CHHATTISGARH SWAMI VIVEKANAND TECHNICAL UNIVERSITY

(UNIVERSITY TEACHING DEPARTMENT)

8th SEMESTER

Cloud Computing Lab Manual

Guided By
Mr. Shesh Narayan Sahu
Assistant Professor
CSE (DS)

Prepared By
Madhurima Rawat
Roll No: 300012821042
Enrolment No: CB4689

Table of Contents

S. No & Title	Content
1. Setting Up a Virtual Machine in a Cloud Environment	In this experiment, students will learn to understand the basics of cloud computing by setting up and configuring a virtual machine on a cloud platform such as AWS, Google Cloud, or Microsoft Azure.
2. Deploying a Web Application on a Cloud Server	This experiment enables students to deploy a basic web application on a cloud server using services like AWS EC2, where they will practice deploying applications on the cloud infrastructure.
3. Working with Cloud Storage Services	Students will explore cloud storage services such as AWS S3, Google Cloud Storage, or Microsoft Azure Blob Storage to store and retrieve data, learning how to manage cloud-based data storage.
4. Setting Up and Configuring Cloud Networking	In this experiment, students will configure and manage networking services in the cloud, such as creating Virtual Private Cloud (VPC) and subnets on cloud platforms like AWS, Google Cloud, or Azure.
5. Using Cloud Functions for Serverless Computing	This experiment provides students with hands-on experience in serverless computing by creating a serverless function using AWS Lambda, Google Cloud Functions, or Azure Functions, enabling them to understand the serverless paradigm.
6. Cloud Load Balancing and Auto Scaling	In this experiment, students will configure load balancing and auto-scaling in the cloud environment to handle varying traffic loads efficiently, using tools such as AWS Elastic Load Balancing (ELB) and Google Cloud Load Balancer.
7. Cloud Databases and Data Management	This experiment introduces students to setting up and managing cloud-based relational databases such as AWS RDS or Google Cloud SQL, enabling them to practice database management in a cloud environment.
8. Cloud Security: Identity and Access Management (IAM)	In this experiment, students will learn to manage users, permissions, and security policies in the cloud using Identity and Access Management (IAM) tools from platforms like AWS IAM, Google Cloud IAM, or Azure Active Directory.
9. Implementing Cloud Monitoring and Logging	This experiment focuses on cloud monitoring services such as AWS CloudWatch, Google Stackdriver, or Azure Monitor, where students will track resources and application performance in a cloud environment.
10. Setting Up Cloud-based CI/CD Pipeline	In this experiment, students will set up a Continuous Integration/Continuous Deployment (CI/CD) pipeline using cloud services like AWS CodePipeline, Google Cloud Build, or Jenkins on the cloud, providing hands-on experience with cloud-based automation tools.

Tools and Technologies

1. AWS CLI

https://docs.aws.amazon.com/cli/latest/userguide/

AWS Command Line Interface (CLI) is a powerful tool that allows users to interact with AWS services directly from the terminal. It simplifies managing cloud resources by providing commands for a wide range of AWS services, enabling tasks such as provisioning, managing, and automating workflows with ease.

2. LocalStack

https://docs.localstack.cloud/

LocalStack is a fully functional, local testing environment for AWS services. It enables developers to simulate AWS services on their local machines, facilitating the development and testing of cloud-based applications without needing access to an actual AWS account.

3. Docker

https://docs.docker.com/

Docker is a containerization platform that allows developers to build, share, and run applications in isolated environments called containers. It ensures consistent environments across development, testing, and production by packaging the application and its dependencies together.

4. Boto3

https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

Boto3 is the official AWS SDK for Python, enabling developers to interact with AWS services programmatically. It provides a high-level, easy-to-use interface for managing AWS resources like S3, EC2, DynamoDB, and more.

5. Flask

https://flask.palletsprojects.com/

Flask is a lightweight and flexible Python web framework designed for building web applications and APIs. It follows a minimalistic approach, providing essential tools while allowing developers to add extensions as needed.

6. Nginx

https://nginx.org/en/

Nginx is a high-performance, open-source web server and reverse proxy server. It's designed for serving static content, load balancing, and handling high concurrency with minimal resource consumption. Nginx is widely used for improving web server scalability and security.

7. PostgreSQL

https://www.postgresql.org/

PostgreSQL is a powerful, open-source relational database management system (RDBMS). Known for its reliability, scalability, and support for advanced data types and performance

optimization features, PostgreSQL is widely used for complex applications requiring robust data integrity.

8. GitHub Actions

https://github.com/features/actions

GitHub Actions is a powerful CI/CD (Continuous Integration and Continuous Deployment) tool integrated into GitHub. It allows developers to automate workflows directly in their repositories, enabling tasks such as testing, building, and deploying applications seamlessly.

Experiment 1

Aim: In this experiment, students will learn to understand the basics of cloud computing by setting up and configuring a virtual machine on a cloud platform such as AWS, Google Cloud, or Microsoft Azure.

Prerequisites:

- **Docker** – A platform for developing, shipping, and running applications in containers.
- **LocalStack** – A fully functional local cloud service emulator for AWS development.
- **AWS CLI** – A command-line tool for managing AWS services and automation.
- **System Requirements** – A computer with **8GB RAM** and at least **16GB free space**.

Key Concepts:

Virtual Machine (VM)

- **Definition:** Software-based emulation of a physical computer.
- **Isolation:** Runs in a secure, independent environment.
- **Use Case:** Ideal for cloud computing, testing, and deployment.

LocalStack

- **Purpose:** Locally simulates AWS cloud services.
- **Benefits:** Enables offline development and testing.
- **Integration:** Supports automation and CI/CD workflows.

AWS CLI

- **Function:** Command-line tool for managing AWS services.
- **Automation:** Enables scripting and infrastructure as code.
- **Use Case:** Efficient for deployment, monitoring, and automation.

Docker

- **Concept:** Platform for containerizing applications.
- **Efficiency:** Ensures consistency across environments.
- **Use Case:** Simplifies development, testing, and deployment.

Installation Guide:

Docker Setup

1. **Install Docker**: Download and install Docker Desktop from the official website. Enable WSL 2 if required.

2. **Verify Installation**: Run docker --version to confirm the installation.
3. **Start Docker**: Launch Docker Desktop and ensure it is running.
4. **Enable WSL 2 (If Needed)**: Enable it from Docker Desktop settings under "General".

LocalStack Installation:

1. **Install LocalStack**:
    - Via pip: pip install localstack
    - Via Docker: docker pull localstack/localstack
2. **Start LocalStack**:
    - Using Docker: docker run -d -p 4566:4566 localstack/localstack
    - If installed locally: localstack start
3. **Install AWS CLI**: pip install awscli
4. **Configure AWS CLI**:
    - Run aws configure with test credentials.
    - Setx AWS_ENDPOINT_URL=http://localhost:4566 for LocalStack.

AWS CLI Output Formats

- **JSON** (default): Structured format, ideal for automation.
- **Table**: Human-readable, good for quick reviews.
- **Text**: Simple, suitable for scripts.
- Change format using aws configure or --output flag.

Troubleshooting

- **LocalStack Connection Issues**: Run docker ps to check running containers. Restart LocalStack if needed.
- **Long Path Issue on Windows**:
    1. Enable long paths via Registry (HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem).
    2. Enable via Group Policy (for Pro/Education editions).
    3. Restart the system and retry pip install localstack.

Starting Services:

1. Ensure the Service is Running
If LocalStack or another AWS emulator is being used, it's essential to confirm that it has started successfully.

For LocalStack:
localstack start

For Docker (if LocalStack is running via Docker):
docker run -p 4566:4566 -d localstack/localstack

After executing this, the following command can be used to confirm the container is running:
docker ps

2. Verify Port Availability
To check if another service is using port 4566, the following command should be run:
netstat -ano | find "4566"

If no output is returned, it indicates the port is free.
If output is returned, it's necessary to identify which process is using the port by executing:
tasklist /FI "PID eq <PID>"
Replace <PID> with the process ID found from the netstat output.

3. Check Firewall or Antivirus Settings
It should be ensured that no firewall or antivirus software is blocking connections to port 4566. Temporarily disabling the firewall or antivirus can help in testing.

4. Confirm localhost Works
To check if localhost is working correctly on the system, the following command can be used:
ping localhost

LocalStack Container Image Download Process
The following output indicates that LocalStack is attempting to pull the container image localstack/localstack to run in Docker mode, which suggests that the setup is working up to this point. Here's what's happening:

C:\Users\rawat>docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
Pulling container image localstack/localstack

What's Happening?

1. **LocalStack CLI Detected Docker Mode:**
   [11:20:31] starting LocalStack in Docker mode confirms that LocalStack is trying to run using Docker.
2. **Container Image Not Found Locally:**
   [11:25:13] container image not found on host indicates that Docker is downloading the localstack/localstack image from Docker Hub since it's not found locally.

3. **Pulling the Container Image:**
   ⁝. Pulling container image localstack/localstack confirms that the image download process is ongoing.

What to Expect Next

If the internet connection is stable, Docker should complete the download in a few minutes. After the image is downloaded, LocalStack will initialize, and the services (e.g., S3, DynamoDB) will start. This will be confirmed by further logs.

How to Verify It's Working

1. After the image download is complete, the user can check if the container is running with the following command:
   docker ps
   The container should be listed with the name localstack.
2. To verify that LocalStack is running, the endpoint should be accessed:
   curl http://localhost:4566
   A response should be returned, confirming that LocalStack is active.

Steps to Set Up a Virtual Machine in LocalStack:

1. Simulate EC2 Service

LocalStack emulates a limited set of EC2 functionalities. The goal is to create mock resources like key pairs, security groups, and instances.

2. Create a Key Pair

Use the AWS CLI to generate a key pair:

aws ec2 create-key-pair --key-name local-key --endpoint-url=%AWS_ENDPOINT_URL%

The output will include the generated public/private key pair.

3. Create a Security Group

Create a security group to define network rules:

aws ec2 create-security-group --group-name local-sg --description "Local Security Group" --endpoint-url=%AWS_ENDPOINT_URL%

4. Run an Instance

Launch a mock EC2 instance using:

aws ec2 run-instances --image-id ami-12345678 --count 1 --instance-type t2.micro --key-name local-key --security-group-ids sg-12345678 --endpoint-url=%AWS_ENDPOINT_URL%

Replace ami-12345678 with an example AMI ID that is recognized by LocalStack.

Example Output:

Security Group ID: sg-2cd410ccd533c7f8b

Image ID: ami-a2678d778fc6

Command:

aws ec2 run-instances --image-id ami-a2678d778fc6 --count 1 --instance-type t2.micro --key-name local-key --security-group-ids sg-2cd410ccd533c7f8b --endpoint-url=%AWS_ENDPOINT_URL%

What Happens When You Run This Command?

- AWS CLI creates a single EC2 instance based on ami-a2678d778fc6.
- The instance is t2.micro, suitable for low-resource tasks.
- Uses local-key key pair for SSH access.
- sg-2cd410ccd533c7f8b security group manages traffic.
- Endpoint URL directs requests to the specified AWS service.

Example Use Case

Set up a local test server (e.g., Ubuntu) with a custom AWS endpoint (http://localhost:4566) and controlled security settings.

5. List Instances

Verify the instance creation:

aws ec2 describe-instances --endpoint-url=%AWS_ENDPOINT_URL%

Conclusion:

In this experiment, we have successfully executed the necessary steps to set up and simulate an EC2 instance using LocalStack. We created a key pair, defined a security group, launched a mock instance, and verified its status using AWS CLI commands.

The output for all the executed commands is provided on the next page for reference.

