EXPERIMENT -05

CREATING LAMBDA FUNCTIONS USING THE AWS SDK FOR PYTHON

Objective

Creating Lambda Functions using the AWS SDK for Python.

Procedure

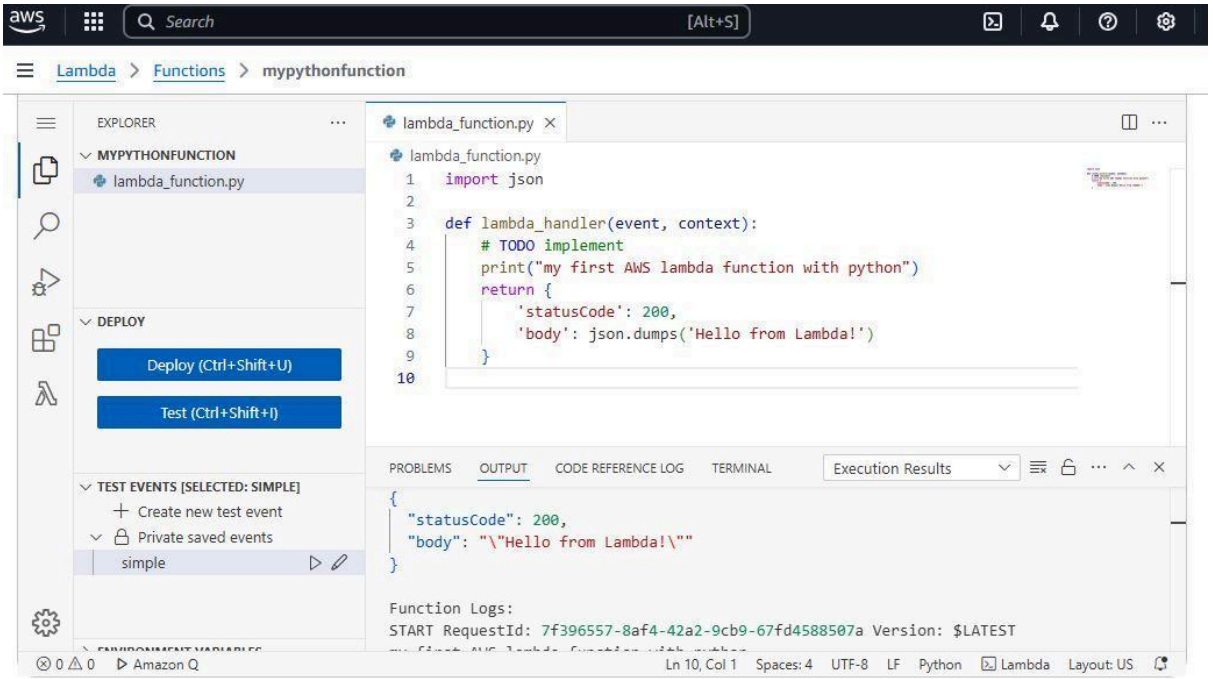
To build your first Python based AWS Lambda function, follow these steps:

1. Log into the AWS console and navigate to the Lambda dashboard.
2. Click the Create Function button.
3. Specify the function's name and the Python version (Python 3.10 is recommended).
4. Edit the Python code in Amazon's embedded code editor.
5. Click Deploy and then Test to see Python Lambda function in action!

Lambda_function.py

```
import json

def lambda_handler(event, context):
    # TODO implement
    print("my first AWS lambda function with python")
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

Output:-



Result:-

Thus, creating lambda function using AWS SDK for python has been done successfully.

EXPERIMENT -06

MIGRATING A WEB APPLICATION TO DOCKER CONTAINERS

Objective

Containerize a simple web application using Docker, allowing it to run in any environment.

Prerequisites

✅ Install **Docker** (Download Here)

✅ Install **Python 3** and **Flask** (pip install flask)

Step 1: Create a Simple Web Application

Create a new directory for your app:

mkdir my-web-app && cd my-web-app

Create a file named **app.py** and add the following Python Flask code:

```
python
CopyEdit
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Dockerized Web App!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Step 2: Create a Dockerfile

In the same directory, create a file named **Dockerfile** (no extension) and add:

```
# Use an official Python runtime as a parent image
FROM python:3.9

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container
COPY . .

# Install required packages
RUN pip install flask

# Make port 5000 available to the world outside this container
EXPOSE 5000
```

```
# Run the application
CMD ["python", "app.py"]
```

