

## Advance Devops Practical Examination: AWS Case Study Assignment

### Topic : Serverless Image Processing Workflow

#### 1. Introduction

**Concepts Used:** AWS Lambda, S3, and CodePipeline.

**Problem Statement:** "Create a serverless workflow that triggers an AWS Lambda function when a new image is uploaded to an S3 bucket. Use CodePipeline to automate the deployment of the Lambda function."

#### Tasks:

- Create a Lambda function in Python that logs and processes an image when uploaded to a specific S3 bucket.
- Set up AWS CodePipeline to automatically deploy updates to the Lambda function.
- Upload a sample image to S3 and verify that the Lambda function is triggered and logs the event.

#### Case Study Overview:

The case study involves building a **serverless image processing workflow** using AWS services such as Lambda, S3, and CodePipeline. The goal is to create a serverless workflow that automatically triggers a Lambda function when a new image is uploaded to an S3 bucket. Additionally, the project sets up CodePipeline to automate the deployment of Lambda function updates.

#### Key Feature and Application:

The case study demonstrates how AWS Lambda can process events triggered by S3 uploads, facilitating **real-time image processing** without manual intervention. The project also highlights how CodePipeline automates Lambda function deployment, ensuring **continuous integration and continuous delivery (CI/CD)**.

#### Guidelines & Best Practices

1. **Use Least Privilege:** Restrict IAM roles to necessary permissions only.
2. **Modular Code:** Keep Lambda functions focused and maintainable.
3. **Efficient Triggers:** Set specific S3 event types to avoid redundant triggers.
4. **Error Handling:** Implement logging and error management in Lambda using CloudWatch.

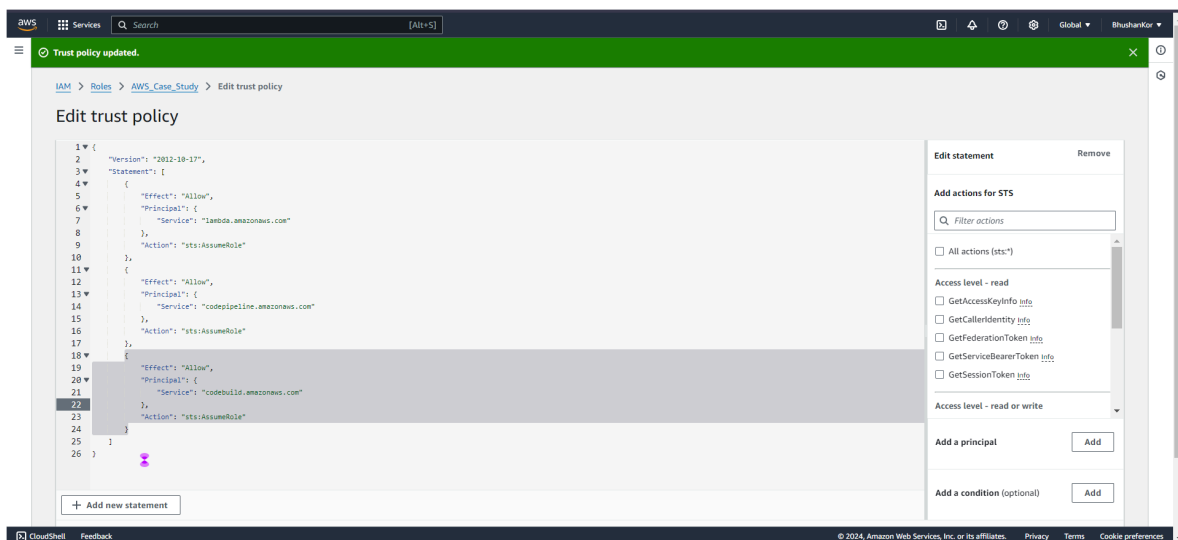
5. **Automate CI/CD:** Use CodePipeline and CodeBuild to update Lambda automatically.
6. **Optimize Costs:** Use AWS Free Tier and manage resource use smartly.

