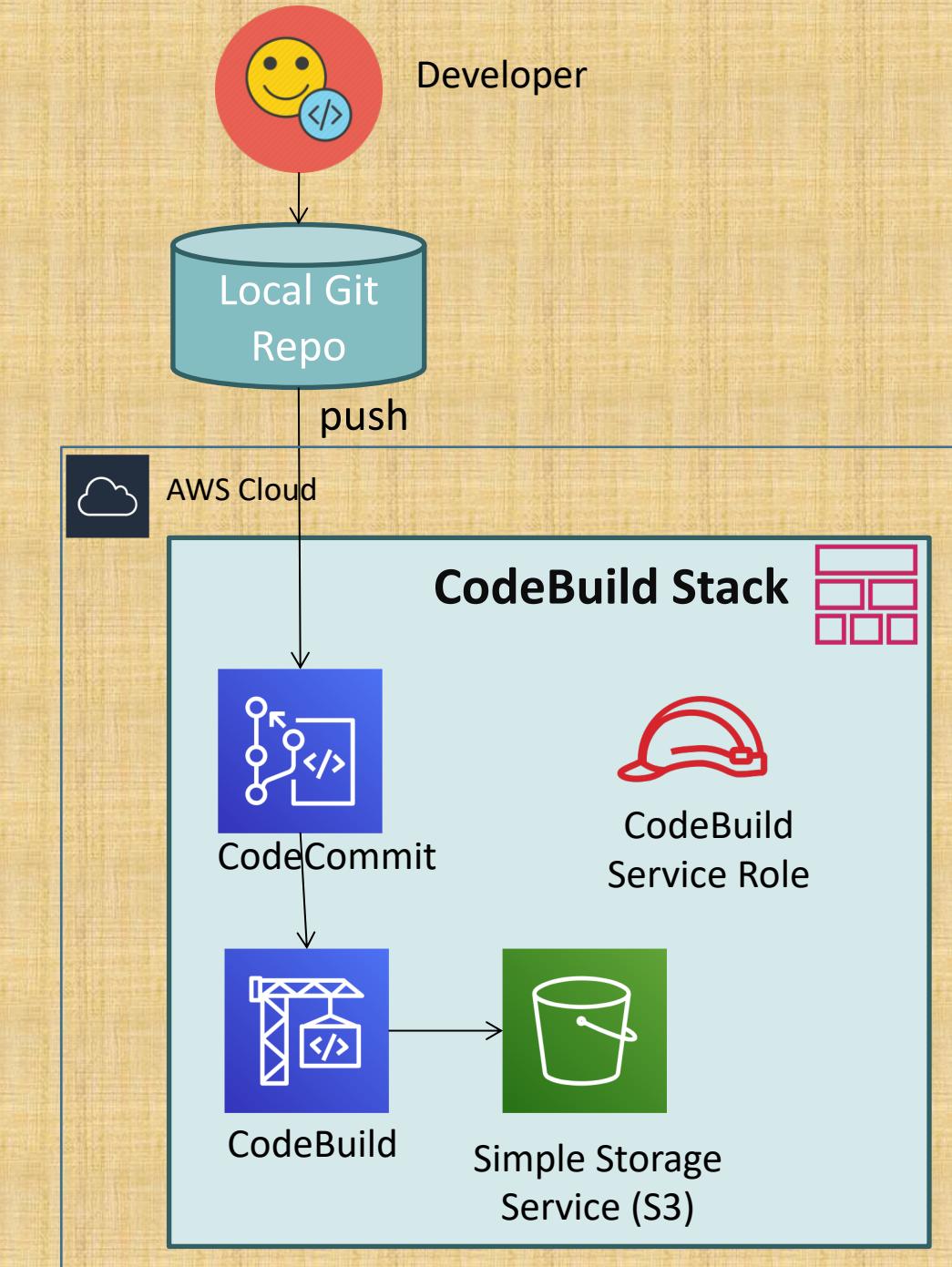
- Build a simple rest service using Java Spring Boot.
- Check-in code to Local Repo and push to CodeCommit.



