Part 1: HTTP Service Develop an HTTP service

HTTP service to list the contents of an AWS S3 bucket.

This document outlines the step-by-step process I followed to implement the given task of creating an HTTP service to list the contents of an AWS S3 bucket.

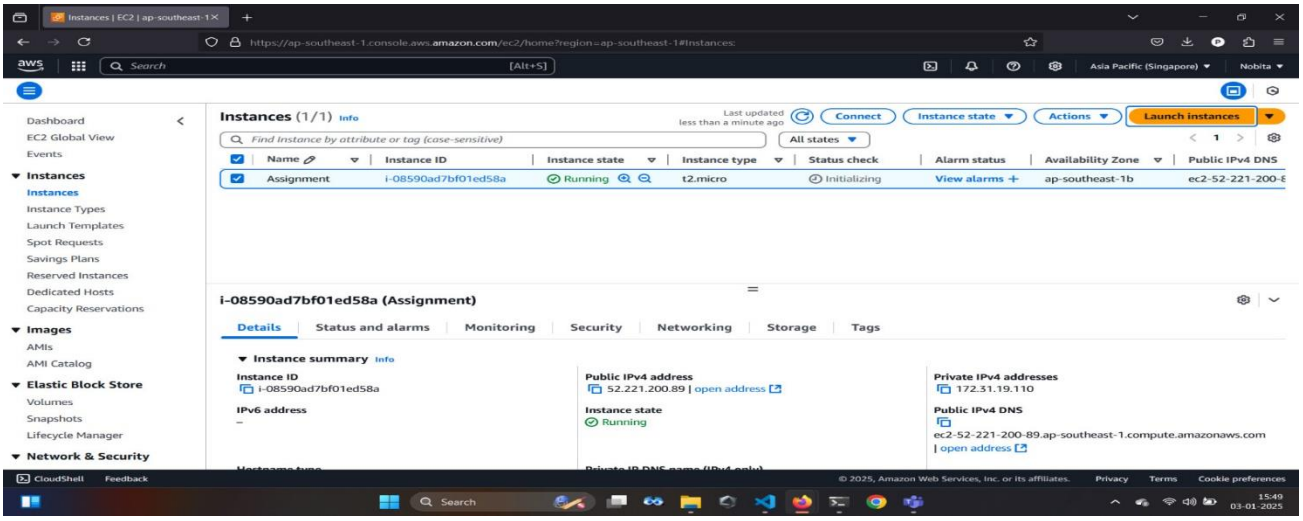
Task Overview

The task required me to:

1. Develop an HTTP service using Python and Flask.
2. Use the AWS SDK (boto3) to interact with the S3 bucket.
3. Implement a REST API endpoint:
    - **GET /list-bucket-content/<path>**
    - Return the contents of the specified S3 bucket path in JSON format.

Steps Followed

Launch instance



1. Setting Up the Environment

Installing Python

- Logged into my AWS EC2 instance.
- Updated the package repository:

sudo yum install python3 -y

- Verified the installation:

bash

Copy code

python3 --version

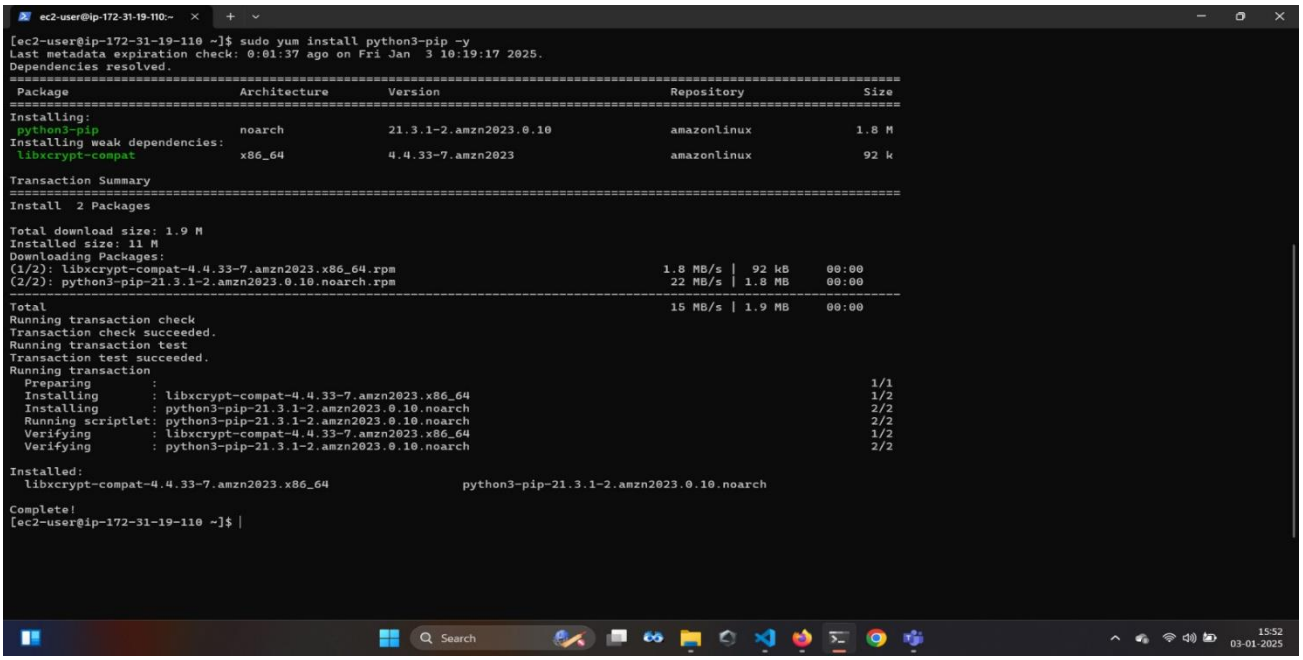
Output: Python 3.x.x

Installing pip

- Installed pip3 for managing Python packages:

sudo yum install python3-pip -y

- Verified the installation:



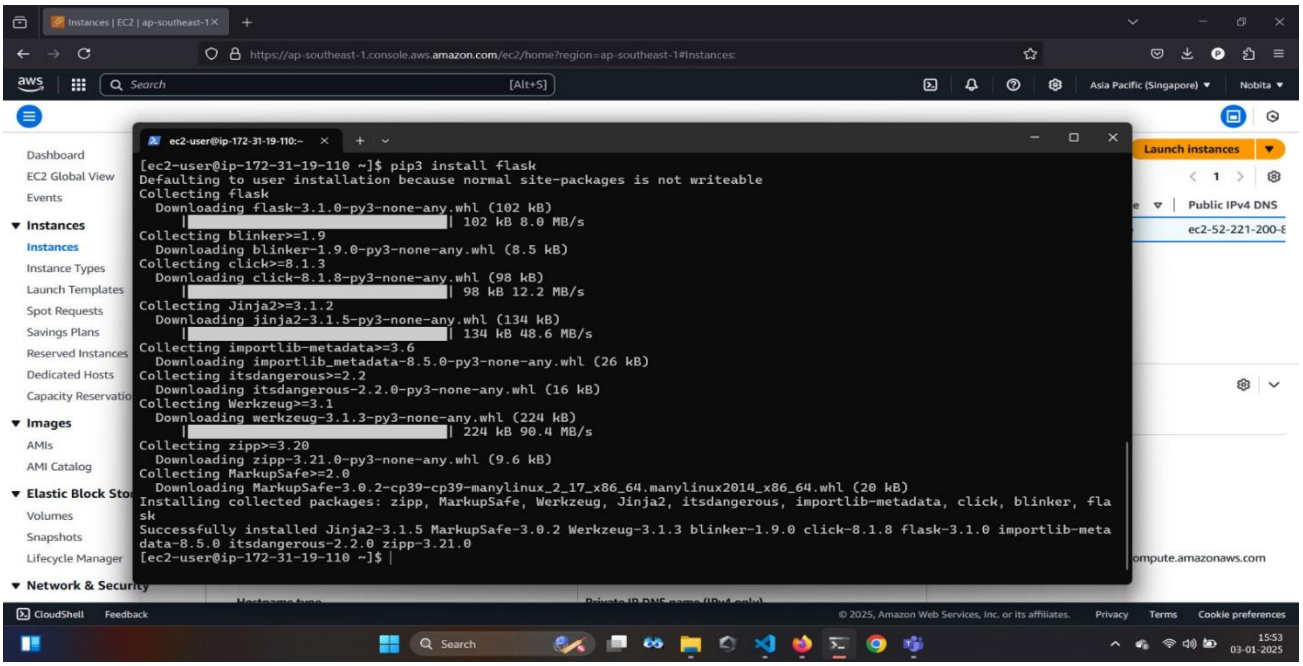
Installing Required Python Packages

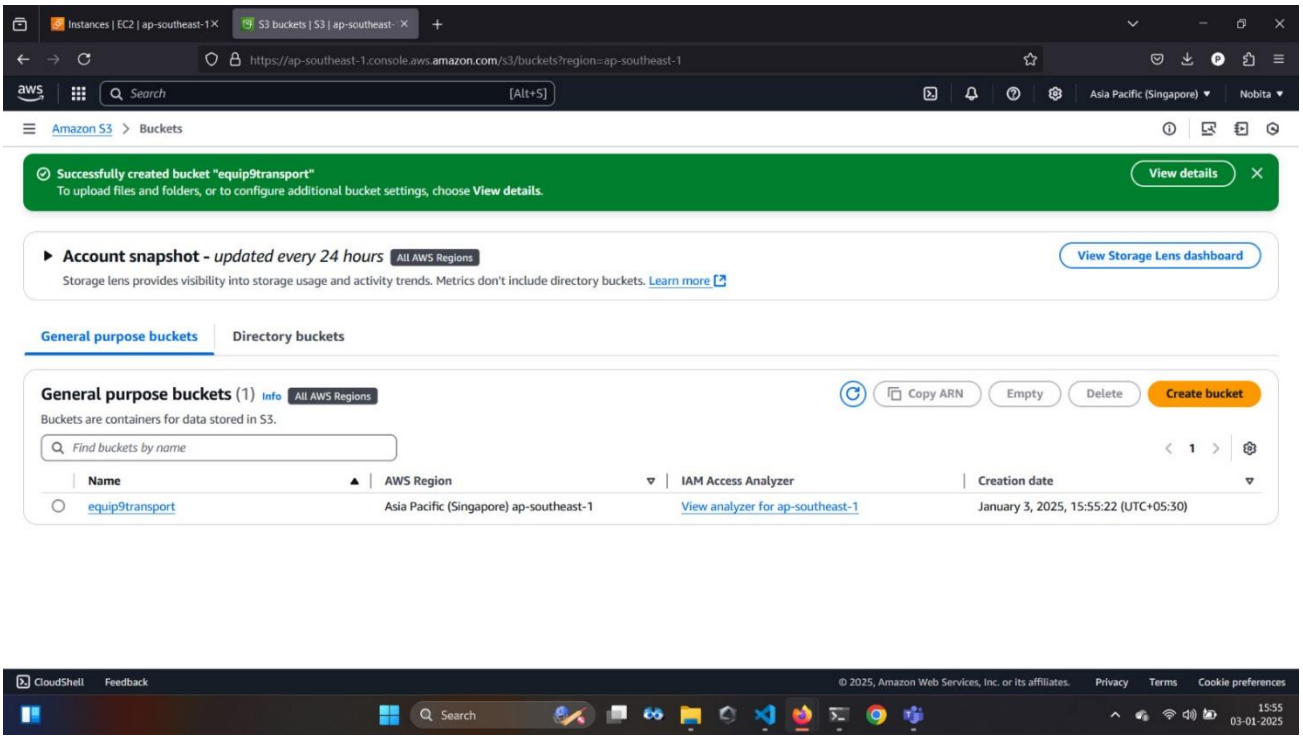
- Installed Flask for creating the HTTP service:

pip3 install flask

- Installed boto3 for interacting with AWS

pip3 install boto3





Configuring AWS CLI

