

# **CHHATTISGARH SWAMI VIVEKANAND TECHNICAL UNIVERSITY**

**(UNIVERSITY TEACHING DEPARTMENT)**

**8<sup>th</sup> 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

```

C:\Users\ravat>localstack start

Microsoft Windows [Version 10.0.22631.4751]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ravat>localstack start

C:\Users\ravat>python "C:\Users\ravat\AppData\Local\Programs\Python\Python310\Scripts\localstack" start

LocalStack CLI 3.6.0
Profile: default

[12:47:07] starting LocalStack in Docker mode localstack.py:503
----- LocalStack Runtime Log (press CTRL-C to quit) -----

LocalStack version: 4.0.4.dev122
LocalStack build date: 2025-01-23
LocalStack build git hash: 81d9f9b79

Ready.
2025-01-23T07:17:18.866 INFO --- [et.reactor-0] localstack.request.http : GET / => 200
2025-01-23T07:17:19.226 INFO --- [et.reactor-0] localstack.request.aws : AWS s3.ListObjects => 404 (NoSuchBucket)
    
```

Fig 1: Starting Localstack in the Console

### Running Docker

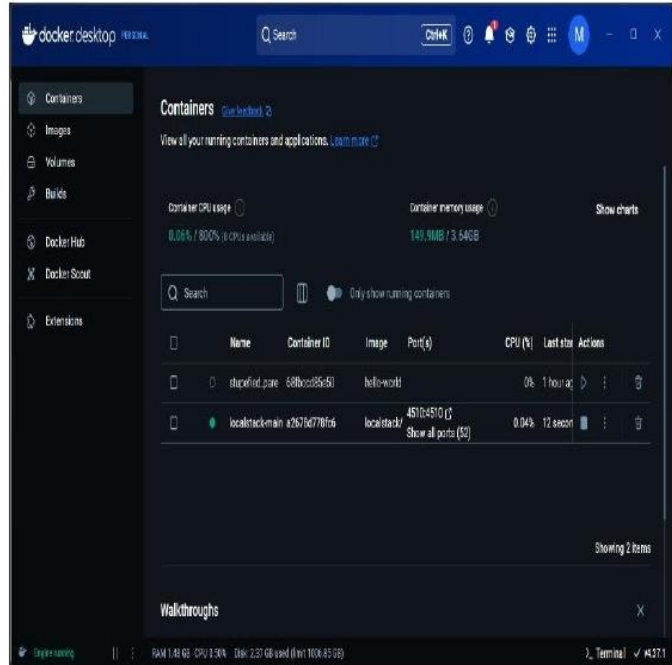


Fig 2: Docker for Desktop in windows

### Creating a New Instance

```

C:\Users\ravat>docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
localstack/localstack latest             b686f3948f42       5 hours ago        1.18GB
hello-world         latest             74cc54e27dc4       32 hours ago        10.1kB

C:\Users\ravat>aws ec2 create-key-pair --key-name local-key --endpoint-url=$AWS_ENDPOINT_URL
Bad value for --endpoint-url "$AWS_ENDPOINT_URL": scheme is missing. Must be of the form http://

C:\Users\ravat>setx AWS_ENDPOINT_URL "http://localhost:4566"
SUCCESS: Specified value was saved.

C:\Users\ravat>aws ec2 create-key-pair --key-name local-key --endpoint-url=$AWS_ENDPOINT_URL
Bad value for --endpoint-url "$AWS_ENDPOINT_URL": scheme is missing. Must be of the form http://

C:\Users\ravat>aws ec2 create-key-pair --key-name local-key --endpoint-url=$AWS_ENDPOINT_URL
    
```

Fig 3: Docker Images and New Instances

### Starting Virtual Machine

RunInstances	
OwnerId	000000000000
ReservationId	r-8daa9365f4eb5ae90
Groups	
GroupId	sg-245f6a01
GroupName	default
Instances	
AmiLaunchIndex	0
Architecture	x86_64
ClientToken	ABCDE00000000000003
EbsOptimized	False
Hypervisor	xen
ImageId	ami-a2678d778fc6
InstanceId	i-239eb58179052805c
InstanceType	t2.micro
KernelId	None
KeyName	local-key
LaunchTime	2025-01-23T07:28:56Z
PrivateDnsName	ip-10-109-41-181.ec2.internal
PrivateIpAddress	10.109.41.181
PublicDnsName	ec2-54-214-27-38.compute-1.amazonaws.com
PublicIpAddress	54.214.27.38
RootDeviceName	/dev/sda1
RootDeviceType	ebs
SourceDestCheck	True
StateTransitionReason	
SubnetId	subnet-6cd350b4143086ff0
VirtualizationType	paravirtual