Step 3: Build the Docker Image
Run the following command in the terminal to build your Docker image:

```
docker build -t my-web-app .
```

Step 4: Run the Docker Container
After building the image, start a container using:

```
docker run -d -p 5000:5000 my-web-app
```

Step 5: Test the Web Application
Open your browser and go to:

http://localhost:5000
Or test it using **cURL** in the terminal:

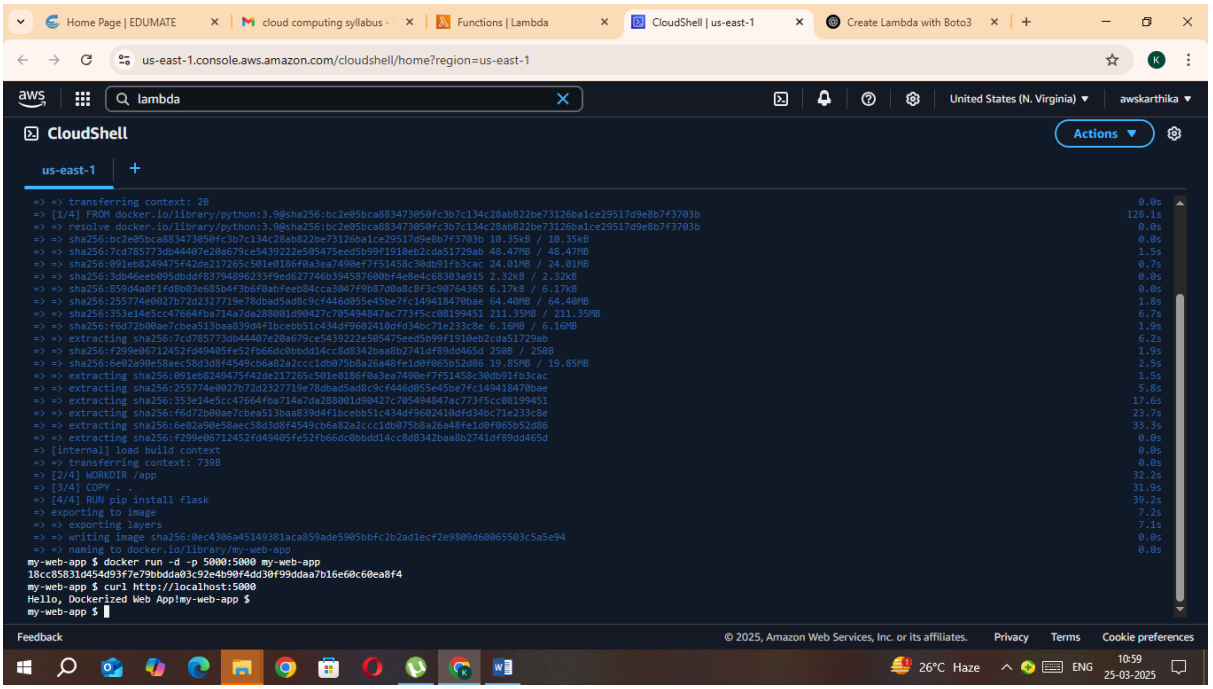
```
curl http://localhost:5000
```
Expected output:

```
Hello, Dockerized Web App!
```

Step 6: Stop and Remove the Container
Stop the running container:

```
docker ps  # Get the container ID
docker stop <container_id>
docker rm <container_id>
```

OUTPUT:-



Result:-

Thus, migrating a Web Application to Docker Containers has been done successfully.

EXPERIMENT -07

CACHING APPLICATION DATA WITH ELASTICACHE, CACHING WITH AMAZON CLOUDFRONT, CACHING STRATEGIES

OBJECTIVE:- Demonstrate how to improve application performance using **Amazon ElastiCache** (data caching) and **Amazon CloudFront** (content caching).