Experiment 1 Output

Running AWS using Localstack

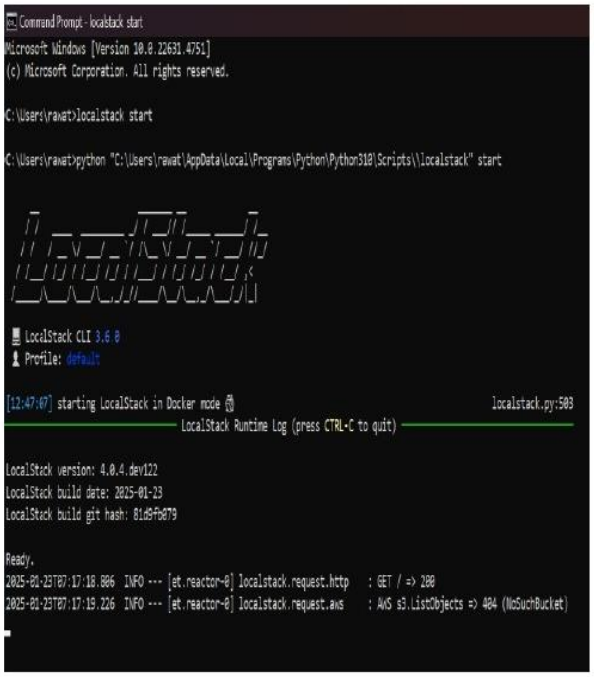


Fig 1: Starting Localstack in the Console

Running Docker

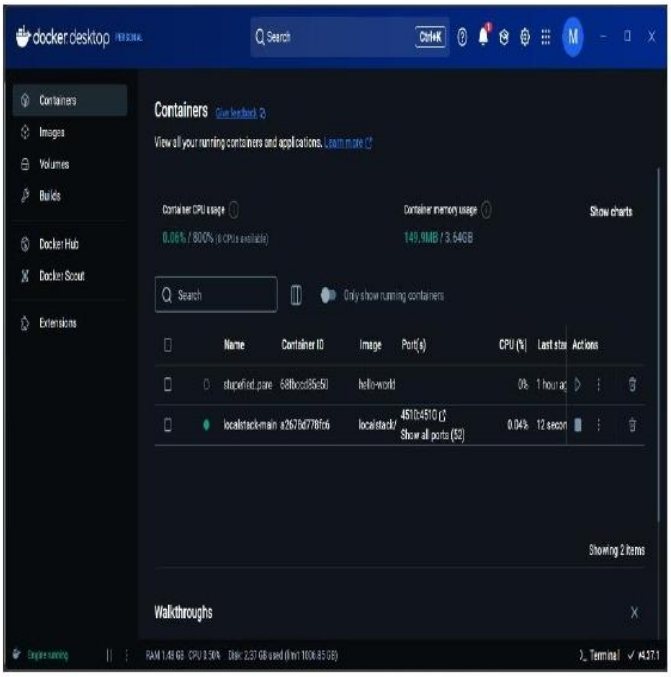


Fig 2: Docker for Desktop in windows

Creating a New Instance

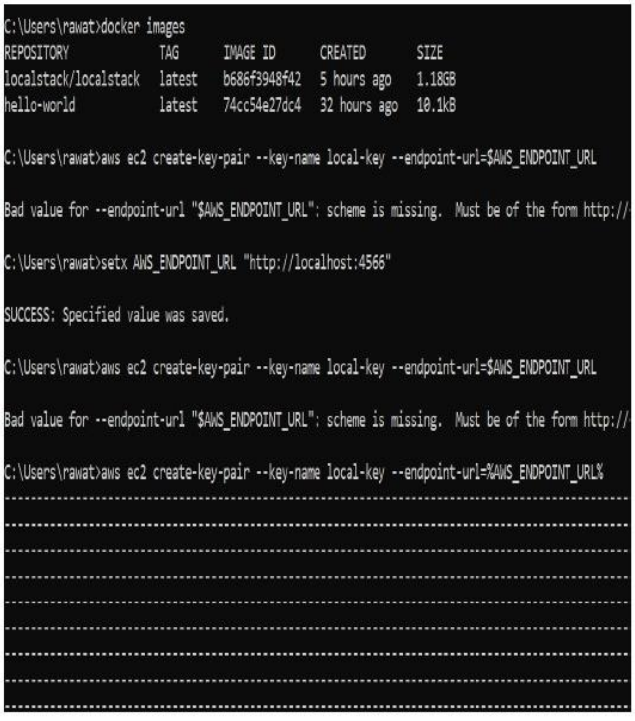


Fig 3: Docker Images and New Instances

Starting Virtual Machine

Fig 4: Virtual Machine & Working

Experiment 2

Aim: This experiment enables students to deploy a basic web application on a cloud server using services like AWS EC2, where they will practice deploying applications on the cloud infrastructure.

Web Application(Flask App Code):

```
"""
Description:
This Flask application interacts with AWS EC2, fetching instance details
and providing a simple API endpoint for cloud deployment testing.
"""
# Importing Required Libraries
from flask import Flask, jsonify
import boto3
import socket
import logging
import os
# Initialize Flask application
app = Flask(__name__)
# Determine if the app is running on LocalStack or AWS
if os.environ.get("LOCALSTACK_URL"):
    endpoint_url = "http://localhost:4566"  # LocalStack endpoint for testing locally
else:
    endpoint_url = None  # Use AWS default endpoints in a real cloud environment
# Initialize EC2 client using boto3
ec2 = boto3.client(
    "ec2", region_name="us-east-1", endpoint_url=endpoint_url
)  # Uses LocalStack if applicable
# Enable logging for debugging
logging.basicConfig(level=logging.DEBUG)
@app.route("/")
def home():
    """
```

```
    Home route that returns a welcome message.
    """
    return "Hello, Cloud Deployment!"
@app.route("/instance-stats")
def instance_stats():
    """
    Fetches and returns metadata of an EC2 instance.
    If running in LocalStack, it will return mock data.
    """
    try:
        logging.debug("Fetching EC2 instance metadata...")
        # Fetch EC2 instance stats using boto3
        response = ec2.describe_instances()
        # Log API response for debugging
        logging.debug(f"API Response: {response}")
        if not response["Reservations"]:
            logging.warning("No EC2 instances found in the response.")
        # Retrieve the first instance's information
        instance_info = response["Reservations"][0]["Instances"][0]
        logging.debug(f"Instance Info: {instance_info}")
        instance_stats = {
            "Instance ID": instance_info["InstanceId"],
            "Instance Type": instance_info["InstanceType"],
            "Public IP": instance_info.get("PublicIpAddress", "N/A"),
            "State": instance_info["State"]["Name"],
            "Region": "us-east-1",
        }
        logging.debug(f"Returning instance stats: {instance_stats}")
        return jsonify(instance_stats)
    except Exception as e:
        logging.error(f"Error: {str(e)}")
        return jsonify({"error": str(e)})
```

```
# Run the application
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Steps to Deploy a Web Application in LocalStack:

1. Creating a Resource in API Gateway

Command:

```
aws --endpoint-url=http://localhost:4566 apigateway create-resource \
--rest-api-id rbx2kdpyxl \
--parent-id regpu0mm3f \
--path-part "flaskapp"
```

Explanation:

- create-resource: Creates a new resource in API Gateway.
- --endpoint-url=http://localhost:4566: Points to the LocalStack endpoint.
- --rest-api-id rbx2kdpyxl: ID of the API Gateway to add the resource to.
- --parent-id regpu0mm3f: ID of the parent resource (usually the root /).
- --path-part "flaskapp": Specifies the path (/flaskapp) for the new resource.

Output Breakdown:

Key	Value
id	bsw1umubix
parentId	regpu0mm3f
path	/flaskapp
pathPart	flaskapp

2. Creating an HTTP Method (GET) for the Resource

Command:

```
aws --endpoint-url=http://localhost:4566 apigateway put-method \
--rest-api-id rbx2kdpyxl \
--resource-id bsw1umubix \
--http-method GET \
```

--authorization-type NONE

Explanation:

- Adds a GET method to the /flaskapp resource.
- No authorization required (authorization-type NONE).

Output Breakdown:

Key	Value
apiKeyRequired	False
authorizationType	NONE
httpMethod	GET

3. Integrating API Gateway with a Backend Service

Command:

aws --endpoint-url=http://localhost:4566 apigateway put-integration \

--rest-api-id rbx2kdpyxl \

--resource-id bsw1umubix \

--http-method GET \

--integration-http-method GET \

--type HTTP_PROXY \

--uri http://localhost:5000/

Explanation:

- Proxies the GET method to the Flask backend at http://localhost:5000/.

Output Breakdown:

Key	Value
httpMethod	GET
type	HTTP_PROXY
uri	http://localhost:5000/
passthroughBehavior	WHEN_NO_MATCH
timeoutInMillis	29000

4. Deploying the API Gateway

Command:

```
aws --endpoint-url=http://localhost:4566 apigateway create-deployment \
--rest-api-id rbx2kdpyxl \
--stage-name prod
```

Explanation:

- Deploys the API under the prod stage.

Output Breakdown:

Key	Value
createdDate	1738825514.0
id	fpmrktu41t

Here is a detailed explanation of each command and the associated output you've shared:

5. Retrieving API Gateway Stage Information

Command:

```
aws --endpoint-url=http://localhost:4566 apigateway get-stage \
--rest-api-id rbx2kdpyxl \
--stage-name prod
```

Explanation:

- This command retrieves information about a specific stage (prod) of a REST API hosted on **LocalStack** (a local AWS emulator).
- --endpoint-url=http://localhost:4566: Points the AWS CLI to LocalStack instead of AWS cloud.
- --rest-api-id rbx2kdpyxl: Specifies the ID of the API.
- --stage-name prod: Specifies the name of the stage whose details are being fetched.

Output Breakdown:

- stageName: The name of the stage (in this case, "prod").
- deploymentId: ID of the current deployment associated with this stage.
- tracingEnabled: Whether AWS X-Ray tracing is enabled for this stage (False here).
- cacheClusterEnabled: Indicates if caching is enabled (False).
- cacheClusterStatus: The current status of the cache cluster (not available).

6. Testing the Flask Application

Command:

curl http://localhost:5000/instance-stats

Explanation:

- This command sends a **GET** request using curl to the Flask application running locally on port 5000.
- /instance-stats is a route in the Flask app, expected to return instance information in JSON format.

Sample Output:

{

"Instance ID": "i-6c9d5e3fa4c23d261",

"Instance Type": "t2.micro",

"Public IP": "54.214.36.25",

"Region": "us-east-1",

"State": "running"

}

- This response typically mimics AWS EC2 instance metadata and is likely mocked or fetched by the Flask app.
- Each key represents a piece of metadata about an instance (possibly fetched using boto3 in the backend).

7. Retrieving API Gateway Resources

Command:

aws --endpoint-url=http://localhost:4566 apigateway get-resources \

--rest-api-id rbx2kdpyxl

Explanation:

- This command lists all the resources defined in a REST API on LocalStack.
- --rest-api-id: Specifies the ID of the API whose resources you want to list.

Resource Breakdown:

1. **Root Resource /**
    - id: regpu0mm3f: Unique identifier for the root path.
2. **/flaskapp Resource**
    - id: bsw1umubix: Unique ID for this specific resource.

- parentId: regpu0mm3f: Indicates it is a child of the root resource.
- path: /flaskapp: The resource path.
- httpMethod: GET: Specifies the HTTP method supported (GET).
- type: HTTP_PROXY: Indicates this resource is set up as a **proxy integration**, forwarding requests to a backend.
- url: http://localhost:5000/: The backend endpoint this resource is proxied to, which is your Flask app.

Conclusion:

In this experiment, we have successfully executed the necessary steps to set up and deploy a web application using **API Gateway** and a **Flask backend**, simulated through **LocalStack**. We created an API resource, configured an HTTP method, integrated it with the Flask app using HTTP proxy integration, and deployed it under a named stage.

The application was successfully tested using curl, and all configurations were validated via AWS CLI commands.

The output for all executed commands is provided on the next page for reference.

Experiment 2 Output

Running AWS using Localstack

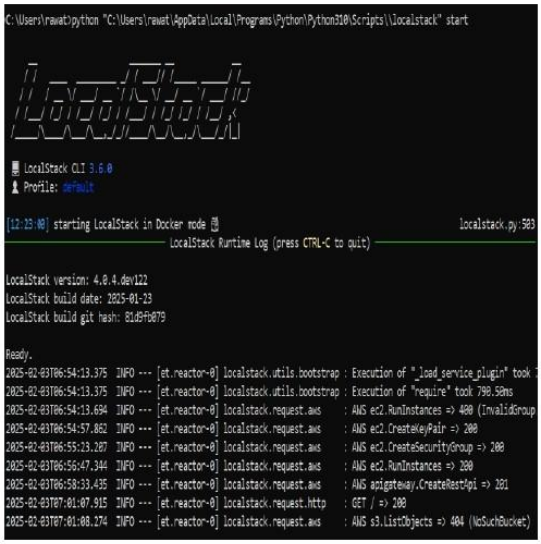


Fig 1: Starting Localstack in the Console

Running Docker

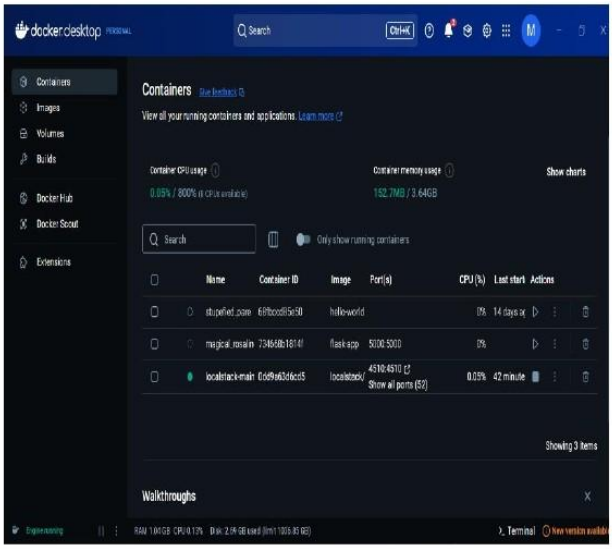


Fig 2: Docker for Desktop in windows

Deploying a Flask App

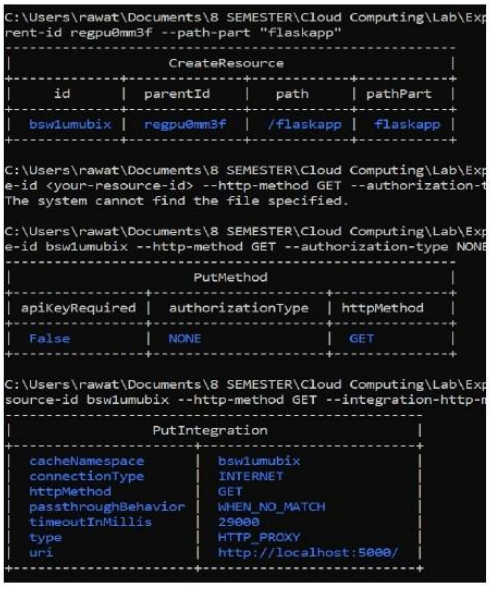


Fig 3: AWS New Instances and deployment of Flask App

Configuration and Resources

Fig 4: Instance Id & Resources

Fig 5: Flask and Cloud Output

Experiment 3

Aim: Students will explore cloud storage services such as AWS S3, Google Cloud Storage, or Microsoft Azure Blob Storage to store and retrieve data, learning how to manage cloud-based data storage.

What is AWS S3?

Amazon S3 (Simple Storage Service) is an object storage service that offers scalability, security, and high availability. It allows users to store and retrieve any amount of data at any time, making it ideal for:

- Backups
- Website hosting
- Data lakes

Key Features:

- Secure and highly available storage solution
- Supports various storage classes for cost optimization
- Provides strong access control and encryption options
- Integrates seamlessly with other AWS services

Use Cases for S3

- Storing images, videos, and static website assets
- Hosting static websites and content delivery
- Backup and disaster recovery solutions
- Log storage for analytics and monitoring
- Machine learning and big data storage

S3 Storage Types

Storage Class	Use Case
S3 Standard	Frequently accessed data
S3 Intelligent-Tiering	Automatic cost optimization based on usage
S3 Standard-IA	Infrequent access, lower cost
S3 Glacier	Long-term archival storage
S3 One Zone-IA	Infrequent access in a single availability zone

Steps to Use S3 in Localstack:

Step 1: Start LocalStack

- Use one of the following to start LocalStack:

localstack start

Or with Docker:

docker run --rm -it -p 4566:4566 localstack/localstack

- Start Docker Desktop and wait until it shows "Docker is running".
- LocalStack simulates AWS services on port **4566** for local development.

Step 2: Creating an S3 Bucket in LocalStack

Commands:

1. **List Existing Buckets:**

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 ls

2. **Create a New Bucket:**

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 mb s3://my-test-bucket

3. **List Buckets Again:**

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 ls

Explanation:

- LocalStack emulates AWS services without needing an AWS account.
- localhost:4566 acts as the endpoint for CLI/API requests.
- CLI outputs confirm the creation and listing of the bucket.

Step 3: Store and Access Images & CSV in S3

1. **Upload Files:**

Navigate to your directory:

cd "C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items"

Upload all files:

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 cp . s3://my-test-bucket/ --recursive

2. **List Uploaded Files:**

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 ls s3://my-test-bucket/

Expected output:

2025-02-18 10:30:00 12345 image1.jpg

2025-02-18 10:30:01 67890 image2.png

2025-02-18 10:30:02 45678 data.csv

3. **Download Files:**

CSV File:

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 cp s3://my-test-bucket/data.csv .

Images:

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 cp s3://my-test-bucket/image1.jpg .

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 cp s3://my-test-bucket/image2.png .

Download to Specific Path:

aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 cp s3://my-test-bucket/Sample_Housing_CSV_File.csv "C:\Users\rawat\Downloads\Sample_Housing_CSV_File.csv"

To verify:

dir "C:\Users\rawat\Downloads\Sample_Housing_CSV_File.csv"

4. **Print CSV Content:**

Windows:

type data.csv

macOS/Linux:

cat data.csv

Expected Output:

Name, Age, City

Alice, 25, New Delhi

Bob, 30, Mumbai

5. **Open Images for Viewing:**

Windows:

start image1.jpg

start image2.png

macOS:

open image1.jpg

open image2.png

Linux:

xdg-open image1.jpg

xdg-open image2.png

Conclusion:

In this experiment, we successfully set up an S3 bucket using LocalStack, uploaded data (images and a CSV file), and verified both upload and retrieval using the AWS CLI. This demonstrated the power and simplicity of working with cloud storage services like Amazon S3 locally using LocalStack.

The output for all executed commands is provided on the next page for reference.

Experiment 3 Output

Running AWS using Localstack

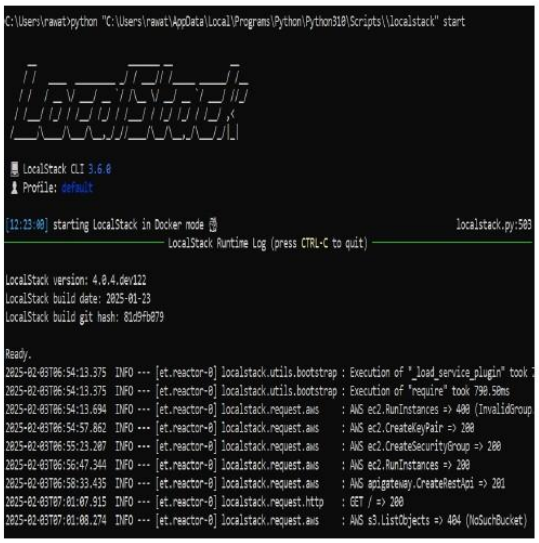


Fig 1: Starting Localstack in the Console

Running Docker

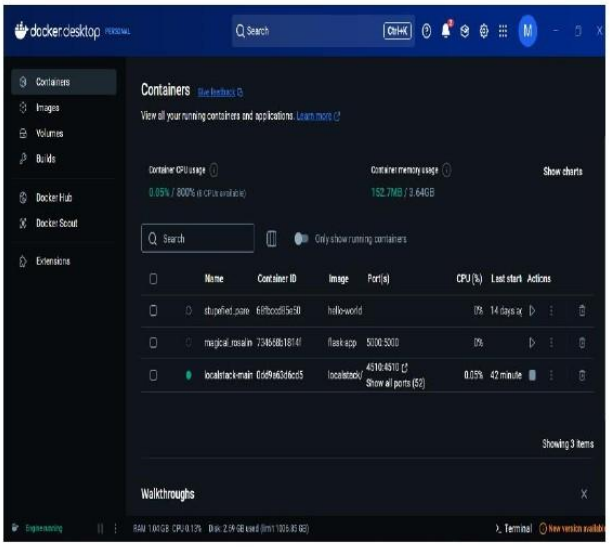


Fig 2: Docker for Desktop in windows

Creating and Storing Data in S3 Bucket

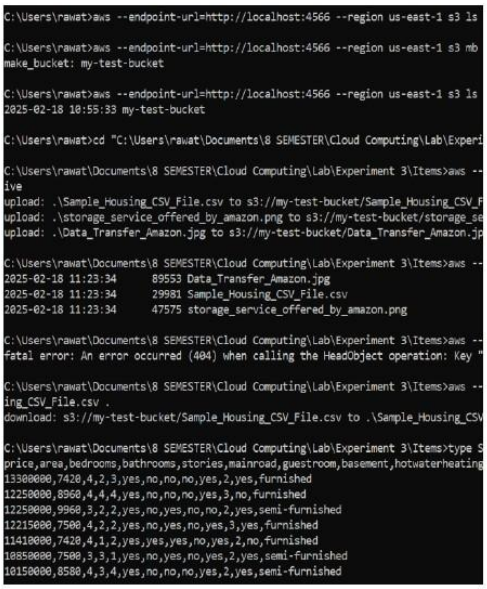


Fig 3: S3 Bucket and contents

Contents in Bucket

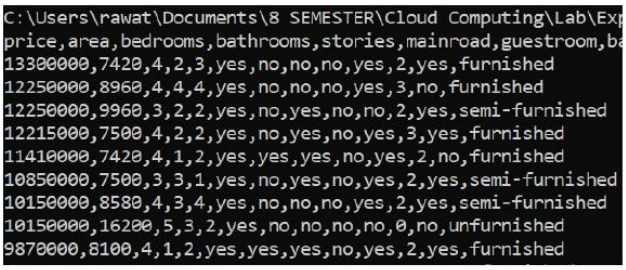


Fig 4: Printing CSV File

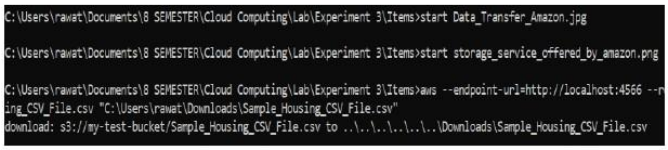


Fig 5: Images Output

Experiment 4

Aim: In this experiment, students will configure and manage networking services in the cloud, such as creating Virtual Private Cloud (VPC) and subnets on cloud platforms like AWS, Google Cloud, or Azure.

Overview of Virtual Private Cloud (VPC):

A **Virtual Private Cloud (VPC)** is a customizable virtual network within the AWS cloud that allows you to manage networking resources securely and privately. It enables the creation of isolated cloud networks and offers complete control over IP address ranges, routing, and connectivity.

- Allows the creation of isolated networks in the cloud
- Provides control over IP address ranges and routing
- Enables creation of public and private subnets
- Facilitates secure, private communication between instances

Key Components of a VPC

A VPC consists of essential components that support a secure and scalable networking environment:

- **Subnets**: Segment the network into smaller, manageable sections
- **Internet Gateway (IGW)**: Provides internet access for public-facing resources
- **Route Tables**: Direct traffic within the VPC and to/from the internet
- **NAT Gateway**: Allows private subnet instances to access the internet
- **Security Groups & Network ACLs**: Define inbound and outbound traffic permissions for resources

Setting Up a VPC in LocalStack

Step 1: Start LocalStack

Start LocalStack using one of the following methods:

Using the CLI:

localstack start

Using Docker:

docker run --rm -it -p 4566:4566 localstack/localstack

Ensure Docker is Running:
Open Docker Desktop and wait until it displays **"Docker is running."**

LocalStack will simulate AWS services on **port 4566**, allowing local development without needing an actual AWS account.

Step 2: Create a Virtual Private Cloud (VPC)

Create a VPC (Virtual Private Cloud) to define a private network:

aws ec2 create-vpc --cidr-block 10.0.0.0/16 --endpoint-url=%AWS_ENDPOINT_URL%

What This Does:

- Defines a private network (10.0.0.0/16) for AWS resources.
- Returns a VPC ID (e.g., "VpcId": "vpc-123456"), which is required for the next steps.

Step 3: Create a Subnet

A subnet allows instances to communicate within the VPC:

aws ec2 create-subnet --vpc-id vpc-123456 --cidr-block 10.0.1.0/24 --endpoint-url=%AWS_ENDPOINT_URL%

What This Does:

- Creates a smaller network (10.0.1.0/24) within the VPC.
- Returns a Subnet ID (e.g., "SubnetId": "subnet-123456"), which is needed for later steps.

Step 4: Create and Attach an Internet Gateway

An Internet Gateway (IGW) allows resources in the subnet to access the internet.

Create the Internet Gateway:

aws ec2 create-internet-gateway --endpoint-url=%AWS_ENDPOINT_URL%

Attach the Internet Gateway to the VPC:

aws ec2 attach-internet-gateway --internet-gateway-id igw-123456 --vpc-id vpc-123456 --endpoint-url=%AWS_ENDPOINT_URL%

What This Does:

- Enables instances in the VPC to communicate with the internet.
- Returns an Internet Gateway ID (e.g., "InternetGatewayId": "igw-123456").

Step 5: Configure Routing for Internet Access

To allow traffic to and from the internet, a Route Table must be created and updated.

Create a Route Table for the VPC:

aws ec2 create-route-table --vpc-id vpc-123456 --endpoint-url=%AWS_ENDPOINT_URL%

Add a Default Route to Enable Internet Access:

aws ec2 create-route --route-table-id rtb-123456 --destination-cidr-block 0.0.0.0/0 --gateway-id igw-123456 --endpoint-url=%AWS_ENDPOINT_URL%

Associate the Route Table with the Subnet:

aws ec2 associate-route-table --route-table-id rtb-123456 --subnet-id subnet-123456 --endpoint-url=%AWS_ENDPOINT_URL%

What This Does:

- Defines a default route (0.0.0.0/0) to allow internet access.
- Links the route table to the subnet so instances can use the Internet Gateway.

Step 6: Verify the Configuration

List All VPCs:

aws ec2 describe-vpcs --endpoint-url=%AWS_ENDPOINT_URL%

List All Subnets:

aws ec2 describe-subnets --endpoint-url=%AWS_ENDPOINT_URL%

What This Does:

- Confirms the VPC and subnets have been successfully created and configured.
- Ensures all components (VPC, subnet, IGW, and routing) are correctly set up.

Conclusion:

In this experiment, we learned about the concept and components of a Virtual Private Cloud (VPC). We implemented a VPC setup locally using LocalStack, simulating key AWS services. Through this, we gained hands-on experience in creating subnets, internet gateways, and routing configurations.

The output for all executed commands is provided on the next page for reference.

Experiment 4 Output

Starting VPC