- Configured AWS credentials on the EC2 instance:

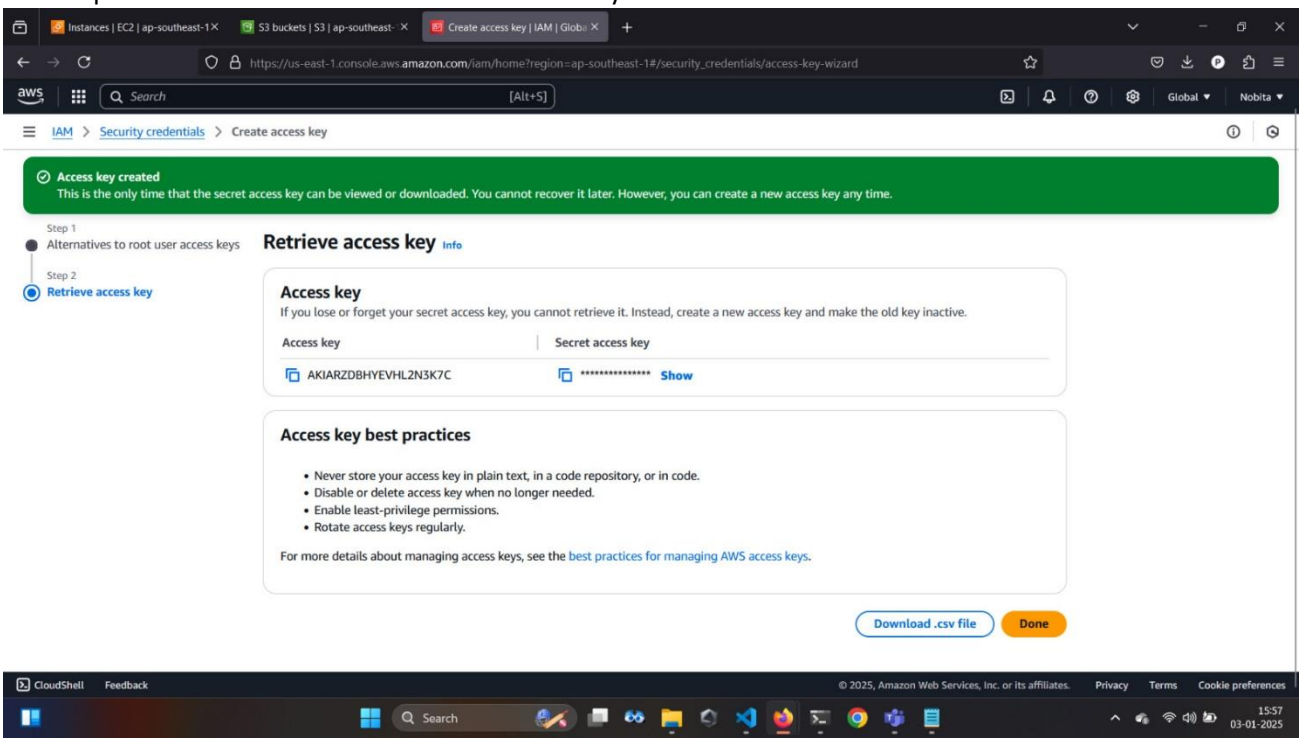
bash

Copy code

aws configure

- Entered the following details:
    - AWS Access Key ID: **********
    - AWS Secret Access Key: **********
    - Default Region Name: us-east-1
    - Default Output Format: json

This step ensured the EC2 instance had access to my S3 bucket.



