

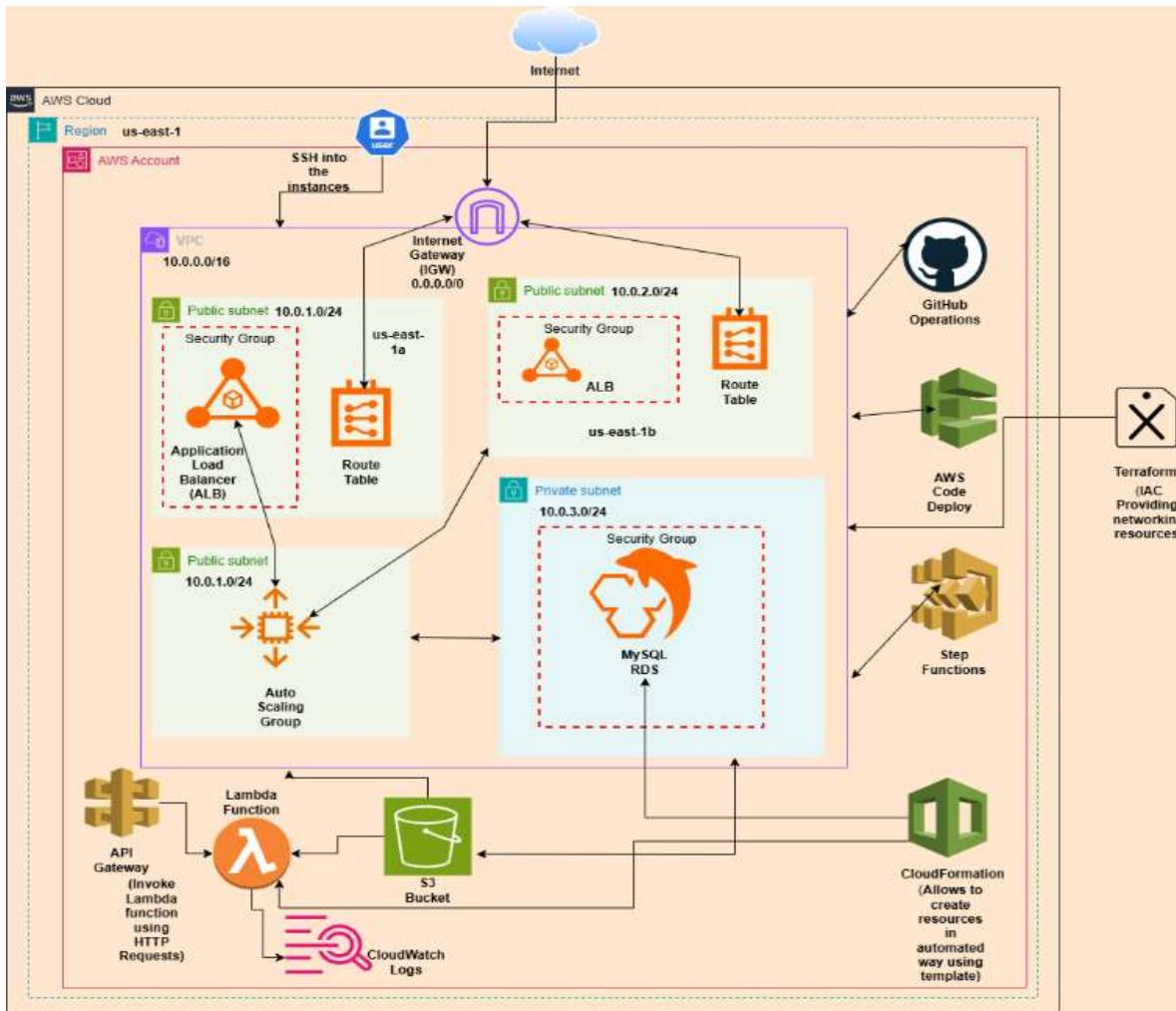
IS-698 Project Report – UB01976

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).

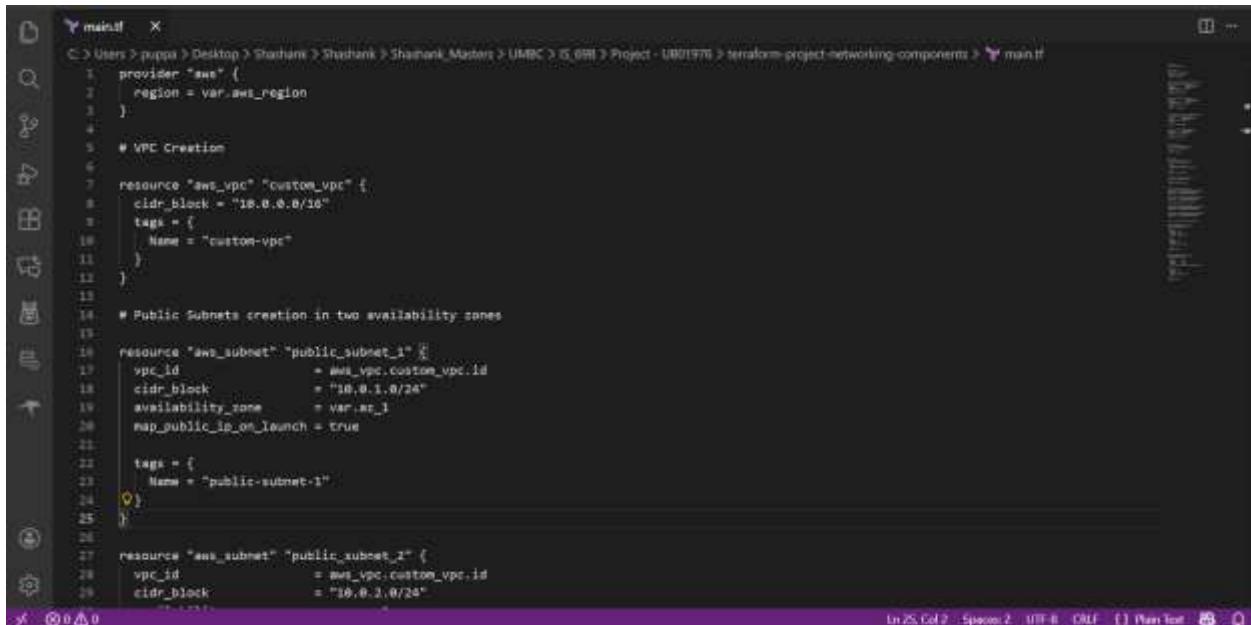
The implementation of the application begins with the creation of networking resources using the Terraform. We create the **VPC**, **Subnets**, **Route tables**, **Associations**, **IGW**, and **Security groups** using terraform. First, create a directory named as **terraform-project-networking-components** 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 -



```
main.tf
C:\Users\ruppa\Desktop\Shashank\Shashank_Mawani\UMBC\15.098\Project - UB01976\terraform-project-networking-components> main.tf
1 provider "aws" {
2   region = var.aws_region
3 }
4
5 # VPC Creation
6
7 resource "aws_vpc" "custom_vpc" {
8   cidr_block = "10.0.0.0/16"
9   tags = {
10     Name = "custom-vpc"
11   }
12 }
13
14 # Public Subnets creation in two availability zones
15
16 resource "aws_subnet" "public_subnet_1" {
17   vpc_id          = aws_vpc.custom_vpc.id
18   cidr_block      = "10.0.1.0/24"
19   availability_zone = var.availability_zones[0]
20   map_public_ip_on_launch = true
21
22   tags = {
23     Name = "public-subnet-1"
24   }
25 }
26
27 resource "aws_subnet" "public_subnet_2" {
28   vpc_id          = aws_vpc.custom_vpc.id
29   cidr_block      = "10.0.2.0/24"
```