```
C:\Users\rawat>aws ec2 create-vpc --cidr-block 10.0.0.0/16 -
-----------------------------------------------------------
|                        CreateVpc                        |
+---------------------------------------------------------+
||                          Vpc                          ||
|+-----------------------+-------------------------------+|
|| CidrBlock             | 10.0.0.0/16                   ||
|| DhcpOptionsId         | default                       ||
|| InstanceTenancy       | default                       ||
|| OwnerId               | 000000000000                  ||
|| State                 | pending                       ||
|| VpcId                 | vpc-e57c31c086cb3ba0c         ||
|+-----------------------+-------------------------------+|
|||              CidrBlockAssociationSet                |||
||+----------------+-----------------------------------+||
||| AssociationId | vpc-cidr-assoc-f48b6c421c5fe1c29  |||
||| CidrBlock     | 10.0.0.0/16                       |||
||+----------------+-----------------------------------+||
||||                 CidrBlockState                    ||||
|||+-------------------+-------------------------------+|||
|||| State             | associated                    ||||
|||+-------------------+-------------------------------+|||
```

Fig 1: VPC Setup

Creating Subnet and API Gateway

```
C:\Users\rawat>aws ec2 create-subnet --vpc-id vpc-e57c31c
-----------------------------------------------------------
|                                              CreateSubne
+---------------------------------------------------------
||                                                Subnet
|+-------------------------------+-----------------------
|| AssignIpv6AddressOnCreation | False
|| AvailabilityZone            | us-east-1a
|| AvailabilityZoneId          | use1-az6
|| AvailableIpAddressCount     | 251
|| CidrBlock                   | 10.0.1.0/24
|| DefaultForAz                | False
|| Ipv6Native                  | False
|| MapPublicIpOnLaunch         | False
|| OwnerId                     | 000000000000
|| State                       | pending
|| SubnetArn                   | arn:aws:ec2:us-east-1:
|| SubnetId                    | subnet-13df13c5c1296a6
|| VpcId                       | vpc-e57c31c086cb3ba0c
|+-------------------------------+-----------------------

C:\Users\rawat>aws ec2 create-internet-gateway --endpoint
-----------------------------------------------
|           CreateInternetGateway             |
+-----------------------------------------------+
||              InternetGateway              ||
|+-----------------------+----------------+|
|| InternetGatewayId    |    OwnerId     ||
|+-----------------------+----------------+|
|| igw-095e7ce5a8f8472d1 | 000000000000  ||
|+-----------------------+----------------+|
```

Fig 2: Subnet and API Gateway Setup

Creating a Route Table

```
C:\Users\rawat>aws ec2 create-route-table --vpc-id vpc-e57c31c086cb3ba0c -
-----------------------------------------------------------------------
|                        CreateRouteTable                             |
+---------------------------------------------------------------------+
||                          RouteTable                               ||
|+-------------+-------------------------+-------------------------+|
|| OwnerId    |      RouteTableId       |          VpcId          ||
|+-------------+-------------------------+-------------------------+|
|| 000000000000| rtb-b16c035cb996d1c2a  | vpc-e57c31c086cb3ba0c  ||
|+-------------+-------------------------+-------------------------+|
||                          Routes                                  ||
|+-------------------------------+----------------+------------+|
|| DestinationCidrBlock         |   GatewayId    |   State    ||
|+-------------------------------+----------------+------------+|
|| 10.0.0.0/16                  | local          | active     ||
|+-------------------------------+----------------+------------+||

C:\Users\rawat>aws ec2 create-route --route-table-id rtb-b16c035cb996d1c2a
T_URL%
-------------------
|   CreateRoute   |
+---------+--------+
| Return | True  |
+---------+--------+
```

Fig 3: Route Table and Creating new Routes

Listing Subnets and Connections

```
C:\Users\rawat>aws ec2 describe-subnets --endpoint-url=%AWS_ENDPOINT_URL%
-----------------------------------------------------------------------
|                                          DescribeSubnets
+---------------------------------------------------------------------
||                                              Subnets
|+-------------------------------+-----------------------------------
|| AssignIpv6AddressOnCreation | False
|| AvailabilityZone            | us-east-1a
|| AvailabilityZoneId          | use1-az6
|| AvailableIpAddressCount     | 4091
|| CidrBlock                   | 172.31.0.0/20
|| DefaultForAz                | True
|| Ipv6Native                  | False
|| MapPublicIpOnLaunch         | True
|| OwnerId                     | 000000000000
|| State                       | available
|| SubnetArn                   | arn:aws:ec2:us-east-1:000000000000:subne
|| SubnetId                    | subnet-0e07399449d53791e
|| VpcId                       | vpc-71f925ef3211ce7cf
|+-------------------------------+-----------------------------------
||                                              Subnets
|+-------------------------------+-----------------------------------
|| AssignIpv6AddressOnCreation | False
|| AvailabilityZone            | us-east-1a
|| AvailabilityZoneId          | use1-az6
|| AvailableIpAddressCount     | 251
|| CidrBlock                   | 10.0.1.0/24
|| DefaultForAz                | False
|| Ipv6Native                  | False
|| MapPublicIpOnLaunch         | False
|| OwnerId                     | 000000000000
|| State                       | available
```

Fig 4: List of Subnets

Experiment 5

Aim: This experiment provides students with hands-on experience in serverless computing by creating a serverless function using AWS Lambda, Google Cloud Functions, or Azure Functions, enabling them to understand the serverless paradigm.

Serverless Computing and AWS Lambda:

What is Serverless Computing?

Serverless computing refers to a cloud computing model where the cloud provider manages the infrastructure and automatically allocates resources as needed. The term "serverless" does not mean that there are no servers involved; rather, it means that the responsibility of managing servers is abstracted away from the developer. The cloud provider handles provisioning, scaling, and managing the servers, allowing the developer to focus on building and deploying the application code.

What is AWS Lambda?

AWS Lambda is a serverless compute service provided by Amazon Web Services (AWS) that allows users to run code without provisioning or managing servers. With AWS Lambda, developers can focus solely on writing the code for their applications, while AWS automatically handles the infrastructure, scaling, and maintenance.

Creation of a Lambda Function with LocalStack:

1. Creating the Lambda Function:

To create a Lambda function using LocalStack, the first step involves writing the Lambda function code. The function will be triggered by an event and should return a simple response.

Example Lambda Function Code (lambda_function.py):