Apply caching to:

- Store frequently accessed user profile data in **ElastiCache (Redis)**
- Deliver static content through **CloudFront CDN**
- Apply basic **caching strategies** (TTL, lazy loading, cache invalidation)

1. **Set up the Application (Local or AWS EC2)**
   - Deploy a simple web app (Node.js, Python Flask, etc.)
   - Example endpoint: /user/123 fetches user data from an RDS or simulated database.
2. **Add Amazon ElastiCache (Redis) for Data Caching**
   - Launch an **ElastiCache Redis cluster** via AWS Console.
   - Modify your app to:
     1. **Check Redis** for the user data.
     2. If not present, **fetch from DB**, store in Redis with a **TTL** (e.g., 60 seconds).
     3. Return result to user.

```
def get_user_profile(user_id):
    cache_key = f"user:{user_id}"
    cached = redis.get(cache_key)
    if cached:
        return json.loads(cached)
    else:
        user = db.query_user(user_id)
        redis.setex(cache_key, 60, json.dumps(user))  # 60s TTL
        return user
```

3. **Use Amazon CloudFront for Static Content**
   - Upload static assets to **S3 bucket**.
   - Create a **CloudFront distribution** for the S3 bucket.
   - Use CloudFront URL to serve assets in your app.

Example:

```
<img src="https://d12345.cloudfront.net/images/logo.png" />
```

Benefits:

- Global edge caching
- Reduced latency
- Offloading traffic from origin

4. **Apply Caching Strategies**

TTL (Time to Live) - Controls how long data stays in cache.

Lazy Loading - Cache data only when requested.

Cache Invalidation - Remove outdated items from cache (e.g., on update).

Example Cache Invalidation:

```
def update_user_profile(user_id, new_data):
    db.update(user_id, new_data)
    redis.delete(f"user:{user_id}")  # Invalidate old cache
```

Result :-

Thus, Implementing Caching Application Data with Elasticache, Caching with Amazon Cloudfront, Caching Strategies has been done successfully.

EXPERIMENT-08

IMPLEMENTING CLOUDFRONT FOR CACHING AND APPLICATION SECURITY

Objective

To set up an Amazon CloudFront distribution in front of an S3 bucket, enabling caching and basic application security features like HTTPS and origin access control.

1. Create an S3 Bucket and Upload Content

- Go to **S3 console**.
- Create a new bucket (e.g., `my-cloudfront-lab`).
- Disable public access settings.
- Upload a sample HTML file (e.g., `index.html`).

2. Enable Static Website Hosting (Optional for testing)

- In S3 bucket properties, enable **Static Website Hosting**.
- Set index document as `index.html`.

3. Create a CloudFront Distribution

- Go to **CloudFront** in the AWS console.
- Click **Create Distribution**.
- Under **Origin domain**, select your S3 bucket.
- Set **Origin access** to **Origin access control (recommended)**:
    - Create a new OAC.
    - Grant access to your S3 bucket.
- Choose **Viewer protocol policy**: Redirect HTTP to HTTPS.
- Set **Default root object** as `index.html`.
- Click **Create distribution**.

