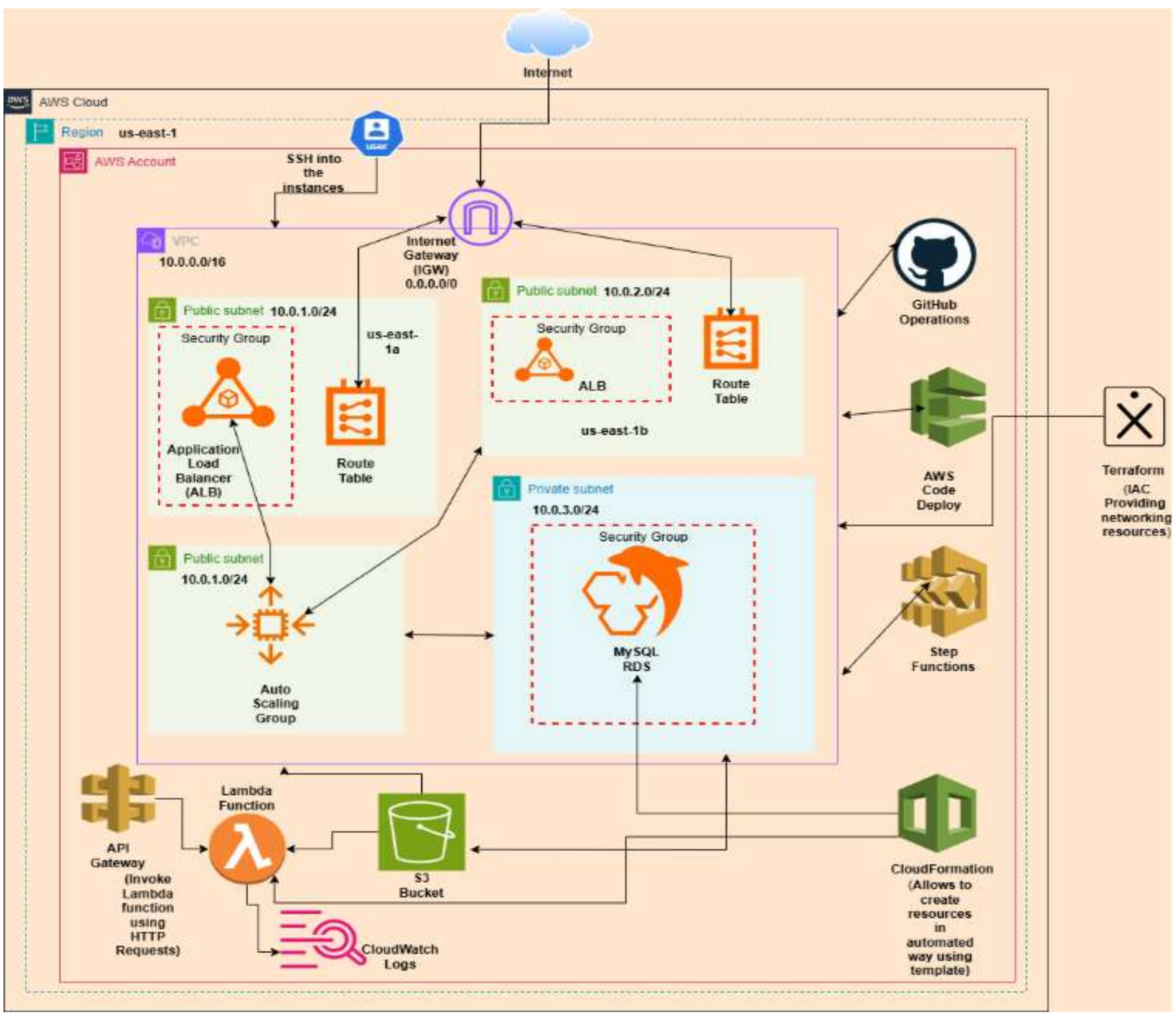
IS-698 Project Report – UB01976

Project: Deploying a Scalable AWS Architecture with Infrastructure as Code

Objective

The primary objective of the project is to design, implement, and test a scalable, cloud-based architecture on AWS. The project requires multiple AWS services including **EC2, VPC, RDS, S3, Lambda, and CloudFormation** alongside **Terraform** for Infrastructure as Code (IaC). Need to interact with AWS through the **Management Console, CLI, and Python Boto3**, and version-control the work using **GitHub**.

1. Designing Project Architecture: -



Architecture Explanation – The project architecture consists of several components to make a scalable solution. Firstly, **Terraform** and **CloudFormation** are used to provide us with the resources required for the project. Both these tools are used to provide us with the resources required. Terraform provides us with the networking resources required such as **VPC, Subnets, Route Tables, Route table Associations with the Subnets, and Security groups.**

CloudFormation is used to create application-level components such as **EC2 instances, RDS in the private subnet, and Lambda functions**. The architecture basically consists of a **web tier** (i.e. ALB (Application Load balancers) connected with Autoscaling groups (ASG)).

Here are the Individual Components: - **VPC** – The virtual private cloud consists of all the resources in the network. It basically provides an isolated network where all the AWS resources can be deployed.

The VPC is attached to the **Internet Gateway (IGW)** which provides internet connectivity to the VPC. Here in our architecture, it allows inbound and outbound traffic for the ALBs and ASGs.

Inside the VPC, we have subnets, public, and private, where the resources are provisioned. **Public subnets** here are used to host all the internet facing resources, which require traffic from outside. These consist of **Route tables** as well which are used to route the traffic going out from the ALB present in the subnet and routing the traffic coming inside into the subnet.

Private Subnet helps to host the resources which should not be exposed publicly and should only be accessed by resources inside VPC with permissions. The RDS (Database) must be in the private subnet, as it must be securely accessed not publicly exposed to traffic.

The **ALB (Application Load Balancers)** are placed in different availability zones (AZ) for high availability to manage the incoming traffic via the route tables. This way only the required traffic is sent into the EC2 instances behind these ALB's and makes sure resources stay healthy. Provides the required DNS endpoint for accessing the web application.

Auto Scaling Group (ASG): The ASG manages the dynamic allocation of the EC2 instances. So, this ASG provides the resources automatically whenever the load is increased on the existing resources and automatically scales down as well whenever those resources are no longer required. It is connected with the ALB to provide high availability and make sure there are no faults when creating resources.

RDS (Relational Database): The MYSQL database is hosted in the private subnet, as it must be accessed only by the EC2 instances of the ASG and must be restricted from public accessibility. It is used to store data, user details, and application related data.

S3 Bucket: - It is a bucket which is used for storing data related to application and user uploads. This S3 is attached to the lambda function, that is whenever a new file is uploaded into the S3 bucket, it triggers the lambda function which logs the event in the CloudWatch logs.

Lambda function: - It is a serverless computing service which allows you to run the code without provisioning the servers. Here the lambda function is invoked as a response to the adding file into the S3 bucket; this lambda function runs on the event driven architecture.

API Gateway: - This API Gateway provides us with the HTTP endpoints through which the external clients can access the backend or front-end services. In our architecture, it is used to call the lambda function from the backend.

CloudWatch Logs: - This is a monitoring system which is used to check the performance of the system, responses to the events taking place, and analyze the metrics. In our application, CloudWatch is used to record the logs generated by the lambda function.

GitHub Operations: - The source code generated in the application is pushed to the GitHub using the GitHub operations. It is used to automate the deploying, testing, and building of the CI/CD flow. It helps in the version control of EC2 instances of user data, lambda code, and python boto3 scripts.

AWS Step Function: - These step functions are used to automate the whole workflow in the application. They help in handling all the workflows such as EC2 Autoscaling with ASG, S3 File Upload process, and even Lambda invocation.

AWS Code Deploy: - Handles all the deployment activities for the application. It helps to deploy the code to EC2 instances in the ASG, provide updates to the Lambda functions, and helps in rollback support as well in case the application fails.

Refer the following GitHub link for code -

https://github.com/UB01976/IS-698-Project-UB01976

2. Implementation

A. Infrastructure Deployment

1. Use Terraform to provision networking components (VPC, subnets, security groups).

The implementation of the application begins with the creation of networking resources using the Terraform. We create the **VPC, Subnets, Route tables, Associations, IGW, and Security groups** using terraform. First, create a directory named as **terraform-project-networking-components** and have 3 separate files under this folder as follows;

terraform-project-networking-components/

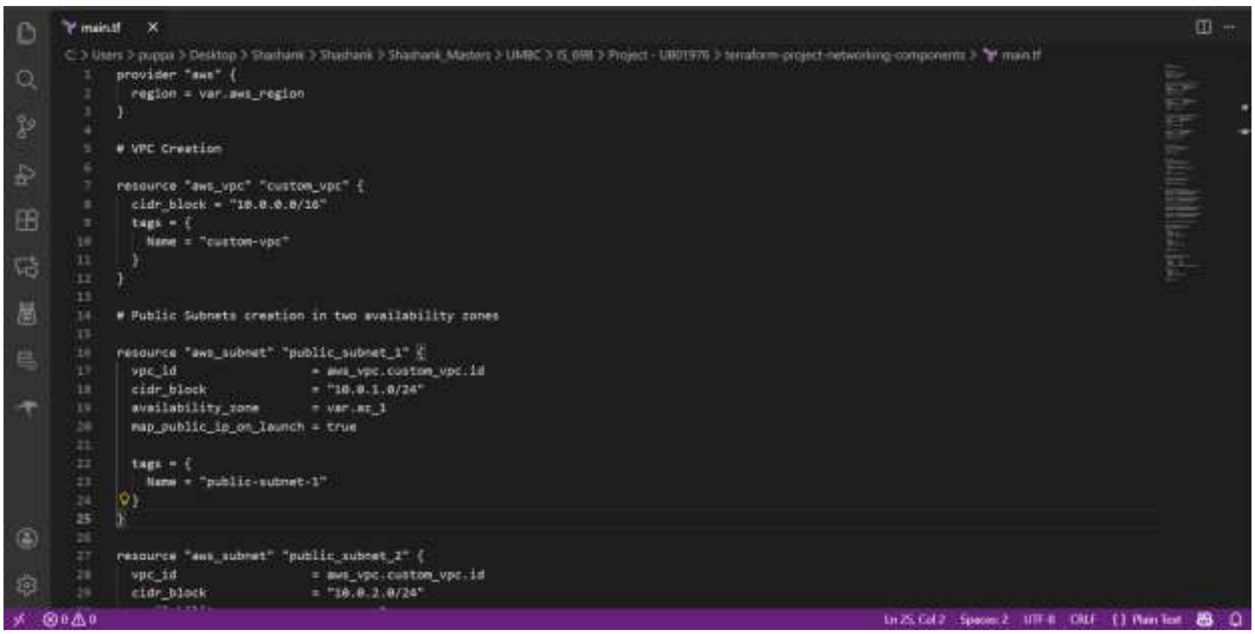
├── main.tf

├── variables.tf

├── outputs.tf

Place the following code inside the respective code blocks;

main.tf -



```
provider "aws" {
  region = var.aws_region
}

# VPC Creation

resource "aws_vpc" "custom_vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "custom-vpc"
  }
}

# Public Subnets creation in two availability zones

resource "aws_subnet" "public_subnet_1" {
  vpc_id                  = aws_vpc.custom_vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = var.az_1
  map_public_ip_on_launch = true

  tags = {
    Name = "public-subnet-1"
  }
}

resource "aws_subnet" "public_subnet_2" {
  vpc_id                  = aws_vpc.custom_vpc.id
  cidr_block              = "10.0.2.0/24"
```

Next the variable.tf and outputs.tf files, place the following codes;

variables.tf -

variables.tf

C: > Users > puppa > Desktop > Shashank > Shashank > Shashank_Masters > UMBC > IS_698 > Project - UB01976 > terraform-project-networking-components >

```
variable "aws_region" {
  default = "us-east-1"
}

variable "az_1" {
  default = "us-east-1a"
}

variable "az_2" {
  default = "us-east-1b"
}
```

outputs.tf -

outputs.tf

C: > Users > puppa > Desktop > Shashank > Shashank > Shashank_Masters > UMBC > IS_698 > Project - UB01976 > terraform-project-networking-components > outputs.tf

```
output "vpc_id" {
  value = aws_vpc.custom_vpc.id
}

output "public_subnet_ids" {
  value = [
    aws_subnet.public_subnet_1.id,
    aws_subnet.public_subnet_2.id
  ]
}

output "private_subnet_ids" {
  value = [
    aws_subnet.private_subnet_1.id,
    aws_subnet.private_subnet_2.id
  ]
}

output "web_security_group_id" {
  value = aws_security_group.web_sg.id
}

output "db_security_group_id" {
  value = aws_security_group.db_sg.id
}
```

Open CMD, then Configure using your AWS Programmatic credentials, type – aws configure and enter your credentials (Access key and Secret access key)