## 2. Step-by-Step Explanation

### 1. Create an IAM Role :

- Log in to the **AWS Management Console**.
- Go to the **IAM Console** in AWS.
- Click on **Roles** and choose **Create Role**.
- Select **AWS service** as the trusted entity type, and choose **Lambda** as the use case.
- Attach policies such as:
  - **AmazonS3FullAccess**
  - **AmazonCodeBuildAdminAccess**
  - **AmazonCodePipeline\_FullAccess**
  - **AWSLambda\_FullAccess**
  - **CloudWatchLogsFullAccess**
- Give the role a name: **AWS\_Case\_Study** and create it.
- In Trust Relationship/Policy attach below policy in Statement.

```
{
  "Effect": "Allow",
  "Principal": {
    "Service": "codepipeline.amazonaws.com"
  },
  "Action": "sts:AssumeRole"
},
{
  "Effect": "Allow",
  "Principal": {
    "Service": "codebuild.amazonaws.com"
  },
  "Action": "sts:AssumeRole"
}
```



## 2. Create an S3 Bucket:

- Navigate to the **S3** service.
- Click on **Create Bucket**.
  - Bucket Type: **General Purpose**.
  - Provide a unique bucket name. (**bhushan-aws-bucket**)
  - **Uncheck the Block all public access option.**
  - Keep Rest of the things to default.
- You can upload a test image to ensure the bucket is working properly.

**Create bucket** [Info](#)

Buckets are containers for data stored in S3.

**General configuration**

AWS Region  
US East (N. Virginia) us-east-1

Bucket type [Info](#)

☒ **General purpose**  
Recommended for most use cases and access patterns. General purpose buckets are the original S3 bucket type. They allow a mix of storage classes that redundantly store objects across multiple Availability Zones.

☐ **Directory**  
Recommended for low-latency use cases. These buckets use only the S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.

Bucket name [Info](#)  
bhushan-aws-bucket

Bucket name must be unique within the global namespace and follow the bucket naming rules. [See rules for bucket naming](#)

Copy settings from existing bucket - optional  
Only the bucket settings in the following configuration are copied.

Format: s3://bucket/prefix

**Object Ownership** [Info](#)

Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

☒ **ACLs disabled (recommended)**  
All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.

☐ **ACLs enabled**  
Objects in this bucket can be owned by other AWS accounts. Access to this bucket and its objects can be specified using ACLs.

☒ **ACLs disabled (recommended)**  
All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.

☐ **ACLs enabled**  
Objects in this bucket can be owned by other AWS accounts. Access to this bucket and its objects can be specified using ACLs.

**Object Ownership**  
Bucket owner enforced

**Block Public Access settings for this bucket**

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. [Learn more](#)

☐ **Block all public access**  
Turning this setting on is the same as turning on all four settings below. Each of the following settings are independent of one another.

☐ **Block public access to buckets and objects granted through new access control lists (ACLs)**  
S3 will block public access permissions applied to newly added buckets or objects, and prevent the creation of new public access ACLs for existing buckets and objects. This setting doesn't change any existing permissions that allow public access to S3 resources using ACLs.

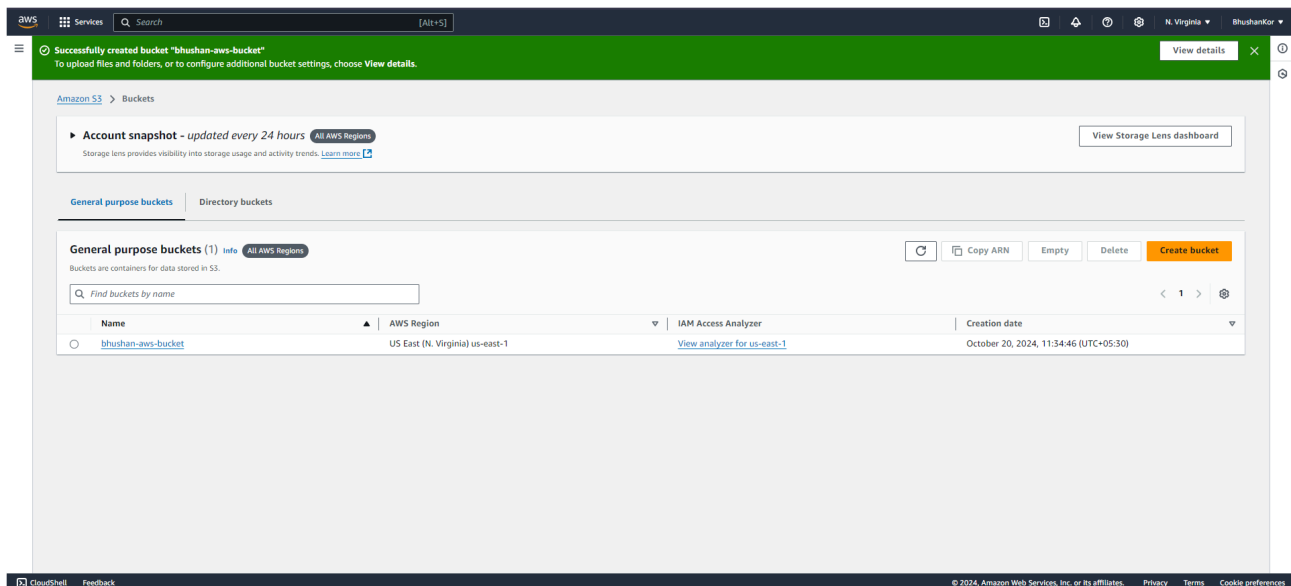
☐ **Block public access to buckets and objects granted through any access control lists (ACLs)**  
S3 will ignore all ACLs that grant public access to buckets and objects.