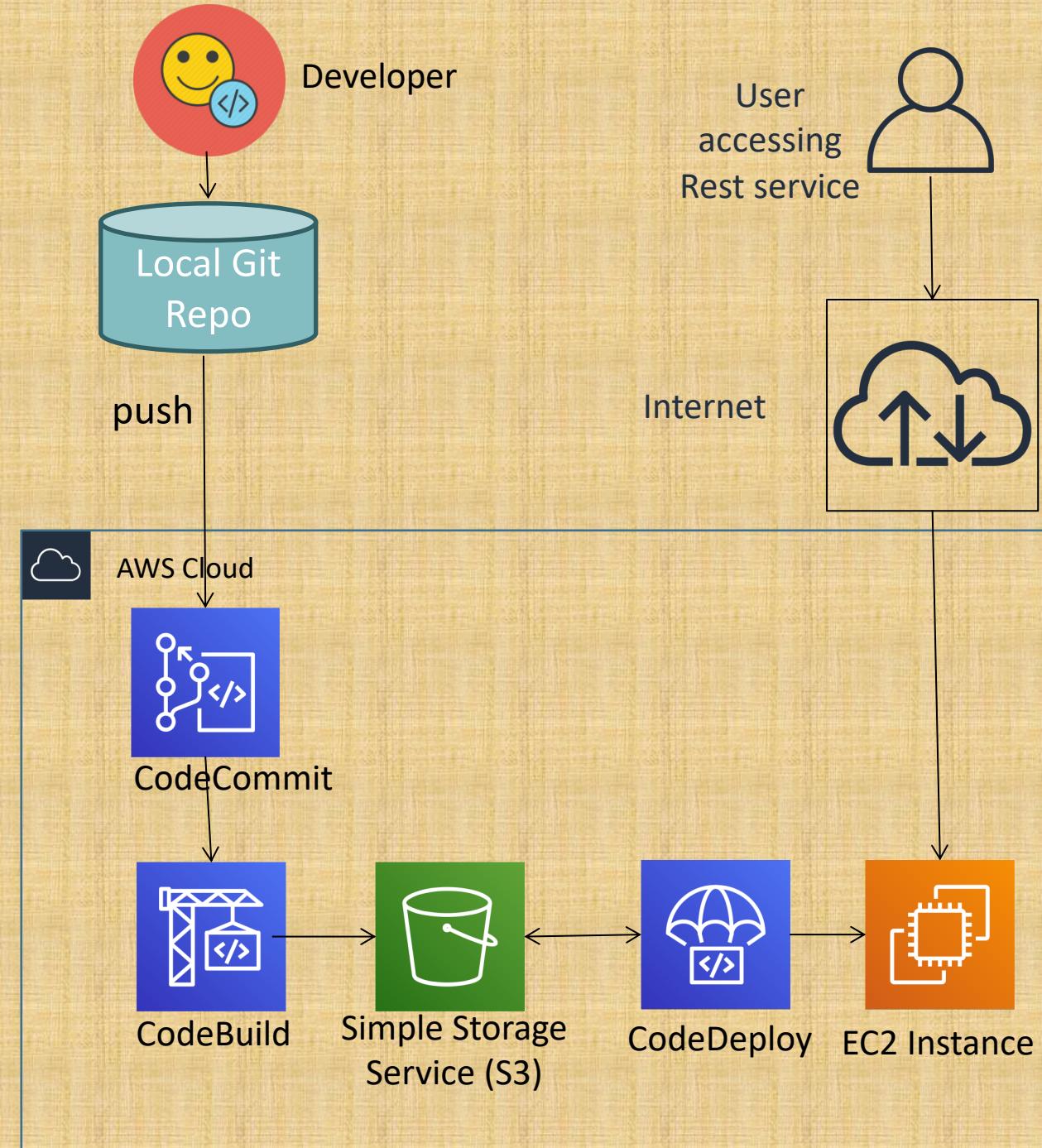
CodeBuild – AWS Web Console



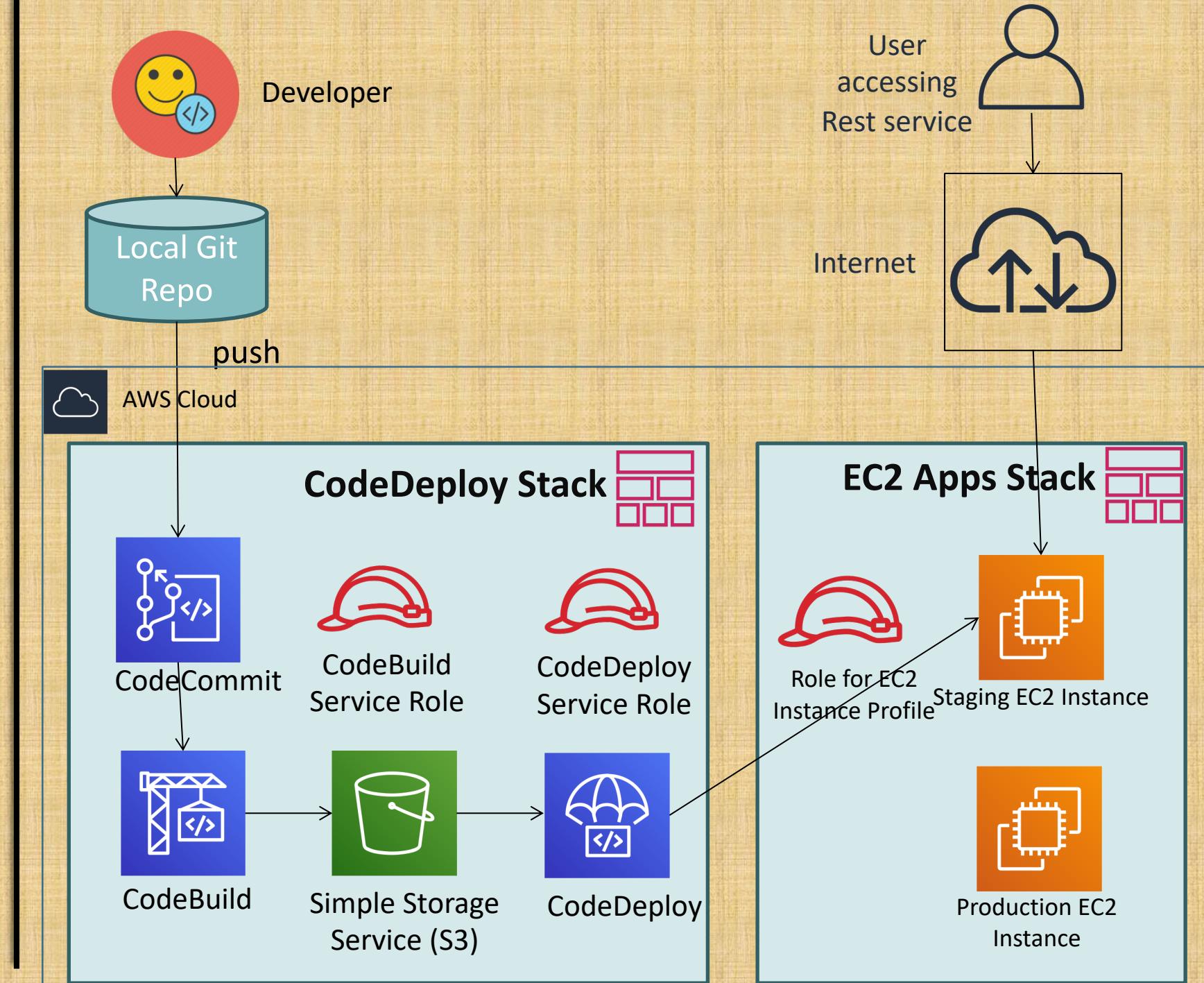
CodeBuild – AWS CloudFormation



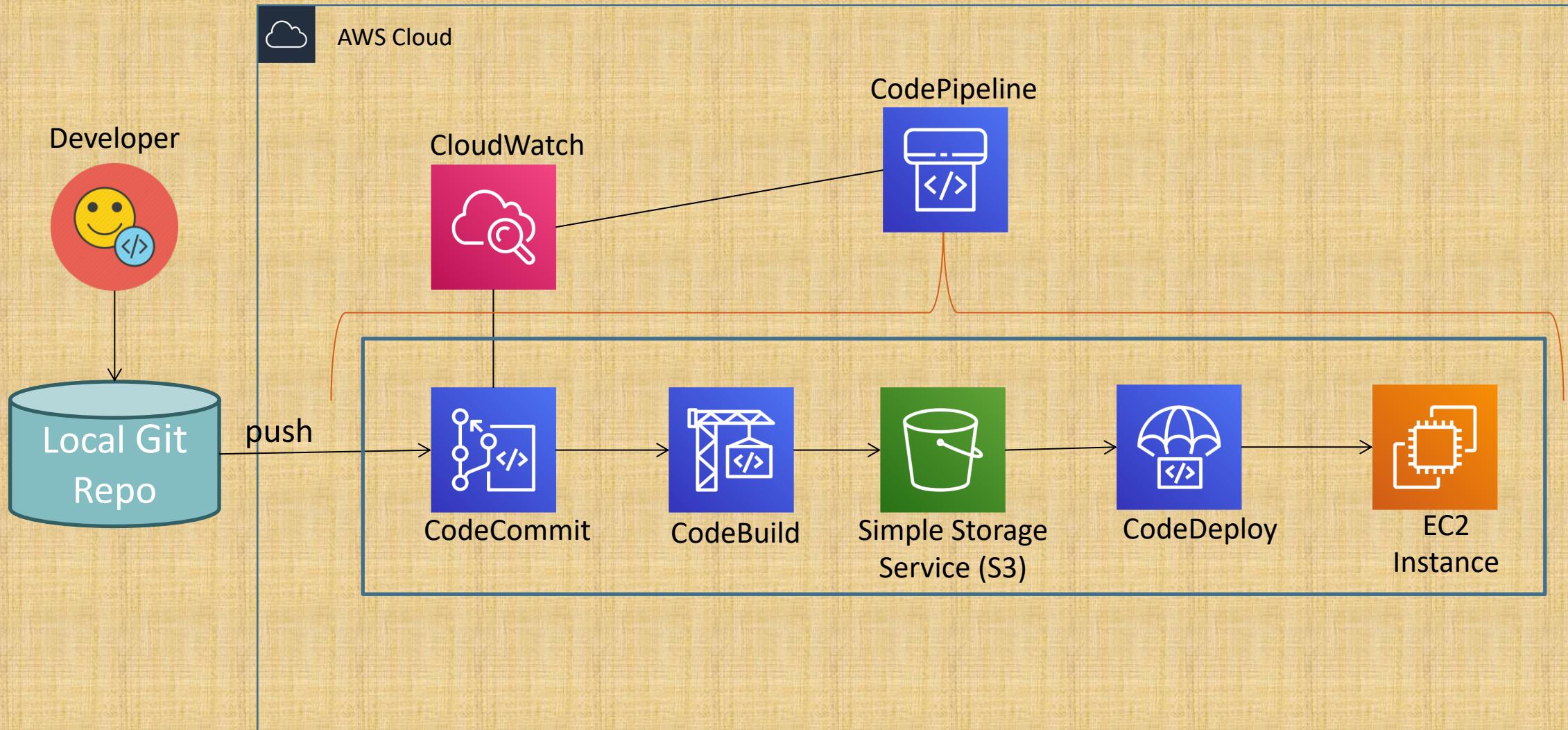
CodeDeploy – AWS Web Console



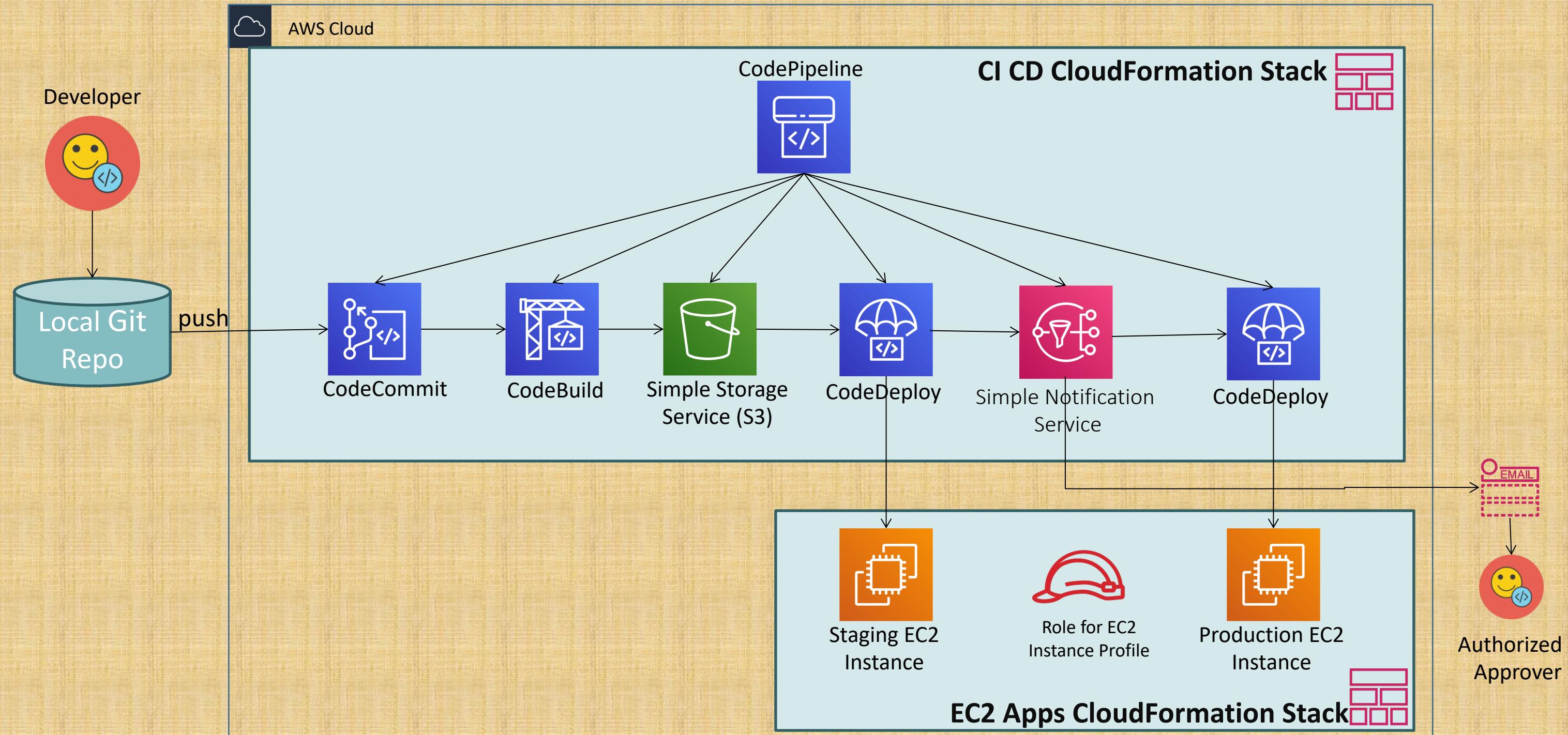
CodeDeploy – AWS CloudFormation



CodePipeline – AWS Web Console



CodePipeline – AWS CloudFormation



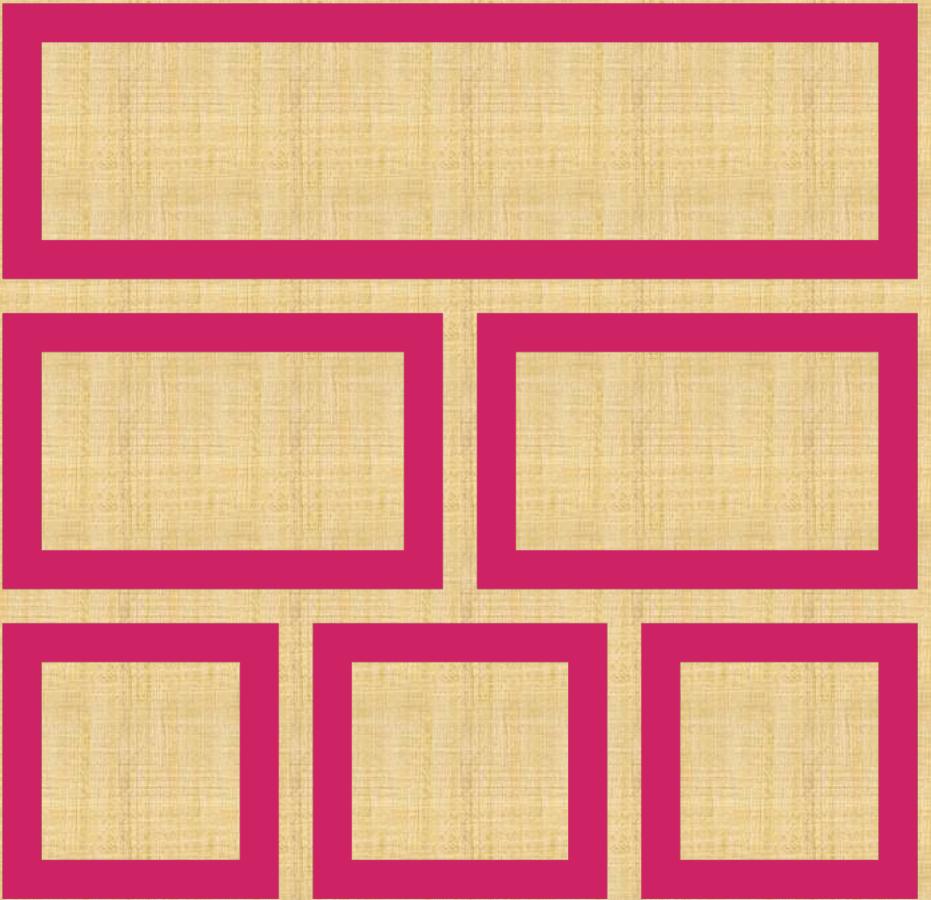
Author: Nho Luong

Skill: DevOps Engineer Lead

Pre-requisites

- **Region: us-east-2 (ohio)**
 - In templates, EC2 Instnace ImageID is hardcoded to this region (Amazon Linux AMI). If you want to test in other regions, please update the templates with ImageId equivalent to that respective region.
- **Default VPC**
 - Ensure we have the default VPC created in the region where we are using these templates.

EC2 CloudFormation Stack

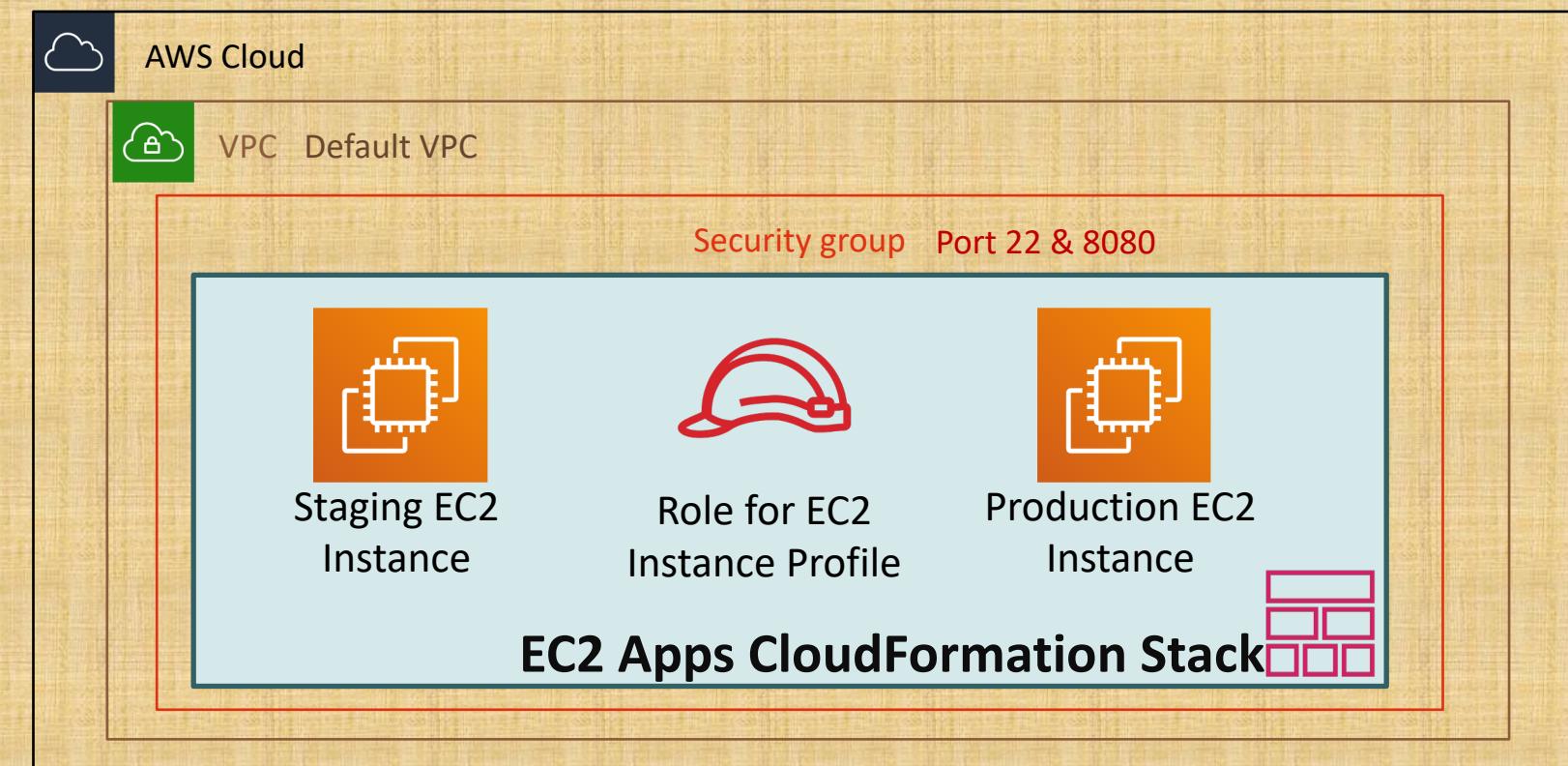


Author: Nho Luong

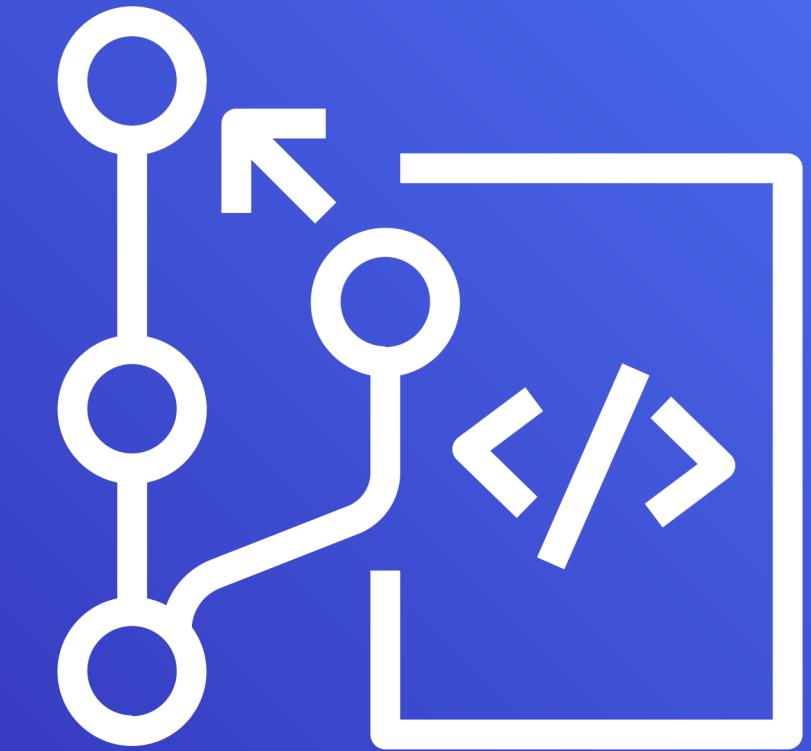
Skill: DevOps Engineer Lead

EC2 CloudFormation Stack

- Step 1: Create Security Group with port 22 and 8080 rules for inbound access.
- Step 2: Create two EC2 Instances
 - Staging
 - Production
 - Update UserData
- Step 3: Create Instance Profile Role and Instance Profile for EC2 Instances to access S3 Buckets.
- Step 4: Create stack and verify.