Then use the following commands in the terraform networking directory (if you are not in the directory, cd to that directory) to deploy the terraform infrastructure from CLI;

terraform init – to initialize the terraform

terraform plan – to preview the resources which will be created in the AWS

terraform apply – to deploy the resources into the AWS cloud

Once, these resources are deployed successfully, you can see the outputs and verify in the console as well;

```
Apply complete! Resources: 14 added, 0 changed, 0 destroyed.

Outputs:

db_security_group_id = "sg-02d1977bd943e10a7"
private_subnet_ids = [
  "subnet-0ccb3181bc86d224b",
  "subnet-07d8c971fa8357be9",
]
public_subnet_ids = [
  "subnet-0a8736df876f2047c",
  "subnet-0fc9864465b7493cf",
]
vpc_id = "vpc-0469b03020a4d9bf8"
web_security_group_id = "sg-0e042b05953c64171"
```

2. Use CloudFormation to deploy EC2, RDS, and Lambda resources.

Once all the networking resources are created and verified in the console, we can go ahead with the creation of EC2 instances, RDS, and Lambda resources using the CloudFormation templates. The CloudFormation templates for the respective resources are as follows;

CloudFormation Template for EC2 (Shashank-puppala-ec2-project.yaml)

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Deploying EC2 instance with and HTML page using the existing VPC, Subnets, and Security groups created earlier with the help c

Parameters:
  KeyName:
    Type: String
    Default: Final
    Description: EC2 Key Pair for SSH

  SubnetId:
    Type: AWS::EC2::Subnet::Id
    Description: Subnet where EC2 will be launched
    Default: "subnet-0a8736df876f2047c"   # public_subnet_id

  SecurityGroupId:
    Type: String
    Description: Security Group ID in the same VPC
    Default: "sg-0e042b05953c64171"       # web_security_group_id