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

variables.tf -

```
variables.tf X
C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698\Project - UB01976\terraform-project-networking-components> variables.tf
1 variable "aws_region" {
2   default = "us-east-1"
3 }
4
5 variable "az_1" {
6   default = "us-east-1a"
7 }
8
9 variable "az_2" {
10  default = "us-east-1b"
11 }
12 |
```

outputs.tf -

```
outputs.tf X
C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698\Project - UB01976\terraform-project-networking-components> outputs.tf
1 output "vpc_id" {
2   value = aws_vpc.custom_vpc.id
3 }
4
5 output "public_subnet_ids" {
6   value = [
7     aws_subnet.public_subnet_1.id,
8     aws_subnet.public_subnet_2.id
9   ]
10 }
11
12 output "private_subnet_ids" {
13   value = [
14     aws_subnet.private_subnet_1.id,
15     aws_subnet.private_subnet_2.id
16   ]
17 }
18
19 output "web_security_group_id" {
20   value = aws_security_group.web_sg.id
21 }
22
23 output "db_security_group_id" {
24   value = aws_security_group.db_sg.id
25 }
```

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.

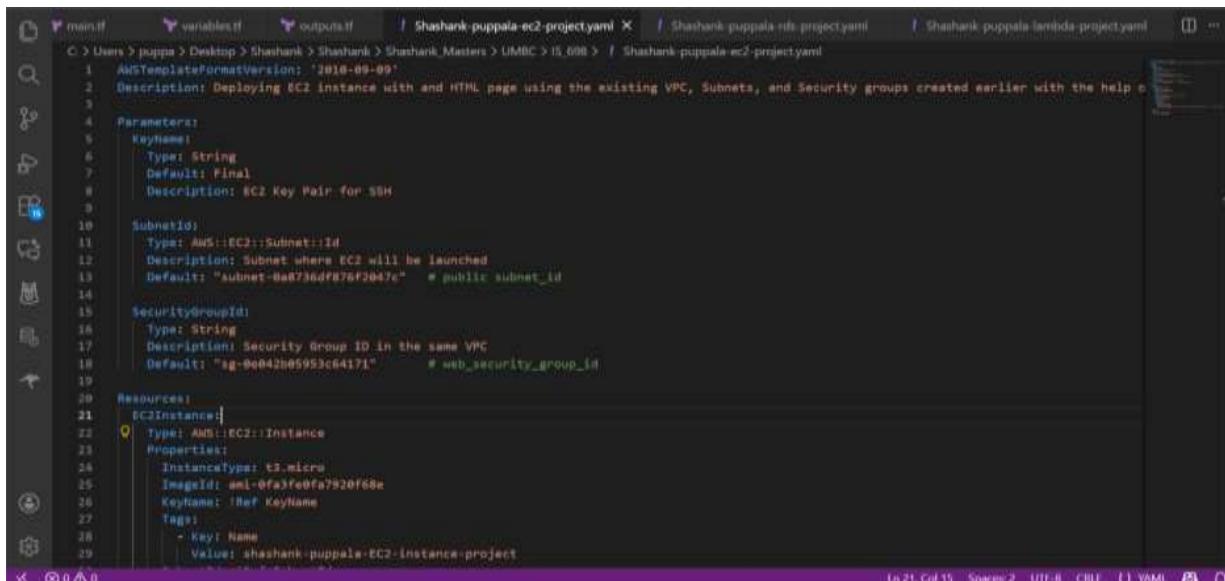
Outputs:

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 for the respective resources are as follows;

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



```
C:\Users\puppa>Desktop>Shashank>Shashank>Shashank_Masters>LIMDE>10_000>/ Shashank-puppala-ec2-project.yaml
1 AWSTemplateFormatVersion: '2010-09-09'
2 Description: Deploying EC2 instance with and HTML page using the existing VPC, Subnets, and Security groups created earlier with the help o
3
4 Parameters:
5   KeyName:
6     Type: String
7     Default: Final
8     Description: EC2 Key Pair for SSH
9
10  SubnetId:
11    Type: AWS::EC2::Subnet::Id
12    Description: Subnet where EC2 will be launched
13    Default: "subnet-0a8736df876f2047c" # public subnet_id
14
15  SecurityGroupId:
16    Type: String
17    Description: Security Group ID in the same VPC
18    Default: "sg-0e042b05953c64171" # web_security_group_id
19
20 Resources:
21   EC2Instance:
22     Type: AWS::EC2::Instance
23     Properties:
24       InstanceType: t3.micro
25       ImageId: ami-0fa3fe0a7920f68e
26       KeyName: !Ref KeyName
27       Tags:
28         - Key: Name
29           Value: shashank-puppala-EC2-instance-project
```