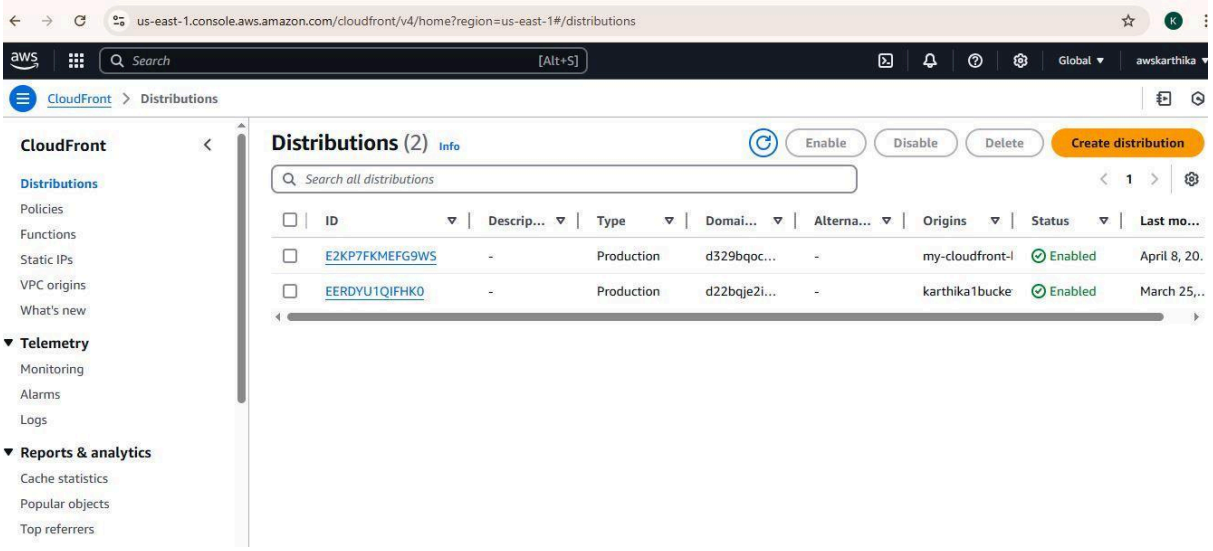
4. Test the Distribution

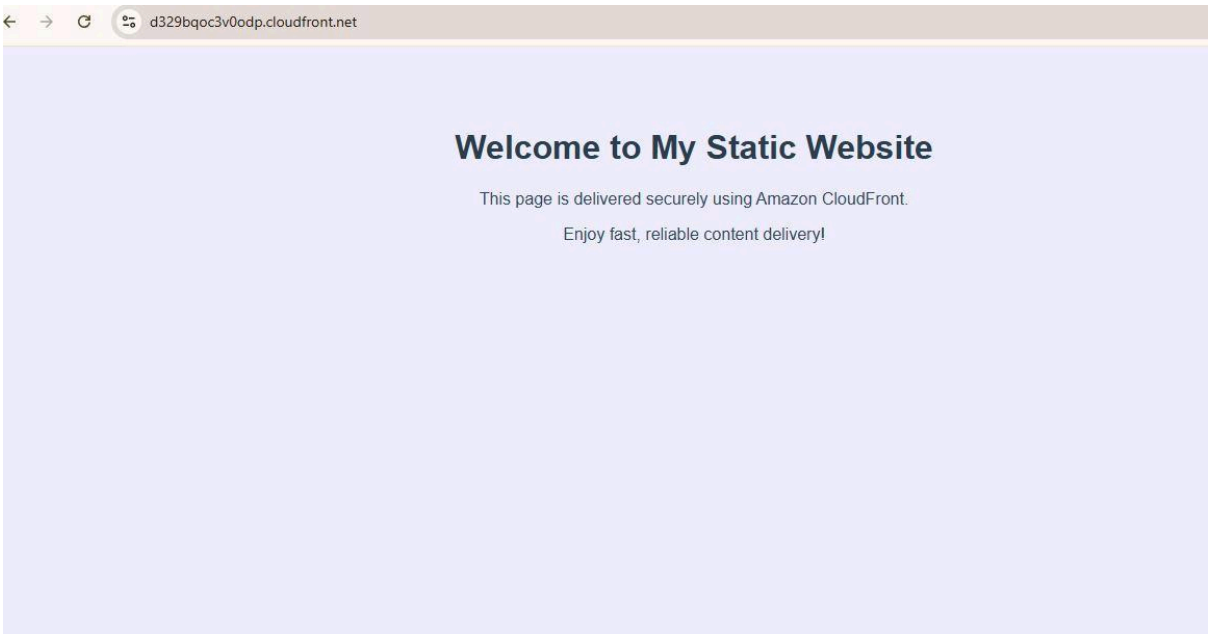
- After deployment (takes a few minutes), copy the **CloudFront domain name** (e.g., `d123abc.cloudfront.net`) and access it in a browser.
- You should see your `index.html` file served securely via HTTPS.

Bucket policy

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowCloudFrontServicePrincipalReadOnly",
         "Effect": "Allow",
         "Principal": {
            "Service": "cloudfront.amazonaws.com"
         },
         "Action": "s3:GetObject",
         "Resource": "arn:aws:s3:::my-cloudfront-lab1/*",
         "Condition": {
            "StringEquals": {
               "AWS:SourceArn":
"arn:aws:cloudfront::842676018658:distribution/E2KP7FKMEFG9WS"
            }
         }
      }
   ]
}
```

Output:-





Result :-

Thus, implementing CloudFront for Caching and Application Security has been done successfully.

EXPERIMENT-09

ORCHESTRATING SERVERLESS FUNCTIONS WITH AWS STEP FUNCTIONS..

Objective
Learn how to create an AWS **Step Function workflow** that orchestrates two **AWS Lambda functions**.

Step 1: Create Two AWS Lambda Functions
We will create two simple Lambda functions:

1. function1 - Takes input and passes it to the next step.
2. function2 - Receives input from function1 and logs it.

Create function1

1. Open **AWS Lambda** in the AWS Console.
2. Click **Create function** → Select **Author from scratch**.
3. Name it **function1**.
4. Select **Runtime**: Python 3.x (or Node.js).
5. Use the following code:
   Python (function1)

```
import json
def lambda_handler(event, context):
    name = event.get("name", "Guest")
    return {"message": f"Hello, {name}!"}
```

Click **Deploy**.

Create function2

1. Repeat steps to create another Lambda function.
2. Name it **function2**.