☐ **Block public access to buckets and objects granted through new public bucket or access point policies**  
S3 will block new bucket and access point policies that grant public access to buckets and objects. This setting doesn't change any existing policies that allow public access to S3 resources.

☐ **Block public and cross-account access to buckets and objects through any public bucket or access point policies**  
S3 will ignore public and cross-account access for buckets or access points with policies that grant public access to buckets and objects.

**Warning:** Turning off block all public access might result in this bucket and the objects within becoming public.  
AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.

☒ I acknowledge that the current settings might result in this bucket and the objects within becoming public.



### 3. Create a Lambda Function:

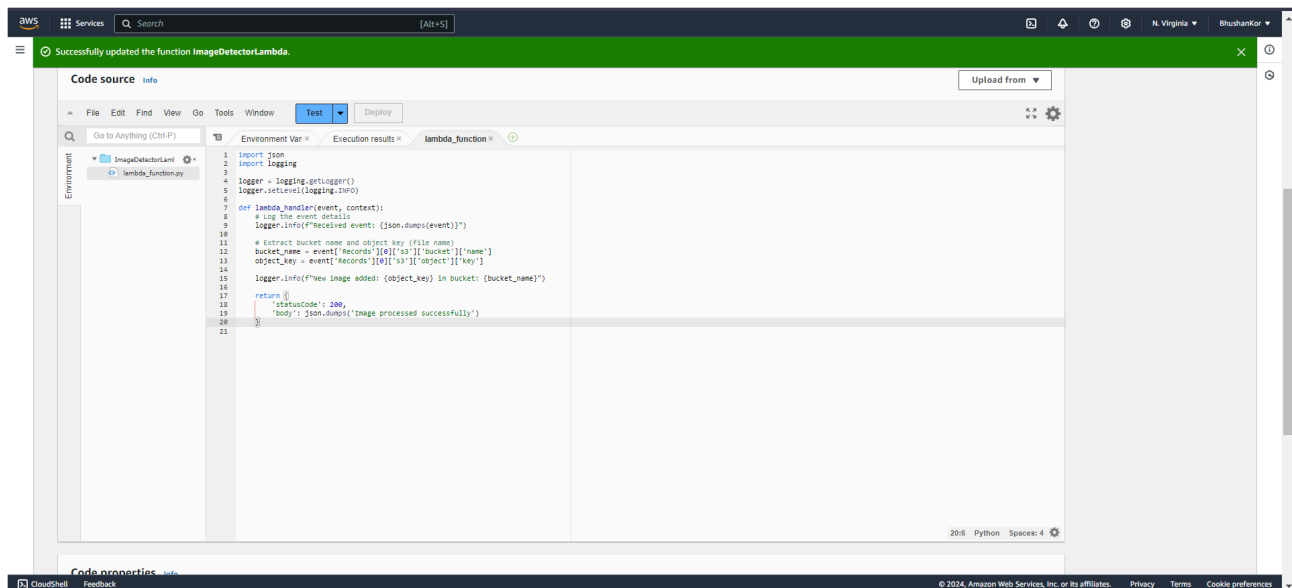
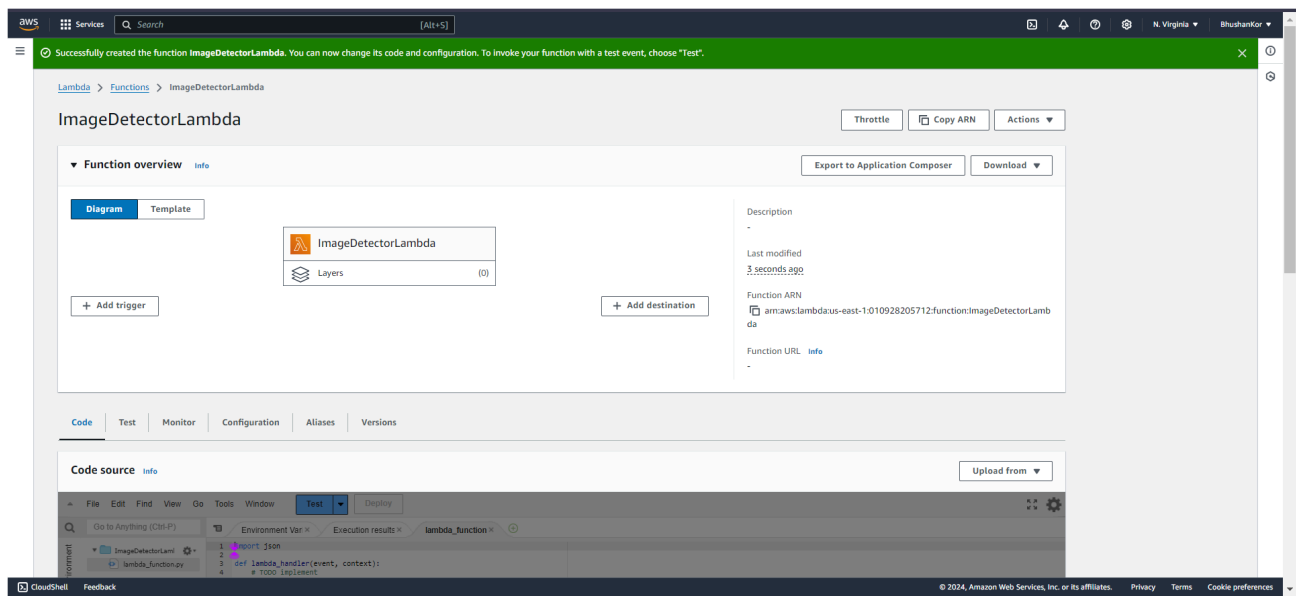
- Navigate to the **Lambda Console**.
- Click **Create Function**.
  - Choose **Author from scratch**.
  - Provide a function name **ImageDetectorLambda**.
  - Choose a free-tier eligible runtime **Python 3.12**.
  - Assign the **IAM role** created earlier **AWS\_Case\_Study**.
  - Keep Rest of things to default.
- In the **Function Code** section, add the logic to log when the image is uploaded to S3.