Resources:
  EC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      InstanceType: t3.micro
      ImageId: ami-0fa3fe0fa7920f68e
      KeyName: !Ref KeyName
      Tags:
        - Key: Name
          Value: shashank-puppala-EC2-instance-project
```

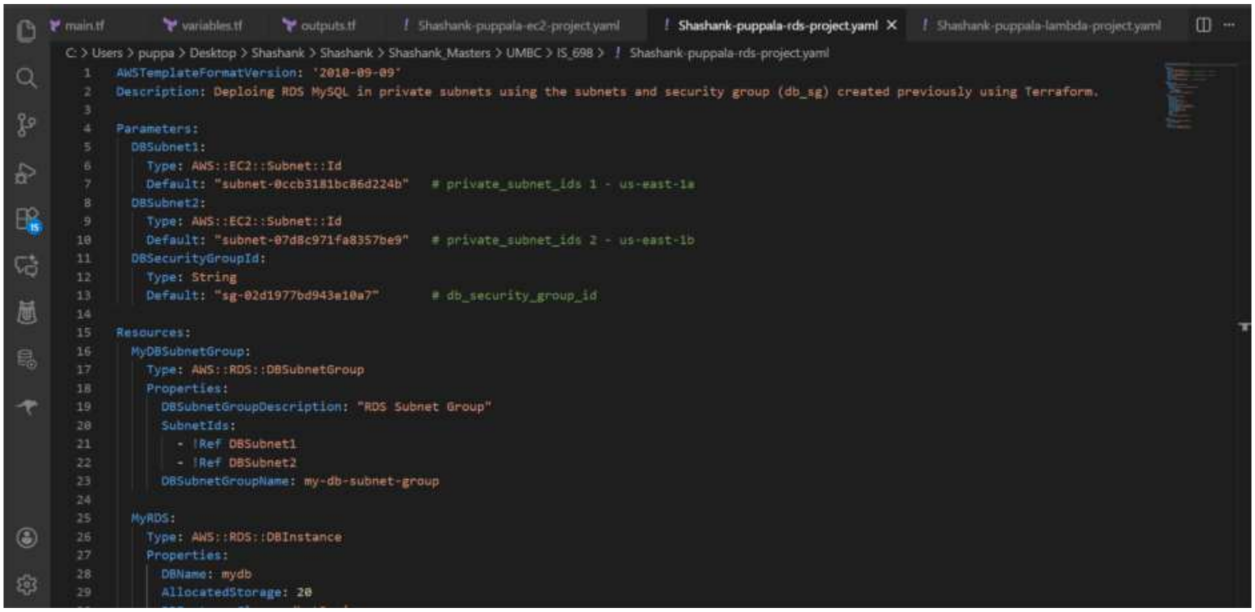
Deploy the stack from the CLI using the following command -

aws cloudformation deploy --stack-name ec2-stack --template-file Shashank-puppala-ec2-project.yaml --parameter-overrides KeyName=shashank-puppala-linux-pair-1 --region us-east-1

(Replace the template file name with the name you are saving the template and KeyName with the actual KeyValue pair name from your AWS account.)

(Note: - Replace the Subnet values in the yaml template with the ones created in the previous step during terraform deployment.)

Next, for the CloudFormation yaml file for RDS creation (Shashank-puppala-rds-project.yaml): -



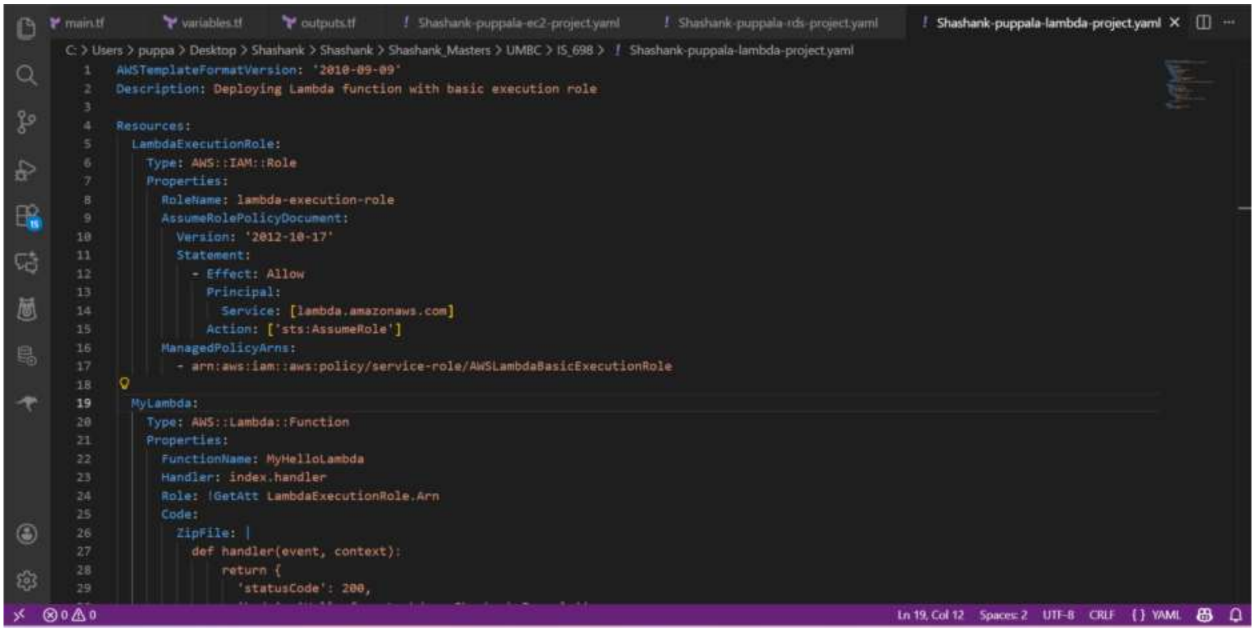
Deploy the rds stack using the following command;

aws cloudformation deploy --stack-name rds-stack --template-file Shashank-puppala-rds-project.yaml --region us-east-1

Replace the template file name with the one you are saving the file with.

Similar to the above step, replace the subnet values with the ones created during the terraform deployment.

CloudFormation yaml file for creating the Lambda function (Shashank-puppala-lambda-project.yaml)

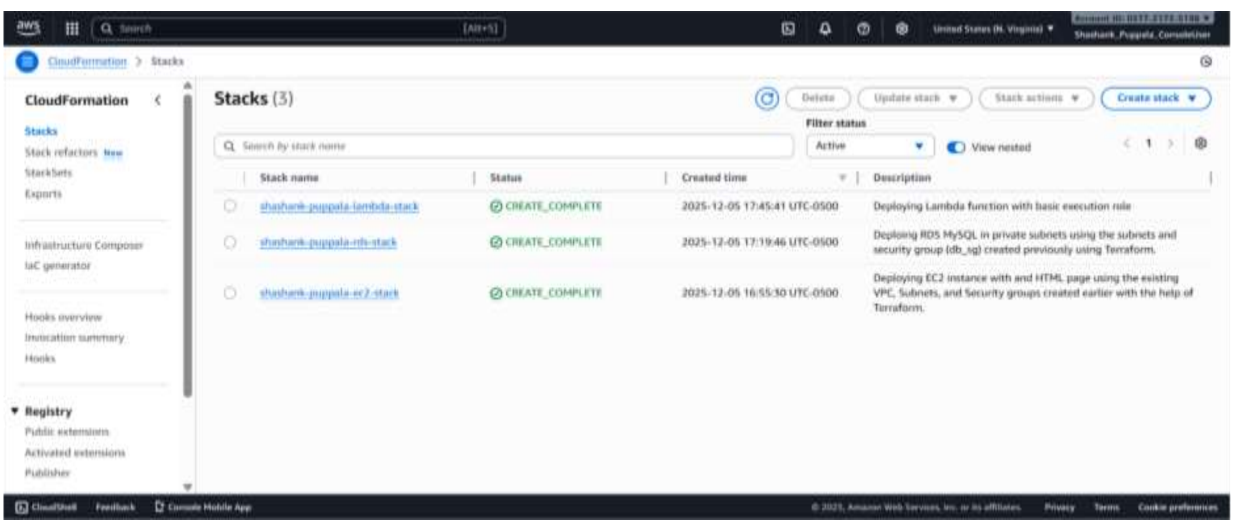


Deploy this above lambda code as well into the AWS cloud using the following CLI command: -

aws cloudformation deploy --stack-name lambda-stack --template-file Shashank-puppala-lambda-project.yaml --capabilities CAPABILITY_NAMED_IAM --region us-east-1

Replace the template file name with the one you are saving the file with.

Once, all the stacks are successfully deployed you can verify in the AWS Console;

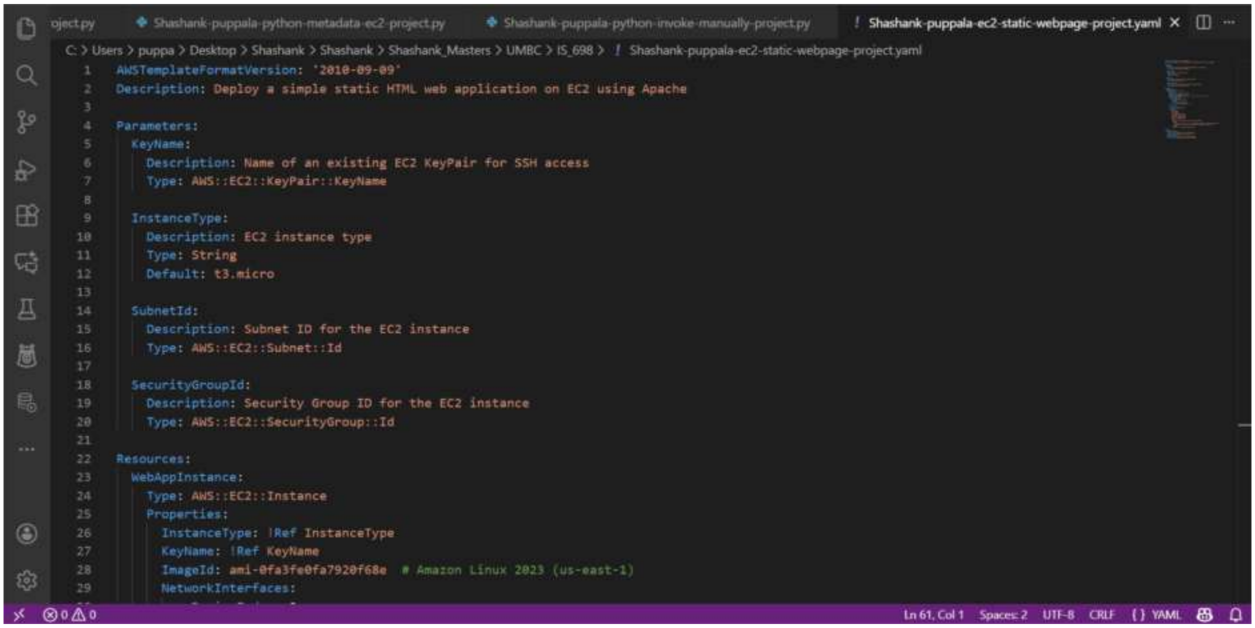


3. Deploy a simple web application on EC2 (e.g., static HTML page).

For this step, we can do it in several ways. We can deploy the HTML code while creating the EC2 in the previous CloudFormation yaml itself directly. Once the stack is deployed, you can directly open the Public IP of the running instance in the browser to view the static. But for convenience, we are even creating a separate yaml template as well to deploy a static web page in the EC2 instance;

(Note: - The CloudFormation yaml file used in previous step, has the required code to view the static page, directly open it in the browser or use the below template to create a new one)

CloudFormation yaml for Static HTML Web page (Shashank-puppala-ec2-static-webpage-project.yaml): -

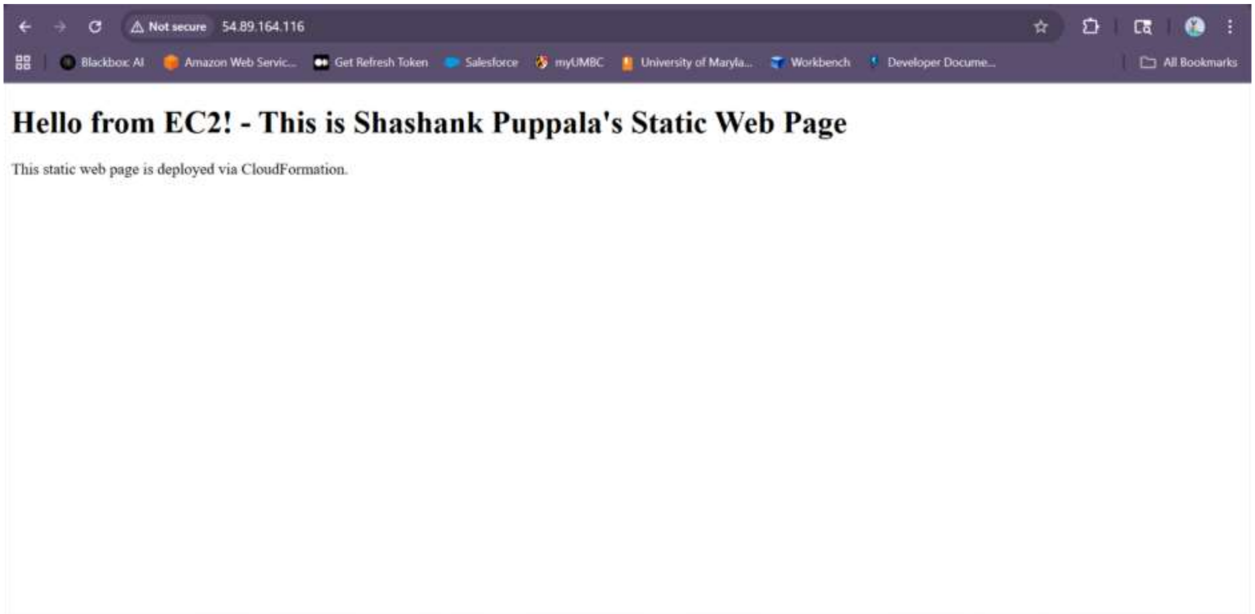


Deploy the above yaml template file into AWS using the following command;

aws cloudformation deploy --stack-name ec2-stack --template-file Shashank-puppala-ec2-static-webpage-project.yaml --parameter-overrides KeyName=shashank-puppala-linux-pair-1 --region us-east-1

Replace the template file with the file name you are saving the file with. In the code, replace the AMI-image ID with the actual one from AWS and use the KeyValue from your account.

The static webpage would be as follows;



4. Configure the database backend for the application.

To configure and verify that the database (RDS) backend for the application to use, we need to first find out the RDS details from the console;

1. **Open** AWS Console, and **Search** for RDS.

2. Open the RDS and Copy the **RDS endpoint**.

3. We have created the RDS using the CloudFormation template, where we have given the Name for the Database, **Username and Password** required for login. Copy them as they are required for logging in the RDS.