AWS CodeCommit



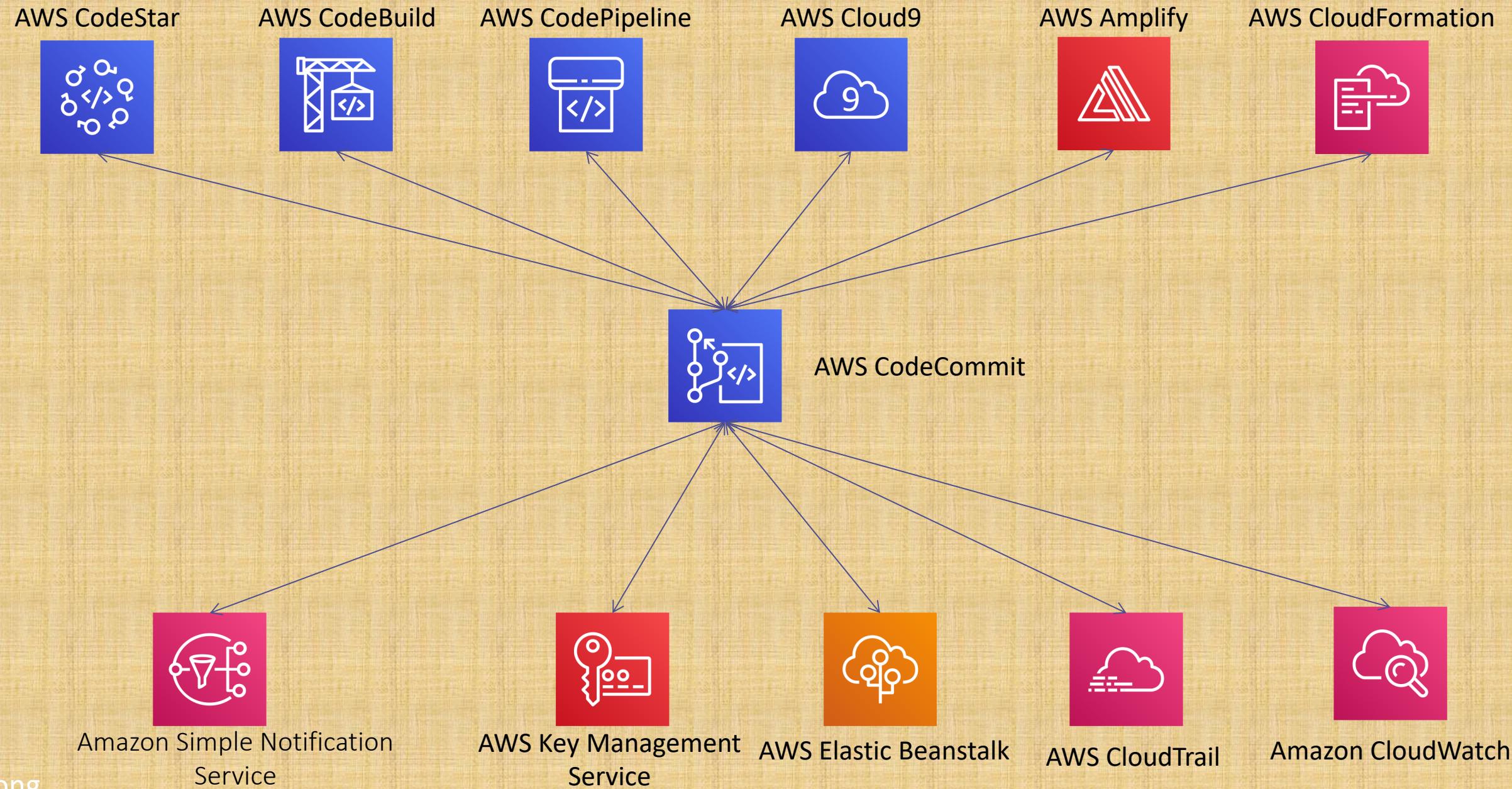
Author: Nho Luong

Skill: DevOps Engineer Lead

AWS CodeCommit - Introduction

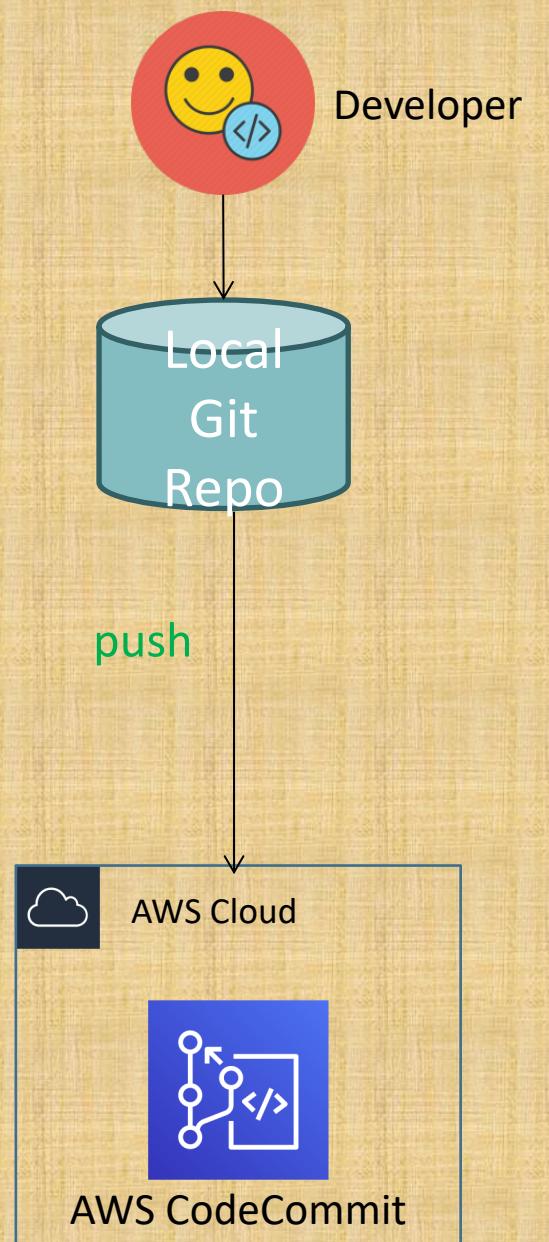
- Version Control Service hosted by AWS
- We can privately store and manage documents, source code, and binary files
- Secure & highly scalable
- Supports standard functionality of Git (CodeCommit supports Git versions 1.7.9 and later.)
- Uses a static user name and password in addition to standard SSH..

CodeCommit – Integration with AWS Services

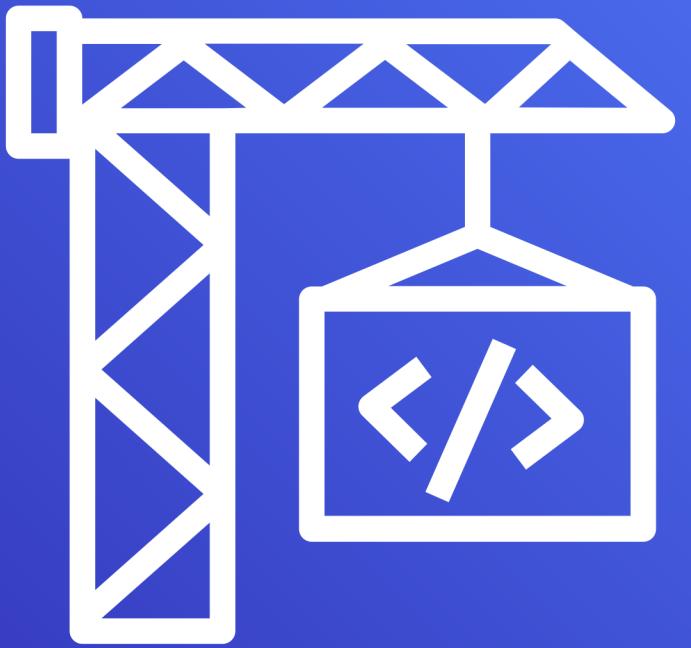


CodeCommit - Steps

- Step#1: Sample Spring Boot Rest Application
 - Pre-requisites
 - Install STS IDE
 - Create Spring boot rest application.
 - Test it.
- Step#2: GIT Repository
 - Create a local git repository and check-in code.
 - Create a remote git repository in AWS Code Commit.
 - Create Code Commit git credentials to connect.
 - Push the code to remote git repository.
 - Verify code in AWS Code Commit.
- Step#3: CodeCommit Features
 - Code, Commits, Branches
 - Settings: Notifications, Triggers
 - Pull Requests