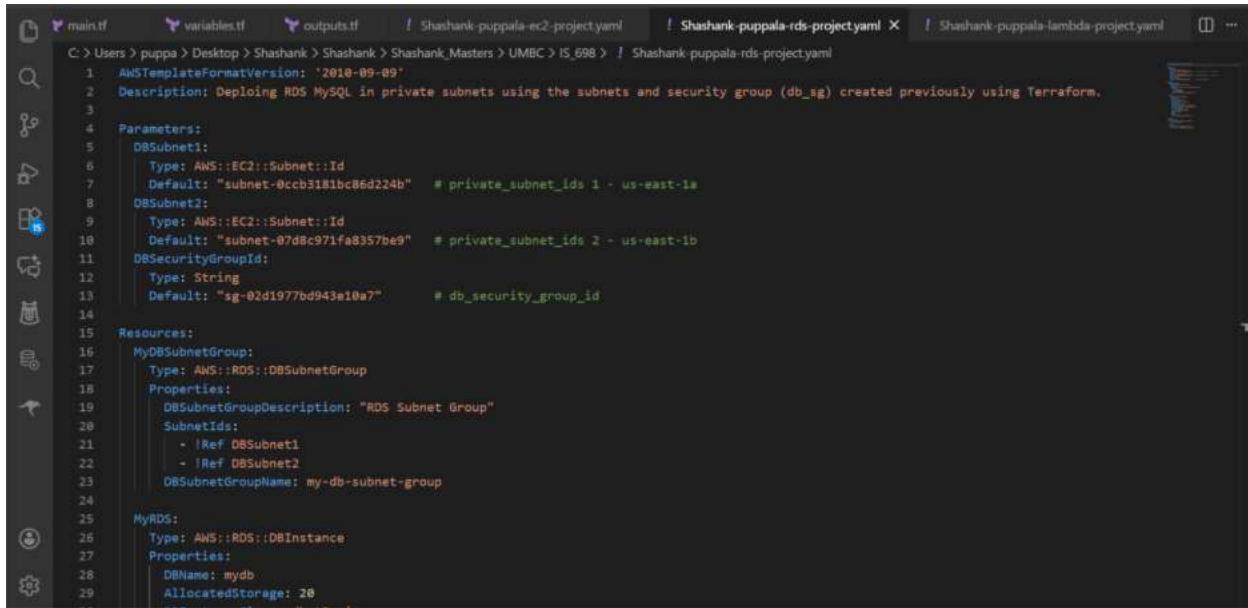
Deploy the stack from the CLI using the following command -

```
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): -



The screenshot shows a code editor with multiple tabs open. The active tab is 'Shashank-puppala-rds-project.yaml'. The code is as follows:

```
C:\ > Users > puppa > Desktop > Shashank > Shashank_Masters > UMBC > IS_698 > Shashank-puppala-rds-project.yaml
1 AWS::TemplateFormatVersion: '2018-09-19'
2 Description: Deploying RDS MySQL in private subnets using the subnets and security group (db_sg) created previously using Terraform.
3
4 Parameters:
5   DBSubnet1:
6     Type: AWS::EC2::Subnet::Id
7     Default: "subnet-0ccb3181bc86d224b" # private_subnet_ids 1 - us-east-1a
8   DBSubnet2:
9     Type: AWS::EC2::Subnet::Id
10    Default: "subnet-97d8c971fa8357be9" # private_subnet_ids 2 - us-east-1b
11   DBSecurityGroupId:
12     Type: String
13     Default: "sg-02d1977bd943e10a7" # db_security_group_id
14
15 Resources:
16   MyDBSubnetGroup:
17     Type: AWS::RDS::DBSubnetGroup
18     Properties:
19       DBSubnetGroupDescription: "RDS Subnet Group"
20       SubnetIds:
21         - !Ref DBSubnet1
22         - !Ref DBSubnet2
23       DBSubnetGroupName: my-db-subnet-group
24
25   MyRDS:
26     Type: AWS::RDS::DBInstance
27     Properties:
28       DBName: mydb
29       AllocatedStorage: 20
```

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)