4. Once we have all the details related to RDS, we can SSH into the instance which we are using to display the static web page and login into RDS to configure it for the application.

5. SSH into the EC2 instance using the command - ssh -i <Key_Value_Pair> ec2-user@<Public_IP>

Windows users – Can use the Putty and .ppk file for logging into the instance

6. Once into the EC2 use the following command to update and install the MySQL to use;

sudo yum update –y

sudo dnf install mariadb105 -y

sudo systemctl enable httpd

sudo systemctl start httpd

mysql -h <RDS-endpoint> -u <username> -p

As the application is about working with Employee data, insert the employee data into the database;

First after getting inside mysql, type - SHOW Databases;

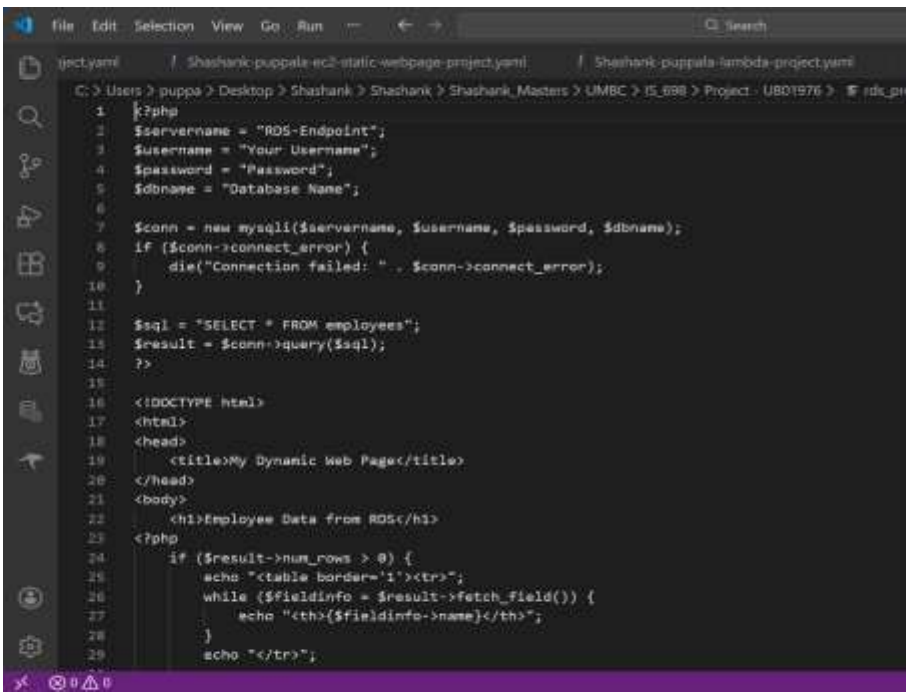
Change to the database which you have created in the step 2 using CloudFormation;

Next create the datatable and insert values into it - CREATE TABLE employees (Id INT PRIMARY KEY, Name VARCHAR(50) NOT NULL, email VARCHAR(100), Department VARCHAR(50));

INSERT INTO employees (Id, name, email, Department) VALUES (1, ‘test', 'shashank@example.com', ‘IS’);

Once, the data is successfully inserted we can use the php to design the dynamic web page to display the employee data from data table;

php code to display the web page with employee records-

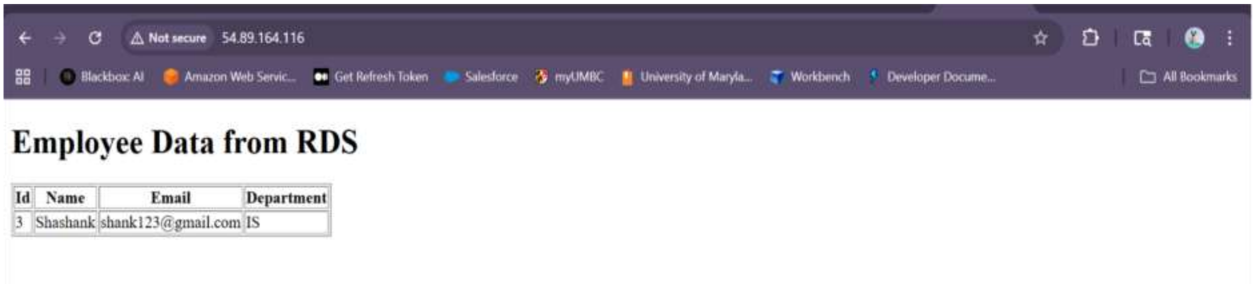


Refer the following GitHub Link for the Codes -

Enter the RDS endpoint, Username, Password, and DB name which you are using and save the code.

sudo systemctl restart httpd – Use this command to restart the web page

If all the steps are completed successfully, the output can be seen on the browser as follows;



4. Implement autoscaling for EC2 instances.

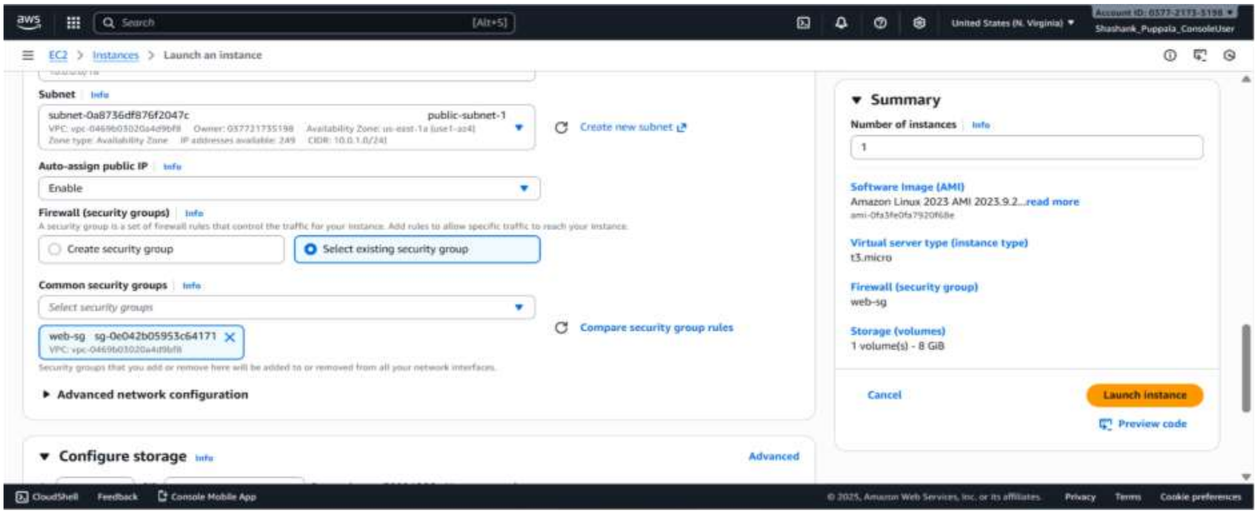
Next, to implement autoscaling we need to first create the AMI Image if it's not existing, design a launch template, create the ALB and target groups to handle the traffic, and then finally ASG to launch the instances based on the CPU utilization for application;

Let's begin with the First step – Web Server Instance creation

You can use the same instance which we were using for the displaying of the database details for application or can create a new auto scaling instance in the same subnet and security group (with SSH access using Port 22 and HTTP port 80) as image reference;

Steps to Create Web Server Instance –

1. Go to **EC2 on Console → Click on Instances → Select Launch Instance**
2. Set:
   Name: WebServer-Base
   AMI: Amazon Linux 2 (ex: - ami-0fa3fe0fa7920f68e)
   Instance Type: t3.micro
   Key Pair: Select existing from your Account
   Network: Choose public subnet which we have created for web server
   Auto-assign Public IP: Enabled
   Security Group: Select the web server sg created using Terraform (Which allows SSH (from 22) & HTTP (80)
3. Launch the instance.



Next SSH into the instance and verify if the instance is able to display the application on the browser. (Existing instance for which application is configured with database will display the web page, for new instance you can create the static html page and check the accessibility) You can use the below code after SSH into instance;

sudo yum update -y

sudo yum install -y httpd

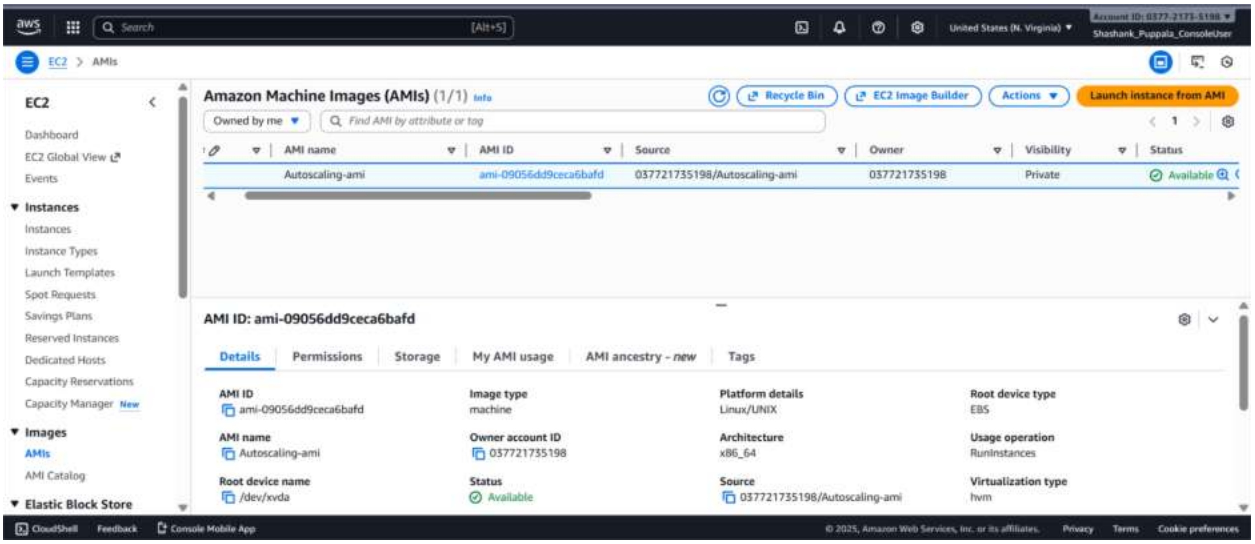
sudo systemctl enable httpd

sudo systemctl start httpd

echo "This Instance is part of EC2 Auto Scaling! - Shashank Puppala" > /var/www/html/index.html

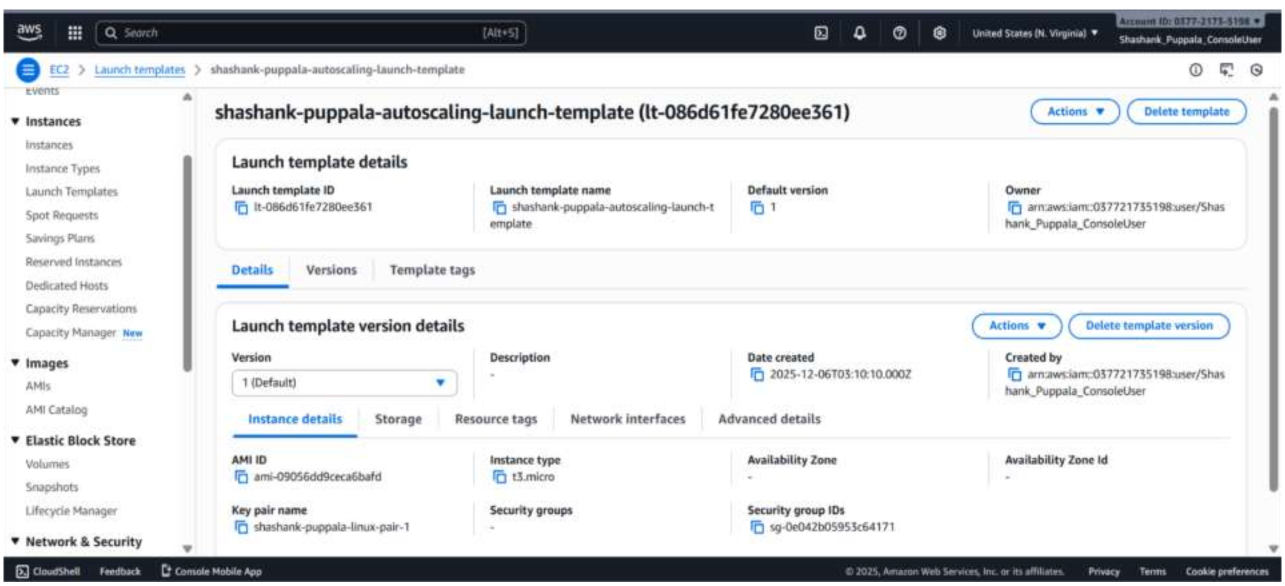