```
def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': 'Hello from LocalStack Lambda!'
    }
```

This function will output a basic response with a 200 OK status code when invoked.

2. Deploying the Lambda Function:

Before deploying the Lambda function, the Python code needs to be packaged into a ZIP file. This ZIP file will be used to deploy the function to LocalStack.

Steps for Packaging the Lambda Function (on Windows):

i. Navigate to the directory where your lambda_function.py file is located.
ii. Right-click the lambda_function.py file and select Send to > Compressed (zipped) folder

from the context menu.
iii. Rename the resulting .zip file to function.zip.

After the Lambda function is packaged, it is ready for deployment.

3. Checking the Function Creation Status:

After the Lambda function is deployed, its status can be checked to ensure it has been successfully created. Initially, the status might be in a Pending state, indicating that LocalStack is still processing the request to create the function.

Command to Check the Function's Status:

aws --endpoint-url=http://localhost:4566 lambda get-function --function-name myLambdaFunction

If the function is still in the Pending state, wait for a few moments before checking again. Once the function's status is Active or Available, it can be invoked.

4. Invoking the Lambda Function:

After the function is in the Active state, it can be invoked using the AWS CLI. The invocation will trigger the Lambda function and return the response in a file (in this case, output.txt).

Command to Invoke the Function:

aws --endpoint-url=http://localhost:4566 lambda invoke --function-name myLambdaFunction output.txt

Docker Image for Lambda:

docker pull public.ecr.aws/lambda/python:3.8

Once this command is executed, Docker will download the required Python 3.8 Lambda image. Below is the expected output when the image is successfully pulled:

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 5\Codes>docker pull public.ecr.aws/lambda/python:3.8
3.8: Pulling from lambda/python
bc2b3a540f9b: Pull complete
7e7a8ab075f3: Pull complete
eb63e7acde8e: Pull complete
35e8cd243a50: Pull complete
d4160264f321: Pull complete
32e94a394cab: Pull complete
Digest: sha256:93e78742873d3ad0c28582366b217ce5169889f4d63d61179598c2a3dc6142ff
Status: Downloaded newer image for public.ecr.aws/lambda/python:3.8
public.ecr.aws/lambda/python:3.8

Command to Create the Lambda Function:

**aws --endpoint-url=http://localhost:4566 lambda create-function **
**--function-name myLambdaFunction **
**--runtime python3.8 **

**--role arn:aws:iam::000000000000:role/execution_role **
**--handler lambda_function.lambda_handler **
--zip-file fileb://lambda_function.zip

Breakdown of the command:

1. aws --endpoint-url=http://localhost:4566 :
   This specifies that the AWS CLI should interact with LocalStack rather than the actual AWS cloud. localhost:4566 is the default endpoint for LocalStack's services.

2. lambda create-function :
   This is the AWS CLI command used to create a new Lambda function.
3. --function-name myLambdaFunction :
   This specifies the name of the Lambda function being created. In this case, the function is named myLambdaFunction.
4. --runtime python3.8: Show the version of python

Viewing the Output:

aws --endpoint-url=http://localhost:4566 lambda invoke --function-name myLambdaFunction output.txt

Command to View the Output:

type output.txt

Expected Output:
{"statusCode": 200, "body": "Hello from LocalStack Lambda!"}

Explanation of the Output:

The content of the output.txt file is a JSON response returned by the Lambda function:

1. statusCode: 200 :
   This indicates that the Lambda function executed successfully and returned an HTTP status code of 200, which signifies a successful response.
2. body: "Hello from LocalStack Lambda!" :
   This is the body of the response. It contains the message "Hello from LocalStack Lambda!", which was defined in the Lambda function. This message is returned as part of the response when the function is triggered.

Conclusion:

In this experiment, we successfully set up and deployed an AWS Lambda function using LocalStack. We created, packaged, deployed, and invoked the function locally without needing access to the actual AWS cloud.

The output for all the executed commands is provided on the next page for reference.

Experiment 5 Output

Creating Lambda Function

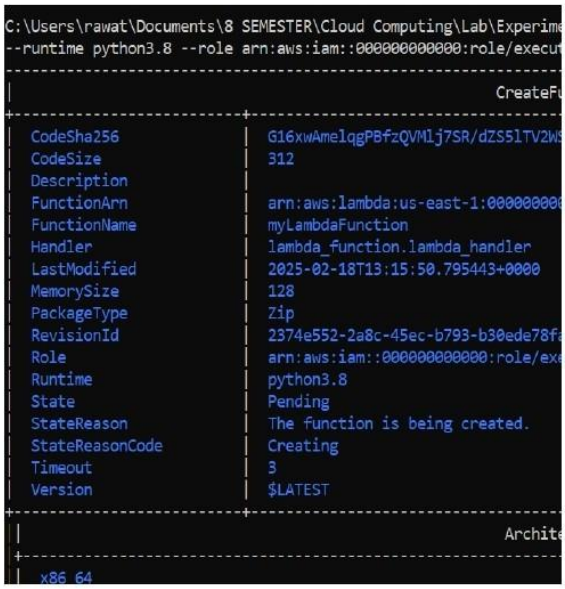


Fig 1: VPC Setup

Pulling Amazon ECR for execution

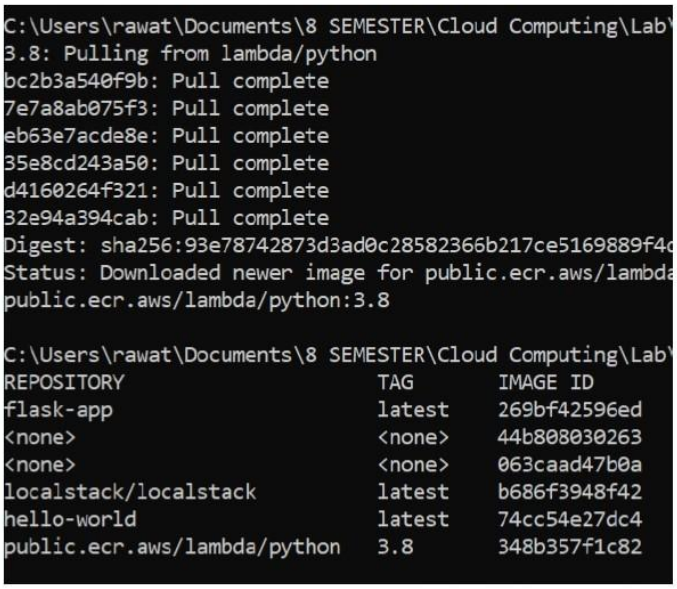


Fig 2: Docker Pulling and Output Image

Listing working Functions

Fig 3: Lambda working Function

Lambda Function invoking with Payload

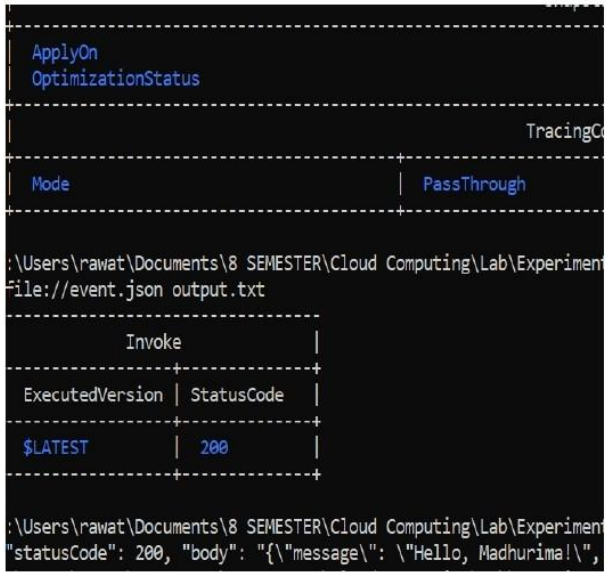


Fig 4: Lambda Function Output

Experiment 6

Aim: In this experiment, students will configure load balancing and auto-scaling in the cloud environment to handle varying traffic loads efficiently, using tools such as AWS Elastic Load Balancing (ELB) and Google Cloud Load Balancer.

Load Balancing:

Load balancing is the process of distributing incoming network traffic across multiple servers to ensure no single server bears too much demand. This improves performance, reliability, and fault tolerance by preventing overload and ensuring continuous availability.

Key Benefits of Load Balancing

Improved Performance: Spreads traffic efficiently across multiple servers.
High Availability & Redundancy: Ensures uptime even if a server fails.
Scalability: Easily accommodates increased traffic by adding more servers.
Optimized Resource Utilization: Ensures servers are used efficiently.

Simple Load Balancing Function with LocalStack and Flask:

1. Creating a Virtual Private Cloud (VPC):

aws ec2 create-vpc --cidr-block 10.0.0.0/16 --endpoint-url=http://localhost:4566

2. Creating a Subnet:

aws ec2 create-subnet --vpc-id vpc-66375b0cbe498b519 --cidr-block 10.0.1.0/24 --endpoint-url=http://localhost:4566

3. Creating an Application Load Balancer:

aws elbv2 create-load-balancer --name my-load-balancer --subnets subnet-3f40f7c6e3a26040f --security-groups default --type application --endpoint-url=http://localhost:4566

Setting Up Nginx Containers with Load Balancing

4. Running Backend Containers:

Navigate to the project directory:
cd "C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 6\Codes"
Run backend containers using Docker:
docker run -d --name backend1 nginx
docker run -d --name backend2 nginx

5. Verify Available Docker Images:

docker images
Expected output includes:
nginx and **flask-app**

6. Creating and Running a Load Balancer Container:

docker run -d --name load-balancer -p 8080:80 -v "C:/Users/rawat/Documents/8 SEMESTER/Cloud Computing/Lab/Experiment 6/Codes/nginx.conf:/etc/nginx/nginx.conf" nginx
Check logs for potential errors:
docker logs load-balancer

7. Fixing 'Host Not Found' Error

The error "host not found in upstream 'backend1:80'" indicates Nginx cannot resolve **backend1**. All containers must be in the same network.

Step 1: Create a Network

docker network create my-network

Step 2: Start Backend Containers in the Same Network

Remove existing containers:
docker rm -f backend1 backend2
Restart within the custom network:
docker run -d --name backend1 --network my-network nginx
docker run -d --name backend2 --network my-network nginx

Step 3: Update nginx.conf

Ensure the following structure in nginx.conf:

```
events { }

http {
  upstream backend_servers {
     server backend1:80;
     server backend2:80;
  }
  server {
     listen 80;
     location / {
        proxy_pass http://backend_servers;
     }
  }
}
```

Restart the load balancer container:
docker rm -f load-balancer

docker run -d --name load-balancer --network my-network -p 8080:80 -v "C:/Users/rawat/Documents/8 SEMESTER/Cloud Computing/Lab/Experiment 6/Codes/nginx.conf:/etc/nginx/nginx.conf" nginx

Step 4: Verify Setup

docker ps
Access via browser:
http://localhost:8080

Step 5: Restart the Load Balancer (if needed)

docker rm -f load-balancer
docker run -d --name load-balancer --network my-network -p 8080:80 -v "C:/Users/rawat/Documents/8 SEMESTER/Cloud Computing/Lab/Experiment 6/Codes/nginx.conf:/etc/nginx/nginx.conf" nginx

Step 6: Final Verification

docker ps
Open:
http://localhost:8080

LocalStack Load Balancer - Flask Simulation

Since Nginx doesn't support load balancing for LocalStack directly, a Python Flask app simulates round-robin load balancing.

Flask-based Round Robin Load Balancer Simulation

Author: Madhurima Rawat
Date: March 6, 2025

Description:
This script simulates round-robin load balancing using Flask, Matplotlib, and Seaborn. It assigns tasks to backend servers cyclically and visualizes the distribution using a bar chart.

Usage

Run the script:
python app.py
Access in browser:
http://localhost:5000

Conclusion:

This project demonstrates a practical implementation of load balancing using Docker, Nginx, and Flask. It ensures efficient traffic distribution across backend containers through round-robin logic. The setup enhances scalability, reliability, and real-time simulation in a local environment.

The output for all the executed commands is provided on the next page for reference.

Experiment 6 Output

Creating VPC

```
C:\Users\rawat>aws ec2 create-vpc --cidr-block 10.0.0.0/16 --endpoint-u
-------------------------------------------------------------
|                        CreateVpc                          |
+-----------------------------------------------------------+
||                          Vpc                            ||
|+----------------------+----------------------------------+|
||  CidrBlock           |  10.0.0.0/16                     ||
||  DhcpOptionsId       |  default                         ||
||  InstanceTenancy     |  default                         ||
||  OwnerId             |  000000000000                    ||
||  State               |  pending                         ||
||  VpcId               |  vpc-66375b0cbe498b519           ||
|+----------------------+----------------------------------+|
|||              CidrBlockAssociationSet                  |||
||+---------------+-------------------------------------+||
|||  AssociationId |  vpc-cidr-assoc-0c82d2b7eaae6089a   |||
|||  CidrBlock     |  10.0.0.0/16                        |||
||+---------------+-------------------------------------+||
||||                 CidrBlockState                     ||||
|||+------------------+-----------------------------+|||
||||  State           |  associated                 ||||
|||+------------------+-----------------------------+|||
```

Fig 1: VPC Setup

Pulling Nginx for Execution

```
943ea0f0c2e4: Pull complete
103f50cb3e9f: Pull complete
Digest: sha256:9d6b58feebd2dbd3c56ab5853333d627cc6e281011cfd6050fa4bcf2072c9496
Status: Downloaded newer image for nginx:latest
63d7914e78bb0791288824fffdcf46d77bd92b4f895de4a403c88012dc87b601

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 6\Codes>docker run -d
e565ff4cf0081c7965a7634fb1de6018e88d927dd80ca179822aa7a33994f87c

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 6\Codes>docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
hackvortex-backend            latest    14e63c26d40b   12 hours ago    1.05GB
postgres                      15        e45d3f5ec589   7 days ago      430MB
flask-app                     latest    269bf42596ed   3 weeks ago     126MB
nginx                         latest    b52e0b094bc0   4 weeks ago     192MB
<none>                        <none>    44b808030263   4 weeks ago     126MB
<none>                        <none>    063caad47b0a   4 weeks ago     126MB
localstack/localstack         latest    b686f3948f42   6 weeks ago     1.18GB
hello-world                   latest    74cc54e27dc4   6 weeks ago     10.1kB
public.ecr.aws/lambda/python  3.8       348b357f1c82   7 weeks ago     575MB
python                        3.9       9f98746a2033   3 months ago    999MB
```

Fig 2: Docker Pulling and Output Image

Running Flask Load balancing App

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment
[+] Building 18.4s (9/9) FINISHED                              dock
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 332B
 => [internal] load metadata for docker.io/library/python:3.9-slim
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/4] FROM docker.io/library/python:3.9-slim
 => [internal] load build context
 => => transferring context: 1.35kB
 => CACHED [2/4] WORKDIR /app
 => [3/4] COPY . .
 => [4/4] RUN pip install --no-cache-dir flask requests
 => exporting to image
 => => exporting layers
 => => writing image sha256:6e4cbd50721df2af76ffcc7bcf3ae2550a3b500
 => => naming to docker.io/library/my-flask-app
```

Fig 3: Flask App

Output with graph showing processes

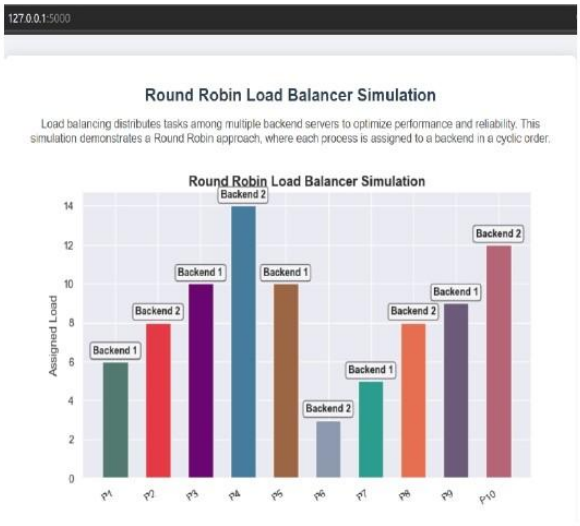


Fig 4: Flask App Output

Experiment 7

Aim: This experiment introduces students to setting up and managing cloud-based relational databases such as AWS RDS or Google Cloud SQL, enabling them to practice database management in a cloud environment.