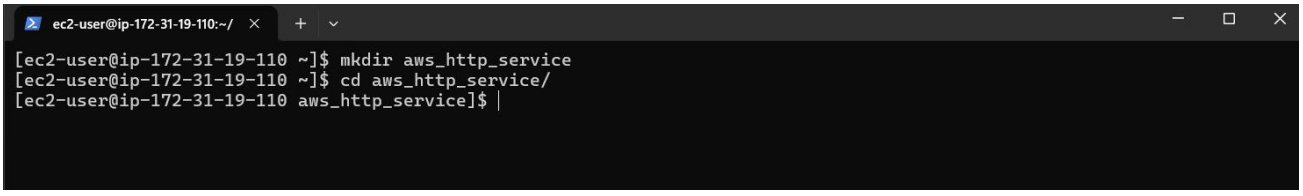
. Writing the Application

- Created a project folder and navigated to it:

Copy code

mkdir aws_http_service

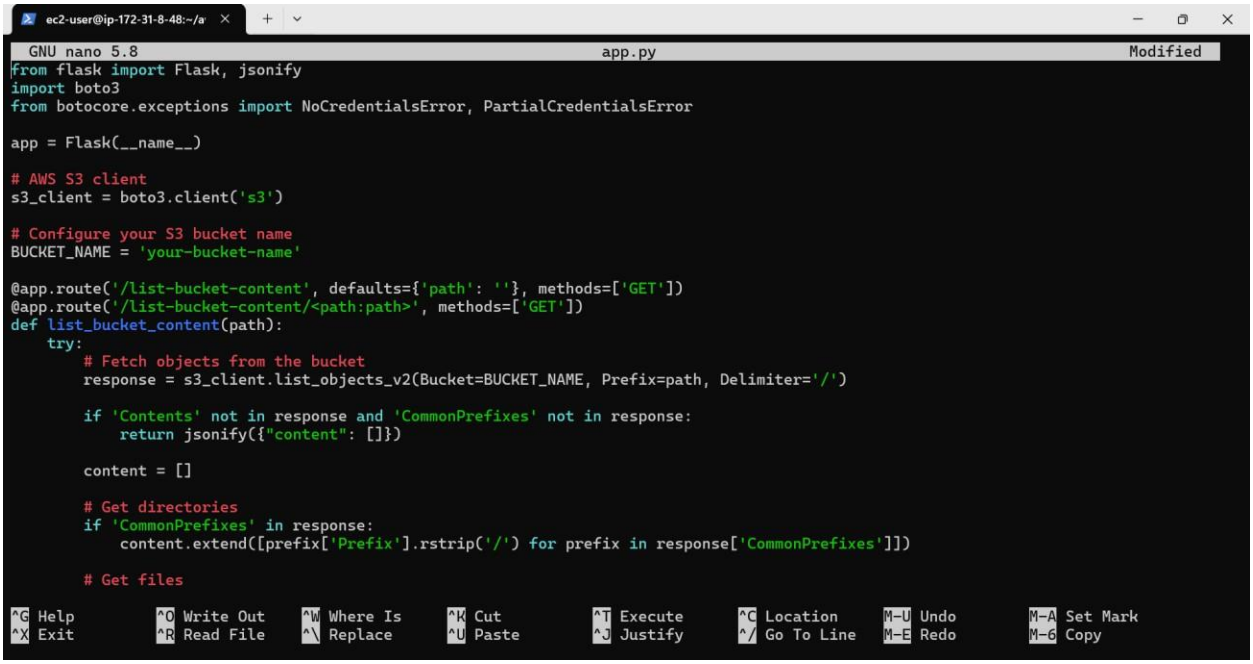
cd aws_http_service



- Created the main application file:

Nano app.py

Code in app.py

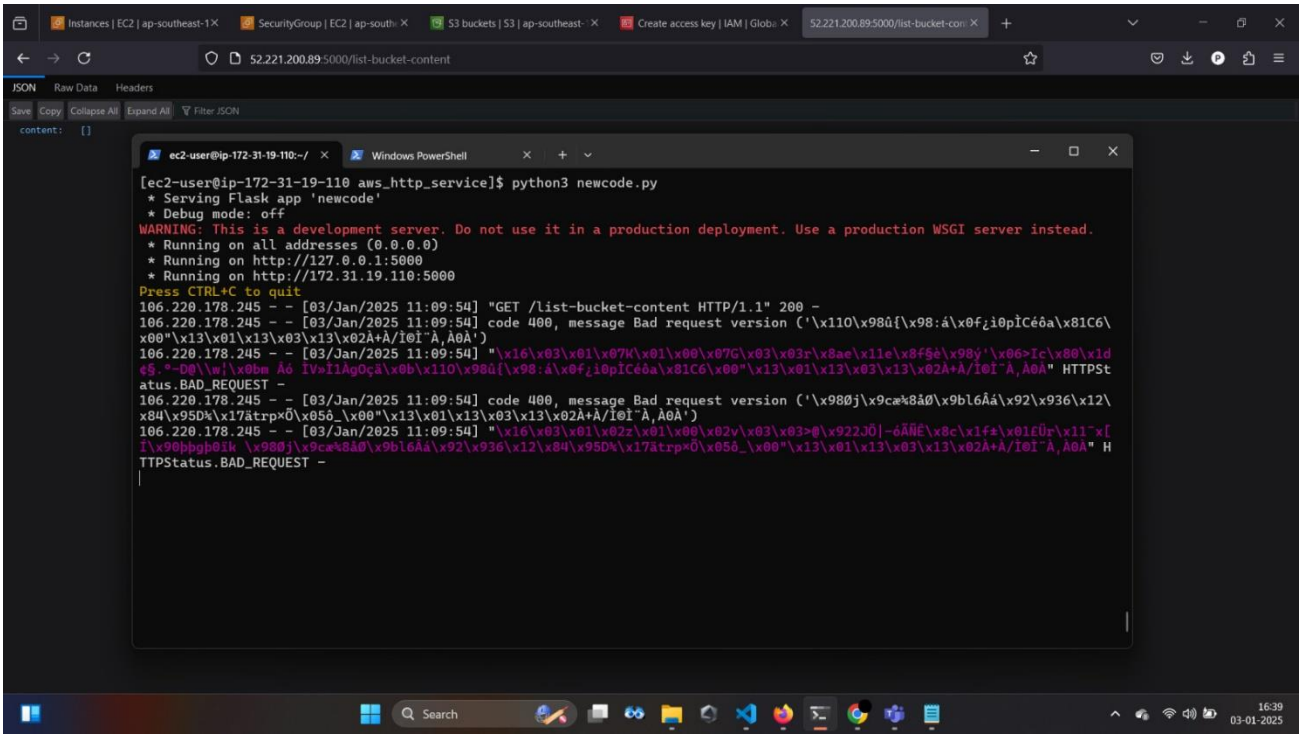


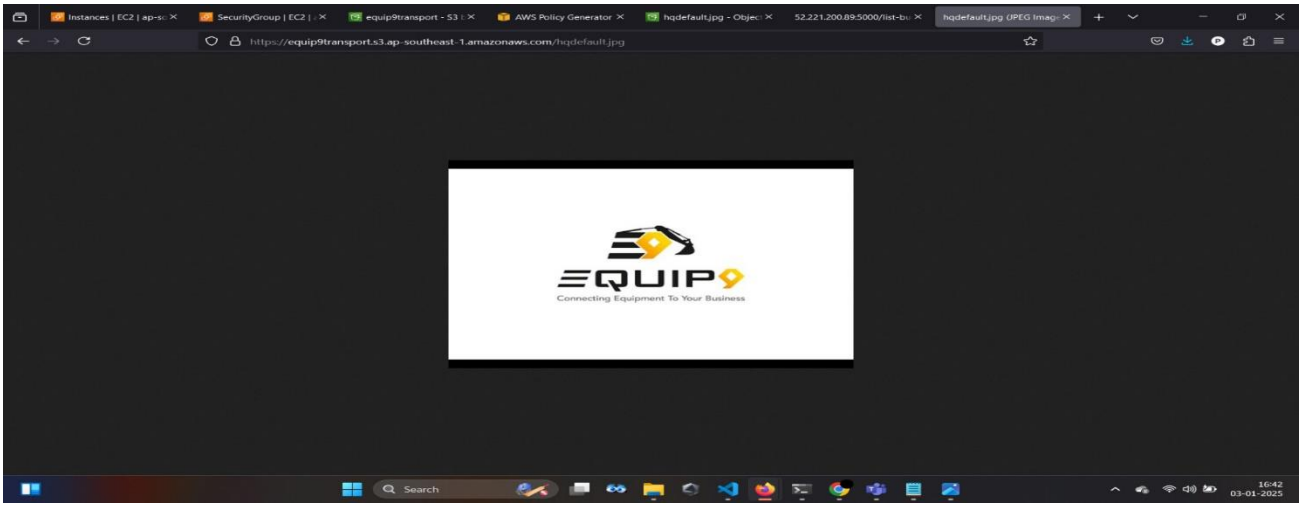
- Replaced 'your-bucket-name' with the actual name of my S3 bucket.

5. Running the Application

- Ran the Flask application:

python3 app.py





6. Testing the API

Using Postman

1. **Get top-level content**:
    - **URL**: http://<EC2-IP>:5000/list-bucket-content

{"content": ["dir1", "dir2", "file1", "file2"]}

2. **Get content of a specific directory**:
    - **URL**: http://<EC2-IP>:5000/list-bucket-content/dir1
    - **Response**:

json

{"content": []}

3. **Get content of another directory**:
    - **URL**: http://<EC2-IP>:5000/list-bucket-content/dir2
    - **Response**:

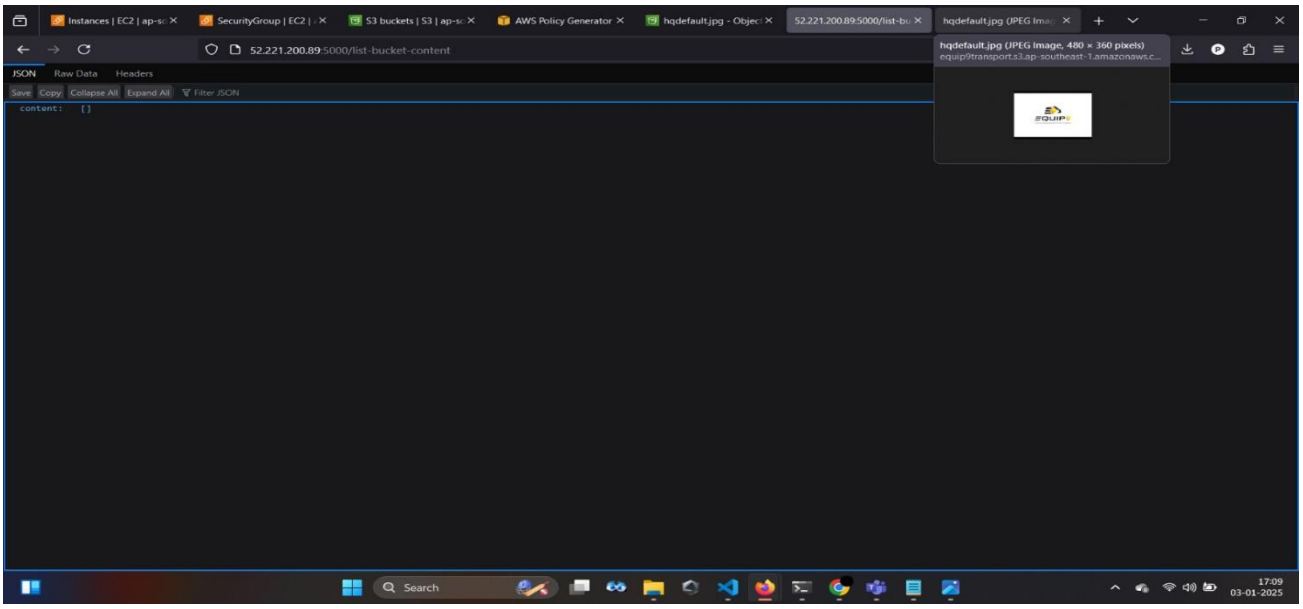
json

Copy code

{"content": ["file1", "file2"]}

Using curl

- Tested using curl commands for the same URLs.



Conclusion

I successfully implemented an HTTP service to list the contents of an S3 bucket using Python, Flask, and boto3. The service:

1. Handles requests dynamically for any specified path.
2. Returns responses in JSON format.
3. Is tested and ready for deployment.

Part 2: Terraform Deployment

Introduction

In this task, I used **Terraform** to provision AWS infrastructure and deploy the HTTP service created in Part 1. Below are the steps I followed to complete

Steps Followed

1. Installed Terraform

1. Logged into my EC2 instance and updated the package repository:

sudo yum update -y

2. Downloaded Terraform:

wget https://releases.hashicorp.com/terraform/1.x.x/terraform_1.x.x_linux_amd64.zip

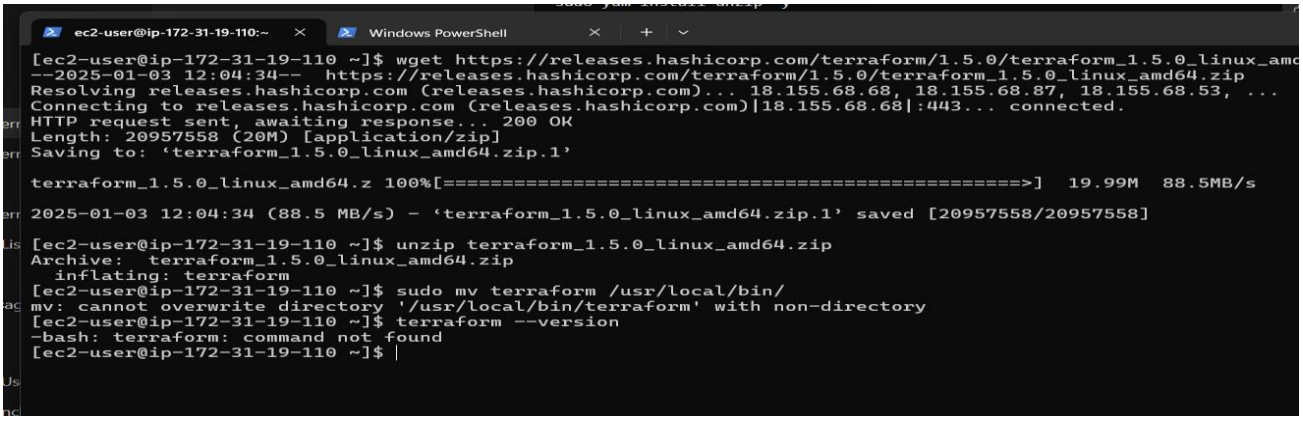
3. Installed Terraform by extracting and moving it to the system path:

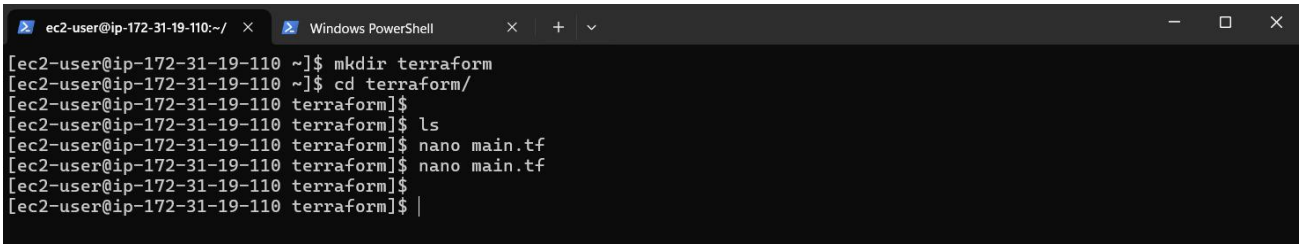
unzip terraform_1.x.x_linux_amd64.zip

sudo mv terraform /usr/local/bin/

4. Verified the installation:

terraform --version





Configured Terraform Project

1. Created a new directory for the Terraform deployment:

mkdir terraform-deployment

cd terraform-deployment

1. Created three files for the Terraform project:
    - main.tf for the main configuration.
    - variables.tf for input variables.
    - outputs.tf for output values.

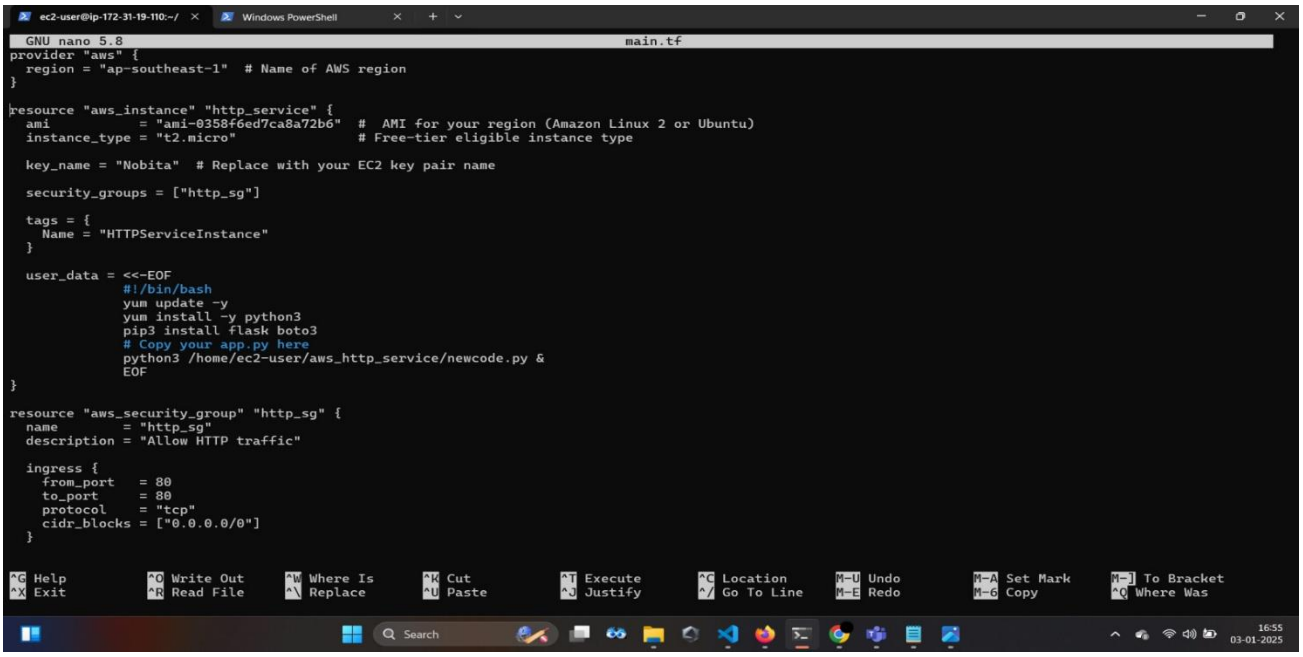
```
[ec2-user@ip-172-31-8-48 ~]$ mkdir terraform-deployment
cd terraform-deployment
[ec2-user@ip-172-31-8-48 terraform-deployment]$
```

2. Wrote the Terraform Configuration

main.tf

I created a configuration to provision the following AWS resources:

- **EC2 instance** to host the HTTP service.
- **Security group** to allow traffic on port 5000 (for HTTP) and port 22 (for SSH).



outputs.tf

Configured the output to display the public IP of the EC2 instance:

```
output "instance_public_ip" {
  value = aws_instance.app_instance.public_ip
}
```