Step 2 would be creating the AMI Image of the instance;

1. Select the instance → **Actions** → **Image** → **Create Image**
2. Name the AMI: Provide a Name to the Image
3. Click Create Image
4. Wait until status is “available.”



Step 3: - Create a Launch Template

1. Go to **EC2 → Launch Templates → Create Template**
2. Use the created AMI from the above step once it is available.
3. Set instance type and security group (Web-sg).
4. Note: - Do not select any subnet while creation.
5. Create a template.

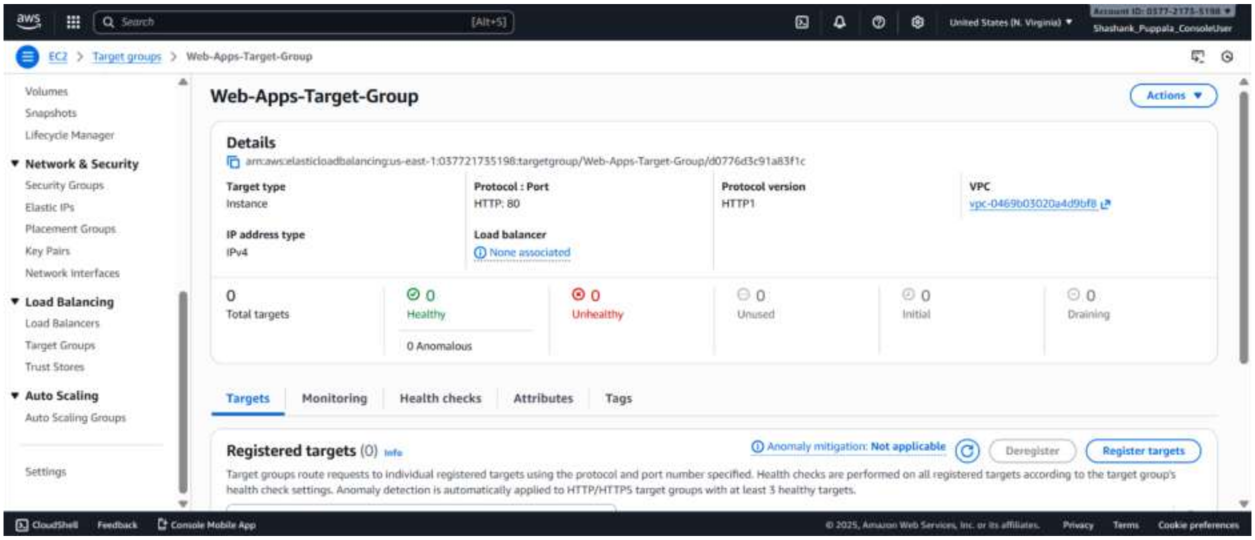


Step 4: - Creating ALB and Target Groups

(Before creating these ALB and Target groups, make sure to have ALB-sg which allows HTTP (80) and Web-sg should allow HTTP (80) from ALB and SSH from port 22)

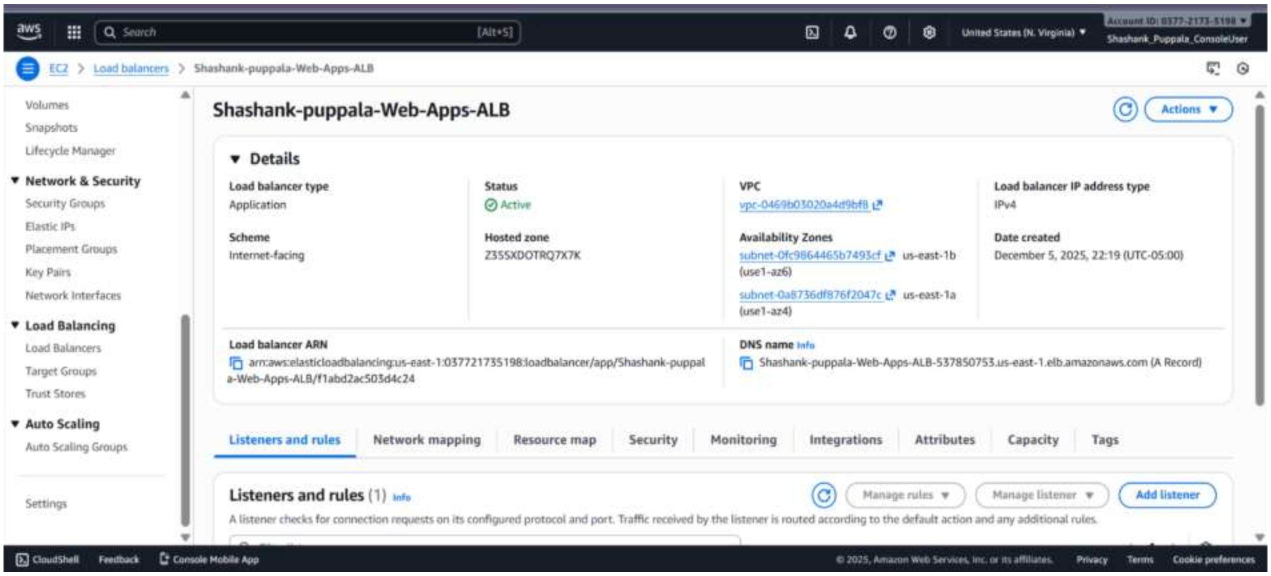
For Target Group creation: -

1. Go to **EC2** → **Target Groups** → **Create**
2. Choose **Instance** target type.
3. Select Protocol:HTTP1
4. Set port 80 , Name the Target group , and health check path to /.
5. Note: - Do not register targets manually; autoscaling will allocate automated resources.



Next for ALB creation;

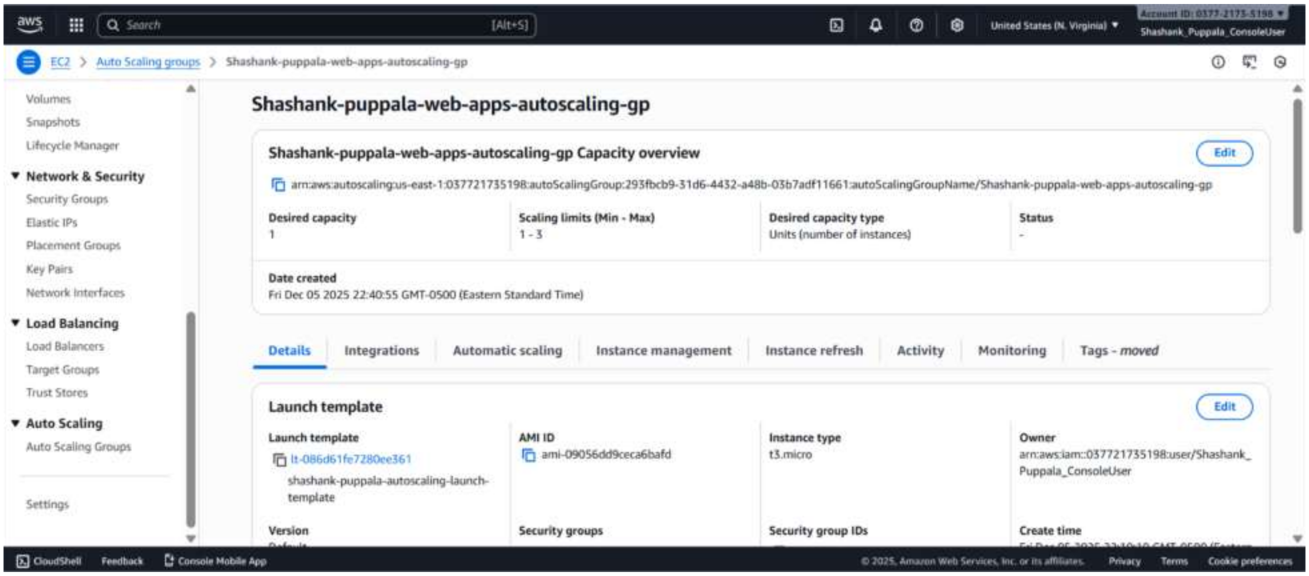
1. Go to **EC2** → **Load Balancers** → **Create ALB**
2. Name ALB (Application Load Balancing)
3. Select the Scheme: Internet-facing
4. Listeners: HTTP 80
5. Subnets: Select Two public subnets in different AZs (Which we have created in the earlier steps using terraform)
6. Security Group: ALB-SG
7. Attach the previously created Target Group from the above step.
8. Click Create Load Balancer and wait until it is active.



Paste the ALB DNS in the web browser and check if the ALB is responding or not

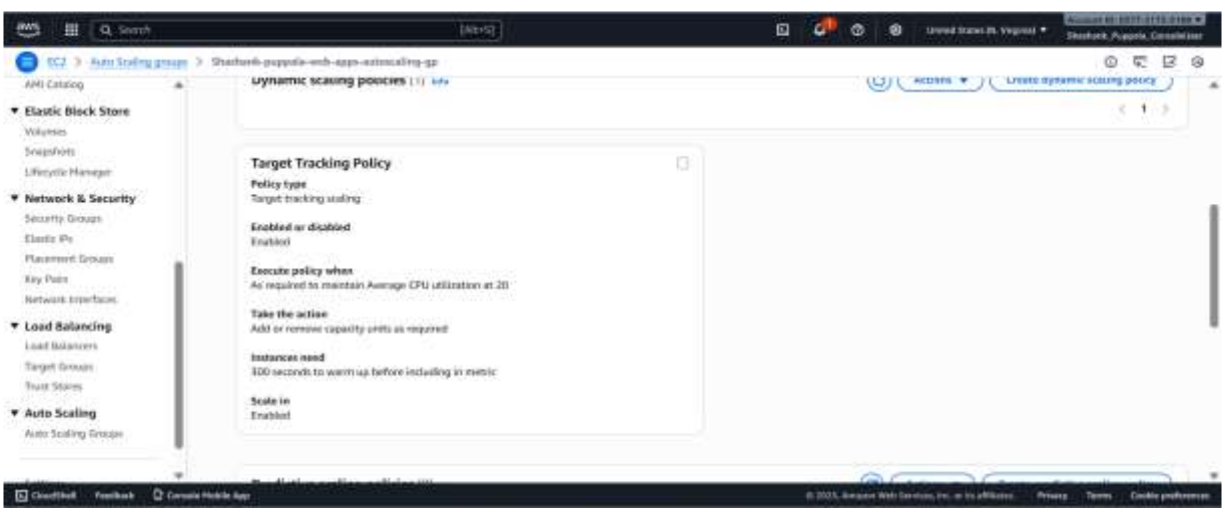
Step 5: - Creation of Auto Scaling Group (ASG)

1. Go to **EC2 → Select Auto Scaling Groups → Click on Create**
2. Select the **Launch Template** which you created earlier.
3. Select the VPC and 2 of the public subnets (Created before, the custom VPC from terraform and respective 2 public subnets in different availability zones).
4. Attach to the ALB by selecting the Target Group.
5. Set capacities: Minimum: 1
   Maximum: 3
6. Click Next and Click on Create Auto Scaling Group.



To check when to launch the instance, ASG requires a Scaling policy, so create a scaling policy to let ASG know when to scale up and scale down;

1. Open **ASG** → Click on **Automatic Scaling** → **Add Policy**
2. Select **Target Tracking Policy**
3. Choose Average CPU Utilization
4. Set target threshold (e.g., 30–50%).
5. Save policy.

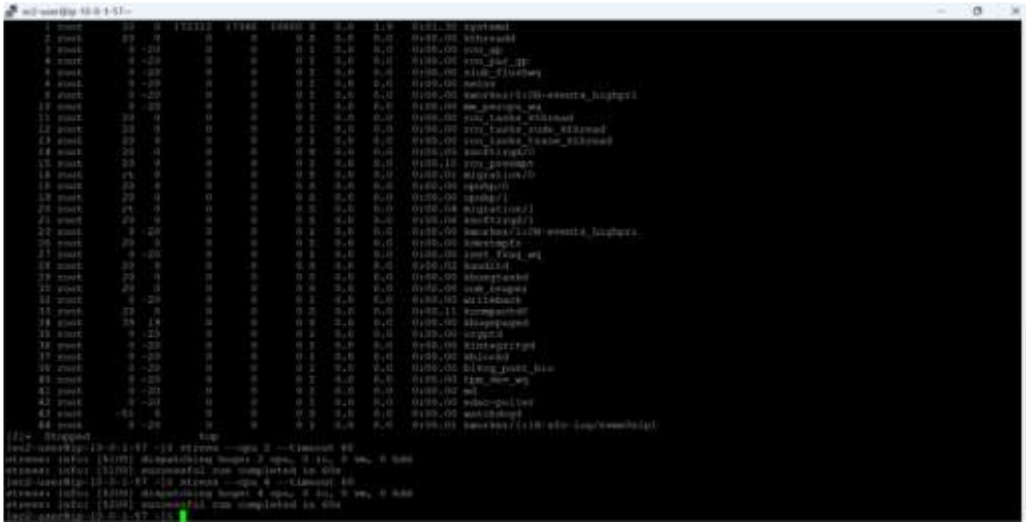


Once, everything is in place, you can stress the system if to check if it is scaling up or not;

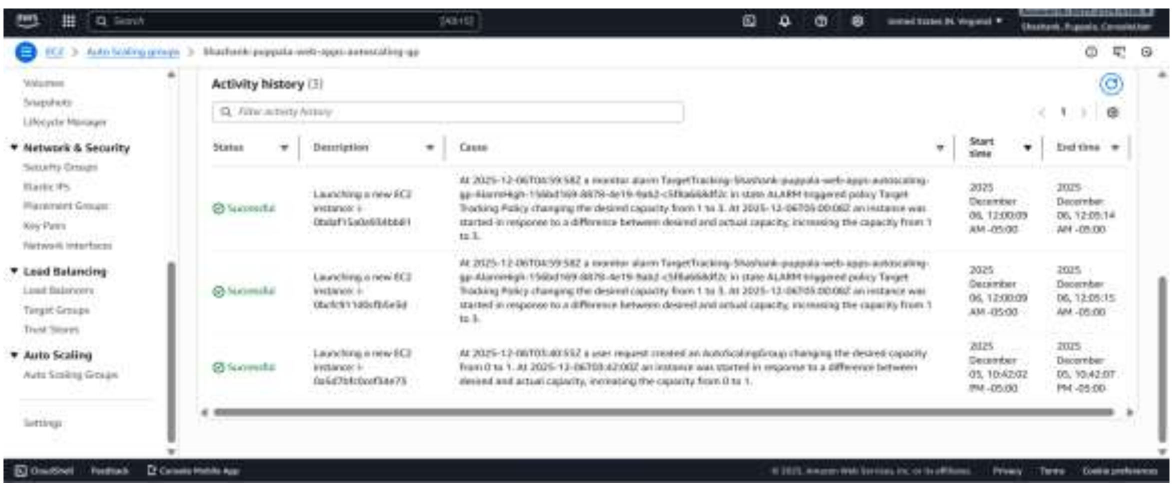
SSH into the ASG instance and type the following command;

sudo yum install -y stress

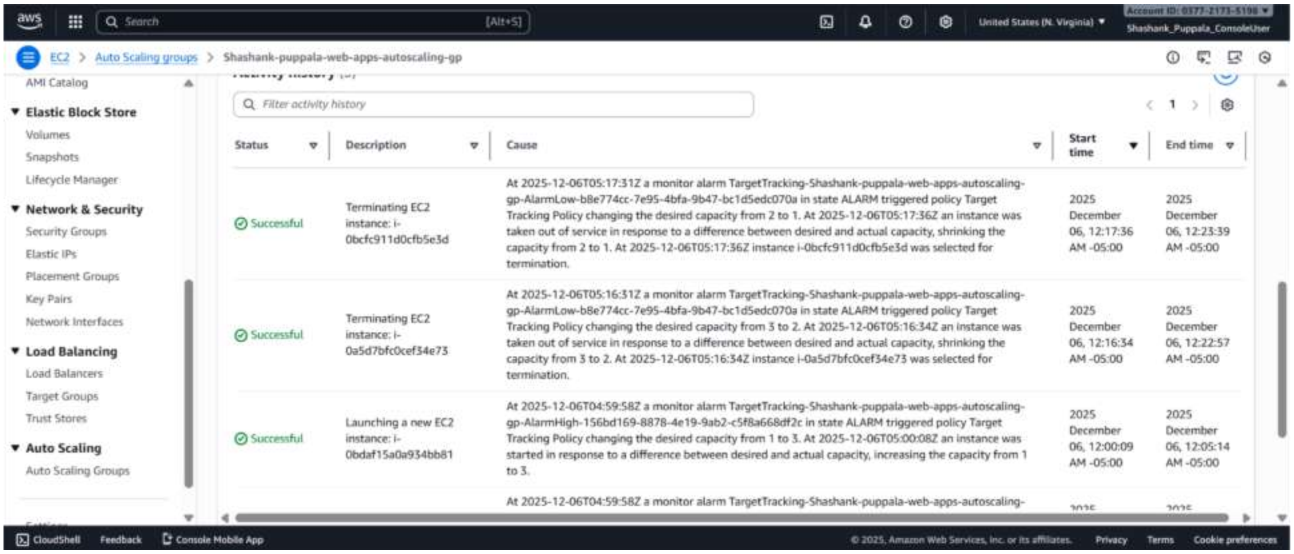
stress –cpu 4 –timeout 60



Once the system cannot handle the stress, ASG automatically creates new instances to handle the stress and reduce CPU load;



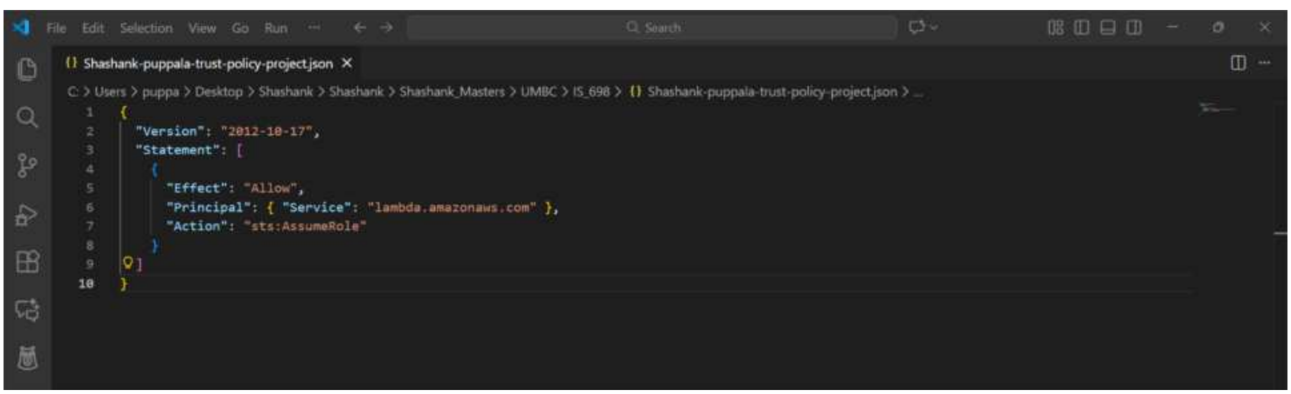
If the Stress is reduced it will automatically scale down the instance;



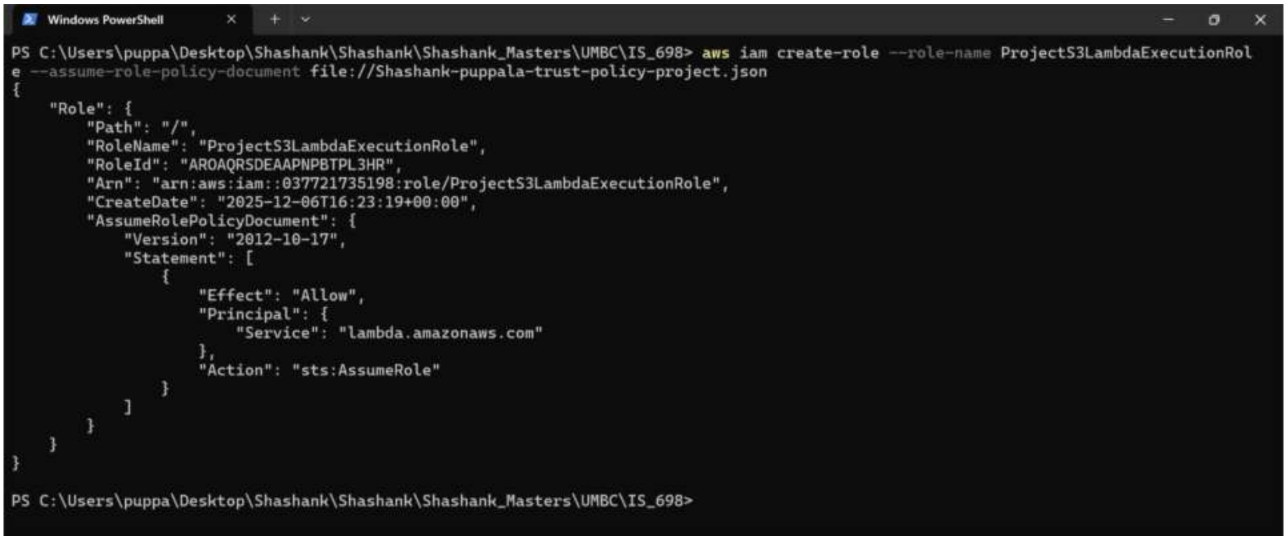
B. AWS Lambda for Logging S3 Uploads

1. Develop a Lambda function (in Python) that logs new S3 uploads to CloudWatch.

First create a Lambda Execution role who can run the function and provide the role with the necessary permissions;



Command to create role: aws iam create-role --role-name ProjectS3LambdaExecutionRole --assume-role-policy-document file://Shashank-puppala-trust-policy-project.json

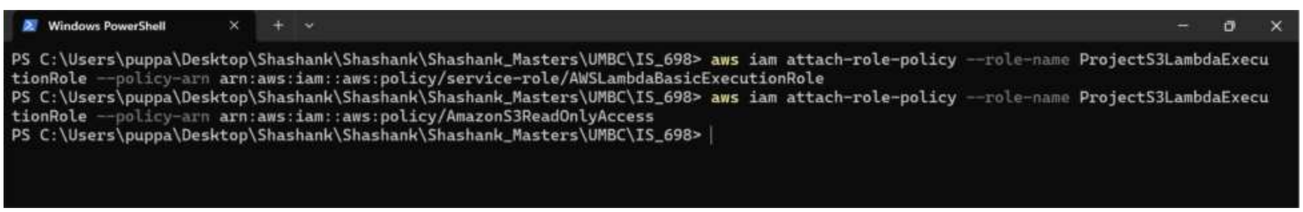