Cloud Databases and Data Management:

Overview
This experiment covers setting up and managing cloud-based relational databases using PostgreSQL, Docker, and LocalStack. It provides hands-on experience with database management in a simulated cloud environment, replicating real-world cloud database operations using containerized solutions.

What are Databases?
Databases store, organize, and manage data efficiently. They are crucial in applications ranging from websites to large-scale enterprise systems.

Database Operations with Postgres:

1. Creating an RDS Instance Using LocalStack
aws rds create-db-instance --db-instance-identifier mydb --db-instance-class db.t3.micro --engine mysql --master-username admin --master-user-password password --allocated-storage 20 --endpoint-url=http://localhost:4566
This command attempts to create a mock RDS instance using LocalStack. The error indicates that either LocalStack is not running or the RDS API requires the pro version.

docker run --rm -d --name localstack_main -p 4566:4566 localstack/localstack
Runs LocalStack in a Docker container to simulate AWS services locally.

2. Starting a PostgreSQL Container
docker start my-postgres
docker start postgres
These commands attempt to start containers named my-postgres and postgres. The error suggests the containers do not exist.

docker run --name my-postgres -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=password -e POSTGRES_DB=mydb -p 5432:5432 -d postgres:15
Creates and starts a new PostgreSQL container.

3. Listing Available Docker Images
docker images
Lists all Docker images available locally. This helps confirm whether required images like postgres or localstack are present.

docker pull postgres
Pulls the official PostgreSQL image from Docker Hub.

4. Starting a PostgreSQL Container Again
docker start postgres
Retries starting the container named postgres, but it doesn't exist.

docker ps -a
Shows all containers (running and exited), allowing verification of container names.

5. Listing All Containers
docker ps -a
Reveals containers that were created. Confirms that the postgres container is not available.

6. Running a New PostgreSQL Container
docker run --name my-postgres -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=password -e POSTGRES_DB=mydb -p 5432:5432 -d postgres:15
Tries to run a new PostgreSQL container on port 5432, but the port is already in use.

7. Resolving Port Conflict and Running PostgreSQL on a Different Port
docker run --name my-new-postgres -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=password -e POSTGRES_DB=mydb -p 5433:5432 -d postgres:15
Runs PostgreSQL on external port 5433 to avoid port conflict.

8. Verifying Running Containers
docker ps
Lists currently running containers. Shows my-new-postgres and LocalStack if they are up.

9. Connecting to PostgreSQL and Performing SQL Operations
docker exec -it my-new-postgres psql -U admin -d mydb
Opens an interactive session to the PostgreSQL instance inside the container.

CREATE TABLE students (id SERIAL PRIMARY KEY, name VARCHAR(100) NOT NULL, email VARCHAR(100) UNIQUE NOT NULL);
Creates a students table with ID, name, and unique email.

INSERT INTO students (name, email) VALUES ('Alice Johnson', 'alice@example.com'), ('Bob Smith', 'bob@example.com'), ('Charlie Brown',

'charlie@example.com');
Inserts three records into the students table.

10. Performing SQL Queries
SELECT * FROM students;
Retrieves all student records.

UPDATE students SET email = 'bob.smith@example.com' WHERE name = 'Bob Smith';
Updates Bob Smith's email address.

DELETE FROM students WHERE name = 'Charlie Brown';
Deletes the record of Charlie Brown.

SELECT * FROM students WHERE name LIKE 'A%';
Selects students whose names start with 'A'.

\q
Exits the PostgreSQL command-line interface.

Conclusion:

This hands-on experiment simulates real-world cloud database deployment using Docker and LocalStack. It highlights common errors, their causes, and practical resolutions for managing PostgreSQL containers. By resolving port conflicts and confirming service coverage, cloud-like database operations can be effectively emulated locally.

The output for all the executed commands is provided on the next page for reference.

Experiment 7 Output

Creating Postgres Container

```
C:\Users\rawat>docker run --name my-new-postgres -e POSTGRES_USER=
b2efdca3c6f0af6cf4154fce236f0b66b5efba0f4f9e14972c94b3e0a5afa9de

C:\Users\rawat>docker ps
CONTAINER ID   IMAGE                  COMMAND                  CR
      NAMES
b2efdca3c6f0   postgres:15            "docker-entrypoint.s…"   42
      my-new-postgres
a10c5a71f625   localstack/localstack  "docker-entrypoint.sh"   3
/tcp   localstack-main
6c3f6fc35cc4   hackvortex-backend     "gunicorn --bind 0.0…"   21
       backend_app
dedb2d1ac081   postgres:15            "docker-entrypoint.s…"   21
       postgres_db
```

Fig 1: Postgres Setup

Accessing via Interactive shell

```
C:\Users\rawat>docker exec -it my-new-postgre
psql (15.12 (Debian 15.12-1.pgdg120+1))
Type "help" for help.

mydb=# CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
INSERT INTO students (name, email) VALUES
('Alice Johnson', 'alice@example.com'),
('Bob Smith', 'bob@example.com'),
('Charlie Brown', 'charlie@example.com');
CREATE TABLE
INSERT 0 3
mydb=# SELECT * FROM students;
 id |     name      |        email
----+---------------+---------------------
  1 | Alice Johnson | alice@example.com
  2 | Bob Smith     | bob@example.com
  3 | Charlie Brown | charlie@example.com
(3 rows)
```

Fig 2: Creating Tables and Data insertion

Data Access & Queries

```
CREATE TABLE
INSERT 0 3
mydb=# SELECT * FROM students;
 id |     name      |        email
----+---------------+---------------------
  1 | Alice Johnson | alice@example.com
  2 | Bob Smith     | bob@example.com
  3 | Charlie Brown | charlie@example.com
(3 rows)

mydb=# UPDATE students SET email = 'bob.smith@example.com' WHERE name = 'Bob Smith';
UPDATE 1
mydb=# DELETE FROM students WHERE name = 'Charlie Brown';
DELETE 1
mydb=# SELECT * FROM students WHERE name LIKE 'A%';
 id |     name      |       email
----+---------------+-------------------
  1 | Alice Johnson | alice@example.com
(1 row)

mydb=# \q
```

Fig 3: Postgres Queries and Output

Experiment 8

Aim: In this experiment, students will learn to manage users, permissions, and security policies in the cloud using Identity and Access Management (IAM) tools from platforms like AWS IAM, Google Cloud IAM, or Azure Active Directory.

Identity and Access Management (IAM):

Identity and Access Management (IAM) is a framework that ensures the right individuals and entities have appropriate access to technology resources. In AWS IAM, users can securely control access to AWS services and resources.

Key Concepts of IAM

- **Identities in IAM:** IAM identities define who can access AWS resources and what actions they can perform. These include:
    - Users – Individual accounts with login credentials.
    - Groups – Collections of users sharing permissions.
    - Roles – Assigned temporary permissions for AWS services or external users.
    - Example: A developer needs access to an S3 bucket. Instead of assigning permissions directly, they are added to the "Developers" IAM group, which has the required permissions.

Identity and Access Management (IAM) with Localstack:

These commands use the AWS CLI to manage IAM users, roles, and policies. The --endpoint-url=%AWS_ENDPOINT_URL% parameter is used to run the commands on LocalStack, which emulates AWS services locally.

1. Create a User

aws iam create-user --user-name test-user --endpoint-url=%AWS_ENDPOINT_URL%

What it does:

Creates a new IAM user named test-user. This user can be assigned roles and policies later.

Expected Output:

```
{
"User": {
"Path": "/",
"UserName": "test-user",
"UserId": "AIDAEXAMPLEID",
"Arn": "arn:aws:iam::123456789012:user/test-user",
"CreateDate": "2024-03-06T12:00:00Z"
}
}
```

2. Create a Role

aws iam create-role --role-name ec2-role --assume-role-policy-document file://trust-policy.json --endpoint-url=%AWS_ENDPOINT_URL%

What it does:

Creates an IAM role named ec2-role. The role uses a trust policy (stored in trust-policy.json) to define which AWS services or users can assume this role.

Explanation of the Trust Policy Document

This Trust Policy is a JSON document that defines who (which AWS service, user, or account) is allowed to assume an IAM role. This is particularly useful when granting permissions to AWS services like EC2, Lambda, or other entities to access AWS resources.

Breakdown of the JSON Policy

```
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Principal": {
"Service": "ec2.amazonaws.com"
},
"Action": "sts:AssumeRole"
}
]
}
```

"Version": "2012-10-17"

Specifies the version of the IAM policy language being used. "2012-10-17" is the latest and recommended version for AWS policies.

"Statement": [...]

Contains one or more statements defining the permission rules.

Inside the Statement Block:

"Effect": "Allow"

Defines whether the policy allows or denies the action. "Allow" means this policy grants permissions.

"Principal": { "Service": "ec2.amazonaws.com" }

Specifies who is allowed to assume this role. "Service": "ec2.amazonaws.com" means that the EC2 service (Amazon Elastic Compute Cloud) is allowed to assume this role.

"Action": "sts:AssumeRole"

Specifies the action that is permitted. "sts:AssumeRole" allows the EC2 service to assume this role using the AWS Security Token Service (STS). This is required when an EC2 instance needs temporary credentials to interact with other AWS services on behalf of the IAM role.

Use Case Example

When you create an IAM Role (e.g., ec2-role) with this trust policy, EC2 instances can assume this role. This is useful when an EC2 instance needs access to S3, DynamoDB, Lambda, etc., without using long-term credentials. AWS securely provides temporary credentials to the EC2 instance through this role.

How This Works in Practice

1. Create an IAM Role with this trust policy.
2. Attach permissions to the role (e.g., S3 Read Access).
3. Assign the IAM Role to an EC2 instance when launching it.
4. The EC2 instance automatically receives temporary credentials and can access AWS resources as defined in the role's permissions.
   Expected Output:
   {
   "Role": {
   "Path": "/",
   "RoleName": "ec2-role",
   "RoleId": "AROAXAMPLEID",
   "Arn": "arn:aws:iam::123456789012:role/ec2-role",
   "CreateDate": "2024-03-06T12:05:00Z",
   "AssumeRolePolicyDocument": {
   "Version": "2012-10-17",
   "Statement": [
   {
   "Effect": "Allow",
   "Principal": {
   "Service": "ec2.amazonaws.com"
   },
   "Action": "sts:AssumeRole"
   }
   ]
   }
   }
   }

3. Attach a Policy to the User

aws iam attach-user-policy --user-name test-user --policy-arn arn:aws:iam::aws:policy/AdministratorAccess --endpoint-url=%AWS_ENDPOINT_URL%

What it does:
Attaches the AdministratorAccess policy to test-user, giving full access to AWS resources.
Expected Output:
No direct output. The policy is attached successfully.
To verify, use the list-attached-user-policies command.

4. List Users

aws iam list-users --endpoint-url=%AWS_ENDPOINT_URL%

What it does:
Lists all IAM users in the account.
Expected Output:
{

"Users": [
{
"Path": "/",
"UserName": "test-user",
"UserId": "AIDAEXAMPLEID",
"Arn": "arn:aws:iam::123456789012:user/test-user",
"CreateDate": "2024-03-06T12:00:00Z" }
]
}

5. List Roles
aws iam list-roles --endpoint-url=%AWS_ENDPOINT_URL%
What it does:
Lists all IAM roles in the account.
Expected Output:
{
"Roles": [
{
"Path": "/",
"RoleName": "ec2-role",
"RoleId": "AROAXAMPLEID",
"Arn": "arn:aws:iam::123456789012:role/ec2-role",
"CreateDate": "2024-03-06T12:05:00Z" }] }

6. List Attached Policies for a User
aws iam list-attached-user-policies --user-name test-user --endpoint-url=%AWS_ENDPOINT_URL%
What it does:
Displays the policies attached to test-user.
Expected Output:
{
"AttachedPolicies": [
{
"PolicyName": "AdministratorAccess",
"PolicyArn": "arn:aws:iam::aws:policy/AdministratorAccess"
}] }

These commands help manage IAM users, roles, and permissions efficiently. Using LocalStack, this can be tested in a local environment before deploying in AWS.