```

main.tf          variables.tf      outputs.tf      Shashank-puppala-ec2-project.yaml      Shashank-puppala-efs-project.yaml      Shashank-puppala-lambda-project.yaml
C: > Users > puppa > Desktop > Shaishank > Shashank > Shashank_Masters > UMBC > IS_698 > Shashank-puppala-lambda-project.yaml
1   AWS::TemplateFormatVersion: '2010-09-09'
2   Description: Deploying Lambda function with basic execution role
3
4   Resources:
5     LambdaExecutionRole:
6       Type: AWS::IAM::Role
7       Properties:
8         RoleName: lambda-execution-role
9         AssumeRolePolicyDocument:
10        Version: '2012-10-17'
11        Statement:
12          - Effect: Allow
13            Principal:
14              Service: [lambda.amazonaws.com]
15            Action: ['sts:AssumeRole']
16        ManagedPolicyArns:
17          - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
18
19   MyLambda:
20     Type: AWS::Lambda::Function
21     Properties:
22       FunctionName: MyHelloLambda
23       Handler: index.handler
24       Role: !GetAtt LambdaExecutionRole.Arn
25       Code:
26         ZipFile: |
27           def handler(event, context):
28             return {
29               'statusCode': 200,
30             }
31
32
33
34
35
36
37
38
39
39

```

In 19, Col 12 Spaces: 2 UTF-8 CRLF { } 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;

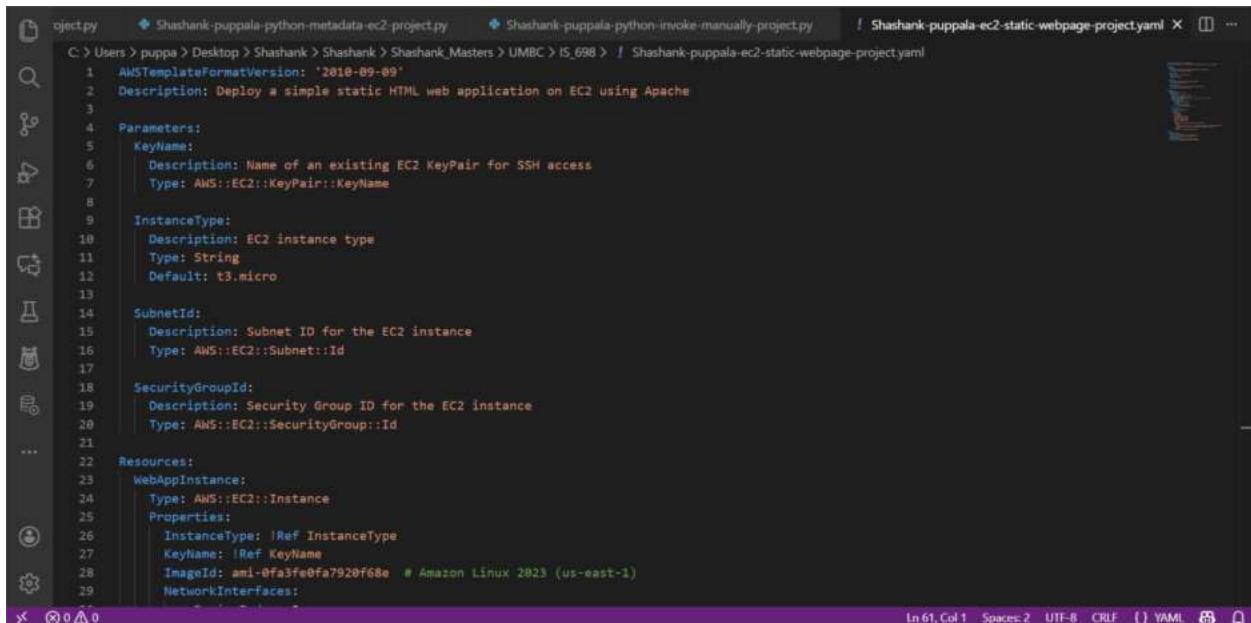
Stack name	Status	Created time	Description
shashank-puppala-lambda-stack	CREATE_COMPLETE	2025-12-05 17:45:41 UTC-0500	Deploying Lambda function with basic execution role
shashank-puppala-efs-stack	CREATE_COMPLETE	2025-12-05 17:19:46 UTC-0500	Deploying RDS MySQL in private subnets using the subnets and security group (db_sg) created previously using Terraform.
shashank-puppala-ec2-stack	CREATE_COMPLETE	2025-12-05 16:55:30 UTC-0500	Deploying EC2 instance with and HTML page using the existing VPC, Subnets, and Security groups created earlier with the help of Terraform.

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): -



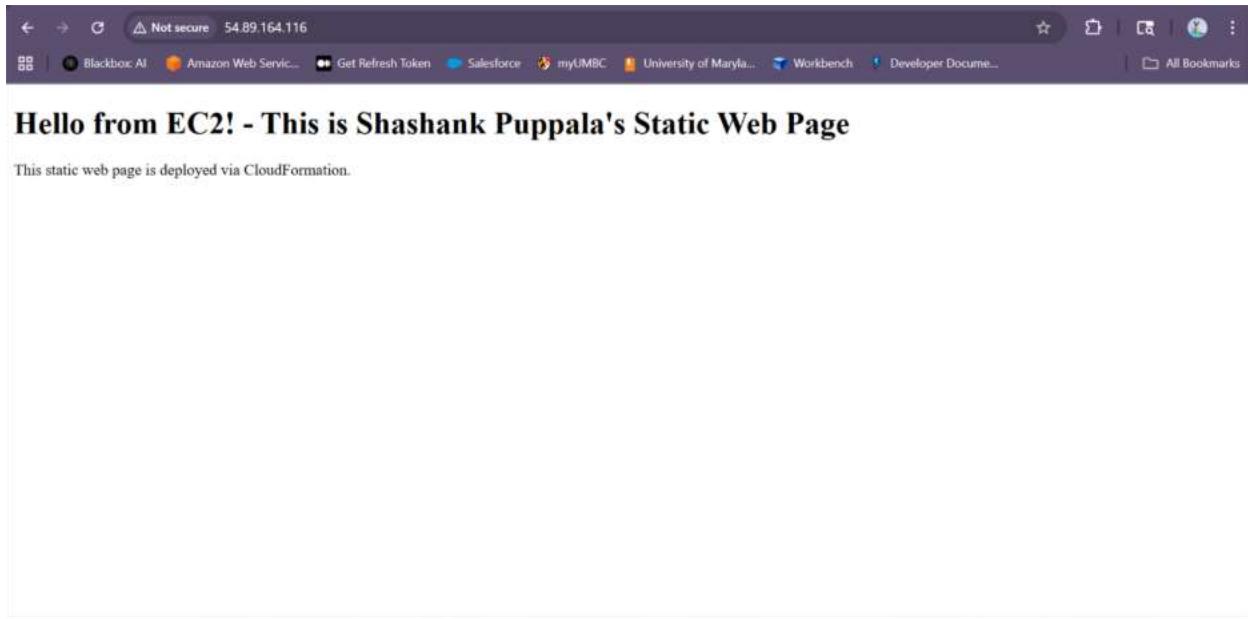
```
object.py  Shashank-puppala-python-metadata-ec2-project.py  Shashank-puppala-python-invoke-manually-project.py  Shashank-puppala-ec2-static-webpage-project.yaml ...  
C: > Users > puppa > Desktop > Shashank > Shashank_Masters > UMBC > IS_698 > Shashank-puppala-ec2-static-webpage-project.yaml  
1 AWSTemplateFormatVersion: '2010-09-09'  
2 Description: Deploy a simple static HTML web application on EC2 using Apache  
3  
4 Parameters:  
5   KeyName:  
6     Description: Name of an existing EC2 KeyPair for SSH access  
7     Type: AWS::EC2::KeyPair::KeyName  
8  
9   InstanceType:  
10    Description: EC2 instance type  
11    Type: String  
12    Default: t3.micro  
13  
14   SubnetId:  
15     Description: Subnet ID for the EC2 instance  
16     Type: AWS::EC2::Subnet::Id  
17  
18   SecurityGroupId:  
19     Description: Security Group ID for the EC2 instance  
20     Type: AWS::EC2::SecurityGroup::Id  
21  
...  
22 Resources:  
23   WebAppinstance:  
24     Type: AWS::EC2::Instance  
25     Properties:  
26       InstanceType: !Ref InstanceType  
27       KeyName: !Ref KeyName  
28       ImageId: ami-0fe3fe0fa7920f68e # Amazon Linux 2023 (us-east-1)  
29       NetworkInterfaces:  
Ln 61, Col 1  Spaces:2  UTF-8  CRLF  { } 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;