```
PS C:\Users\puppa\Desktop\Shashank\Shashank\Shashank_Masters\UMBC\IS_698> aws iam create-role --role-name ProjectS3LambdaExecutionRol
e --assume-role-policy-document file://Shashank-puppala-trust-policy-project.json
{
    "Role": {
        "Path": "/",
        "RoleName": "ProjectS3LambdaExecutionRole",
        "RoleId": "AROAQRSDEAAPNPBTPL3HR",
        "Arn": "arn:aws:iam::037721735198:role/ProjectS3LambdaExecutionRole",
        "CreateDate": "2025-12-06T16:23:19+00:00",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}

PS C:\Users\puppa\Desktop\Shashank\Shashank\Shashank_Masters\UMBC\IS_698>
```

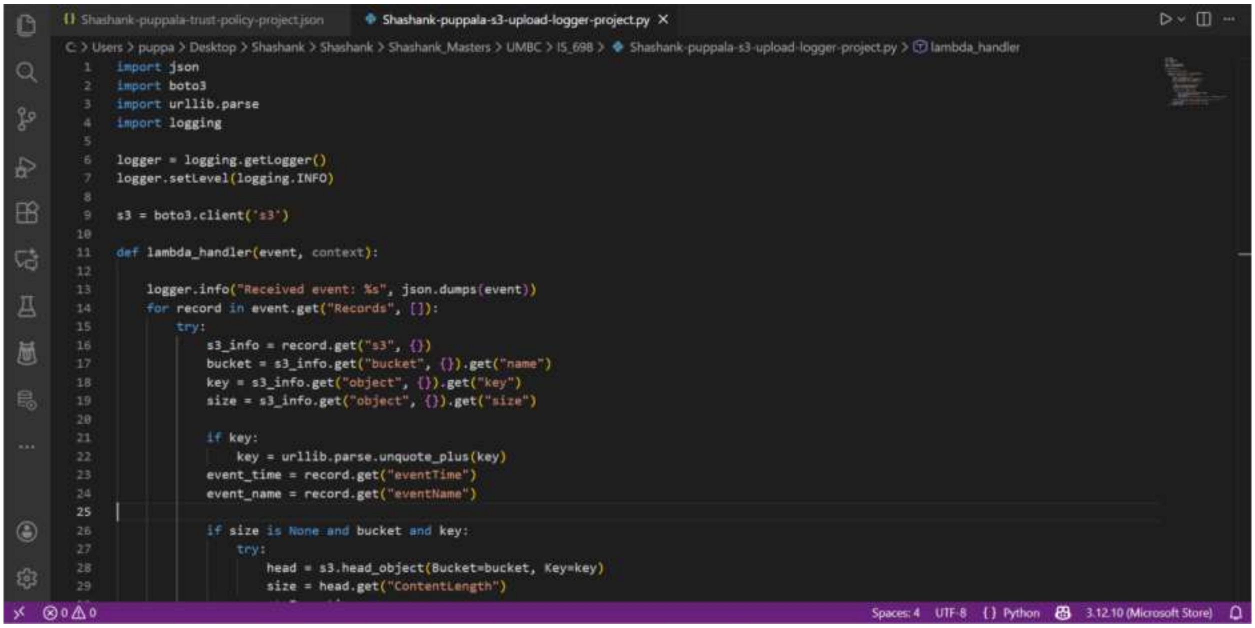
Commands to add permissions: aws iam attach-role-policy --role-name S3LambdaExecutionRole --policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess

aws iam attach-role-policy --role-name ProjectS3LambdaExecutionRole --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole



```
PS C:\Users\puppa\Desktop\Shashank\Shashank\Shashank_Masters\UMBC\IS_698> aws iam attach-role-policy --role-name ProjectS3LambdaExecu
tionRole --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
PS C:\Users\puppa\Desktop\Shashank\Shashank\Shashank_Masters\UMBC\IS_698> aws iam attach-role-policy --role-name ProjectS3LambdaExecu
tionRole --policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
PS C:\Users\puppa\Desktop\Shashank\Shashank\Shashank_Masters\UMBC\IS_698> |
```

Add the python script to show the details of the file uploaded to S3;



Zip it using the command: Compress-Archive -Path Shashank-puppala-s3-upload-logger-project.py -DestinationPath function.zip

Deploy the function from AWS CLI to Console using the following command;

aws lambda create-function --function-name ProjectS3UploadTrigger --zip-file fileb://C:/Users/puppa/Desktop/Shashank/Shashank/Shashank_Masters/UMBC/IS_698/function.zip --handler Shashank-puppala-s3-upload-logger-project.lambda_handler --runtime python3.12 --role arn:aws:iam::037721735198:role/ProjectS3LambdaExecutionRole

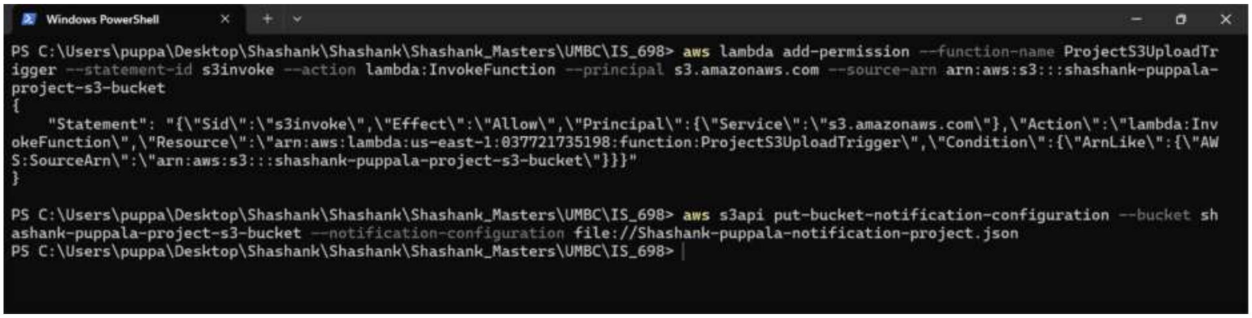
(Note: - Replace the Zip file path with the actual path where the function is in your local system and give the correct execution role name after verifying in the console)

2. Configure an S3 event trigger for Lambda.

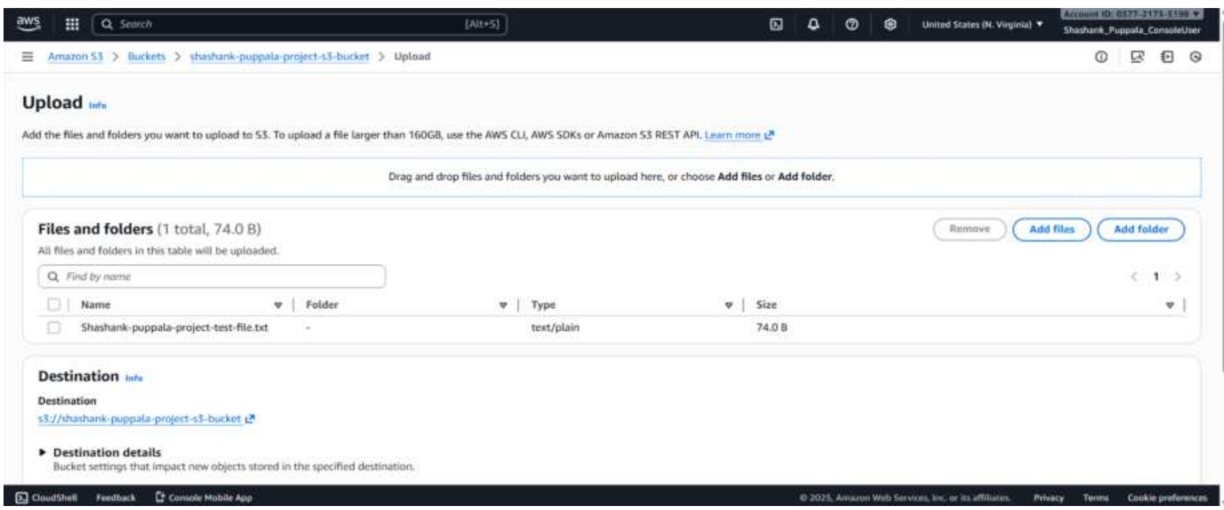
Once uploaded the function to the Console, create a S3 bucket and add the necessary permissions to the bucket to invoke the function. That can be done using the following command;

aws lambda add-permission --function-name ProjectS3UploadTrigger --statement-id s3invoke --action lambda:InvokeFunction --principal s3.amazonaws.com --source-arn arn:aws:s3:::shashank-puppala-project-s3-bucket

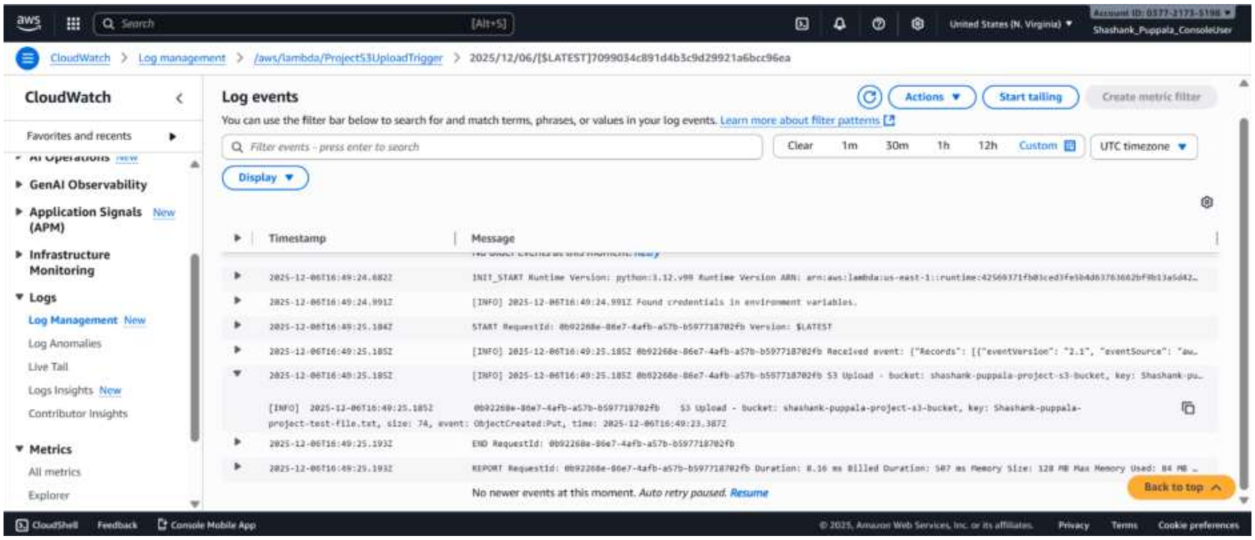
(Replace with your actual bucket name created in your AWS account) aws s3api put-bucket-notification-configuration --bucket shashank-puppala-project-s3-bucket --notification-configuration file://Shashank-puppala-notification-project.json



Now, if you upload any test file into S3 then lambda gets invoked automatically and log is generated;



c. Validate functionality by checking CloudWatch logs for uploaded file details.



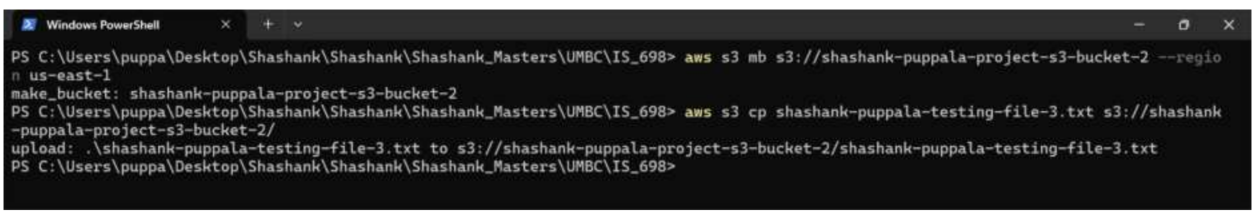
C. AWS Interaction

Following are the AWS CLI command and Python boto3 scripts to interact with AWS for various purposes;

1. Create an S3 bucket and upload a file.

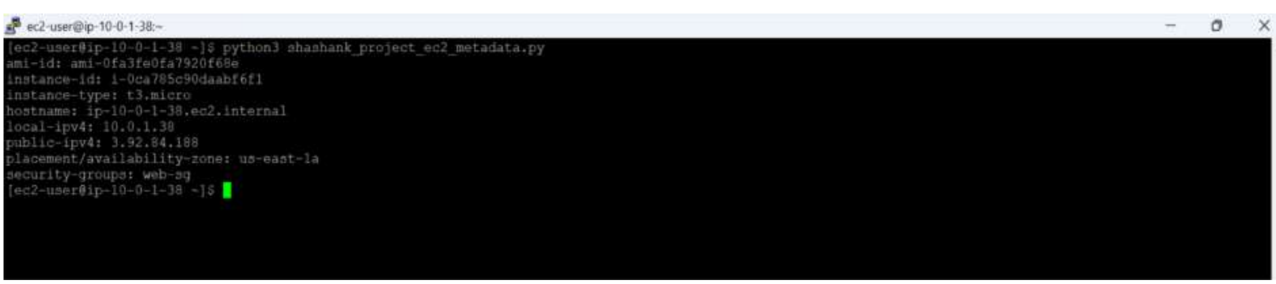
AWS CLI commands - aws s3 mb s3://shashank-puppala-project-s3-bucket-2 --region us-east-1

Uploading a file to bucket - aws s3 cp shashank-puppala-testing-file-3.txt s3://shashank-puppala-project-s3-bucket-2/

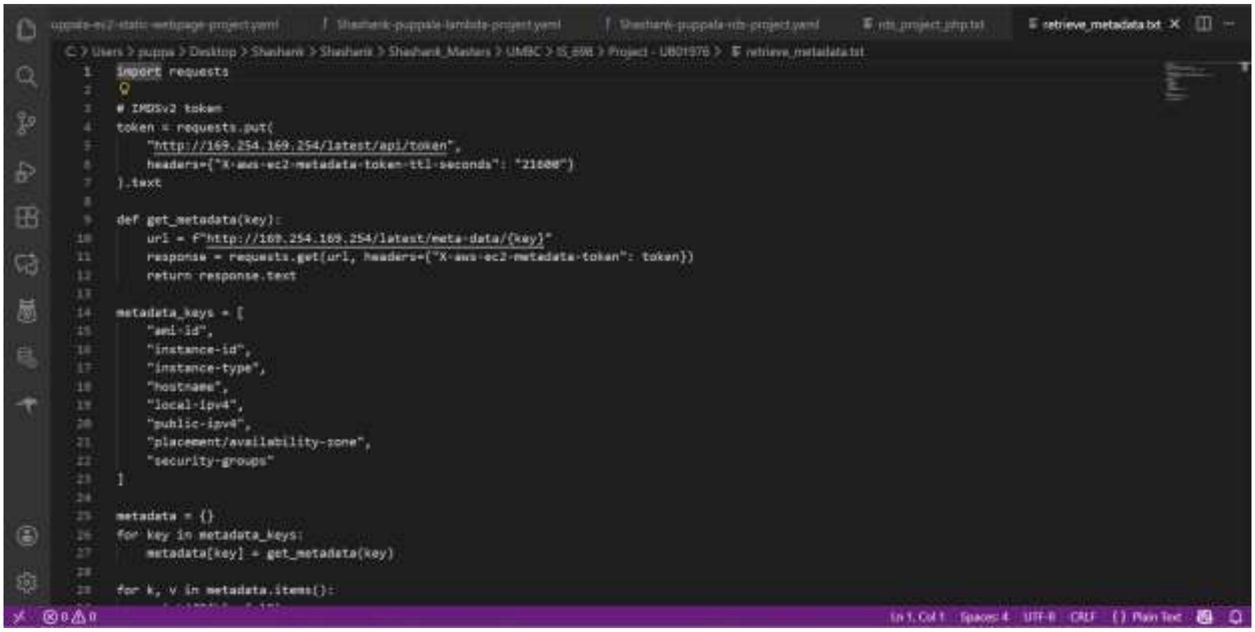


2. Retrieve EC2 metadata.

(Can be only retrieved from SSH into EC2 instance)

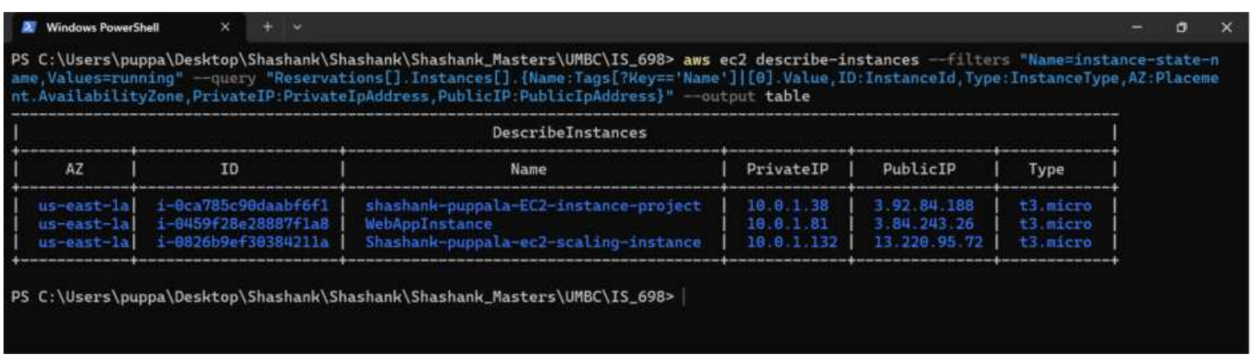


Use the following code to to retrieve metadata from ec2 instance;



3. List running EC2 instances.

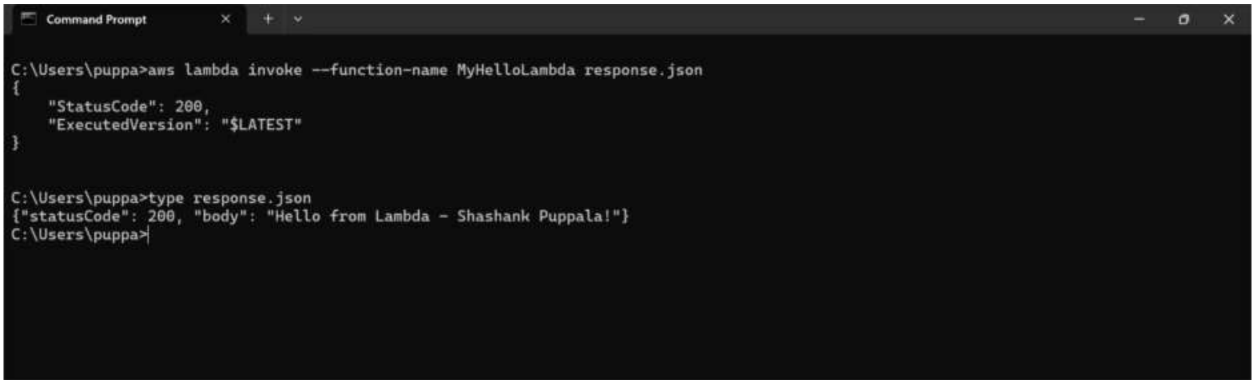