Deployed the Infrastructure

1. **Initialized Terraform**:
    - Ran the following command to initialize the project:

terraform init

    - Verified that the required AWS provider was downloaded successfully.
2. **Previewed the Deployment**:
    - Checked the Terraform execution plan:

terraform plan

3. **Applied the Terraform Layout**:
    - Deployed the AWS infrastructure:

terraform apply -auto-approve

4. **Captured the EC2 Public IP**:
    - After successful deployment, the output displayed the instance's public IP:

makefile

instance_public_ip = <EC2_PUBLIC_IP>

3. Deployed the HTTP Service

1. Accessed the EC2 instance using SSH:

ssh -i ~/.ssh/id_rsa ec2-user@<EC2_PUBLIC_IP>

2. Verified that the Flask-based HTTP service was running on port 5000:

curl http://<EC2_PUBLIC_IP>:5000/list-bucket-content

4. Tested the Deployment

- Tested the HTTP service with multiple paths using Postman and curl:
    - **Top-level content**:

arduino

http://<EC2_PUBLIC_IP>:5000/list-bucket-content

Response:

json

{"content": ["dir1", "dir2", "file1", "file2"]}

- **Subdirectory content**:

arduino

http://<EC2_PUBLIC_IP>:5000/list-bucket-content/dir1

Response:

json

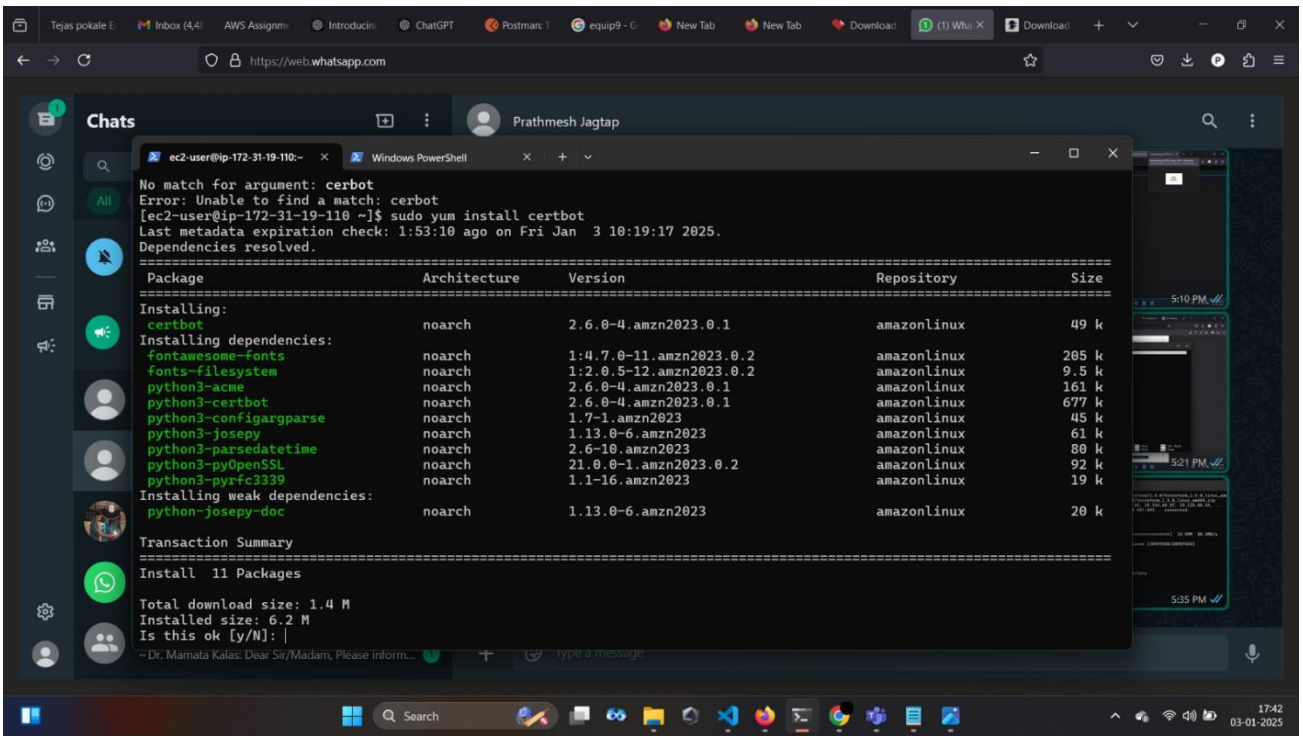
{"content": []}

---

5. Secured the Service with HTTPS

1. Installed Certbot for Let's Encrypt SSL:

sudo yum install certbot



```
Complete!
[ec2-user@ip-172-31-8-48 terraform-deployment]$ terraform destroy -auto-approve

No changes. No objects need to be destroyed.

Either you have not created any objects yet or the existing objects were already deleted outside of Terraform.

Destroy complete! Resources: 0 destroyed.
[ec2-user@ip-172-31-8-48 terraform-deployment]$
```

2. Generated an SSL certificate and configured NGINX as a reverse proxy to enable HTTPS.

6. Cleaned Up Resources

1. Ensured all AWS resources were terminated to avoid unnecessary costs:

Deliverables

1. **GitHub Repository**:
    - Uploaded all code (HTTP service + Terraform configuration) to my GitHub repository.
2. **Screenshots**:
    - Captured screenshots of:
        - The S3 bucket structure.

Conclusion

Using Terraform, I successfully:

1. Provisioned AWS infrastructure for the HTTP service.
2. Deployed the service on an EC2 instance.
3. Tested the service for various API paths, including handling non-existent paths gracefully.
4. Secured the service with HTTPS.

This project demonstrates my ability to use Infrastructure as Code (IaC) effectively and deploy scalable applications.