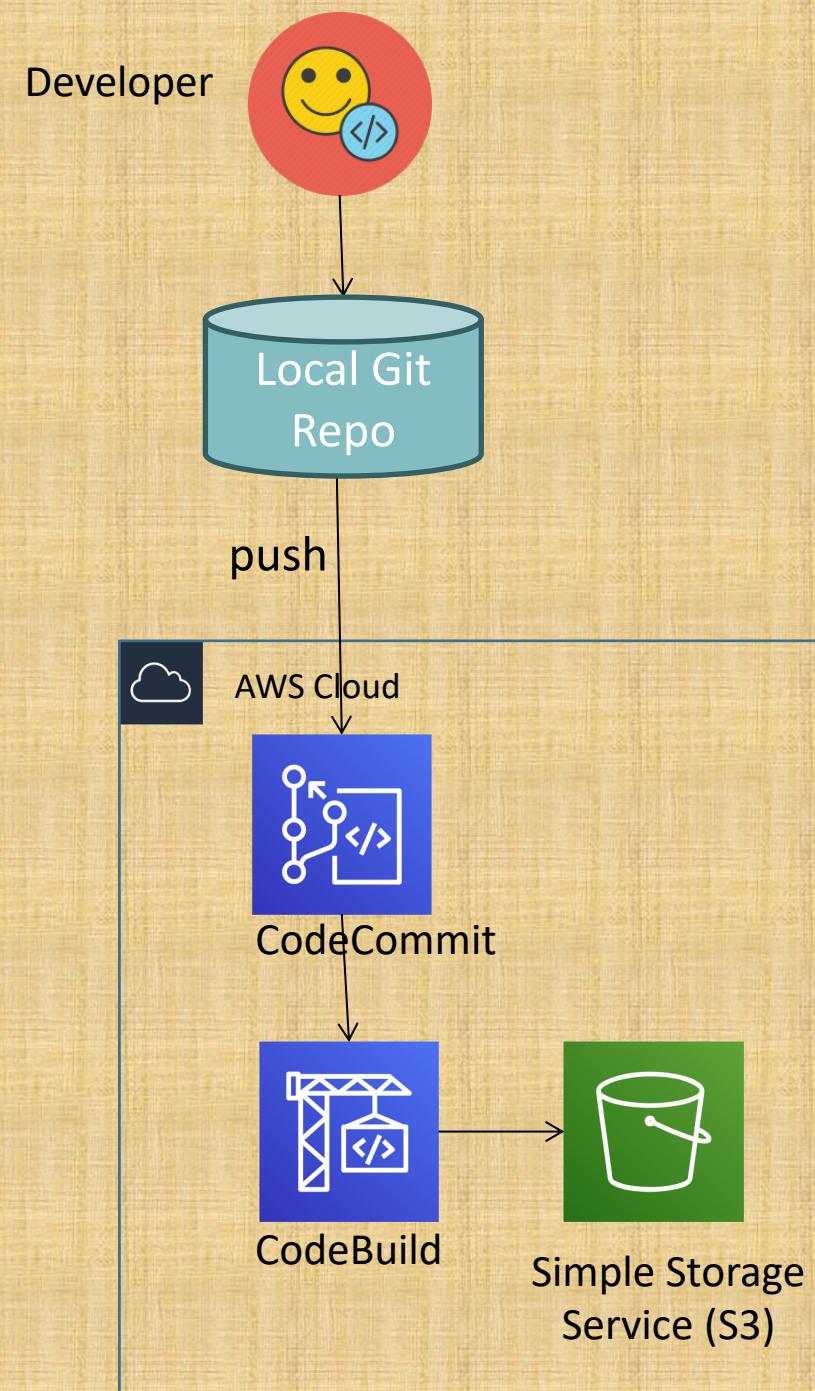
AWS CodeBuild



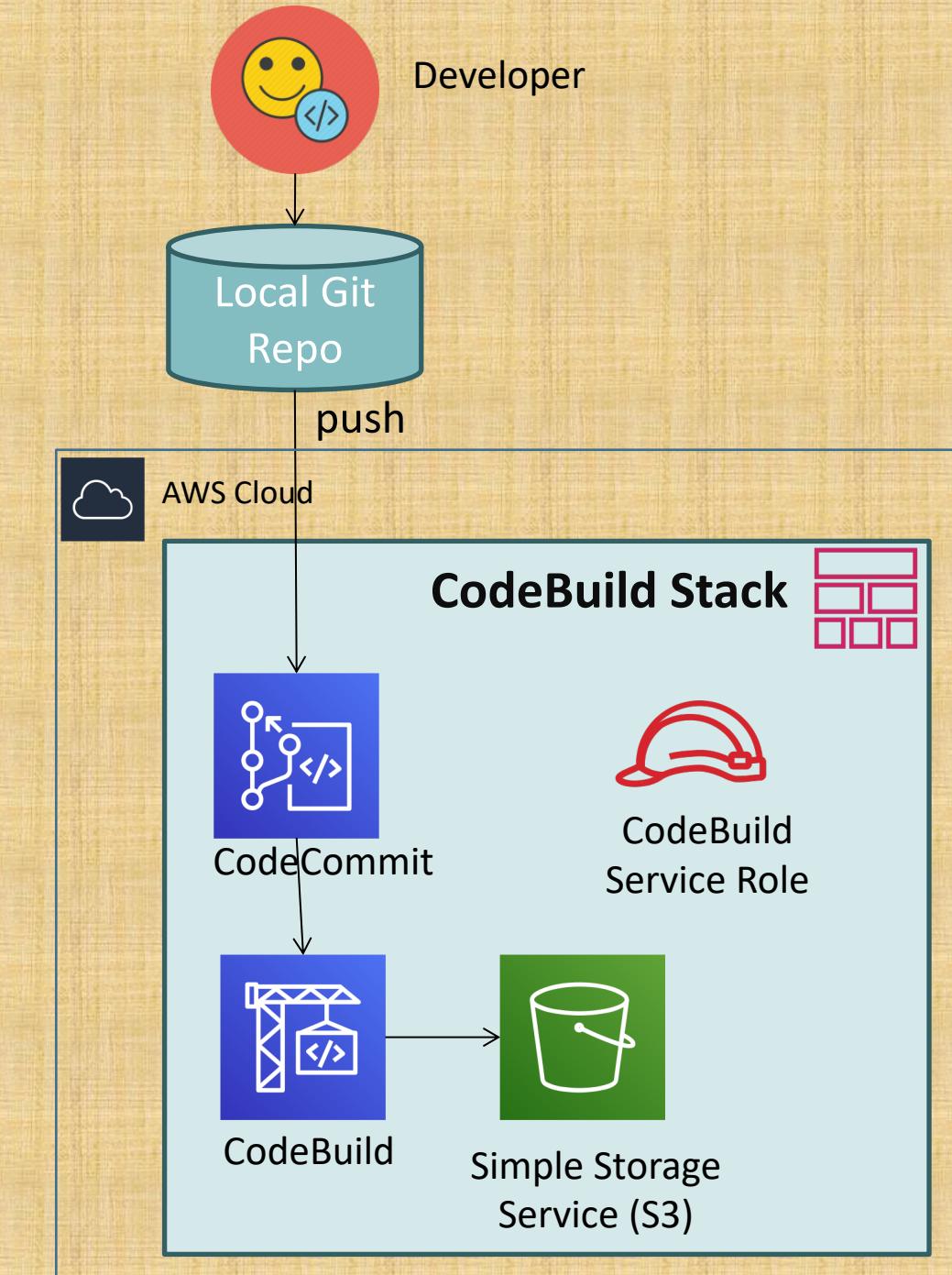
Author: Nho Luong

Skill: DevOps Engineer Lead

CodeBuild – AWS Web Console



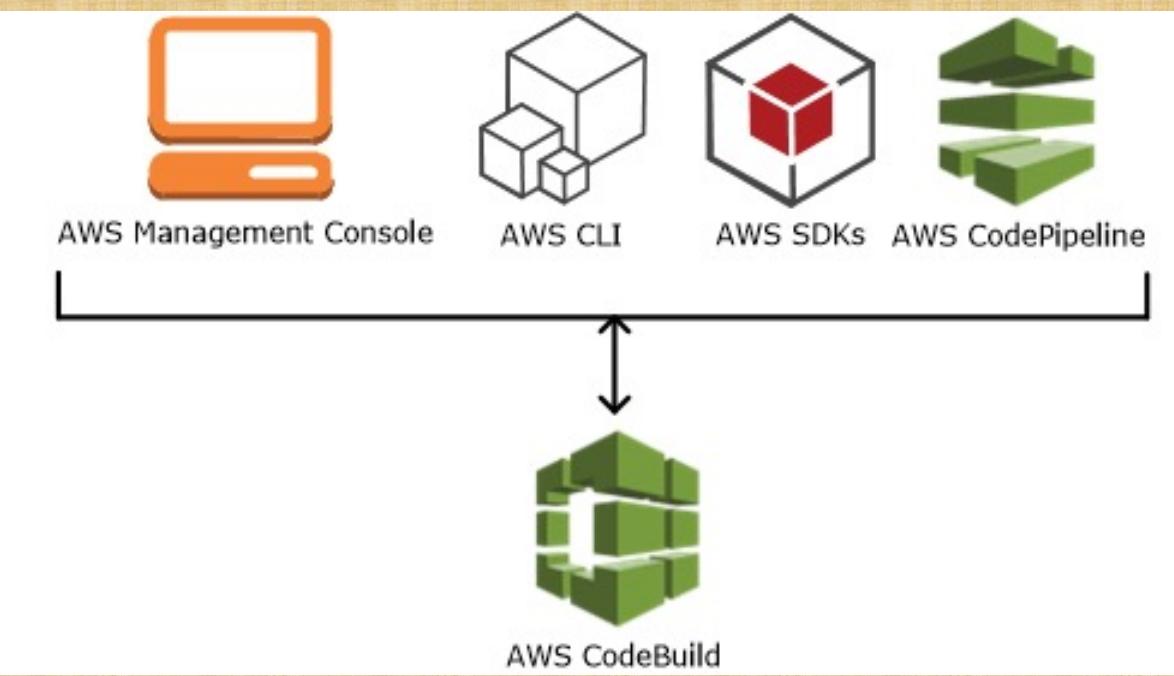
CodeBuild – AWS CloudFormation



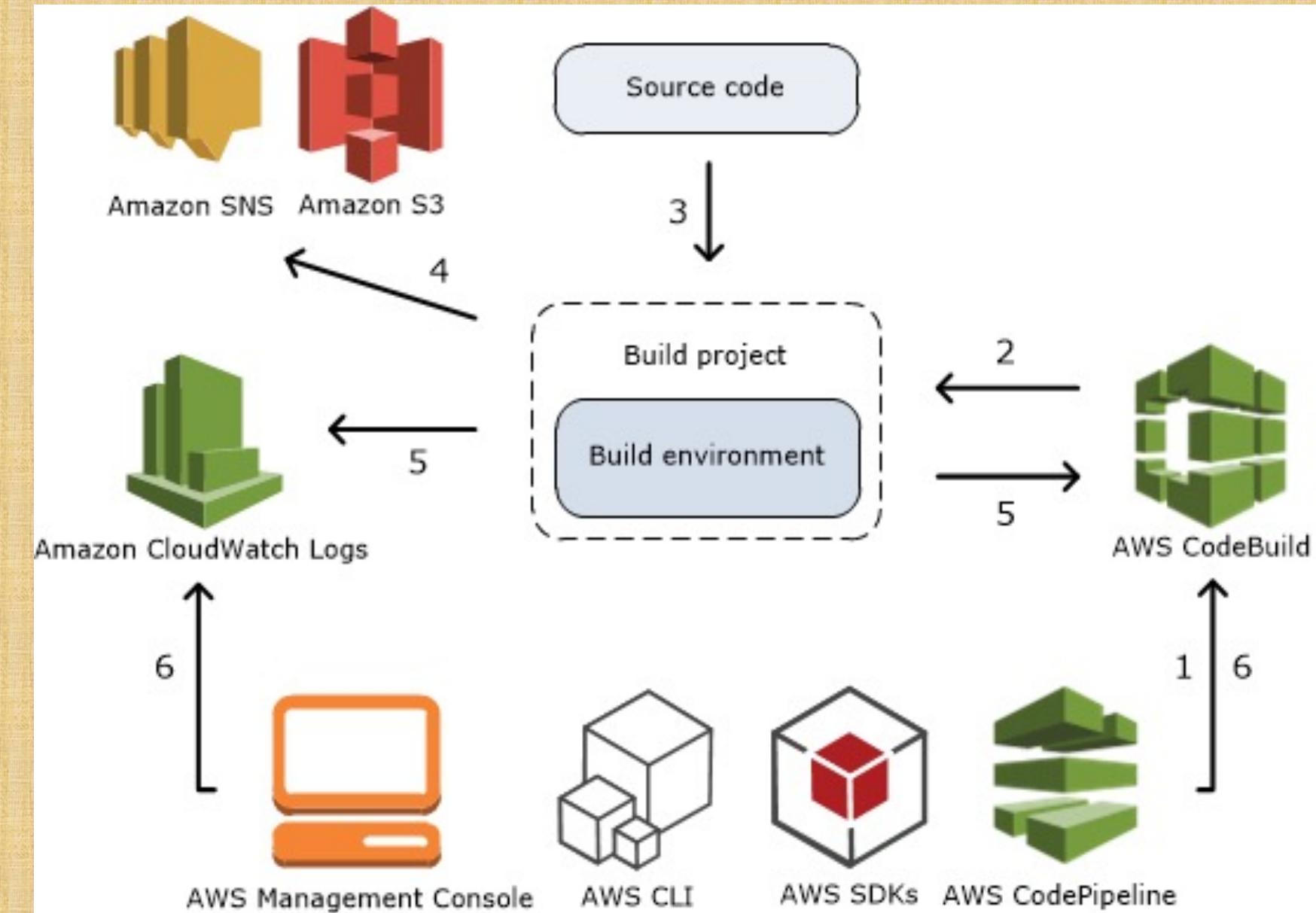
CodeBuild - Introduction

- CodeBuild is a **fully managed** build service in the cloud.
- Compiles your source code, runs unit tests, and produces artifacts that are ready to deploy.
- Eliminates the need to provision, manage, and scale your own build servers.
- It provides **prepackaged build environments** for the most popular programming languages and build tools such as Apache Maven, Gradle, and more.
- We can also customize build environments in CodeBuild to use our own build tools.
- **Scales automatically** to meet peak build requests.

How to run CodeBuild?

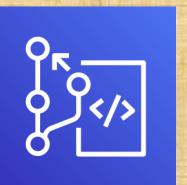


How CodeBuild works?



Source

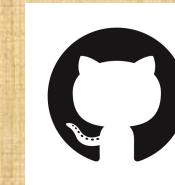
AWS CodeCommit



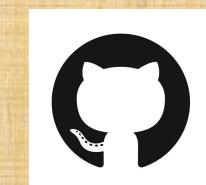
Amazon Simple Storage Service (S3)



GitHub



GitHub Enterprise



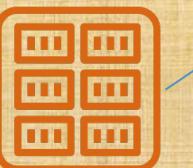
Bitbucket



Managed Image

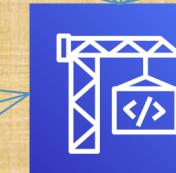


Amazon EC2 Container Registry



External Container Registry Environment

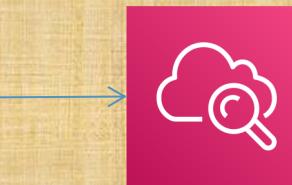
Environment



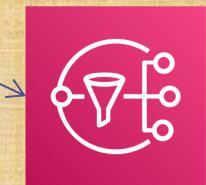
AWS CodeBuild



Artifacts



Amazon CloudWatch Logs



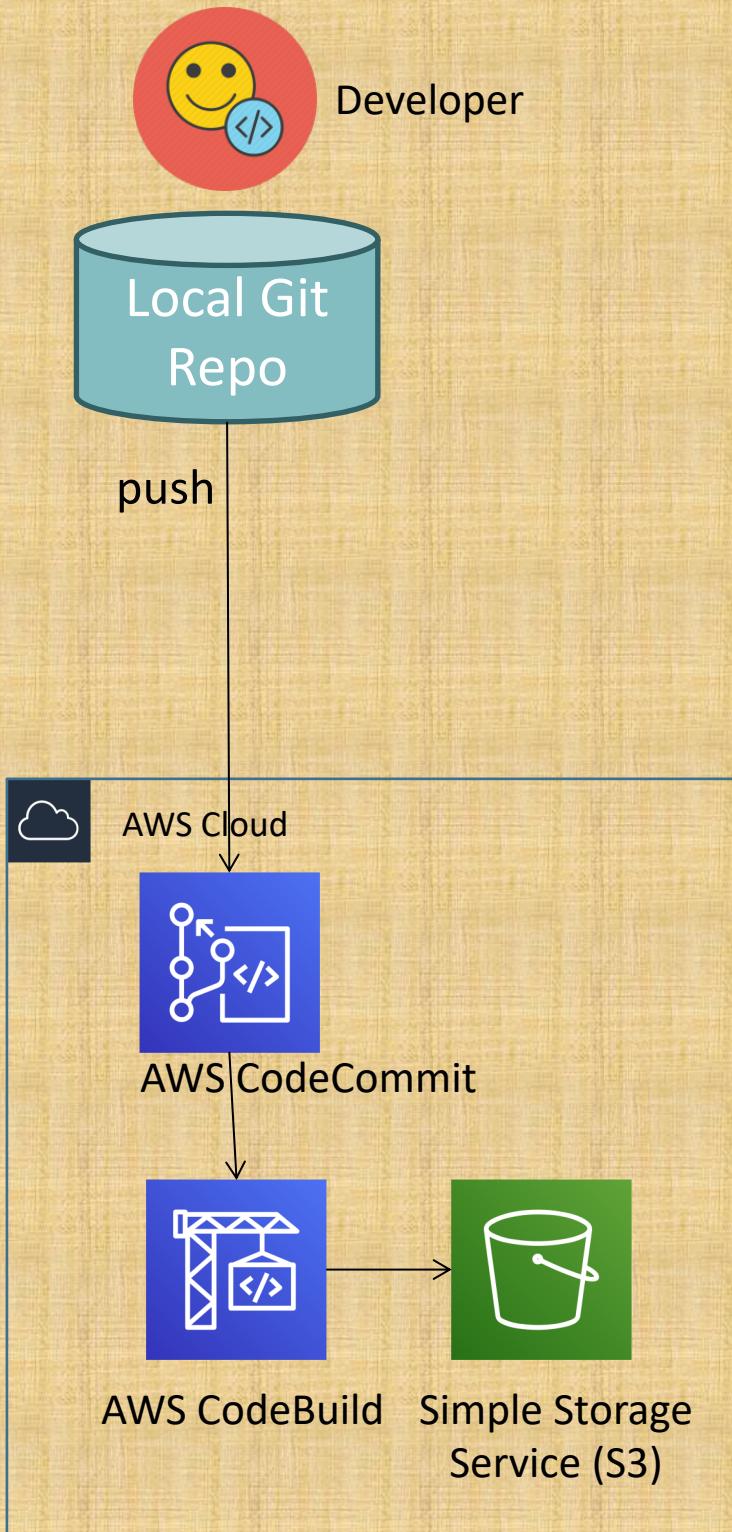
Amazon Simple Notification Service

Notifications

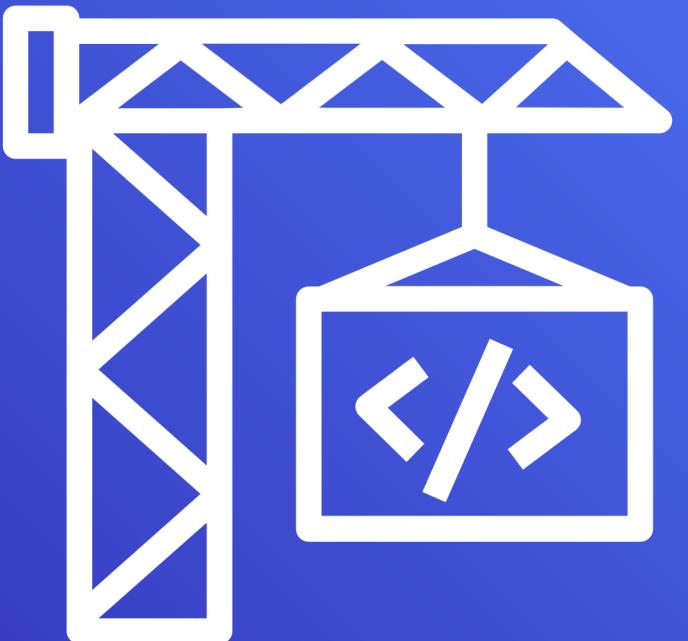
AWS CodeBuild Architecture

CodeBuild - Steps

- Step#1: Create CodeBuild Project
 - Create a S3 bucket and folder
 - Create CodeBuild project
 - Start build, Verify build logs, Verify build phase details
- Step#2: buildspec.yml & Start Build
 - Create buildspec.yml and check-in code
 - Start build, Verify build logs, Verify build phase details
 - Download the artifacts from S3, unzip and review
 - Run one more build and see versioning in S3.
- Step#3: Create Build Notifications
 - Create state change notification
 - Create Phase change notification



AWS CodeBuild using CloudFormation

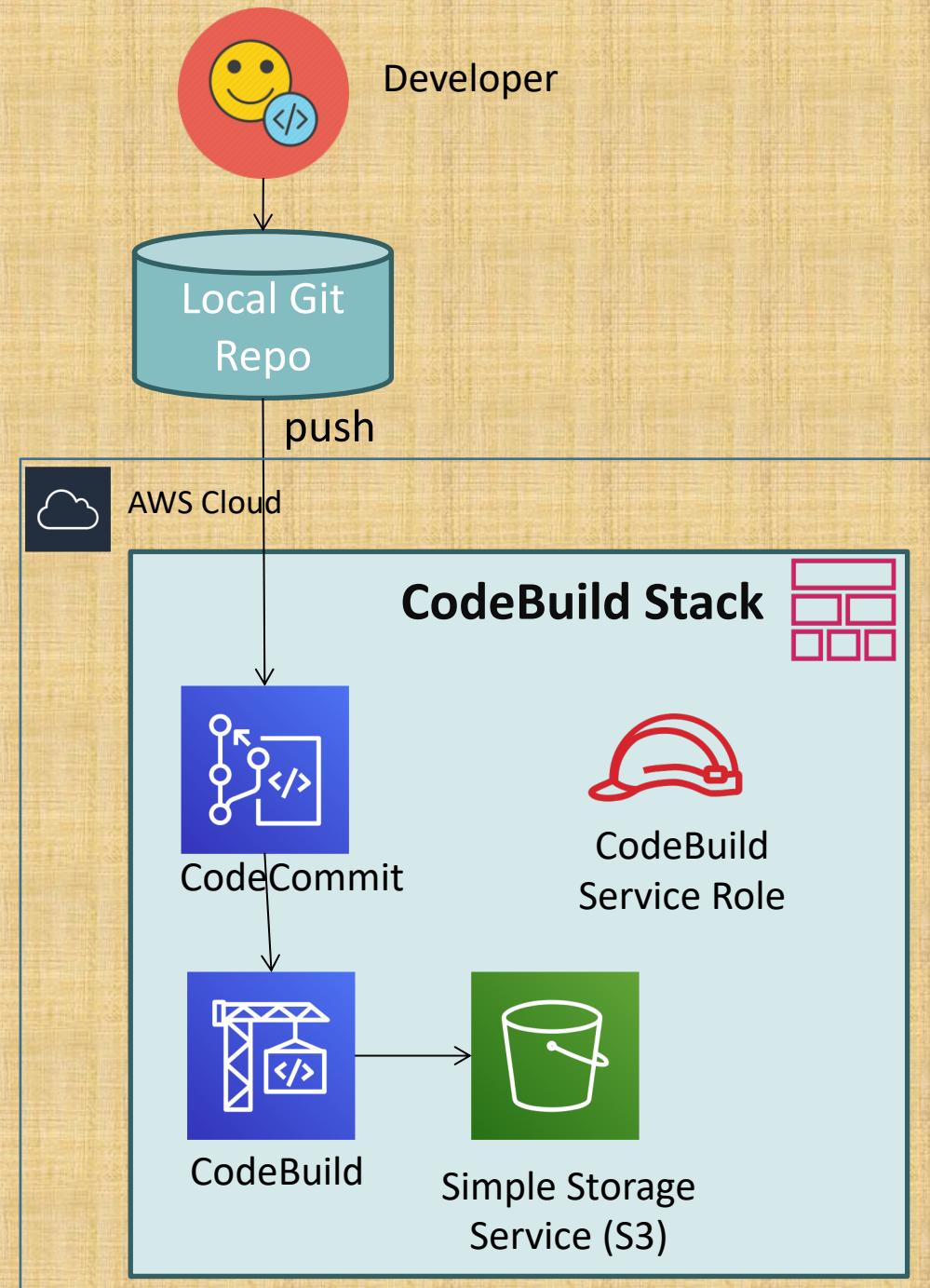


Author: Nho Luong

Skill: DevOps Engineer Lead

CodeBuild – CloudFormation Steps

- Step 1 : Create S3 bucket and enable versioning or use existing bucket.
- Step 2: Create `buildspec.yml` in our rest application and check-in code
- Step 3: Create CodeBuild Stack Template
 - Create CodeBuild Role.
 - Create CodeBuild project.
 - Parameters
- Step 4: Create Stack and Test the build.
 - Click on Start Build
 - Verify logs
 - Verify artifacts in S3.