#### Code:

```
import json
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def lambda_handler(event, context):
    # Log the event details
    logger.info(f"Received event: {json.dumps(event)}")
    # Extract bucket name and object key (file name)
    bucket_name = event['Records'][0]['s3']['bucket']['name']
    object_key = event['Records'][0]['s3']['object']['key']
    logger.info(f"New image added: {object_key} in bucket: {bucket_name}")
    return {
        'statusCode': 200,
```

```
'body': json.dumps('Image processed successfully.')
}
```

- Now Add the **Trigger** (Click on Trigger).
  - In Trigger Configuration Select S3.
  - Select Bucket which we have created. (**bhushan-aws-bucket**)
  - Select Event Types
    - All object create events
    - PUT
    - POST
    - COPY
    - Multipart upload completed



Name: Bhushan Mukund Kor

Academic Year: 2024-2025

Division: D15C

Roll No: 28

The screenshot displays the AWS Lambda console interface. The top navigation bar shows the user is logged in as 'BhushanKor' in the 'N. Virginia' region. The main content area is titled 'Add trigger' and shows the configuration for a new trigger on the 'ImageDetectorLambda' function. The trigger is configured for an S3 bucket named 's3/bhushan-aws-bucket' in the 'us-east-1' region. The event types selected are 'All object create events', 'PUT', 'POST', 'COPY', and 'Multipart upload completed'. The function overview section shows the function is successfully added and is now receiving events from the trigger. The function details include the function name 'ImageDetectorLambda', the layers '(0)', and the function ARN 'arn:aws:lambda:us-east-1:010928205712:function:ImageDetectorLambda'. The function URL is also provided.

- Now Let's test that Lambda is properly working by uploading image and checking logs.

The screenshot displays the AWS CloudWatch console interface. The top navigation bar shows the user is logged in as 'BhushanKor' in the 'N. Virginia' region. The main content area is titled 'Log events' and shows the logs for the 'ImageDetectorLambda' function. The logs include the event details, the function's response, and the execution metrics. The logs show the function successfully received an event from the S3 bucket and processed it. The logs include the event details, the function's response, and the execution metrics.

#### 4. Create a Github Repository:

- Create Repository with name **AWS-CodePipeline**.
- Add the file **buildspec.yml** and **lambda\_function.py**.

##### buildspec.yml

version: 0.2

phases:

install:

commands:

- pip install awscli # Ensure AWS CLI is available

build:

commands:

- echo "Packaging Lambda function..."