Conclusion:

IAM enables secure access control by managing users, roles, and permissions effectively. Using LocalStack allows safe local testing of IAM configurations without impacting live AWS environments. This ensures smoother deployment and better security compliance in production setups.

The output for all the executed commands is provided on the next page for reference.

Experiment 8 Output

Creating IAM User

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experim
-----------------------------------------------------------------
|                         CreateUser                            |
+-----------------------------------------------------------------+
||                           User                                ||
|+------------+-------------------------------------------------+|
||  Arn       |  arn:aws:iam::000000000000:user/test-user       ||
||  CreateDate|  2025-03-06T15:43:02.050000Z                    ||
||  Path      |  /                                              ||
||  UserId    |  7ll7td3a2x1shy4p7x2b                           ||
||  UserName  |  test-user                                      ||
|+------------+-------------------------------------------------+|
```

Fig 1: IAM User

Listing Users

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experim
-----------------------------------------------------------------
|                         ListUsers                             |
+-----------------------------------------------------------------+
||                           Users                               ||
|+------------+-------------------------------------------------+|
||  Arn       |  arn:aws:iam::000000000000:user/test-user       ||
||  CreateDate|  2025-03-06T15:43:02.050000Z                    ||
||  Path      |  /                                              ||
||  UserId    |  7ll7td3a2x1shy4p7x2b                           ||
||  UserName  |  test-user                                      ||
|+------------+-------------------------------------------------+|
```

Fig 3: Users List Table

Listing Policies

```
-----------------------------------------------------------------
              ListAttachedUserPolicies
-----------------------------------------------------------------
                  AttachedPolicies
-------------------------------------+---------------------------
          PolicyArn                  |       PolicyName
-------------------------------------+---------------------------
:iam::aws:policy/AdministratorAccess |  AdministratorAccess
-------------------------------------+---------------------------
```

Fig 4: Policies List Table

Creating Roles

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experime
-endpoint-url=%AWS_ENDPOINT_URL%
-----------------------------------------------------------------
|                         CreateRole                            |
+-----------------------------------------------------------------+
||                           Role                                ||
|+------------+-------------------------------------------------+|
||  Arn       |  arn:aws:iam::000000000000:role/ec2-role        ||
||  CreateDate|  2025-03-06T15:44:48.493000Z                    ||
||  Path      |  /                                              ||
||  RoleId    |  AROAQAAAAAAANMF4UL4W3                          ||
||  RoleName  |  ec2-role                                       ||
|+------------+-------------------------------------------------+|
|||              AssumeRolePolicyDocument                       |||
||+-----------------------+-------------------------------------+||
|||  Version              |  2012-10-17                         |||
||+-----------------------+-------------------------------------+||
||||                       Statement                            ||||
|||+-----------------+-------------------------------------------+|||
||||  Action         |  sts:AssumeRole                           ||||
||||  Effect         |  Allow                                    ||||
|||+-----------------+-------------------------------------------+|||
||||                       Principal                            ||||
|||+----------------+-------------------------------------------+||||
||||  Service       |  ec2.amazonaws.com                        ||||
|||+----------------+-------------------------------------------+||||
```

Fig 2: Roles Table

Listing Roles

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 8\Co
-----------------------------------------------------------------
|                         ListRoles                             |
+-----------------------------------------------------------------+
||                           Roles                               ||
|+-----------------------+-------------------------------------+|
||  Arn                  |  arn:aws:iam::000000000000:role/ec2-role  ||
||  CreateDate           |  2025-03-06T15:44:48.493575Z                ||
||  MaxSessionDuration   |  3600                                       ||
||  Path                 |  /                                          ||
||  RoleId               |  AROAQAAAAAAANMF4UL4W3                      ||
||  RoleName             |  ec2-role                                   ||
|+-----------------------+-------------------------------------+|
|||                  AssumeRolePolicyDocument                   |||
||+-----------------------+-------------------------------------+||
|||  Version              |  2012-10-17                         |||
||+-----------------------+-------------------------------------+||
||||                       Statement                            ||||
|||+-----------------+-------------------------------------------+|||
||||  Action         |  sts:AssumeRole                           ||||
||||  Effect         |  Allow                                    ||||
|||+-----------------+-------------------------------------------+|||
||||                       Principal                            ||||
|||+-----------------+-------------------------------------------+||||
||||  Service        |  ec2.amazonaws.com                        ||||
|||+-----------------+-------------------------------------------+||||
```

Fig 5: Roles Output

Experiment 9

Aim: This experiment focuses on cloud monitoring services such as AWS CloudWatch, Google Stackdriver, or Azure Monitor, where students will track resources and application performance in a cloud environment.

What is AWS CloudWatch?

AWS CloudWatch is a monitoring and observability service that collects and tracks metrics, logs, and events to provide insights into cloud resources and applications.

Key Features of CloudWatch:

- **Metrics Monitoring:** Collects and analyzes performance metrics for AWS services.
- **Log Management:** Stores and processes logs for applications and infrastructure.
- **Alarms & Notifications:** Triggers alerts based on threshold values.
- **Dashboards & Visualization:** Provides real-time monitoring dashboards.
- **Event-driven Automation:** Automates actions based on events.

Understanding Log Management

Log management is a crucial part of cloud monitoring and involves the collection, storage, and analysis of logs.

Six Components of Log Management:

- **Log Collection:** Logs are gathered from various sources (servers, applications, cloud services).
- **Log Aggregation:** Logs are combined and stored in a centralized system like CloudWatch Logs.
- **Log Storage:** Logs are kept in CloudWatch Log Groups and Log Streams.
- **Log Analysis:** Tools like AWS CloudWatch Insights analyze log data for patterns and trends.
- **Log Monitoring & Alerts:** CloudWatch can send alerts based on predefined conditions.
- **Compliance & Retention:** Logs are archived based on security and compliance requirements.

What Are Log Streams?

Log streams represent a sequence of log events from the same source (e.g., an application instance, EC2 instance, or microservice).

A web server might have separate log streams for each instance it runs on.
CloudWatch logs are organized in log groups, and each log group contains multiple log streams.

Real-life Use Case: CloudWatch Logs & Metrics

Scenario:

A company hosts its web application on AWS EC2 instances and wants to monitor CPU usage and analyze logs for errors.

- Enable CloudWatch Logs on EC2 instances to store application logs.
- Create a log group /my/app/logs and define log streams for different instances.
- Send logs & metrics (e.g., CPU usage, errors, response time) to CloudWatch.
- Set up alarms to notify the DevOps team when CPU usage exceeds 80%.
- Analyze logs using CloudWatch Insights to detect anomalies.

AWS CLI Commands for LocalStack (CloudWatch and Logs):

1. Create a Log Group

aws --endpoint-url=http://localhost:4566 logs create-log-group --log-group-name /my/app/logs
Description: Creates a log group named /my/app/logs in AWS CloudWatch Logs.

2. Create a Log Stream

aws --endpoint-url=http://localhost:4566 logs create-log-stream --log-group-name /my/app/logs --log-stream-name my-stream
Description: Creates a log stream named my-stream within the /my/app/logs log group.

3. Put Log Events

aws --endpoint-url=http://localhost:4566 logs put-log-events --log-group-name /my/app/logs --log-stream-name my-stream --log-events "[{\"timestamp\":1741348140000,\"message\":\"Test log entry\"}]"
Description: Adds a log entry with a timestamp (1741348140000) and message (Test log entry) to the my-stream log stream.

4. Put Metric Data

aws --endpoint-url=http://localhost:4566 cloudwatch put-metric-data --namespace "MyApp" --metric-name "CPUUsage" --value 75
Description: Publishes a custom CloudWatch metric CPUUsage with a value of 75 in the MyApp namespace.

5. List CloudWatch Metrics

aws --endpoint-url=http://localhost:4566 cloudwatch list-metrics --namespace "MyApp"
Description: Lists all CloudWatch metrics under the MyApp namespace.

6. Describe Log Groups

aws --endpoint-url=http://localhost:4566 logs describe-log-groups
Description: Retrieves details of all log groups available in CloudWatch Logs.

7. Retrieve Log Events

aws --endpoint-url=http://localhost:4566 logs get-log-events --log-group-name /my/app/logs --log-stream-name my-stream
Description: Fetches the log events from my-stream under /my/app/logs.

Conclusion:

This experiment demonstrates how CloudWatch logging and monitoring work in a local environment using Docker, LocalStack, and AWS CLI. It helps in log management, real-time monitoring, and alerting before deploying applications to AWS.

The output for all the executed commands is provided on the next page for reference.

Experiment 9 Output

Starting Docker

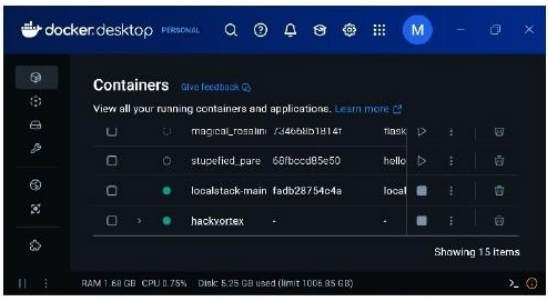


Fig 1: Starting Docker

Starting Localstack

Fig 2: Starting Localstack

Creating Logs and Putting time stamps in logs

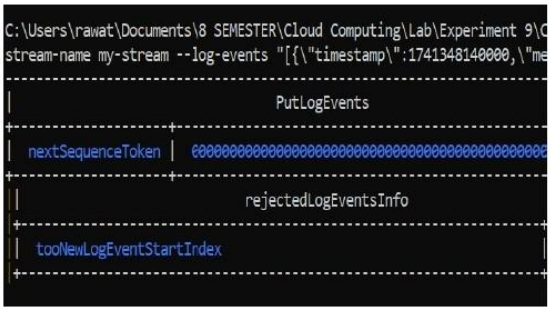


Fig 3: Logs Table

Listing Metrics

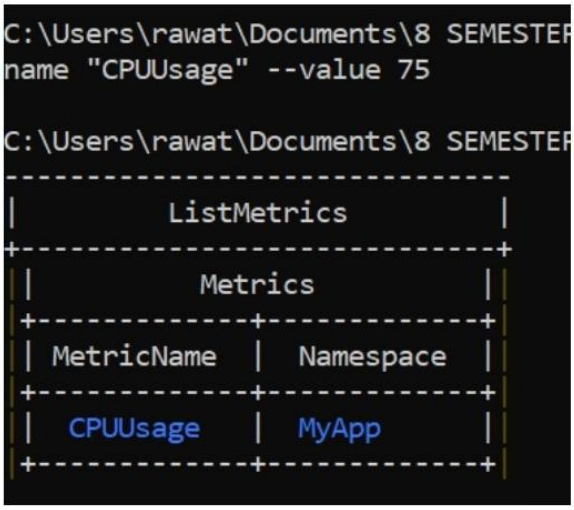


Fig 4: Metrics List

Describing Log Groups

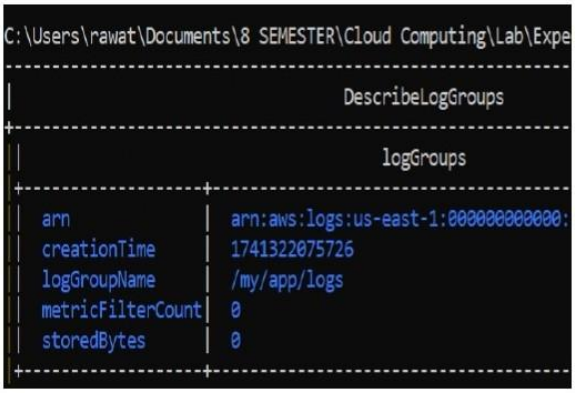


Fig 5: Log Group Table

Log Events

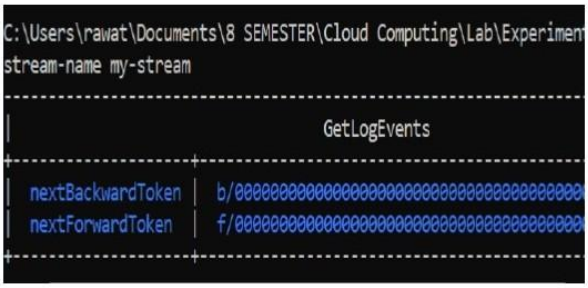


Fig 6: Log Events Table

Experiment 10

Aim: In this experiment, students will set up a Continuous Integration/Continuous Deployment (CI/CD) pipeline using cloud services like AWS CodePipeline, Google Cloud Build, or Jenkins on the cloud, providing hands-on experience with cloud-based automation tools.