AWS CodeDeploy

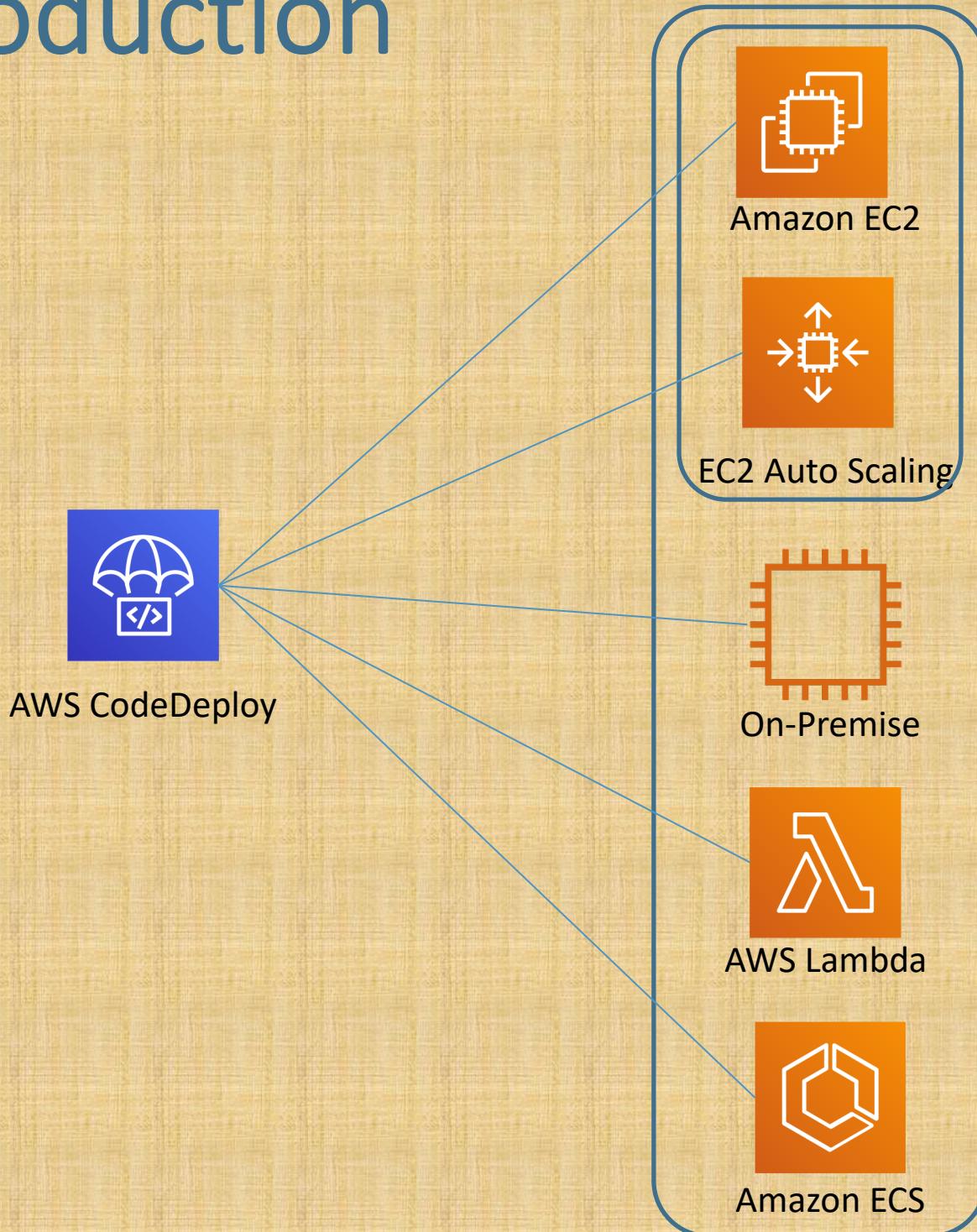


Author: Nho Luong

Skill: DevOps Engineer Lead

CodeDeploy - Introduction

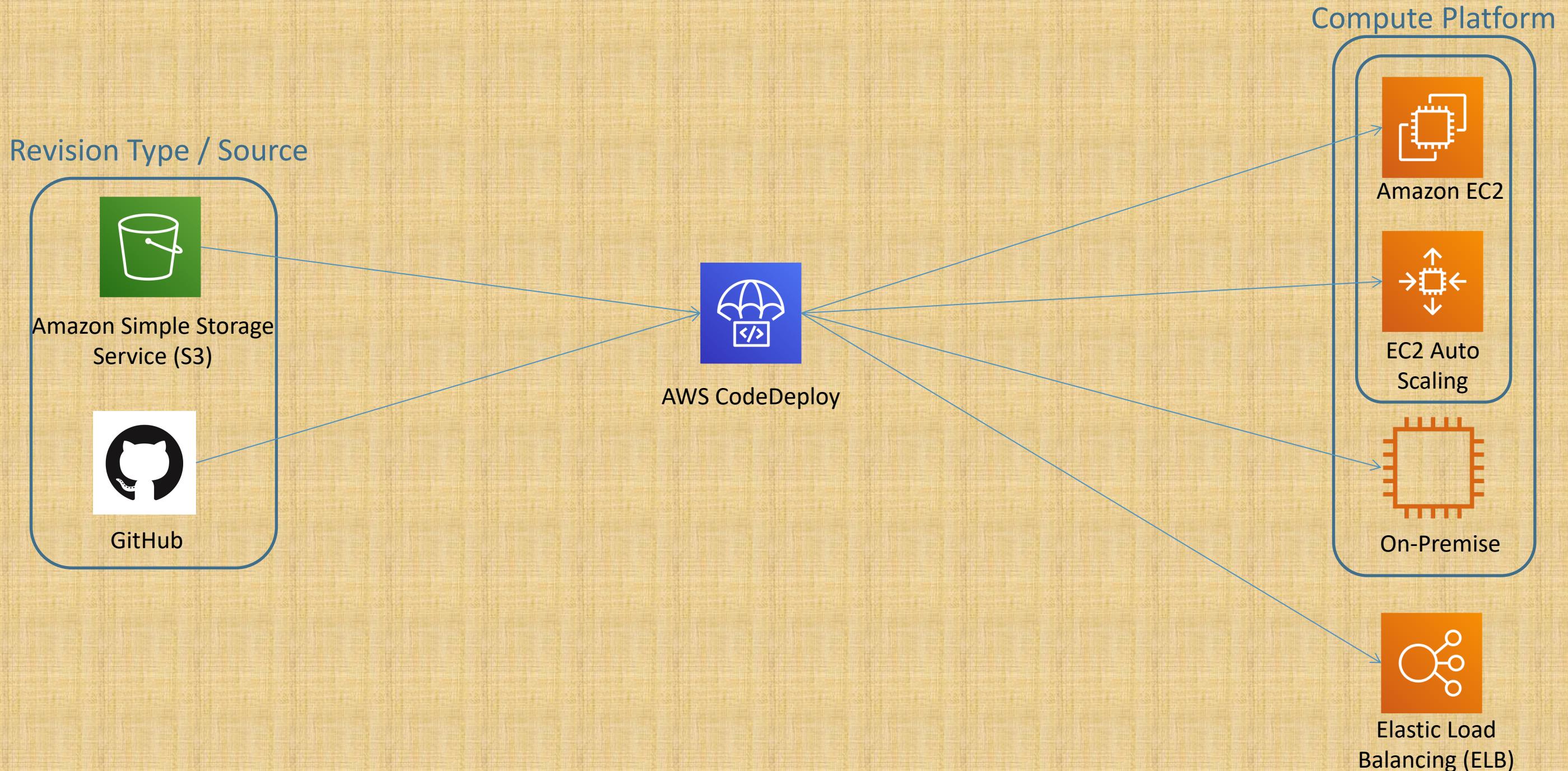
- CodeDeploy is a deployment service that automates application deployments to
 - EC2 instances
 - On-premises instances
 - AWS Lambda
 - AWS ECS
- We can deploy **unlimited variety** of application content
 - code
 - serverless AWS Lambda functions
 - web and configuration files
 - executables
 - packages
 - scripts
 - multimedia files



CodeDeploy - Introduction

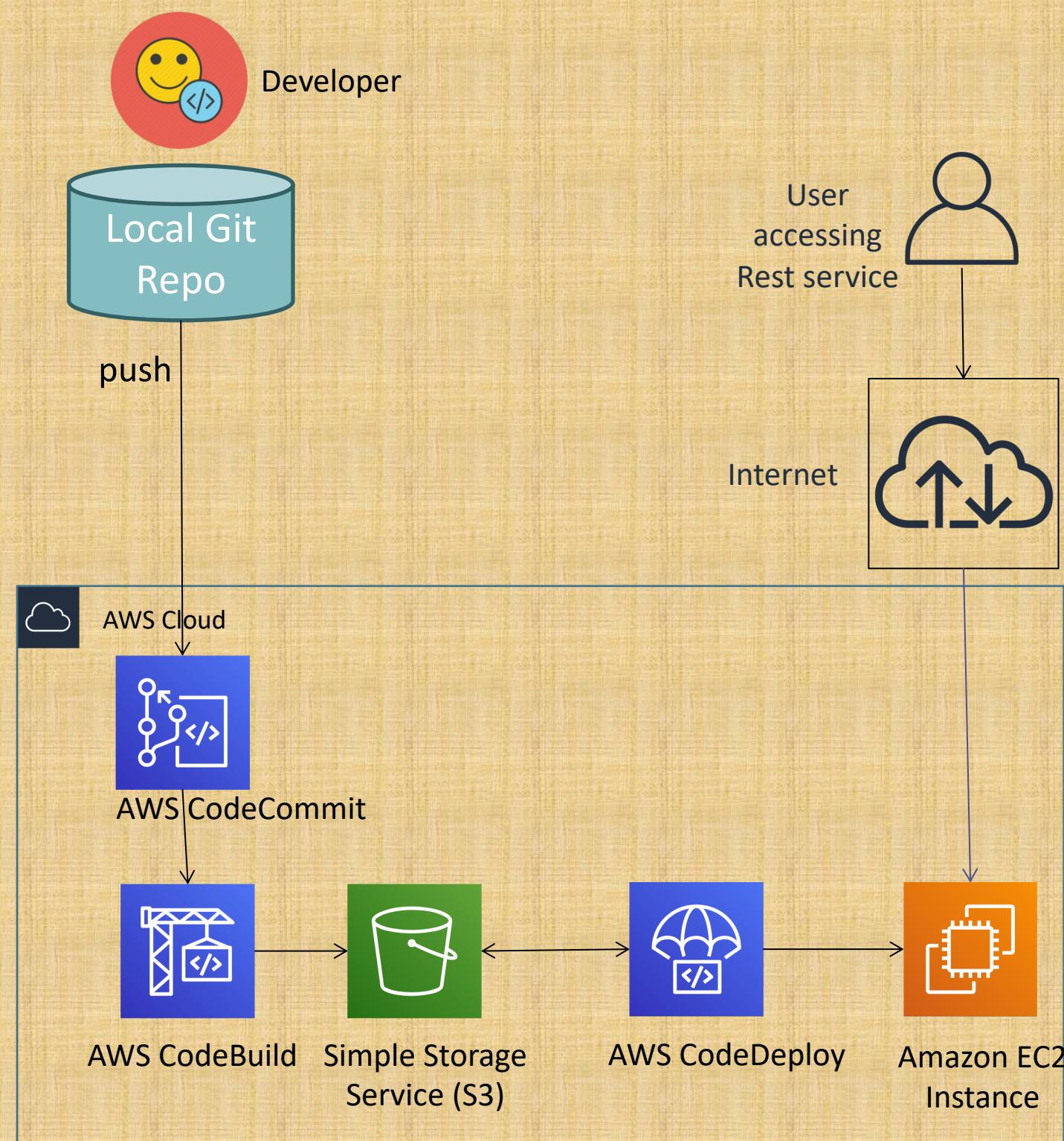
- Benefits
 - We can rapidly release new features.
 - Update AWS Lambda function versions.
 - Avoid downtime during application deployment.
 - Reduces the complexity of updating applications when compared to error-prone manual deployments.
 - Service scales with our infrastructure so we can easily deploy to one instance or thousands.

CodeDeploy - When compute is EC2/On-Premise



CodeDeploy - Steps

- Step#1: Create CodeDeploy pre-requisite roles
 - Create a service role for codeDeploy.
 - Create an IAM Instance profile.
- Step#2: Create a EC2 VM
 - Create EC2 VM
 - During creation associate IAM instance profile.
 - Discuss about “Userdata” containing tomcat and codeDeploy Agent
- Step#3: Create codeDeploy objects
 - Create Application
 - Create Deployment Group
 - Create Deployment
- Step#4: Create codeDeploy files and scripts
 - Create appspec.yml
 - Create scripts (before_install script, after_install script, Start up script, Shutdown script) and check-in
- Step#5: Run CodeBuild and Create Deployment
- Step#6: Verify Deployment
 - Verify the deployment Events
 - Verify the tomcat deployment
 - Verify the codeDeploy agent log
 - Verify by accessing app
- Step#7: New App Release: Make change to Application and re-deploy



Author: Nho Luong

Skill: DevOps Engineer Lead

AWS CodeDeploy using CloudFormation

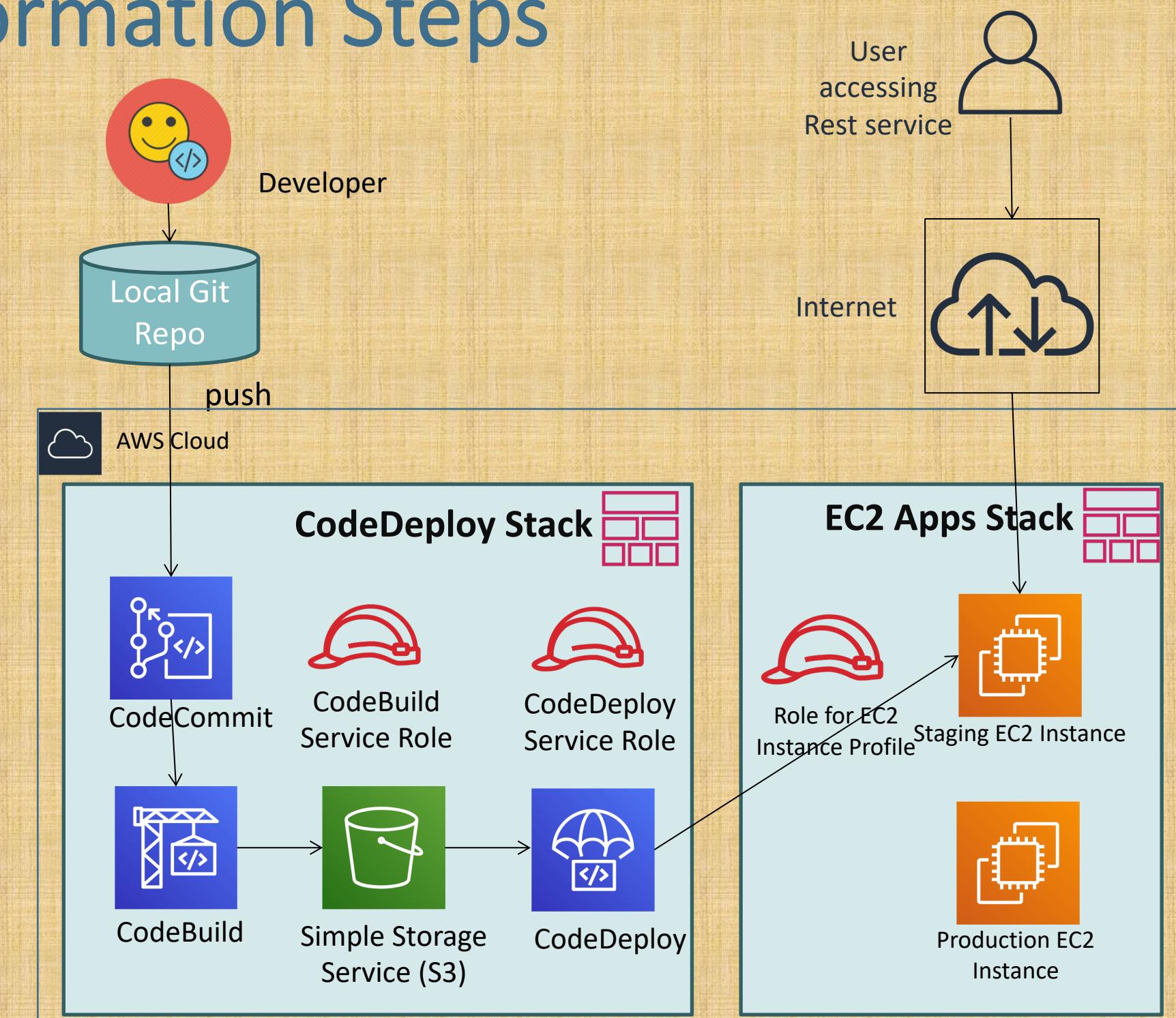


Author: Nho Luong

Skill: DevOps Engineer Lead

CodeDeploy – CloudFormation Steps

- Step 1 : Discuss about [appspec.yml](#) and [scripts](#).
- Step 2: Create CodeDeploy service role
- Step 3: Create CodeDeploy Application
- Step 4: Create CodeDeploy Deployment Group and also change CodeBuild packaging to ZIP.
- Step 5: Create stack and verify the following
 - Application
 - Deployment Group
- Step 6: Create CodeDeploy Deployment
 - Create Deployment Object
 - Run CodeBuild and Verify S3 for ZIP
 - Update Stack
 - Verify Deployment
 - Access Application