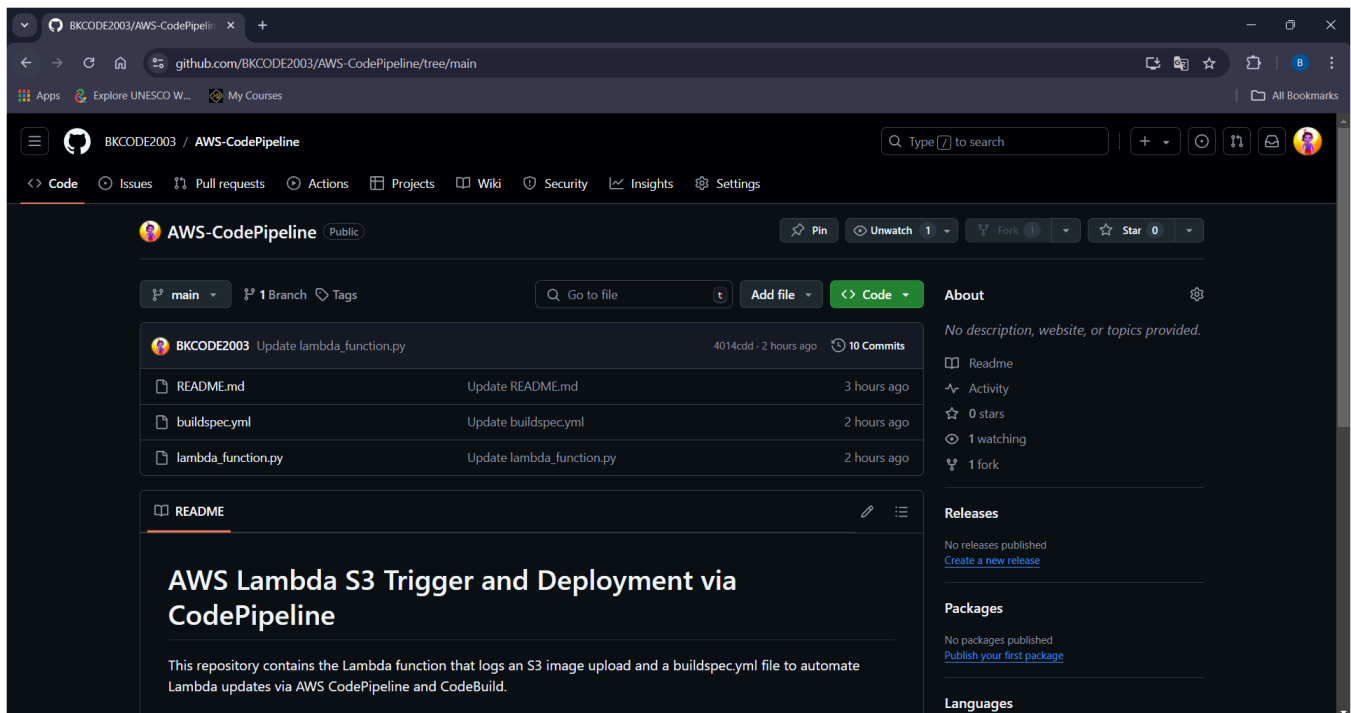
- zip -r lambda\_function.zip .

- echo "Updating Lambda function in AWS..."

- aws lambda update-function-code --function-name ImageDetectorLambda --zip-file

fileb://lambda\_function.zip

##### lambda\_function.py Same as in step 3



#### 5. Create a CodePipeline:

- Go to **AWS CodePipeline** in the AWS Management Console and create a new pipeline.