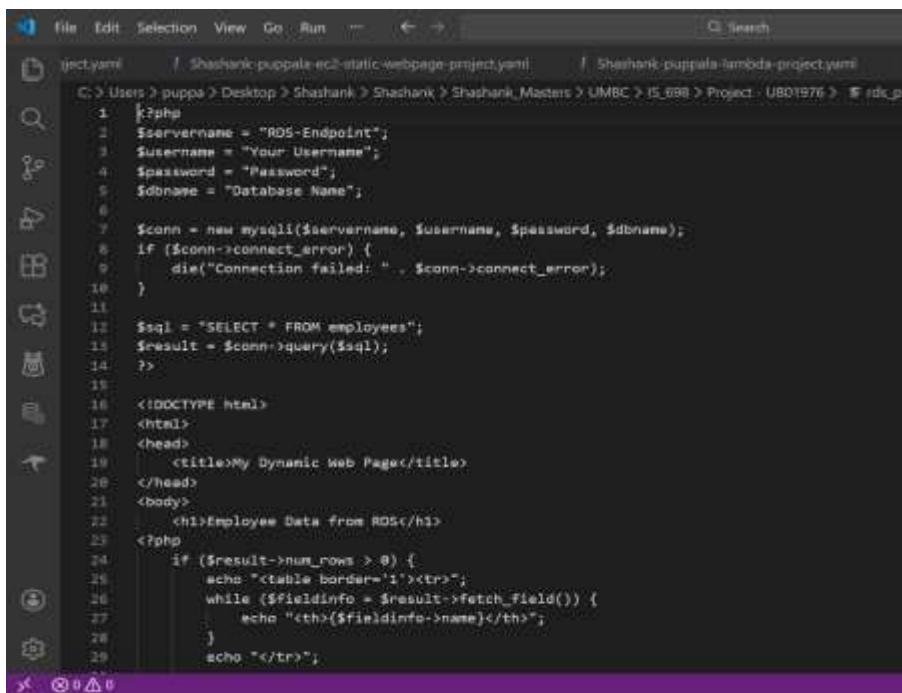
Change to the database which you have created in the step 2 using CloudFormation;

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-



```
File Edit Selection View Go Run ... ← → ⌘ Search
project.yaml    / Shashank_puppalapu2-static-webpage-project.yaml    | Shashank_puppalapu-lambda-project.yaml
C:\Users\2>puppa > Desktop > Shashank > Shashank_Masters > UMBC > IS_698 > Project - UBO1976 > rds.php
1 <?php
2 $servername = "RDS-Endpoint";
3 $username = "Your Username";
4 $password = "Password";
5 $dbname = "Database Name";
6
7 $conn = new mysqli($servername, $username, $password, $dbname);
8 if ($conn->connect_error) {
9     die("Connection failed: " . $conn->connect_error);
10 }
11
12 $sql = "SELECT * FROM employees";
13 $result = $conn->query($sql);
14 ?
15
16 <!DOCTYPE html>
17 <html>
18 <head>
19     <title>My Dynamic Web Page</title>
20 </head>
21 <body>
22     <h1>Employee Data from RDS</h1>
23 </php>
24     if ($result->num_rows > 0) {
25         echo "<table border='1'><tr>";
26         while ($fieldinfo = $result->fetch_field()) {
27             echo "<th>{$fieldinfo->name}</th>";
28         }
29         echo "</tr>";
```

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;

The screenshot shows a web browser window with the address bar displaying 'Not secure 54.89.164.116'. The page title is 'Employee Data from RDS'. Below the title is a table with four columns: Id, Name, Email, and Department. The single data row has values: Id=3, Name=Shashank, Email=shank123@gmail.com, and Department=IS.

Id	Name	Email	Department
3	Shashank	shank123@gmail.com	IS

4. Implement autoscaling for EC2 instances.

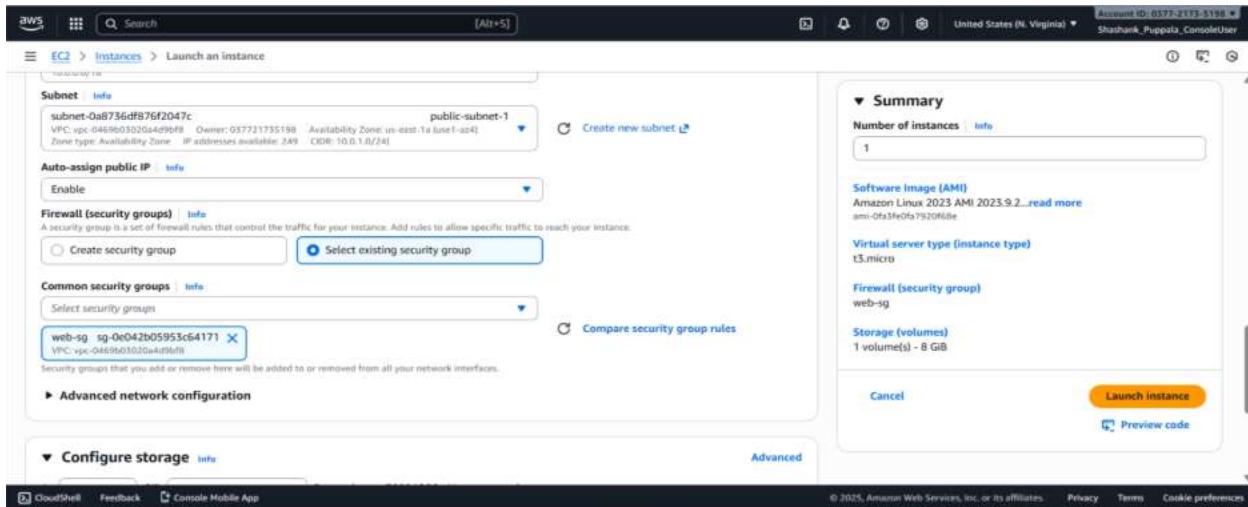
Next, to implement autoscaling we need to first create the AMI Image if it's not existing, design a launch template, create the ALB and target groups to handle the traffic, and then finally ASG to launch the instances based on the CPU utilization for application;

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.”

The screenshot shows the AWS EC2 AMIs page. In the left sidebar, under 'Images', 'AMIs' is selected. The main content area displays the 'Amazon Machine Images (AMIs) (1/1)' section. A single row is listed: 'Autoscaling-ami' with AMI ID 'ami-09056dd9ceca6bafdf'. The status is 'Available'. Below this, a detailed view for 'AMI ID: ami-09056dd9ceca6bafdf' is shown with tabs for Details, Permissions, Storage, My AMI usage, AMI ancestry - new, and Tags. The 'Details' tab shows the following information:

AMI ID	ami-09056dd9ceca6bafdf	Image type	machine	Platform details	Linux/UNIX	Root device type	EBS
AMI name	Autoscaling-ami	Owner account ID	037721735198	Architecture	x86_64	Usage operation	RunInstances
Root device name	/dev/xvda	Status	Available	Source	037721735198/Autoscaling-ami	Virtualization type	hvm

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.