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	<a href="http://localhost:5000/">http://localhost:5000/</a>
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 rbx2kdpysl
```

### 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.

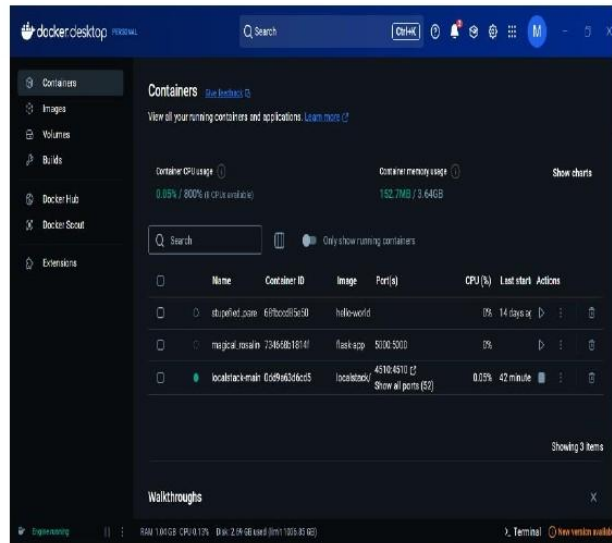
### Conclusion:

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.

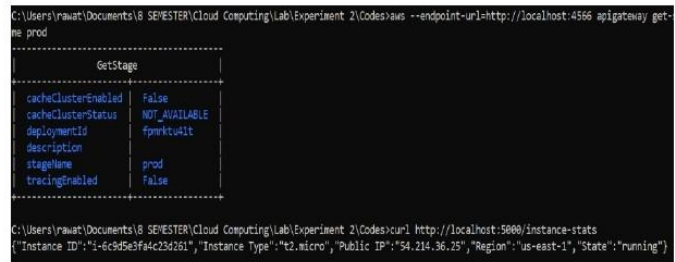
The output for all executed commands is provided on the next page for reference.

## Running Docker



*Fig 2: Docker for Desktop in windows*

### Configuration and Resources



localhost:5000/instance-status

localhost:5000/instance-status

Pretty-print

```
{\"Instance ID\": \"i-6c9d5e3f4c73d261\", \"Instance Type\": \"t2.micro\", \"Public IP\": \"54.234.36.25\", \"Region\": \"us-east-1\", \"State\": \"running\"}
```

localhost:5000

localhost:5000

Hello, Cloud Deployment!

**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

```
C:\Users\ravati>python "C:\Users\ravati\AppData\Local\Programs\Python\Python310\Scripts\localstack" start

LocalStack CLI 3.6.0
Profile: default

[12:13:40] starting LocalStack in Docker mode
LocalStack Runtime Log (press CTRL-C to quit)

LocalStack version: 4.0.4.dev122
LocalStack build date: 2025-01-19
LocalStack build git hash: 81d9f0679

Ready.
2025-02-02 09:18:54-13:375 INFO --- [et.reactor-0] localstack.utils.bootstrap : Execution of "load_service_plugins" took
2025-02-02 09:18:54-13:375 INFO --- [et.reactor-0] localstack.utils.bootstrap : Execution of "require" took 790.50ms
2025-02-02 09:18:54-13:694 INFO --- [et.reactor-0] localstack.request.aws : AWS ec2.RunInstances -> 400 (InvalidGroup
2025-02-02 09:18:54-57:862 INFO --- [et.reactor-0] localstack.request.aws : AWS ec2.CreateSecurityGroup -> 200
2025-02-02 09:18:55-13:207 INFO --- [et.reactor-0] localstack.request.aws : AWS ec2.CreateSecurityGroup -> 200
2025-02-02 09:18:56-47:344 INFO --- [et.reactor-0] localstack.request.aws : AWS ec2.RunInstances -> 200
2025-02-02 09:18:56-33:435 INFO --- [et.reactor-0] localstack.request.aws : AWS ec2.RunInstances -> 200
2025-02-02 09:18:56-07:915 INFO --- [et.reactor-0] localstack.request.http : GET / -> 200
2025-02-02 09:18:57-01:08:274 INFO --- [et.reactor-0] localstack.request.aws : AWS s3.ListObjects -> 404 (NoSuchBucket)
```

Fig 1: Starting Localstack in the Console

### Creating and Storing Data in S3 Bucket