Name: **Bhushan Mukund Kor**

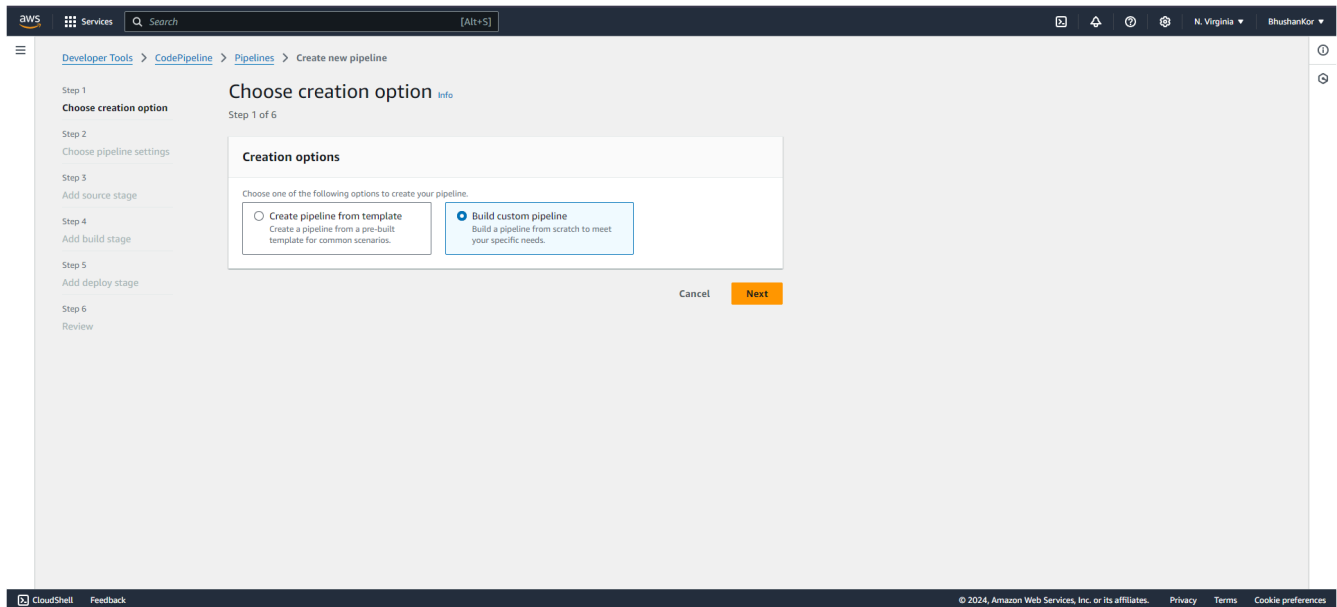
Academic Year: **2024-2025**

Division: **D15C**

Roll No: **28**

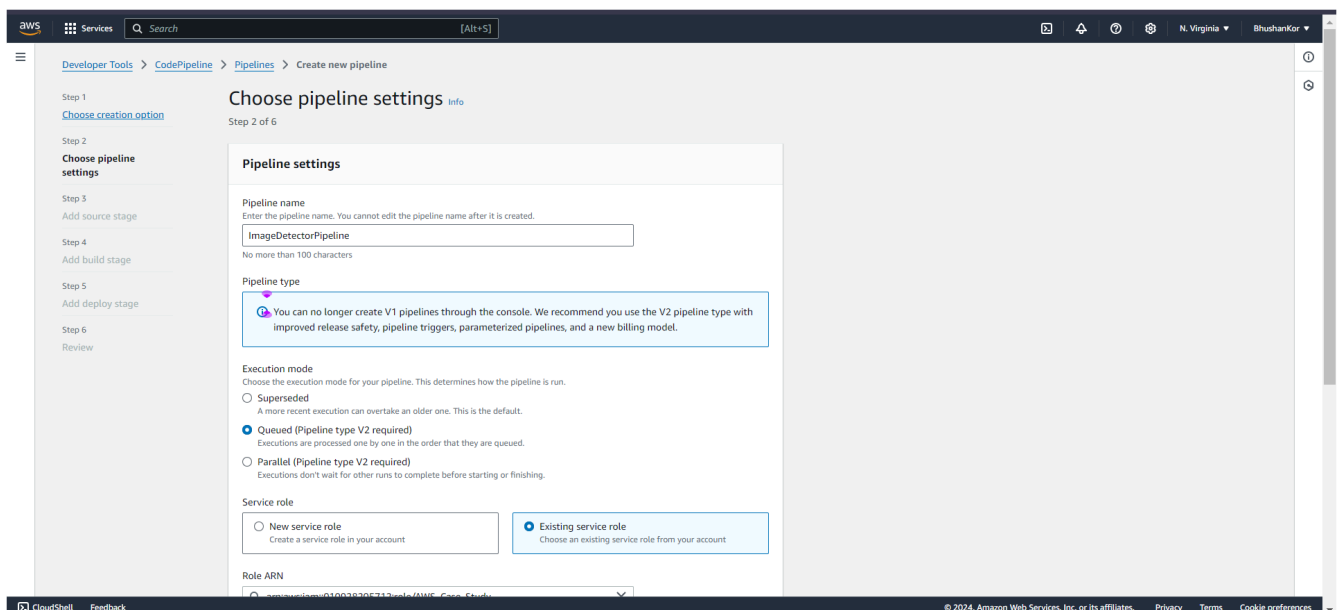
## Step 1: Choose Creation option.

- In creation option Select **Build Custom pipeline**.