The screenshot shows the AWS EC2 Launch Templates page. In the left sidebar, under 'Launch Templates', 'Launch templates' is selected. The main content area displays the 'shashank-puppala-autoscaling-launch-template (lt-086d61fe7280ee361)' page. The 'Launch template details' section includes:

- Launch template ID: lt-086d61fe7280ee361
- Launch template name: shashank-puppala-autoscaling-launch-template
- Default version: 1
- Owner: arnaws:iam:037721735198:user/Shashank_Puppala_ConsoleUser

The 'Launch template version details' section shows:

Version	Description	Date created	Created by
1 (Default)	-	2025-12-06T03:10:10.000Z	arnaws:iam:037721735198:user/Shashank_Puppala_ConsoleUser

Below this, the 'Instance details' section lists:

AMI ID	ami-09056dd9ceca6bafdf	Instance type	t3.micro	Availability Zone	-	Availability Zone Id	-
Key pair name	shashank-puppala-linux-pair-1	Security groups	-	Security group IDs	sg-0e042b05953c64171	Availability Zone	-

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.

The screenshot shows the AWS EC2 Target Groups console. On the left, there's a navigation sidebar with sections like Volumes, Snapshots, Lifecycle Manager, Network & Security (Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces), Load Balancing (Load Balancers, Target Groups, Trust Stores), Auto Scaling (Auto Scaling Groups), and Settings. The main area is titled "Web-Apps-Target-Group". It displays the following details:

Target type	Protocol : Port	Protocol version
Instance	HTTP: 80	HTTP1

IP address type: IPv4, Load balancer: None associated.

Statistics table:

Total targets	Healthy	Unhealthy	Unused	Initial	Draining
0	0	0	0	0	0

0 Anomalous.

Registered targets (0) info: Anomaly mitigation: Not applicable. Buttons: Register targets, Dereister.

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.

The screenshot shows the AWS CloudWatch Metrics Metrics Insights page. The main title is "CloudWatch Metrics Metrics Insights". Below it, the metric is specified as "aws_lambda_function_invocation". The chart area shows a single data series named "aws_lambda_function_invocation" with a blue line graph. The graph shows a sharp increase in the number of invocations starting around December 5, 2025, peaking at approximately 1000 invocations. The X-axis is labeled "Time" and shows dates from "Nov 2025" to "Jan 2026". The Y-axis is labeled "Value" and ranges from 0 to 1000. The chart has a grid and a legend.

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

The screenshot shows a web browser window with the URL "shashank-puppala-web-apps-alb-537850753.us-east-1.elb.amazonaws.com". The page content is a simple text message: "This instance is part of EC2 Auto Scaling! - Shashank puppala". The browser interface includes a back button, forward button, search bar, and various tabs and icons at the top.

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.

The screenshot shows the AWS EC2 Auto Scaling Groups console. The left sidebar has sections for Volumes, Snapshots, Lifecycle Manager, Network & Security (Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces), Load Balancing (Load Balancers, Target Groups, Trust Stores), and Auto Scaling (Auto Scaling Groups, Settings). The main content area is titled "Shashank-puppala-web-apps-autoscaling-gp Capacity overview". It shows the following details:

Desired capacity	Scaling limits (Min - Max)	Desired capacity type	Status
1	1 - 3	Units (number of instances)	-

Date created: Fri Dec 05 2025 22:40:55 GMT-0500 (Eastern Standard Time)

Below this, there are tabs for Details, Integrations, Automatic scaling, Instance management, Instance refresh, Activity, Monitoring, and Tags - moved. The Details tab is selected. Under "Launch template", it shows:

Launch template	AMI ID	Instance type	Owner
lt-086d61fe7280ee361 shashank-puppala-autoscaling-launch-template	ami-09056dd9ceca6bafd	t3.micro	arn:aws:iam::037721735198:user/Shashank_Puppala_ConsoleUser

Version: Default, Security groups, Security group IDs, Create time: Fri Dec 05 2025 22:40:55 GMT-0500 (EST) (2025-12-05T22:40:55Z).

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.

The screenshot shows the AWS EC2 Auto Scaling Groups console. The left sidebar has sections for Volumes, Snapshots, Lifecycle Manager, Network & Security (Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces), Load Balancing (Load Balancers, Target Groups, Trust Stores), and Auto Scaling (Auto Scaling Groups, Settings). The main content area is titled "Create dynamic scaling policy" for the "Shashank-puppala-web-apps-autoscaling-gp" group. The dialog box contains the following configuration:

Target Tracking Policy

- Policy type: Target tracking using
- Enabled
- Execute policy when: As required to maintain Average CPU utilization at 20
- Take the action: Add or remove capacity units as required
- Instances need: 100 seconds to warm up before including in metric
- Scale in: Enabled

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:

Activity history (3)						
Status		Description	Cause	Start time	End time	
Successful	Launching a new EC2 instance -> 0aef1f0e0f8bbf1	At 2025-12-06T03:00:58Z a monitor alarm TargetTracking-MainCloud-pipeline-web-app-autoscaling-pp-MetricHigh-1-failed-0079-de1f-f0d3-cfbfa0dd02, in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 3. At 2025-12-06T03:00:06Z, an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 3.		2025 December 06, 12:00:09 AM -05:00	2025 December 06, 12:05:14 AM -05:00	
Successful	Launching a new EC2 instance -> 0aef1f0e0f8bbf1	At 2025-12-06T03:00:58Z a monitor alarm TargetTracking-MainCloud-pipeline-web-app-autoscaling-pp-MetricHigh-1-failed-0079-de1f-f0d3-cfbfa0dd02, in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 3. At 2025-12-06T03:00:06Z, an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 3.		2025 December 06, 12:00:09 AM -05:00	2025 December 06, 12:05:15 AM -05:00	
Successful	Launching a new EC2 instance -> 0aef1f0e0f8bbf1	At 2025-12-06T03:00:58Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 1. At 2025-12-06T03:02:00Z, an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 1.		2025 December 06, 10:42:02 AM -05:00	2025 December 06, 10:42:07 PM -05:00	

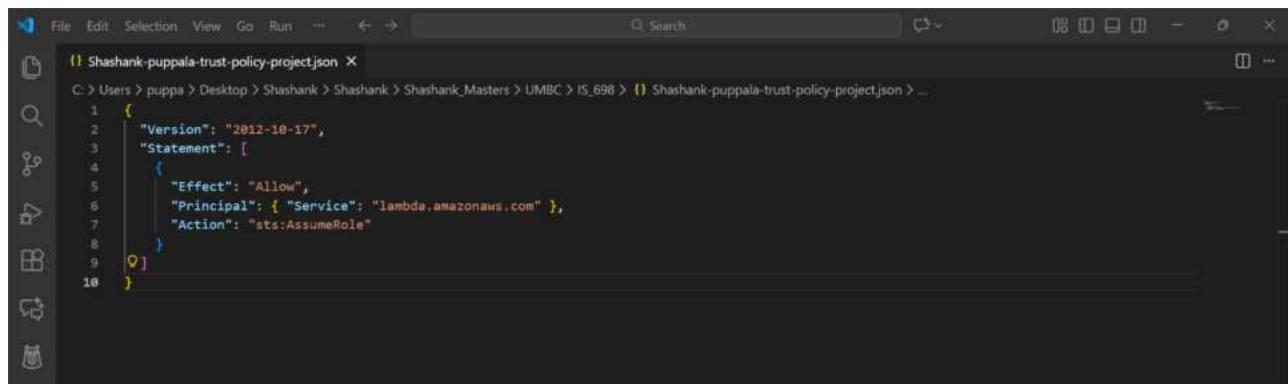
If the Stress is reduced it will automatically scale down the instance:

Activity history					
Status	Description	Cause	Start time	End time	
Successful	Terminating EC2 instance: i-0bcfc911d0cfb5e3d	At 2025-12-06T05:17:31Z a monitor alarm TargetTracking-Shashank-puppala-web-apps-autoscaling-gp-AlarmLow-b8e774cc-7e95-4bfa-9b47-bc1d5edc070a in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-12-06T05:17:36Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 2 to 1. At 2025-12-06T05:17:56Z instance i-0bcfc911d0cfb5e3d was selected for termination.	2025 December 06, 12:17:36 AM -05:00	2025 December 06, 12:23:39 AM -05:00	
Successful	Terminating EC2 instance: i-0a5d7bfc0cef34e73	At 2025-12-06T05:16:31Z a monitor alarm TargetTracking-Shashank-puppala-web-apps-autoscaling-gp-AlarmLow-b8e774cc-7e95-4bfa-9b47-bc1d5edc070a in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 3 to 2. At 2025-12-06T05:16:34Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2025-12-06T05:16:34Z instance i-0a5d7bfc0cef34e73 was selected for termination.	2025 December 06, 12:16:34 AM -05:00	2025 December 06, 12:22:57 AM -05:00	
Successful	Launching a new EC2 instance: i-0bda15a0a934bb81	At 2025-12-06T04:59:58Z a monitor alarm TargetTracking-Shashank-puppala-web-apps-autoscaling-gp-AlarmHigh-156bd169-8878-4e19-9ab2-c5f8a668df2c in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 3. At 2025-12-06T05:00:08Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 3.	2025 December 06, 12:00:09 AM -05:00	2025 December 06, 12:05:14 AM -05:00	
		At 2025-12-06T04:59:58Z a monitor alarm TargetTracking-Shashank-puppala-web-apps-autoscaling-gp-AlarmHigh-156bd169-8878-4e19-9ab2-c5f8a668df2c in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 3. At 2025-12-06T05:00:08Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 3.			

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;



```

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

```

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

```
Windows PowerShell
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMB\IS_698> aws iam create-role --role-name ProjectS3LambdaExecutionRole --assume-role-policy-document file://Shashank-puppala-trust-policy-project.json
{
    "Role": {
        "Path": "/",
        "RoleName": "ProjectS3LambdaExecutionRole",
        "RoleId": "AROAQRSDAAPNPBTPL3HR",
        "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_Masters\UMB\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

```
Windows PowerShell
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMB\IS_698> 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_Masters\UMB\IS_698> aws iam attach-role-policy --role-name ProjectS3LambdaExecutionRole --policy-arn arn:aws:iam:aws:policy/AmazonS3ReadOnlyAccess
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMB\IS_698> |
```

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

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface with the following details:

- Title Bar:** The title bar displays the file path "Shashank-puppala-trust-policy-project.json" and the active file "Shashank-puppala-s3-upload-logger-project.py".
- Status Bar:** The status bar at the bottom right indicates "Spaces: 4" and "UTF-8" encoding.
- Code Editor:** The main area contains Python code for a Lambda function. The code imports json, boto3, urllib.parse, and logging, and defines a lambda_handler function that processes S3 event records to log object metadata (bucket, key, size) and head object details if the object exists.
- Sidebar:** On the left, there is a sidebar with various icons representing different features like file operations, search, and help.
- Activity Bar:** At the bottom, there is an activity bar with icons for file operations, search, and help.

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)

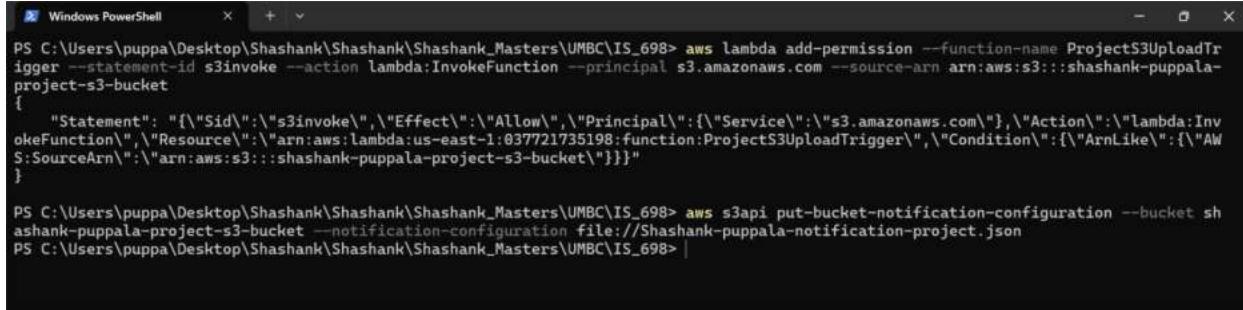
2. Configure an S3 event trigger for Lambda.

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)

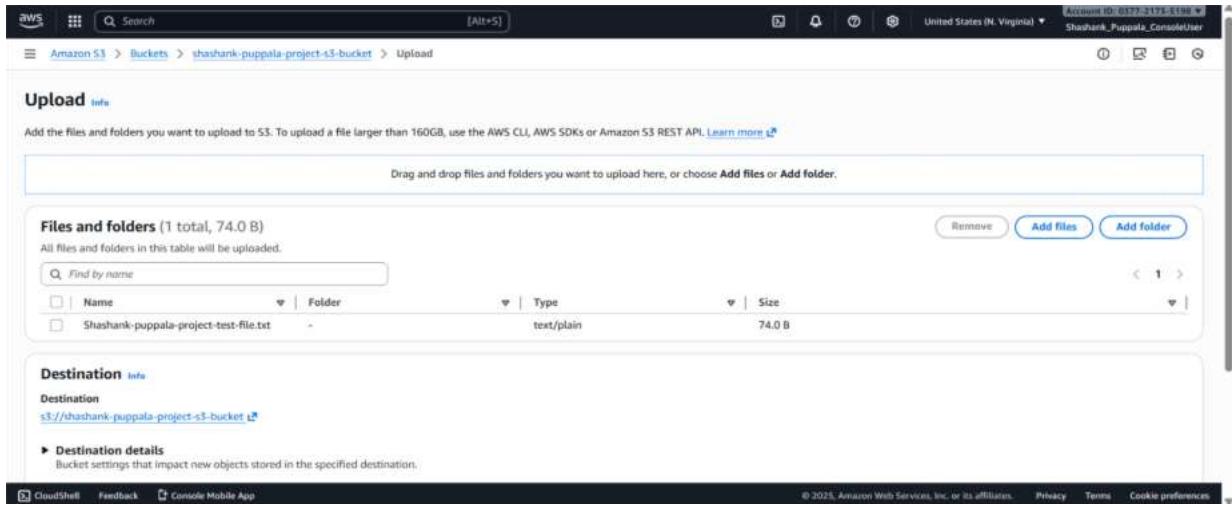
```
aws s3api put-bucket-notification-configuration --bucket shashank-puppala-project-s3-bucket --notification-configuration file://Shashank-puppala-notification-project.json
```