aws ec2 describe-instances --filters "Name=instance-state-name,Values=running" --query "Reservations[].Instances[].{ID:InstanceId,Type:InstanceType,AZ:Placement.AvailabilityZone,PrivateIP:PrivateIpAddress,PublicIP:PublicIpAddress}" --output table



4. Invoke Lambda manually.

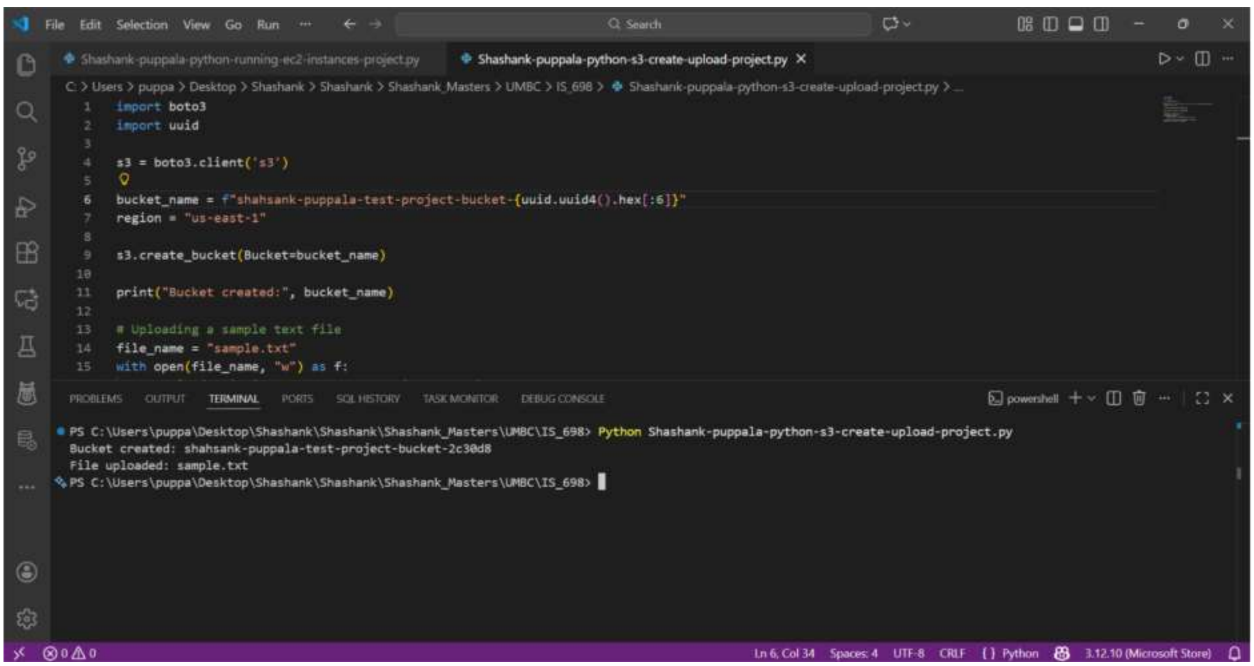
aws lambda invoke --function-name MyHelloLambda reponse.json

type response.json



Python Boto 3 scripts to perform the operations;

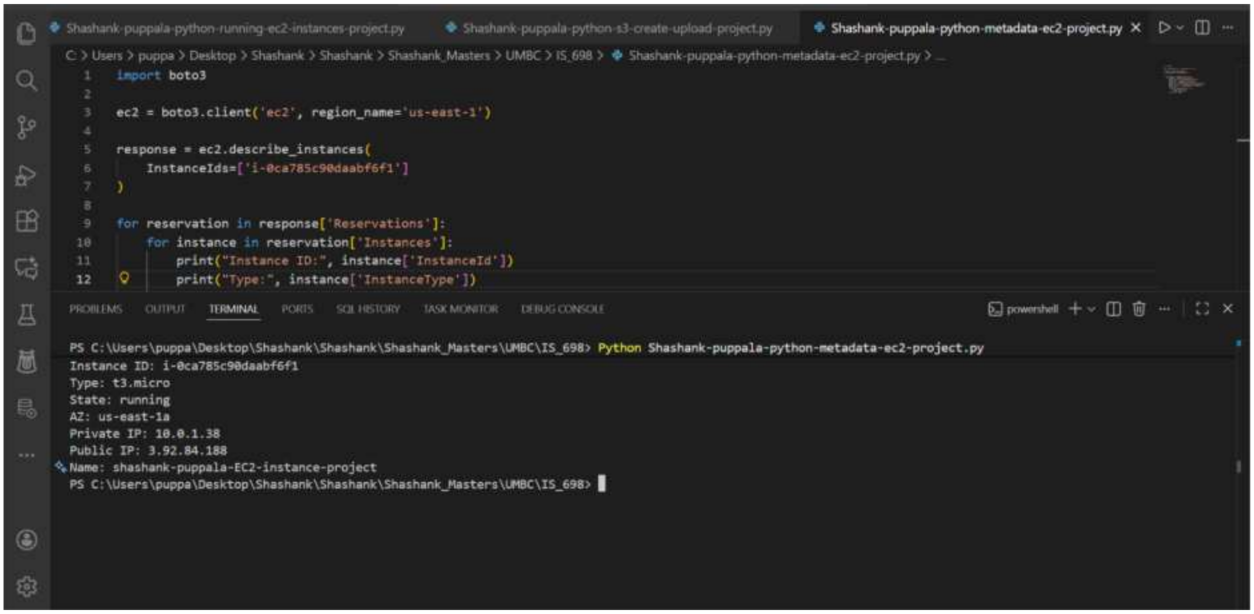
1. Create an S3 bucket and upload a file.



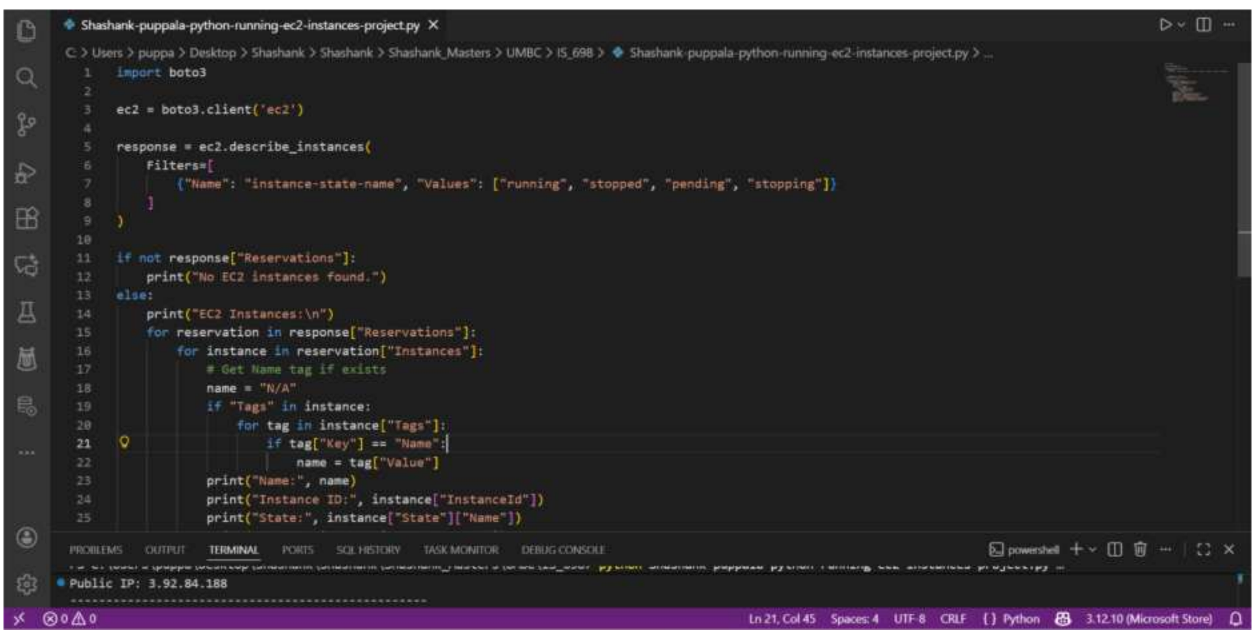
Run the Python commands in the terminal as follows: - Python <filename.py>

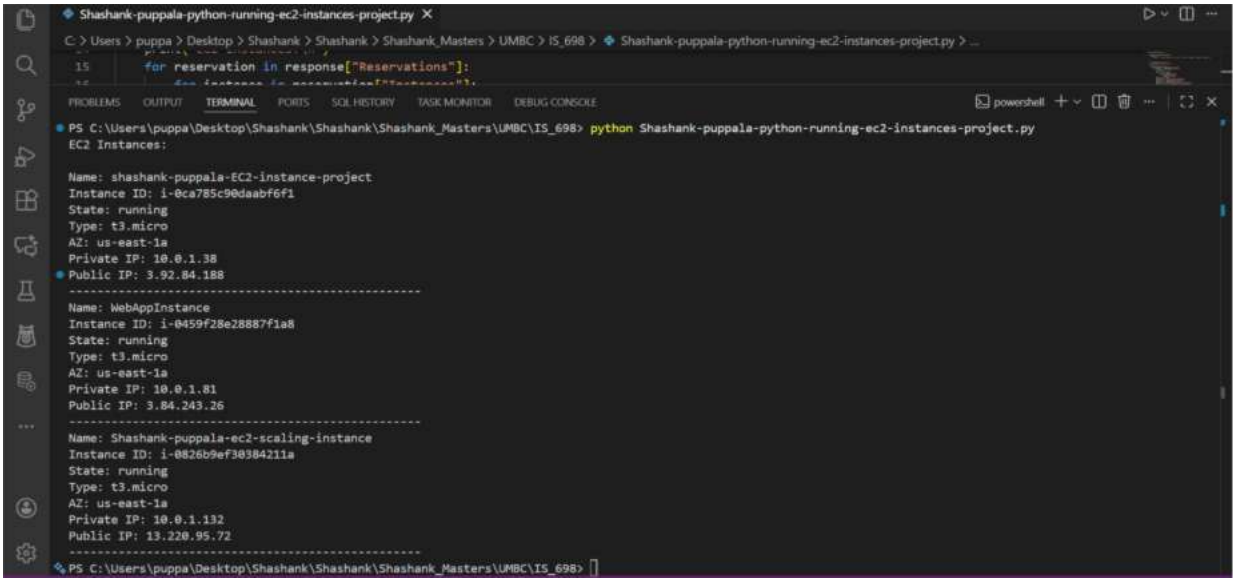
2. Retrieve EC2 metadata.

(Can only be done from SSH into the instance, Python can only describe the instances)

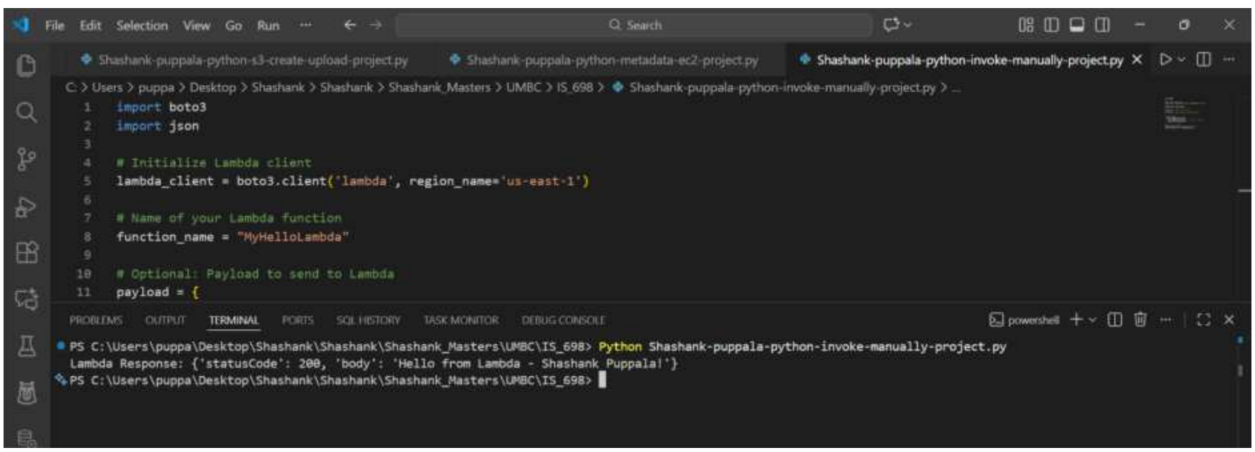


3. List running EC2 instances.





4. Invoke Lambda Function manually



3. GitHub Integration

Please find below the Link to the GitHub repository consisting of all the file related to project - https://github.com/UB01976/IS-698-Project-UB01976

4. Challenges Encountered and Future Work –

1. Issues with Auto-scaling testing – It was difficult to check whether the instances were being under stress or not. Had to increase the Stress level for CPU to 4 and time of 120 seconds to stress the system to ASG to create new instances to handle that stress.

2. Monitoring the CloudWatch Metrics – Had to create check the CloudWatch metrics to see whether the instance is stressing out or not. And had to reduce the target value (to 20 – 30 range) as to get the system stressed, because the max CPU utilization was at 40 percent.

Future Work: -

1. Enable Web Application Firewall – That would protect the ALB from any of the external attacks such as SQL injection or XSS.

2. Serverless Architecture – Using of Fargate or extending the lambda functionalities will reduce the usage of servers or EC2 instances.

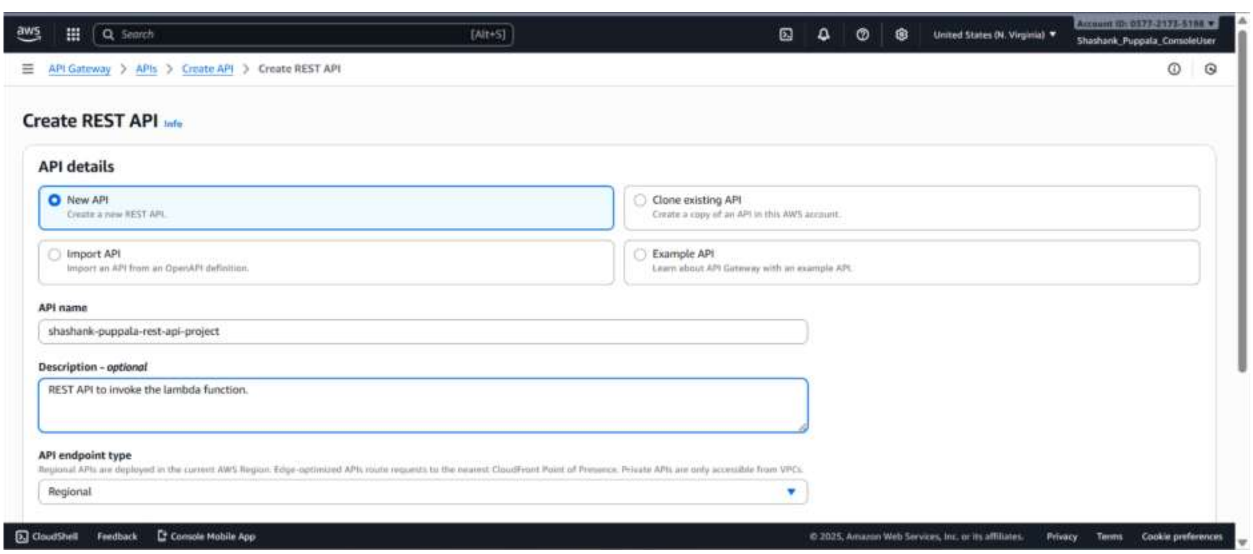
3. Apply least privilege access principle for IAM roles and enable bucket versioning, and have bucket policies in place, to increase the security measures for the systems.

4. **Better monitoring system –** Use CloudWatch advanced dashboards to check the CPU Utilization, and RDS connections. Can use amazon managed Grafana as well for better monitoring of ALB requests and lambda performance.

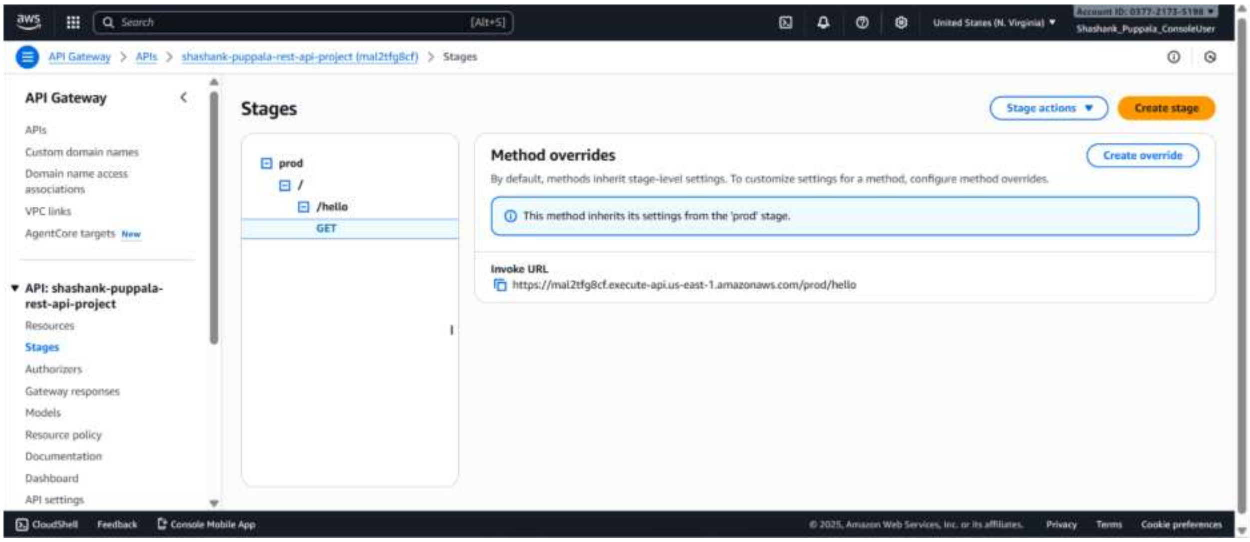
Bonus Challenge: -

Deploy **API Gateway** to invoke Lambda via HTTP requests.

1. Go to **API Gateway -> Click Create API -> Select REST API -> Click on Build** -> Give a name to the API
2. Click Create API
3. After the API is created, click on **Create Resource**



4. Select **Integration type as Lambda function** and select the lambda function you want to invoke.
5. Click on Save.
6. Once resource, is created Click on **Deploy API, and choose the stage (like prod or test)**
7. Click on Deploy



8. Once you get an invoke URL, you can copy and paste it in the browser to check if it is working or not.