## Step 2: Choose Pipeline Settings.

- Pipeline Name: **ImageDetectorPipeline** .
- Select Existing Role : **AWS\_Case\_Study** .
- Keep Rest all to default.



## Step 3: Add Source Storage.



- Source provider: **GitHub (Version 2)** .
- Connect Your account where you have created the repository. (**BKCODE2003**)
- Select Repository. (**BKCODE2003/AWS-CodePipeline**)
- Select Branch to **main**.
- In Trigger Just add **main** in include.
- Keep Rest all to default.

The screenshot shows the AWS CodePipeline console interface. The breadcrumb navigation is 'Developer Tools > CodePipeline > Pipelines > Create new pipeline'. The left sidebar shows the steps: Step 1: Choose creation option, Step 2: Choose pipeline settings, Step 3: Add source stage (selected), Step 4: Add build stage, Step 5: Add deploy stage, Step 6: Review. The main content area is titled 'Add source stage' and 'Step 3 of 6'. It contains the following sections: 'Source' with a dropdown for 'Source provider' set to 'GitHub (Version 2)'; a callout box for 'New GitHub version 2 (app-based) action'; 'Connection' with a dropdown showing 'arn:aws:codeconnections:us-east-1:010928205712:connection/6f08433b-a3...' and a 'Connect to GitHub' button; 'Repository name' with a dropdown showing 'BKCODE2003/AWS-CodePipeline'; 'Default branch' with a dropdown showing 'main'; and 'Output artifact format' with a dropdown.

## Step 4: Add Build Stage.

- Select **Other build Provider**.
- Select **AWS CodeBuild**.
  - Now Click on **Create Project**.
  - Project name: **Image\_Detector\_Build**
  - Enable public access.
  - Select Existing Role : **AWS\_Case\_Study** On 2 Places.
  - Buildspec: **Use a buildspec file** .
  - Buildspec name: **buildspec.yml** .
  - Keep Rest all to Default.
- Select Created Project.
- Keep Rest all to default.

Name: Bhushan Mukund Kor

Academic Year: 2024-2025

Division: D15C

Roll No: 28

The screenshot shows the 'Add build stage' step in the AWS CodePipeline console. The left sidebar lists steps from 1 to 6, with 'Add build stage' selected as Step 4. The main area is titled 'Add build stage' and 'Step 4 of 6'. It contains a 'Build - optional' section with a 'Build provider' dropdown set to 'AWS CodeBuild'. Below this is a 'Project name' field with 'Image\_Detector\_Build' and a 'Create project' button. A green success message states 'Successfully created Image\_Detector\_Build in CodeBuild.' There is also an 'Environment variables - optional' section with an 'Add environment variable' button. At the bottom, the 'Build type' is set to 'Single build'.

Step 5: Skip The Deploy Stage.

Step 6: Review the Pipeline and Click on Create.