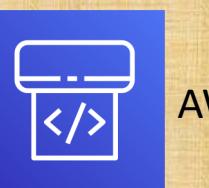


AWS CodePipeline



Author: Nho Luong

Skill: DevOps Engineer Lead



AWS CodePipeline



AWS CodeCommit



Amazon EC2 Container Registry



Simple Storage Service (S3)



GitHub



AWS CodeBuild



Jenkins



AWS CloudFormation



AWS CodeDeploy



AWS Elastic Beanstalk



AWS Service Catalog



Amazon Elastic Container Service



Amazon Elastic Container Service (Blue/Green)



Simple Storage Service (S3)



Amazon CloudWatch



GitHub Webhooks

Source

Author: Nho Luong

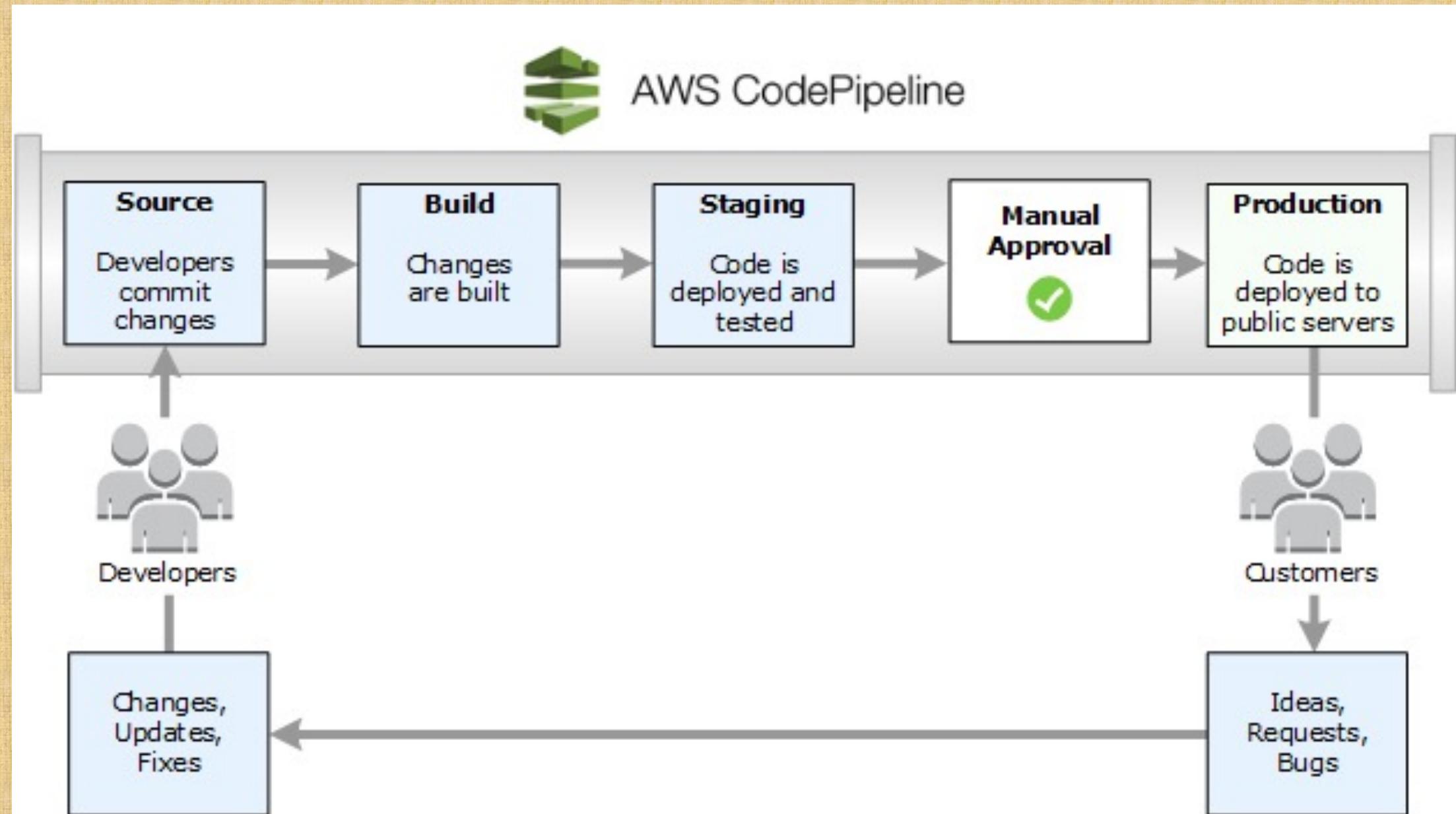
Skill: DevOps Engineer Lead

Build

Deploy

Monitor Source Changes

Continuous Delivery

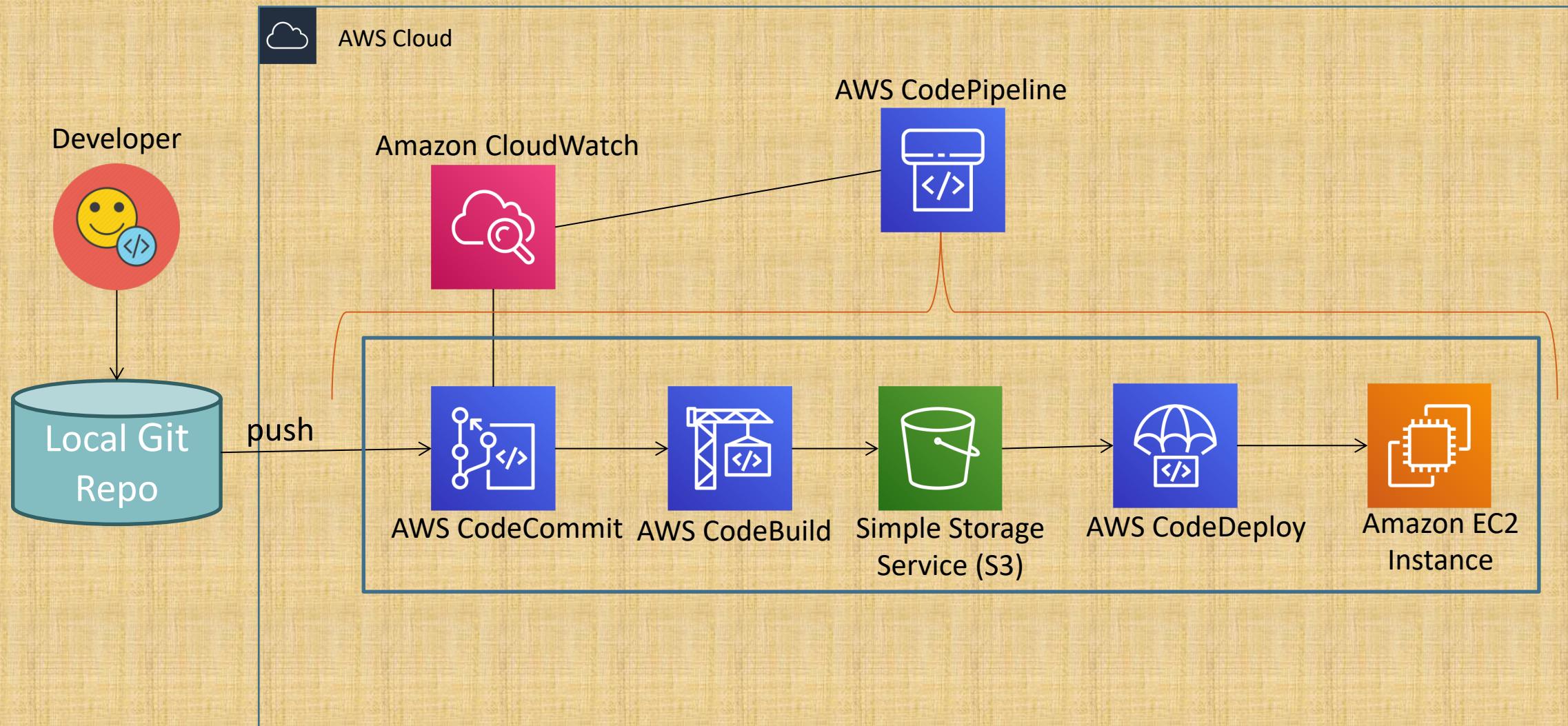


CodePipeline - Introduction

- AWS CodePipeline is a continuous delivery service to model, visualize, and automate the steps required to release your software.
- Benefits
 - Automate your release processes.
 - Establish a consistent release process.
 - Speed up delivery while improving quality.
 - Supports external tools integration for source, build and deploy.
 - View progress at a glance
 - View pipeline history details.

CodePipeline - Steps

- Step#1: Create Pipeline
 - Artifacts: S3
 - Source: CodeCommit
 - Build: CodeBuild
 - Deploy: CodeDeploy
 - Server: EC2 Instance
- Step#2: Make changes & Check-In Code
 - Make changes to rest app and check-in
 - Pipeline should trigger the build automatically.



CodePipeline – Manual Approval & Prod Deployment

- Step#1: Create new EC2 Instance with tag name as prod
- Step#2: Create new deployment group for prod
- Step#3: Create [Manual Approval stage](#) in CodePipeline
- Step#4: Create [Prod Deployment stage](#) in CodePipeline .
- Step#5: Check-in changed code to trigger pipeline and monitor the pipeline process.

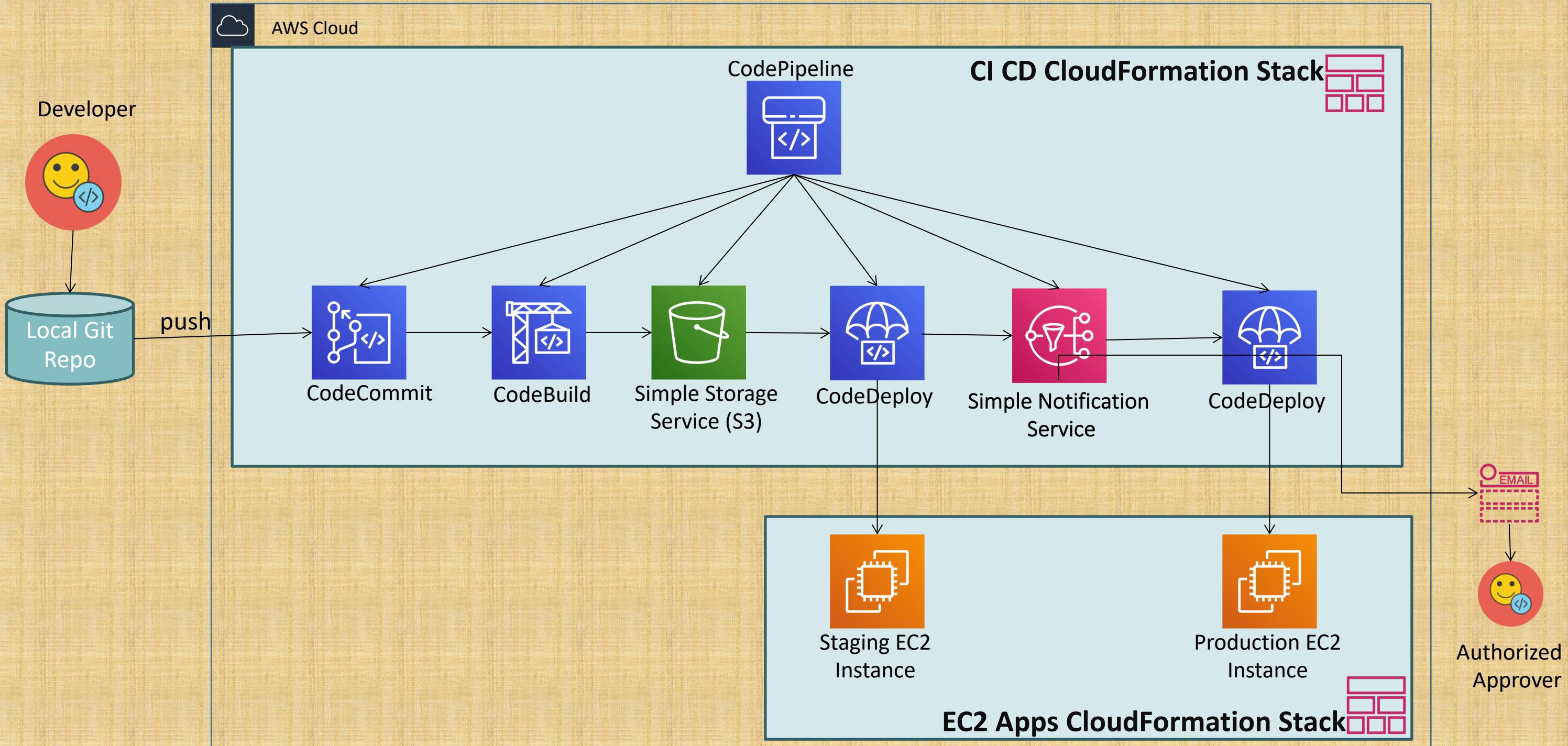
AWS CodePipeline using CloudFormation



Author: Nho Luong

Skill: DevOps Engineer Lead

Continuous Integration & Continuous Delivery using CloudFormation



CodePipeline – CloudFormation Steps

- Step 1: Create CodePipeline role
- Step 2: Create Pipeline stages for staging deployment
 - Stage 1: Source Stage
 - Stage 2: Build Stage
 - Stage 3: Deploy To Staging
- Step 3: Create stack and verify the following
 - **Stages:** Source, Build, Deploy to Staging
 - Access Application in staging
- Step 4: From IDE make changes to rest app and check-in code and verify the following
 - **Stages:** Source, Build, Deploy to Staging
 - Access Application in staging

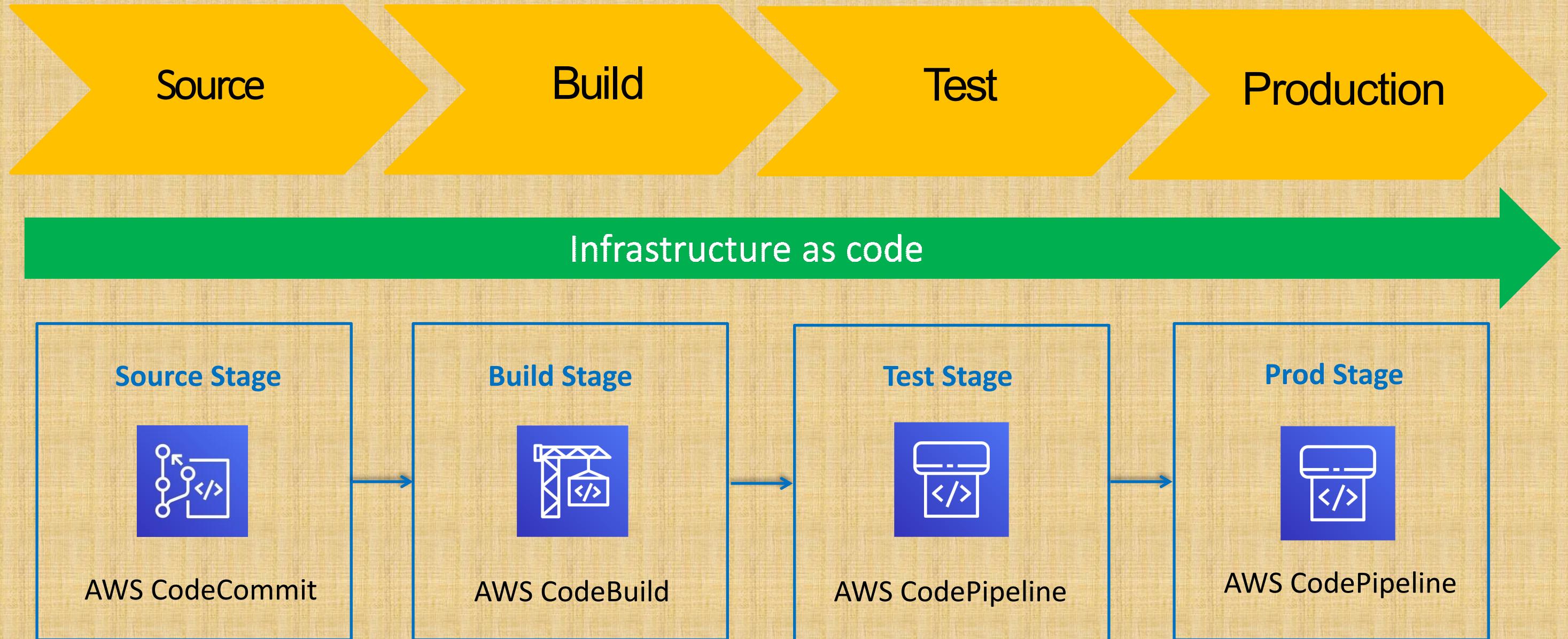
CodePipeline – CloudFormation Steps

- Step 5: Create SNS Topic and its equivalent parameter and add Production DeploymentGroup
- Step 6: Create Pipeline stages for Production deployment
 - Stage 4: Production email Approval
 - Stage 5: Deploy To Production
- Step 7: Create stack and verify the following
 - Confirm SNS Subscription in email
 - Stages: Source, Build, Deploy to Staging, Production email approval and Deploy to production.
 - Access Application in staging and production
- Step 8: From IDE make changes to rest app and check-in code and verify the following
 - Stages: Source, Build, Deploy to Staging, Production email approval and Deploy to production.
 - Access Application in staging and production