```
C:\Users\ravati>aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 ls
C:\Users\ravati>aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 mb
make_bucket: my-test-bucket
C:\Users\ravati>aws --endpoint-url=http://localhost:4566 --region us-east-1 s3 ls
2025-02-18 10:55:33 my-test-bucket
C:\Users\ravati>cd "C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items"
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>aws --
ive
upload: .\Sample_Housing_CSV_File.csv to s3://my-test-bucket/Sample_Housing_CSV_F
upload: .\storage_service_offered_by_amazon.png to s3://my-test-bucket/storage_se
upload: .\Data_Transfer_Amazon.jpg to s3://my-test-bucket/Data_Transfer_Amazon.jp
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>aws --
2025-02-18 11:23:34 89553 Data_Transfer_Amazon.jpg
2025-02-18 11:23:34 29981 Sample_Housing_CSV_File.csv
2025-02-18 11:23:34 47575 storage_service_offered_by_amazon.png
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>aws --
fatal error: An error occurred (404) when calling the HeadObject operation: Key
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>aws --
ing_CSV_File.csv
download: s3://my-test-bucket/Sample_Housing_CSV_File.csv to .\Sample_Housing_CSV
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>type S
price,area,bedrooms,bathrooms,stories,mainroad,guestroom,basement,hotwaterheating
13300000,7420,4,2,3,yes,no,no,no,yes,2,yes,furnished
12250000,8960,4,4,4,yes,no,no,no,yes,3,no,furnished
12250000,9960,3,2,2,yes,no,yes,no,no,2,yes,semi-furnished
12215000,7500,4,2,2,yes,no,yes,no,yes,3,yes,furnished
11410000,7420,4,1,2,yes,yes,yes,no,yes,2,no,furnished
10850000,7500,3,3,1,yes,no,yes,no,yes,2,yes,semi-furnished
10150000,8580,4,3,4,yes,no,no,no,yes,2,yes,semi-furnished
10150000,16200,5,3,2,yes,no,no,no,no,0,no,unfurnished
9870000,8100,4,1,2,yes,yes,yes,no,yes,2,yes,furnished
```

Fig 3: S3 Bucket and contents

### Running Docker

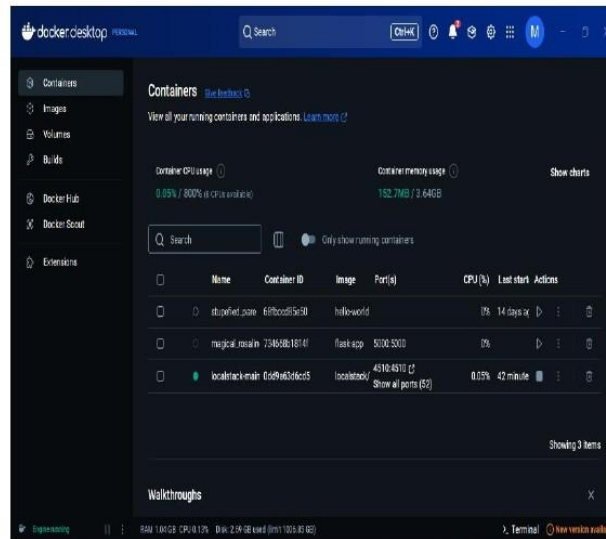


Fig 2: Docker for Desktop in windows

### Contents in Bucket

```
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>start Data_Transfer_Amazon.jpg
price,area,bedrooms,bathrooms,stories,mainroad,guestroom,basement,hotwaterheating
13300000,7420,4,2,3,yes,no,no,no,yes,2,yes,furnished
12250000,8960,4,4,4,yes,no,no,no,yes,3,no,furnished
12250000,9960,3,2,2,yes,no,yes,no,no,2,yes,semi-furnished
12215000,7500,4,2,2,yes,no,yes,no,yes,3,yes,furnished
11410000,7420,4,1,2,yes,yes,yes,no,yes,2,no,furnished
10850000,7500,3,3,1,yes,no,yes,no,yes,2,yes,semi-furnished
10150000,8580,4,3,4,yes,no,no,no,yes,2,yes,semi-furnished
10150000,16200,5,3,2,yes,no,no,no,no,0,no,unfurnished
9870000,8100,4,1,2,yes,yes,yes,no,yes,2,yes,furnished
```

Fig 4: Printing CSV File

```
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>start Data_Transfer_Amazon.jpg
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>start storage_service_offered_by_amazon.png
C:\Users\ravati\Documents\8 SEMESTER\Cloud Computing\Lab\Experiment 3\Items>aws --endpoint-url=http://localhost:4566 --r
ing_CSV_File.csv "C:\Users\ravati\Downloads\Sample_Housing_CSV_File.csv"
download: s3://my-test-bucket/Sample_Housing_CSV_File.csv to ..\..\..\..\Downloads\Sample_Housing_CSV_File.csv
```



Fig 5: Images Output

## 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](https://dev.to/madhurima_rawat/series/31049)