Understanding CI/CD Pipelines:

How It Works:

1. **GitHub Actions**: Automates build, test, and deployment processes directly from GitHub. It triggers workflows on code commits, pull requests, or scheduled intervals.
2. **Docker**: Creates containerized environments for running applications, ensuring that the pipeline runs consistently across different systems.
3. **LocalStack**: Simulates AWS cloud services locally (S3, Lambda, DynamoDB, etc.). This allows developers to test AWS-related workflows without real AWS costs.
4. **AWS CLI & S3**: AWS CLI automates interactions with AWS services. S3 (Simple Storage Service) acts as a storage bucket for deployment artifacts.

Use Cases

- **Automated Deployment Pipelines**: Code is automatically tested, built, and deployed to cloud environments. Reduces manual intervention, ensuring faster release cycles.
- **Simulating AWS Services Locally**: Developers can test AWS-dependent applications without incurring AWS costs. Ideal for offline development or local testing of cloud-native applications.
- **Cloud-Based Workflow Testing**: Ensures infrastructure as code (IaC) principles by defining cloud setups in version control. Useful for DevOps teams deploying applications on AWS.

Real-Life Examples

- **Mobile App Backend Development**: Automatically deploy backend APIs (hosted on AWS Lambda) after each successful commit. Use LocalStack to test S3 storage operations without using real AWS resources.
- **AI/ML Model Deployment**: Automate pushing trained ML models to S3 for cloud inference. Use GitHub Actions to validate the model before deployment.

Key Benefits

- **Faster Development Cycles**: Reduces manual deployment efforts.
- **Cost-Efficient Testing**: Simulates AWS without incurring costs.

CI/CD Pipelines with GitHub Actions, Docker, and LocalStack:

1. **Creating an S3 Bucket**
   Command:
   aws --endpoint-url=http://localhost:4566 s3 mb s3://my-ci-cd-artifacts
   Explanation:
   aws s3 mb → Creates a new S3 bucket.
   s3://my-ci-cd-artifacts → The name of the bucket being created.
   --endpoint-url=http://localhost:4566 → Uses LocalStack to simulate AWS services.
   Output:
   make_bucket: my-ci-cd-artifacts

2. **Attempting to Create a CodeCommit Repository**
   Command:
   aws --endpoint-url=http://localhost:4566 codecommit create-repository --repository-name my-repo
   Explanation:
   aws codecommit create-repository → Creates a new AWS CodeCommit repository.
   --repository-name my-repo → Assigns the repository name as my-repo.
   --endpoint-url=http://localhost:4566 → Uses LocalStack.
   Error Output:
   An error occurred (InternalFailure) when calling the CreateRepository operation: API for service 'codecommit' not yet implemented or pro feature.

3. **Initializing a Git Repository**
   Command:
   git init
   Explanation:
   git init → Initializes a new Git repository in the current directory.
   Output:
   Initialized empty Git repository.

4. **Staging and Committing Files**
   Commands:
   git add .
   git commit -m "Initial commit"
   Explanation:
   git add . → Stages all files for commit.
   git commit -m "Initial commit" → Commits the staged files with a message.
   Output:
   [master (root-commit) 2dfb5b6] Initial commit

5. **Uploading a ZIP File to S3**
   Command:
   aws --endpoint-url=http://localhost:4566 s3 cp my-code.zip s3://my-ci-cd-artifacts/
   Explanation:
   aws s3 cp → Copies a file to S3.
   my-code.zip → The file being uploaded.
   s3://my-ci-cd-artifacts/ → Destination bucket in S3.

Error Output:
The user-provided path my-code.zip does not exist.

6. **Creating a ZIP Archive**
   Command:
   powershell Compress-Archive -Path * -DestinationPath my-code.zip
   Explanation:
   Compress-Archive -Path * -DestinationPath my-code.zip → Creates a ZIP archive of all files in the directory.

7. **Uploading the ZIP File Again**
   Command:
   aws --endpoint-url=http://localhost:4566 s3 cp my-code.zip s3://my-ci-cd-artifacts/
   Output:
   upload: .\my-code.zip to s3://my-ci-cd-artifacts/my-code.zip

8. **Listing the Uploaded Files in S3**
   Command:
   aws --endpoint-url=http://localhost:4566 s3 ls s3://my-ci-cd-artifacts/
   Output:
   2025-03-08 10:32:42 289415 my-code.zip

9. **Setting Up a Remote Git Repository**
   Commands:
   git remote add origin https://github.com/madhurimarawat/CloudComputing.git
   git branch -M main
   git push -u origin main
   Explanation:
   git remote add origin <repo-url> → Links the local repository to GitHub.
   git branch -M main → Renames the current branch to main.
   git push -u origin main → Pushes the code to GitHub.
   Error Output:
   Updates were rejected because the remote contains work that you do not have locally.

10. **Pulling the Latest Changes from GitHub**
    Command:
    git pull origin main --rebase
    Explanation:
    Fetches changes from the remote repository and applies them using rebase instead of a merge.
    Output:
    Successfully rebased and updated refs/heads/main.

11. **Staging All Changes**
    Command:
    git add .
    Explanation:
    Stages all modified and newly created files for the next commit.

12. **Checking for an Ongoing Rebase**
    Command:
    git rebase --continue
    Explanation:
    Used to continue an ongoing rebase operation if there are conflicts.
    Output:
    fatal: no rebase in progress

13. **Pushing Changes to GitHub**
    Command:
    git push -u origin main
    Explanation:
    Pushes local changes to the remote repository, setting main as the upstream branch.
    Output:
    Successfully pushed changes.

14. **Viewing the YAML Deployment Workflow**
    Link: View the deployment YAML file
    Purpose: This GitHub Actions workflow automates a manual deployment process.

Conclusion:

This CI/CD pipeline experiment with GitHub Actions, Docker, and LocalStack demonstrates how to automate deployments, streamline cloud workflows, and reduce AWS costs. It is an essential practice for teams looking to improve deployment efficiency and cloud-native development.

The output for all the executed commands is provided on the next page for reference.

Experiment 10 Output

Creating S3 Bucket and Storing data

```
>aws --endpoint-url=http://localhost:4566 s3 cp my-code.zip s3://my-ci-cd-artifacts/


>powershell Compress-Archive -Path * -DestinationPath my-code.zip

>aws --endpoint-url=http://localhost:4566 s3 cp my-code.zip s3://my-ci-cd-artifacts/


>aws --endpoint-url=http://localhost:4566 s3 ls s3://my-ci-cd-artifacts/
```

Fig 1: S3 Bucket

Initialising a git repo

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git init
Initialized empty Git repository in C:/Users/rawat/Documents/8 SEMESTER/Cloud Computi

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git add .

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git commi
[master (root-commit) 2dfb5b6] Initial commit
 3 files changed, 1153 insertions(+)
 create mode 100644 Command Prompt Input and Output Explanation.md
 create mode 100644 Command Prompt Input and Output Explanation.pdf
 create mode 100644 Command Prompt Input and Output.txt
```

Fig 2: Git Repository Intitialization

Pulling the Remote Repository

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git push -u origin main
To https://github.com/madhurimarawat/Cloud-Computing.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/madhurimarawat/Cloud-Computing.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git pull origin main --rebase
remote: Enumerating objects: 240, done.
remote: Counting objects: 100% (240/240), done.
remote: Compressing objects: 100% (212/212), done.
remote: Total 240 (delta 100), reused 43 (delta 21), pack-reused 0 (from 0)
Receiving objects: 100% (240/240), 9.22 MiB | 1.11 MiB/s, done.
Resolving deltas: 100% (100/100), done.
From https://github.com/madhurimarawat/Cloud-Computing
 * branch            main       -> FETCH_HEAD
 * [new branch]      main       -> origin/main
Successfully rebased and updated refs/heads/main.
```

Fig 3: Remote Repository

Updating in Remote Repository

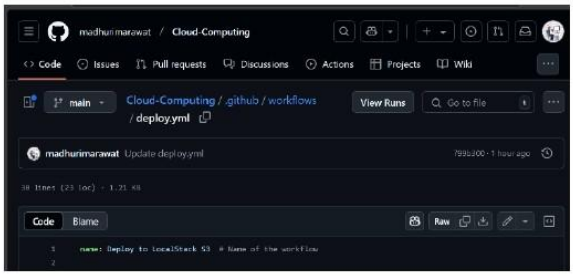
```
C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git add .

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git rebase --continue
fatal: no rebase in progress

C:\Users\rawat\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 10\Codes>git push -u origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 282.43 KiB | 31.38 MiB/s, done.
Total 5 (delta 1), reused 1 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/madhurimarawat/Cloud-Computing.git
   b201b02..eb4fef7  main -> main
branch 'main' set up to track 'origin/main'.
```

Fig 4: Updating in Repository

Remote Repository and Workflow




```yaml
name: Deploy to LocalStack S3  # Name of the workflow

on:
  workflow_dispatch:  # This makes the workflow run ONLY when manually triggered

jobs:
  deploy:
    runs-on: ubuntu-latest  # Uses the latest Ubuntu runner in GitHub Actions

    steps:
      - name: Checkout repository  # Pulls the repository code into the GitHub Actions runner
        uses: actions/checkout@v4

      # - name: Install AWS CLI  # Installs AWS CLI to allow S3 operations
      #   run: |
```

Fig 5: Remote Repository and Workflow File

Workflow Run

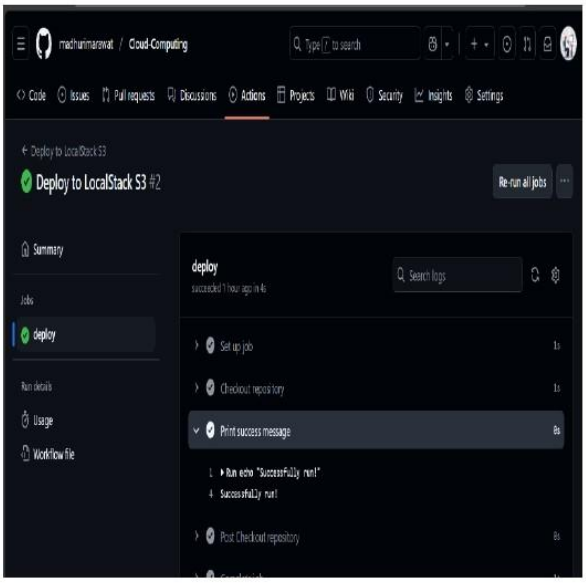


Fig 6: GitHub Actions Workflow

Resources

1. **GitHub Repository: Cloud Computing**
   This repository focuses on cloud computing and demonstrates how to set up virtual machines, S3, and other services using LocalStack. It provides a comprehensive guide to simulating AWS services locally for development and testing purposes. Additionally, it contains detailed documentation for every experiment conducted.

   Link: https://github.com/madhurimarawat/Cloud-Computing

2. **LocalStack Official Documentation**
   This resource provides in-depth guidance on setting up and using LocalStack to emulate AWS cloud services for development and testing.

   Link: https://docs.localstack.cloud/

3. **AWS Documentation**
   The official AWS documentation offers detailed explanations, best practices, and tutorials for working with various AWS services, including EC2, S3, Lambda, and more.

   Link: https://docs.aws.amazon.com/

4. **Cloud Computing Concepts - Coursera**
   A well-structured online course that covers fundamental cloud computing concepts, including virtualization, networking, and cloud service models like IaaS, PaaS, and SaaS.

   Link: https://www.coursera.org/learn/cloud-computing

5. **Dev.to Series: Cloud Experiments**
   This series on Dev.to by documents various cloud computing experiments, including the use of LocalStack, AWS CLI, and S3 storage. It serves as a practical reference for learners and developers exploring local cloud service simulations.
   Link: https://dev.to/madhurima_rawat/series/31049