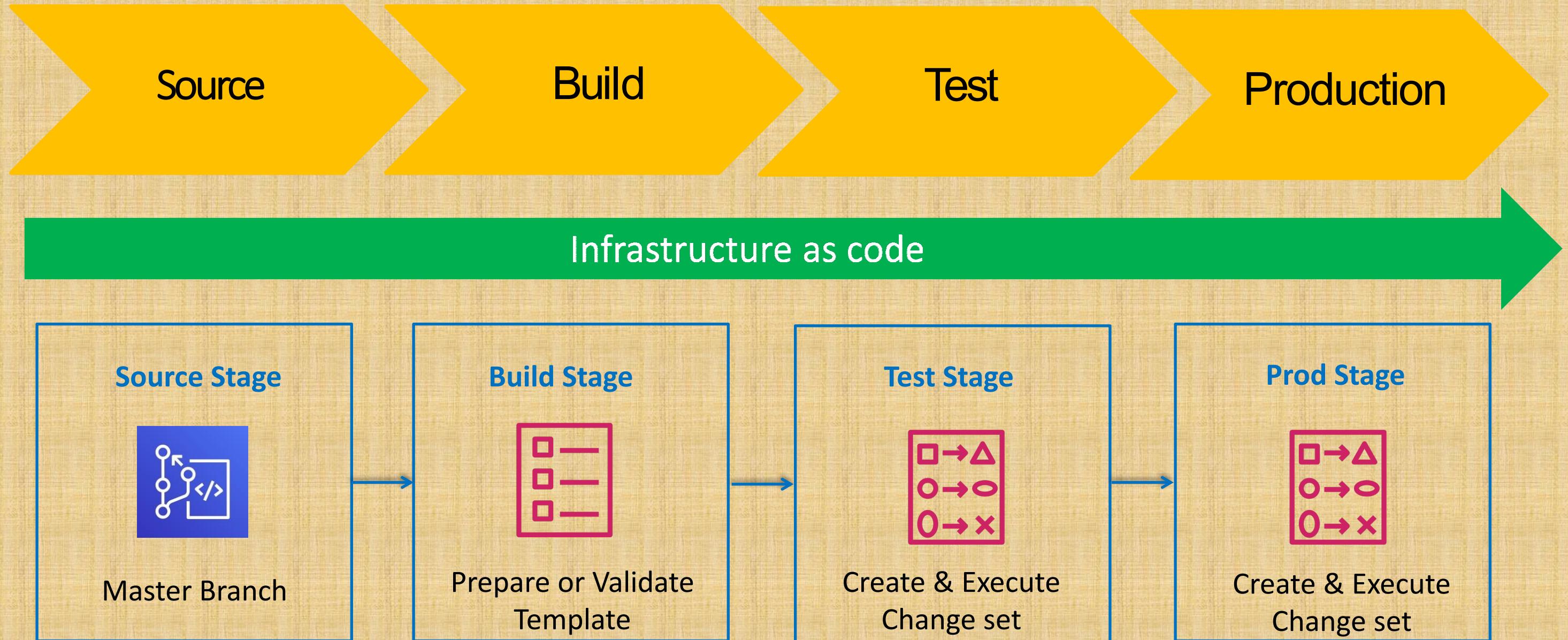
Infrastructure as Code using **AWS CloudFormation** and **AWS Web Console**



Infrastructure as Code



Infrastructure as Code



Infrastructure as Code



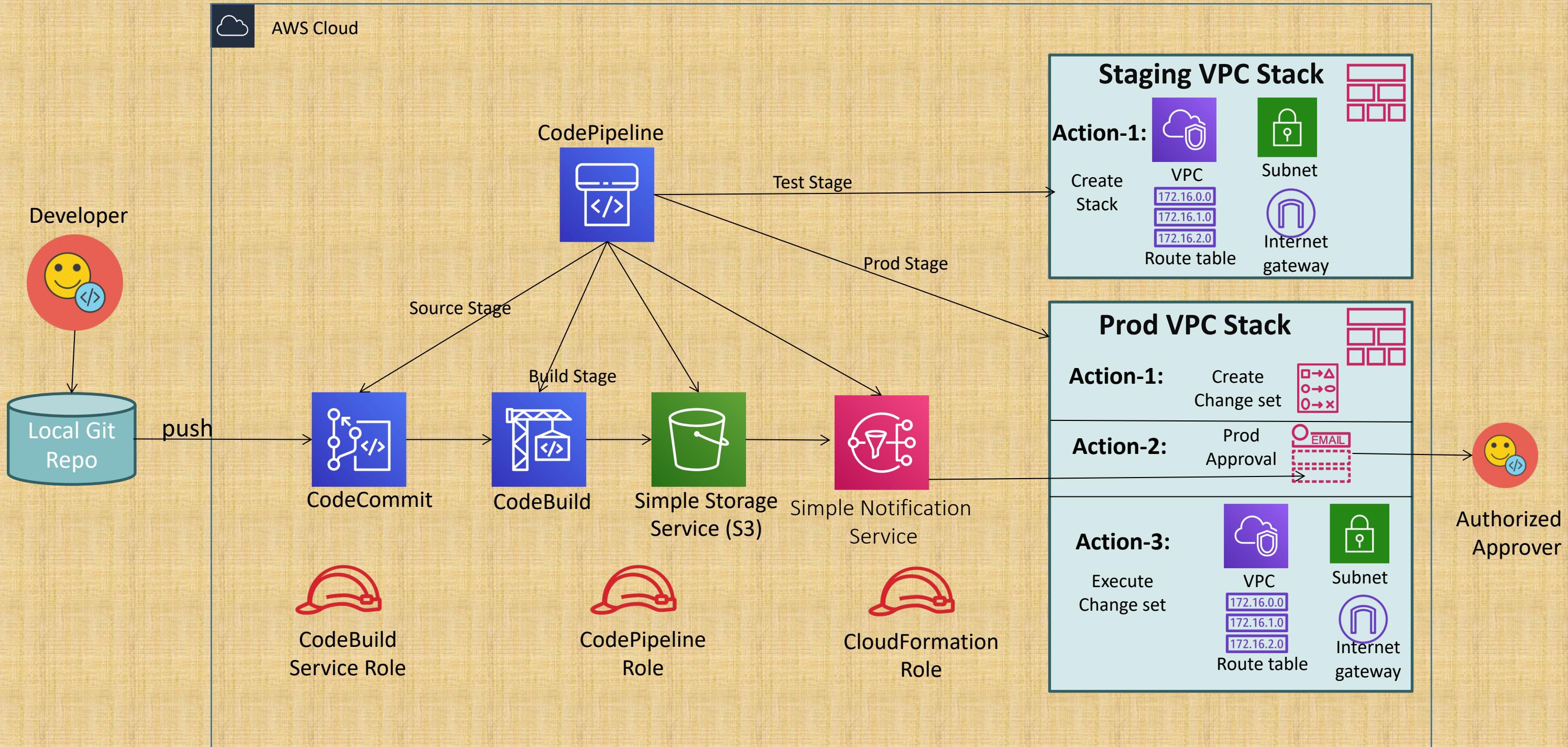
Benefits

- Track Infrastructure changes using version control system like [AWS CodeCommit](#).
- Release infrastructure changes using the same tools as code changes ([AWS CodeCommit](#), [CodeBuild](#) and [CodePipeline](#)).
- Replicate production environment in any environment as desired for continuous testing.
- Make infrastructure changes repeatable.
- Minimize infrastructure buildout time.
- Seamless provisioning and de-provisioning of infrastructure resources in minutes or even reduced to seconds.

Author: Nho Luong

Skill: DevOps Engineer Lead

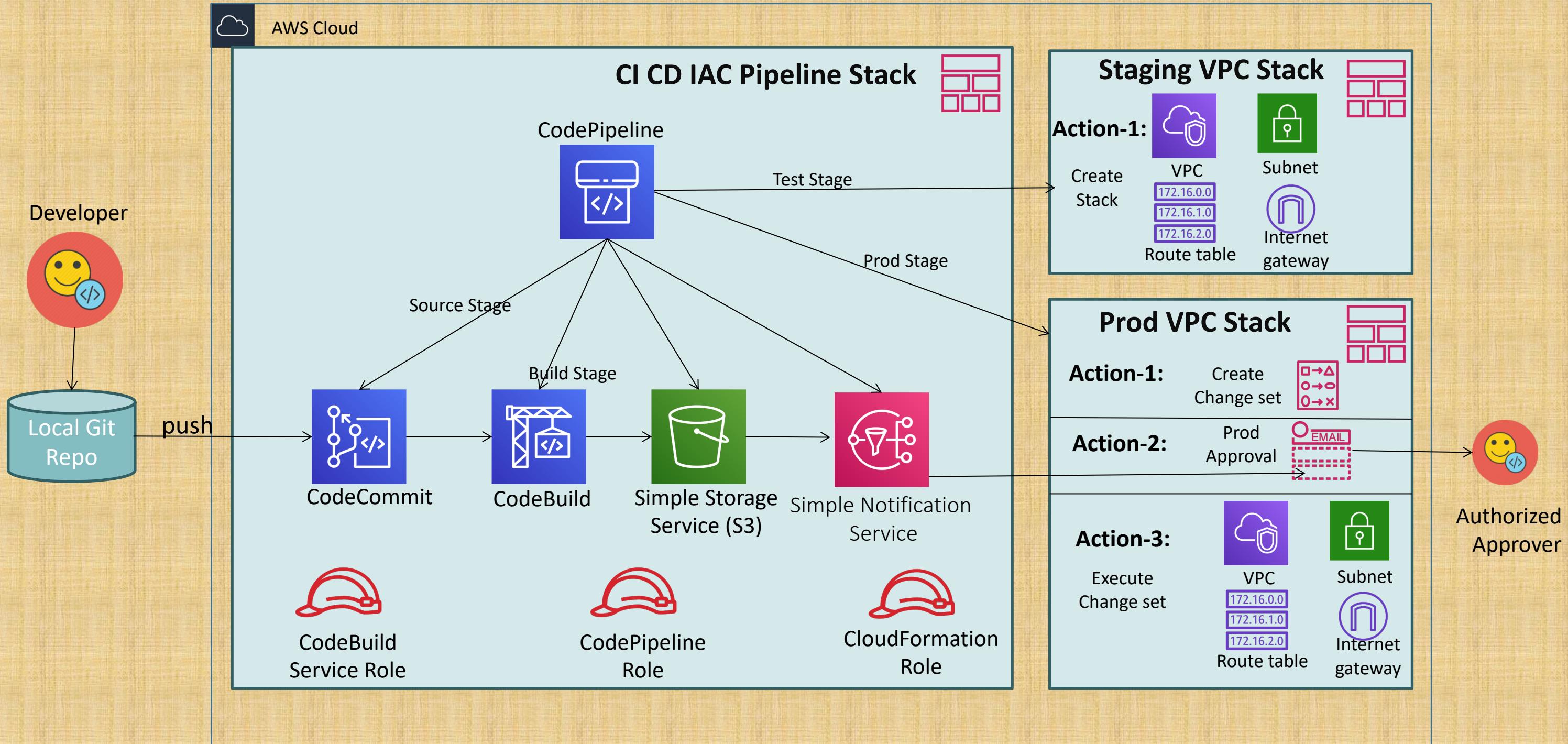
Infrastructure as Code – Manual AWS Web Console