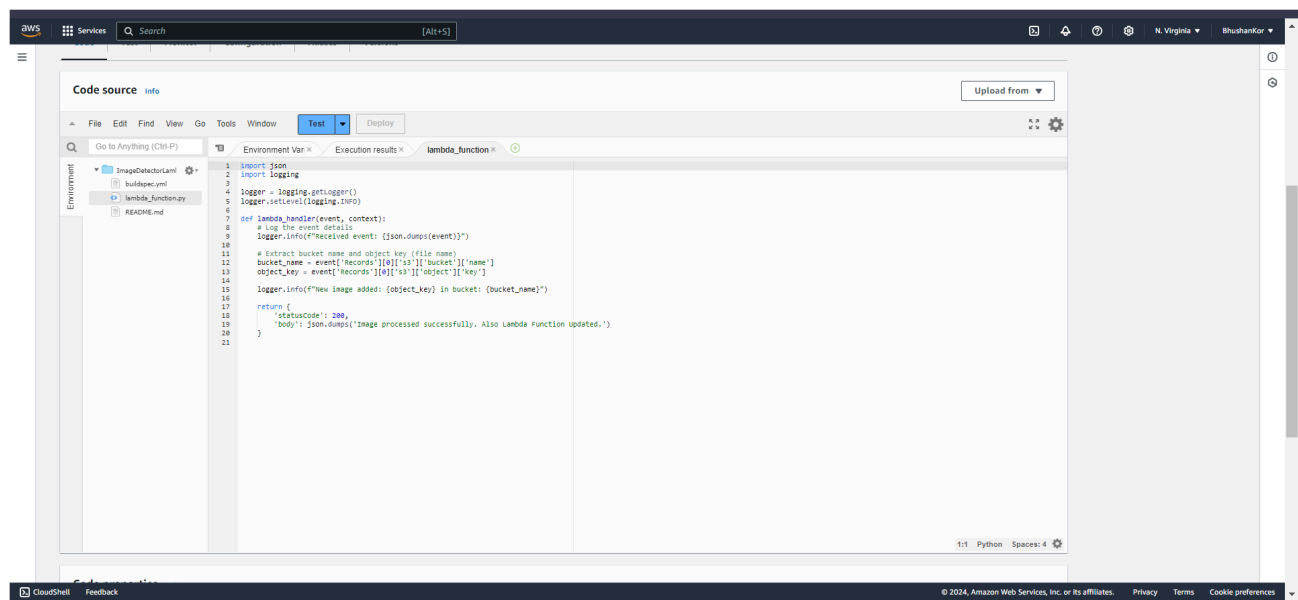
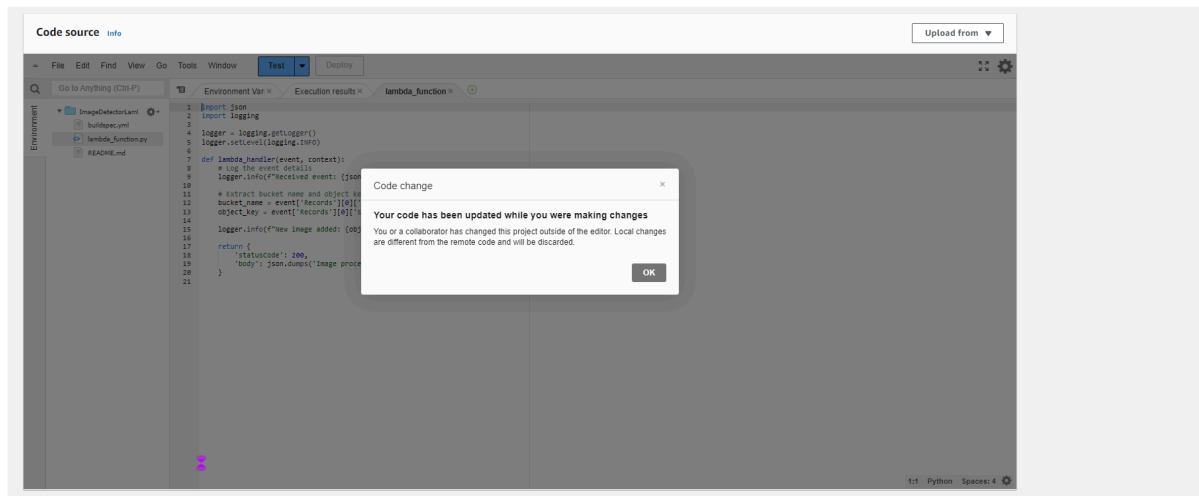
The screenshot shows the 'Review' step in the AWS CodePipeline console. The left sidebar lists steps from 1 to 6, with 'Review' selected as Step 6. The main area is titled 'Review' and 'Step 6 of 6'. It contains a 'Step 1: Choose pipeline settings' section with a 'Pipeline settings' table. The table lists the pipeline name as 'ImageDetectorPipeline', type as 'V2', execution mode as 'QUEUED', and artifact location as 'A new Amazon S3 bucket will be created as the default artifact store for your pipeline'. Below this is a 'Variables' table with the heading 'No variables'.

The screenshot shows the 'ImageDetectorPipeline' details in the AWS CodePipeline console. The left sidebar lists the pipeline's components, including 'Source', 'Build', 'Deploy', and 'Settings'. The main area shows the pipeline's status as 'QUEUED' and a list of stages. The 'Source' stage is successful, and the 'Build' stage is also successful. A green success message at the top states 'Congratulations! The pipeline ImageDetectorPipeline has been created.' There are buttons for 'Notify', 'Edit', 'Stop execution', 'Clone pipeline', and 'Release change'.

## 6. Test the Pipeline:

- Push/Update code to your GitHub repository.
- CodePipeline will trigger the CodeBuild project, which will package the code and update the Lambda function using the AWS CLI command `aws lambda update-function-code`.

### Lambda After push of new code in repository .



**Conclusion:** In this case study, we successfully implemented a serverless image processing workflow using AWS Lambda, S3, and CodePipeline. The solution demonstrated how Lambda functions can be triggered by S3 events, allowing real-time image processing without manual intervention. By leveraging CodePipeline, we automated the deployment of updates to the Lambda function, ensuring a streamlined CI/CD process. Testing confirmed that the workflow functions as expected, with images uploaded to S3 triggering the Lambda function, which logs the event and processes the image.