```
import json
def lambda_handler(event, context):
    print(f"Received message: {event['message']}")
    return {"status": "Processed successfully!"}
```

Click **Deploy**.

Step 2: Create an AWS Step Function

1. Open **AWS Step Functions**.
2. Click **Create State Machine**.
3. Choose **Author with code**.
4. Select **Standard** type.
5. Replace the default JSON with this:

```
{
  "Comment": "A simple Step Function workflow",
  "StartAt": "InvokeFunction1",
  "States": {
    "InvokeFunction1": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:function1",
      "Next": "InvokeFunction2"
    },
    "InvokeFunction2": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:function2",
      "End": true
    }
  }
}
```

Replace:

- REGION with your AWS region (e.g., us-east-1).
- ACCOUNT_ID with your AWS account ID.

Click **Next** → Name it **MyStepFunction**.

Click **Create**.

Step 3: Grant Step Functions Permission to Invoke Lambda

1. Open **IAM** → Roles.
2. Find the role used by Step Functions.
3. Attach the following **policy**:

```
{
    "Effect": "Allow",
    "Action": "lambda:InvokeFunction",
    "Resource": [
        "arn:aws:lambda:REGION:ACCOUNT_ID:function:function1",
        "arn:aws:lambda:REGION:ACCOUNT_ID:function:function2"
    ]
}
```

Save the role.

Step 4: Execute the Step Function

1. Go to **AWS Step Functions** → Click **Start Execution**.

2. Provide input JSON:

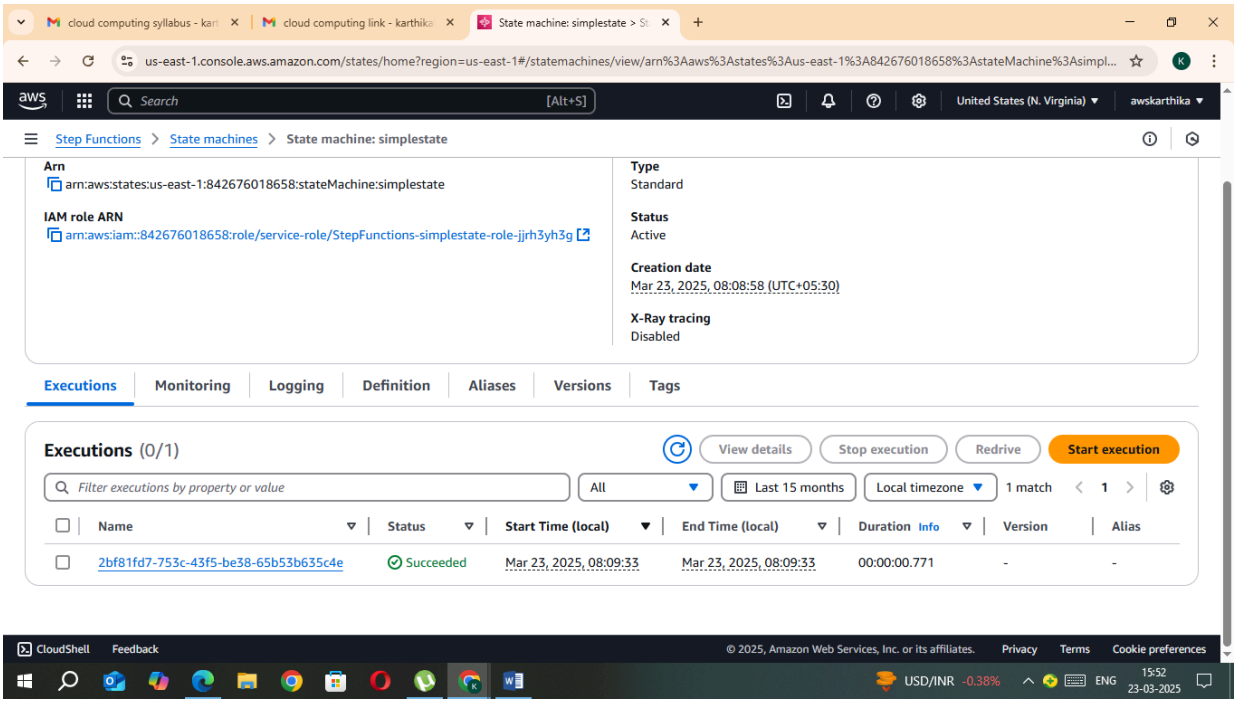
```
{
  "name": "Alice"
}
```

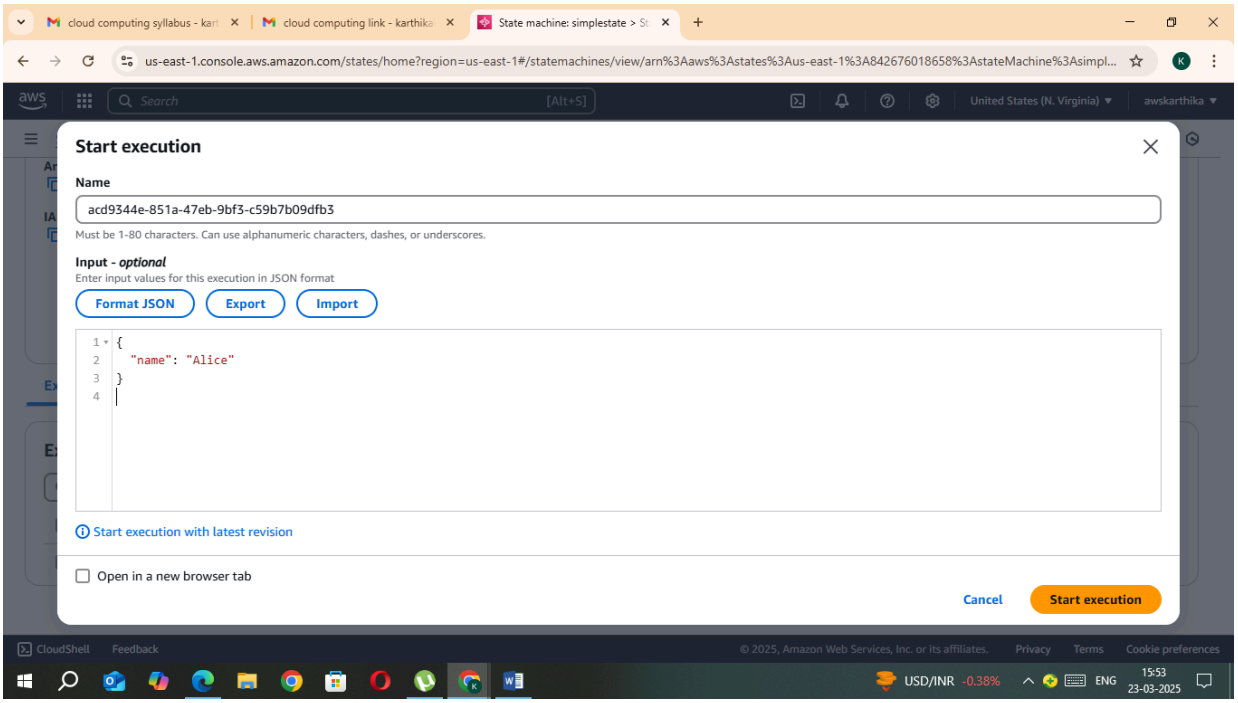
Click **Start Execution**.

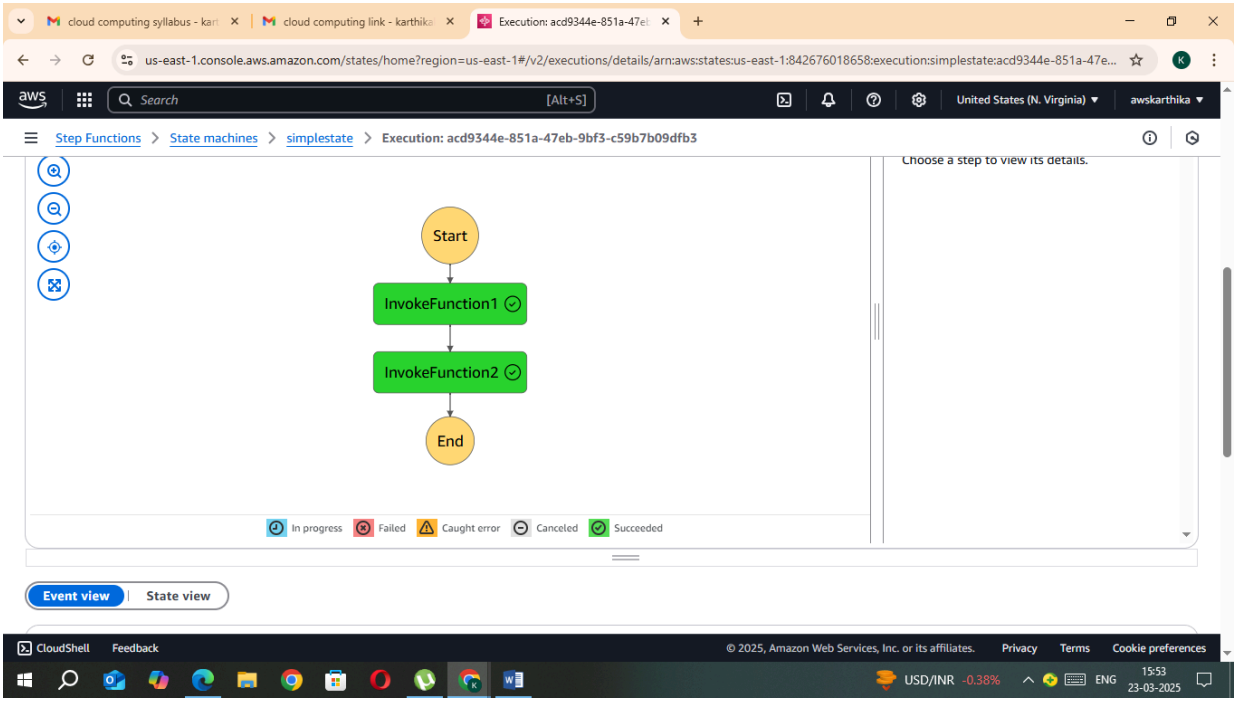
Step 5: Verify the Output

- Check **Step Functions Execution History**.

OUTPUT:-







Result:-

Thus, Orchestrating Serverless Functions with AWS Step Functions has been done successfully.

EXPERIMENT -10

AUTOMATING APPLICATION DEPLOYMENT USING A CI/CD PIPELINE

Objective:

Automatically deploy a basic website whenever code is pushed to GitHub.

.1. **Create a Simple Website**

Create a file named index.html:

```
html
CopyEdit
<!DOCTYPE html>
<html>
<head>
  <title>My CI/CD Site</title>
</head>
<body>
  <h1>Welcome to my website!</h1>
</body>
</html>
```

2. **Create a GitHub Repository**

- Go to https://github.com

- Click **New Repository**
- Name it simple-cicd-site
- Upload the index.html file

3. **Add GitHub Actions for Deployment**

1. In your repo, create folders and a file:

.github/workflows/main.yml

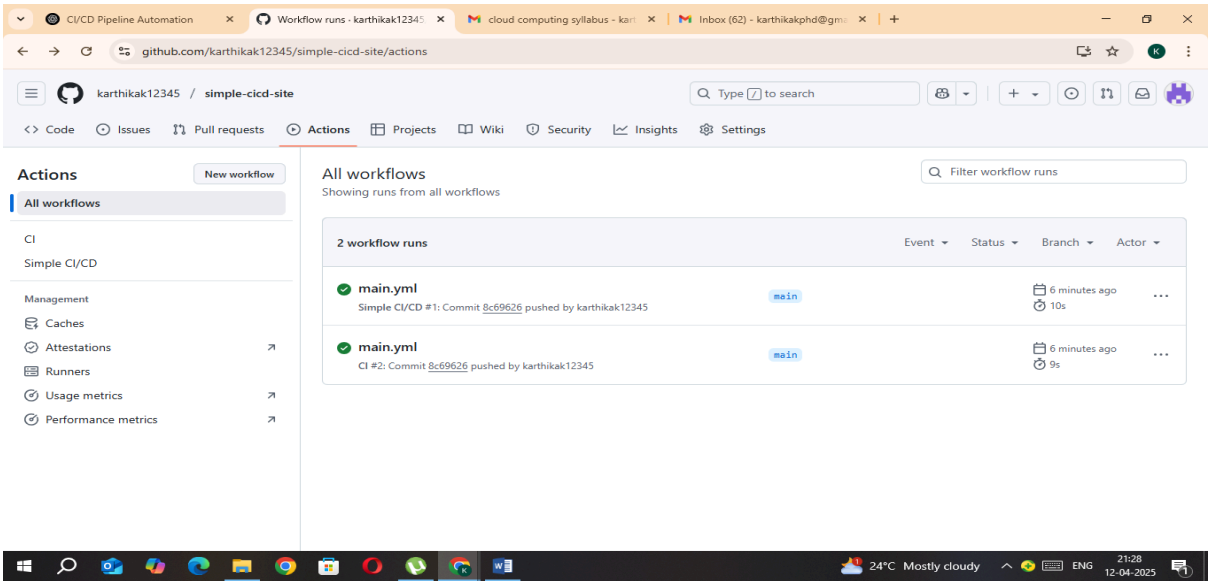
2. Add the following content in main.yml:

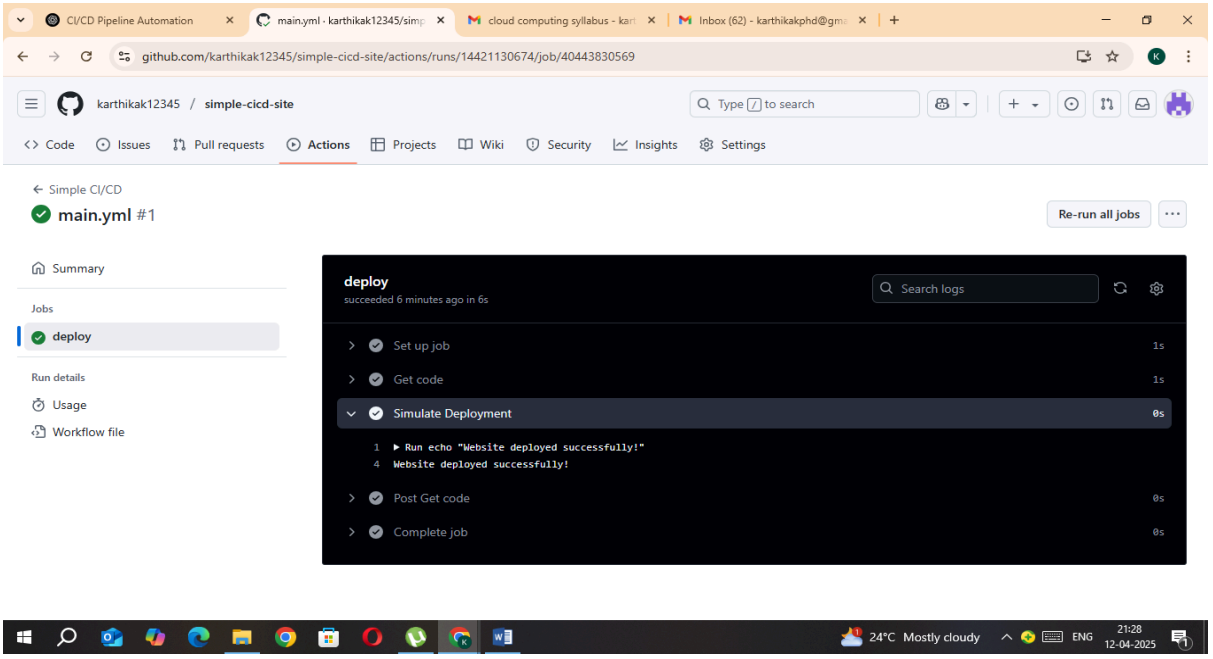
```
name: Simple CI/CD
on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Simulate Deployment
        run: echo "Website deployed successfully!"
```

4. **Watch the Pipeline Run**

1. Go to your GitHub repository.
2. Click on the **"Actions"** tab.
3. You'll see a workflow running — click it.
4. Click the job (e.g., "deploy") to see details.
5. Under "Simulate Deployment," you'll see:

OUTPUT:-





Result:-

Thus, automatically deploy a basic website whenever code is pushed to GitHub has been done successfully.