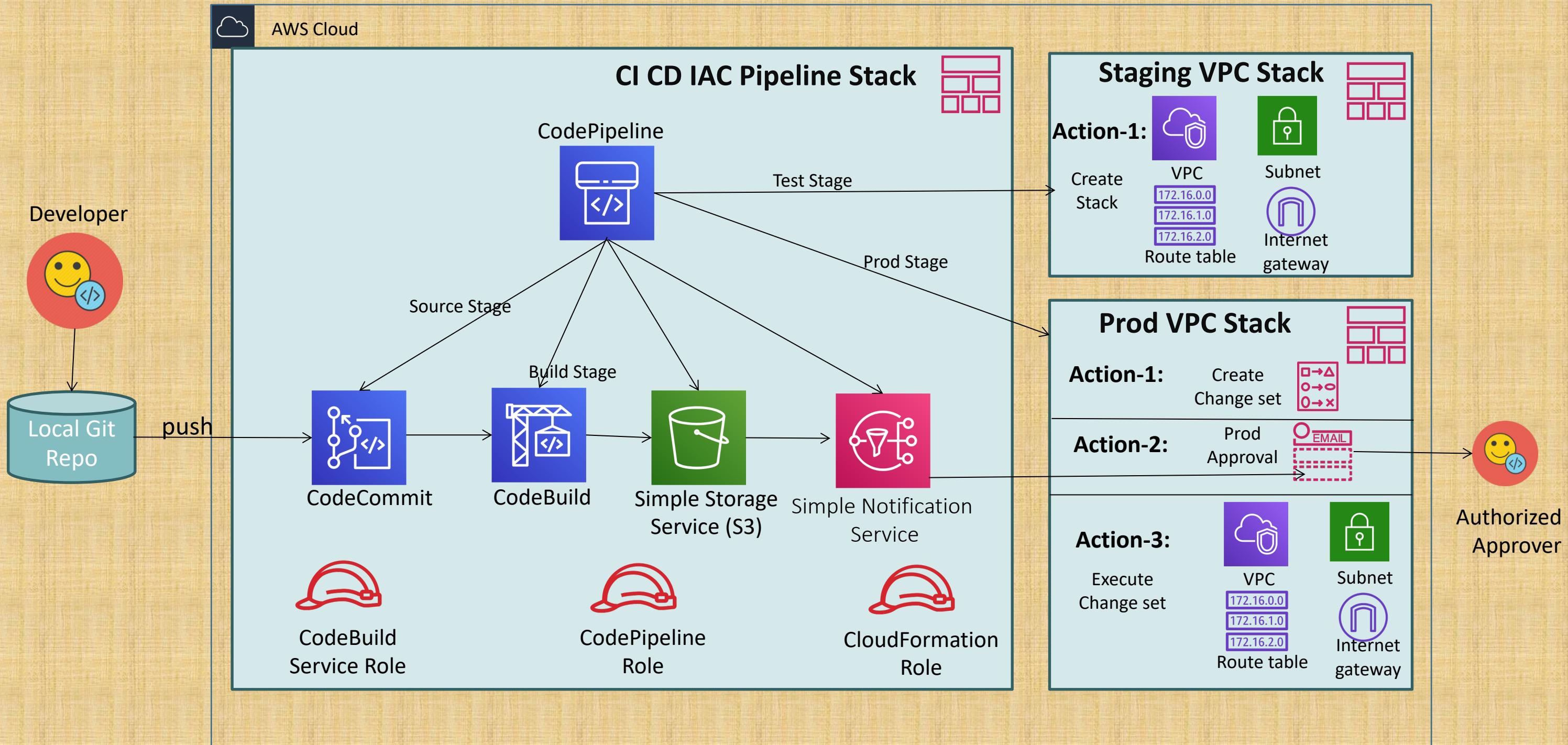
Author: Nho Luong

Skill: DevOps Engineer Lead

Infrastructure as Code – CFN Template creation Flow



Infrastructure as Code – Execution Flow



Author: Nho Luong

Skill: DevOps Engineer Lead

Infrastructure as Code using AWS Developer Tools on AWS Web Console



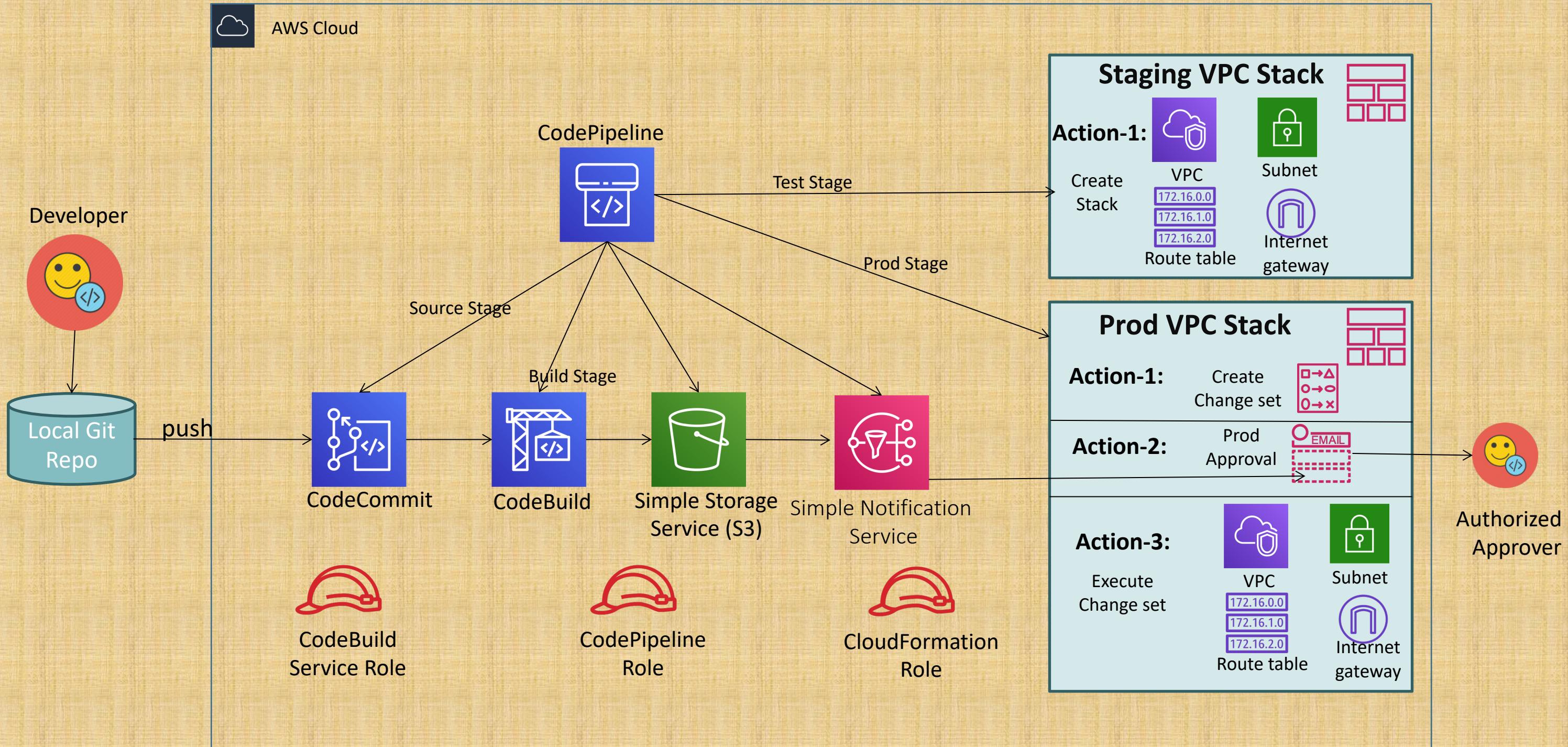
Author: Nho Luong

Skill: DevOps Engineer Lead

Usecase

- We will create a [VPC CloudFormation stack](#) using AWS CodePipeline.
- Manage the AWS VPC infrastructure components like Subnets, Routes, Route Tables everything with [AWS Developer Tools](#) and [CloudFormation](#).
- Any changes to vpc infra, we will change the `vpc.yml` cfn template and check-in the code to AWS CodeCommit.
- CodePipeline will trigger pipeline and push the changes to [staging VPC Stack](#).
- CodePipeline creates a [Change Set](#) for production.
- Approve the Change set using [SNS notification](#)
- Changes will be pushed to [production VPC Stack](#) after approval.
- Finally we will achieve [Continuous Integration](#), [Continuous Delivery](#) & [Infrastructure as code](#) after this usecase implementation.

Infrastructure as Code – Manual AWS Web Console



Author: Nho Luong

Skill: DevOps Engineer Lead

Infrastructure as Code – Manual AWS Web Console

- Step 1: Understand about the VPC source files.
 - `vpc.yml`
 - `vpc-config.json`
 - `buildspec.yml`
- Step 2: Create CodeCommit repository `vpcrepo` and check-in vpc source files.
- Step 3: Create a pipeline with source and build stages.
 - Build stage fails and we will understand the causes of failure and fix it.
 - Create a CFN policy with validate CFN template permission and associate to `CodeBuildRole`.
 - Rerun the pipeline.
 - Build stage should pass now.
- Step 4: Create a Test Stage which creates `TestVPCStack`
 - Role#1: Create CFN full access policy and associate it with `CodePipeline` role
 - Role#2: Create CFN VPC full access Role by associating “[VPC Full Access policy](#)” which is required by CloudFormation to create the VPC Stack.
 - Create Pipeline stage named [VPCTest](#)
 - Click “[Release Change](#)” after stage creation and verify the stack got created in CFN Console.

Infrastructure as Code – Manual AWS Web Console

- Step 5: Create Prod Stage
 - Pre-requisite: Create SNS Topic
 - Action#1: Create Prod Change Set
 - Action#2: Create Prod Approval
 - Action#3: Create Execute Change Set
 - Verify the stack got created in CloudFormation console.
- Step 8: Update `vpc.yml` with new subnet (`subnet02`) and check-in file to CodeCommit
 - Verify the pipeline stages
 - Source, Build, VPCStage
 - VPCProd
 - Create Change Set
 - Prod Approval
 - Execute Change Set
 - Verify the same in VPC

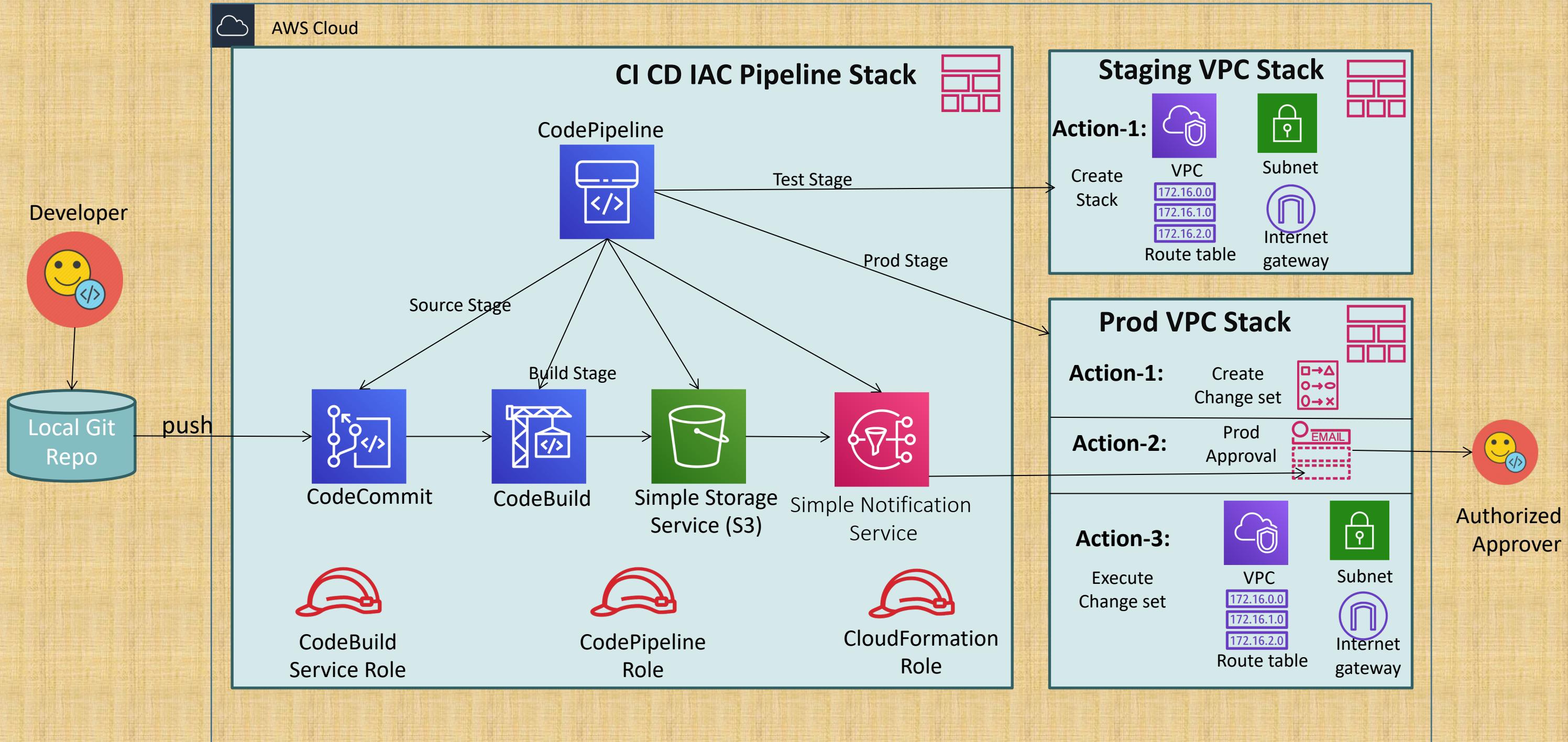
Infrastructure as Code using AWS CloudFormation



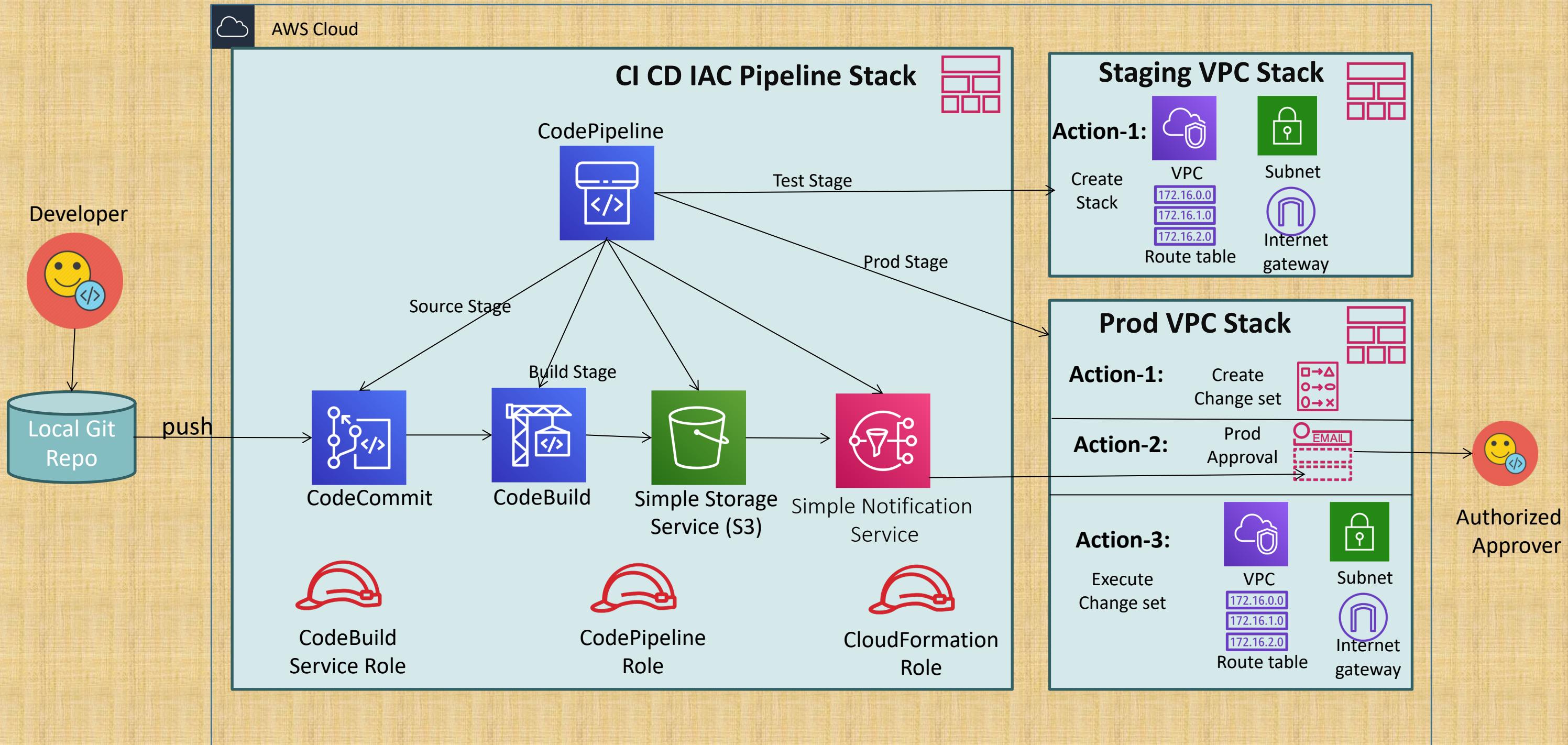
Author: Nho Luong

Skill: DevOps Engineer Lead

Infrastructure as Code – CFN Template creation Flow



Infrastructure as Code – Execution Flow



Infrastructure as Code – CloudFormation Pipeline

- Step 1: Understand about the VPC source files.
 - vpc.yml
 - vpc-config.json
 - buildspec.yml
- Step 2: Create CodeCommit repository vpcrepo and check-in vpc source files. (Note: check-in base vpc.yml and vpc-config.json)
- Step 3: Create a CodeBuild related template objects
 - Create Parameters (Repo Name, Artifact storage bucket)
 - Create CodeBuild Role
 - Create CodeBuild Project
- Step 4: Create Other roles
 - Create CodePipeline Role
 - Create CloudFormation Role

Infrastructure as Code – CloudFormation Pipeline

- Step 5: Create Pipeline stages
 - Source Stage
 - Build Stage
 - Test Stage
- Step 6: Create Pipeline stage and actions for production
 - Create SNS Topic Resource
 - Create Parameter for email notifications.
 - Prod Stage
 - Action-1: Create Change Set
 - Action-2: Prod Approval
 - Action-3: Execute Change Set
- Step 7: Add subnet02 in vpc.yml and verify the pipeline end to end.



Thank You