```
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> 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
{
  "Statement": "{\"Sid\":\"s3invoke\",\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"s3.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":\"arn:aws:lambda:us-east-1:037721735198:function:ProjectS3UploadTrigger\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:s3:::shashank-puppala-project-s3-bucket\"}}}"
}

PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> aws s3api put-bucket-notification-configuration --bucket shashank-puppala-project-s3-bucket --notification-configuration file://Shashank-puppala-notification-project.json
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698>
```

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.

The screenshot shows the AWS CloudWatch Log Management interface. The left sidebar has a tree view with 'Logs' expanded, showing 'Log Management' and 'Logs Insights'. The main area is titled 'Log events' and contains a table with two columns: 'Timestamp' and 'Message'. The table lists several log entries, including one from 2025-12-06T16:49:25.185Z which describes a file upload to an S3 bucket.

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/

```
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> aws s3 mb s3://shashank-puppala-project-s3-bucket-2 --region us-east-1
make_bucket: shashank-puppala-project-s3-bucket-2
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> aws s3 cp shashank-puppala-testing-file-3.txt s3://shashank-puppala-project-s3-bucket-2/
upload: .\shashank-puppala-testing-file-3.txt to s3://shashank-puppala-project-s3-bucket-2/shashank-puppala-testing-file-3.txt
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698>
```

2. Retrieve EC2 metadata.

(Can be only retrieved from SSH into EC2 instance)

```
[ec2-user@ip-10-0-1-38:~]
[ec2-user@ip-10-0-1-38 ~]$ python3 shashank_project_ec2_metadata.py
ami-id: ami-0fa3fe0fa7920f68e
instance-id: i-0ca785c90daabf6f1
instance-type: t3.micro
hostname: ip-10-0-1-38.ec2.internal
local-ipv4: 10.0.1.38
public-ipv4: 3.92.84.108
placement/availability-zone: us-east-1a
security-groups: web-sg
[ec2-user@ip-10-0-1-38 ~]$
```

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

```

update-ec2-static webpage-project.pyw      Shashank-puppala-lambda-project.pyw      Shashank-puppala-ec2-project.pyw      mls-project.php      retrieve_metadata.txt - X
C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> Project - UBD1976 > retrieve_metadata.pyw

1 import requests
2
3 # IAMSV2 token
4 token = requests.get(
5     "http://169.254.169.254/latest/api/token",
6     headers={"X-aws-ec2-metadata-token-ttl-seconds": "21600"}
7 ).text
8
9 def get_metadata(key):
10     url = f"http://169.254.169.254/latest/meta-data/{key}"
11     response = requests.get(url, headers={"X-aws-ec2-metadata-token": token})
12     return response.text
13
14 metadata_keys = [
15     "ami-id",
16     "instance-id",
17     "instance-type",
18     "hostname",
19     "local-ipv4",
20     "public-ipv4",
21     "placement/availability-zone",
22     "security-groups"
23 ]
24
25 metadata = {}
26 for key in metadata_keys:
27     metadata[key] = get_metadata(key)
28
29 for k, v in metadata.items():
30     print(f'{k}: {v}')

```

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

```

DescribeInstances						
AZ	ID	Name	PrivateIP	PublicIP	Type	
us-east-1a	i-0ca785c90daabf6f1	shashank-puppala-EC2-instance-project	10.0.1.38	3.92.84.188	t3.micro	
us-east-1a	i-0459f28e28887f1a8	WebAppInstance	10.0.1.81	3.84.243.26	t3.micro	
us-east-1a	i-0826b9ef30384211a	Shashank-puppala-ec2-scaling-instance	10.0.1.132	13.220.95.72	t3.micro	

4. Invoke Lambda manually.

```

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

```

```
Command Prompt

C:\Users\puppa>aws lambda invoke --function-name MyHelloLambda response.json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}

C:\Users\puppa>type response.json
{"statusCode": 200, "body": "Hello from Lambda - Shashank Puppala!"}
C:\Users\puppa>
```

Python Boto 3 scripts to perform the operations;

1. Create an S3 bucket and upload a file.

The screenshot shows a Microsoft Visual Studio Code interface. The top bar includes 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', and other standard options. A search bar is located at the top right. The left sidebar contains icons for file operations like Open, Save, Find, and others. The main editor area displays a Python script named 'Shashank-puppala-python-s3-create-upload-project.py'. The code creates an S3 bucket and uploads a sample text file. The terminal below shows the command run and its output, confirming the bucket creation and file upload.

```
Shashank-puppala-python-running-ec2-instances-project.py Shashank-puppala-python-s3-create-upload-project.py X
C:\Users>puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> Shashank-puppala-python-s3-create-upload-project.py ...
1 import boto3
2 import uuid
3
4 s3 = boto3.client('s3')
5
6 bucket_name = f"shashank-puppala-test-project-bucket-{uuid.uuid4().hex[:6]}"
7 region = "us-east-1"
8
9 s3.create_bucket(Bucket=bucket_name)
10
11 print("Bucket created:", bucket_name)
12
13 # Uploading a sample text file
14 file_name = "sample.txt"
15 with open(file_name, "w") as f:
```

PROBLEMS OUTPUT TERMINAL PORTS SQL HISTORY TASK MONITOR DEBUG CONSOLE

powerhell +

```
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> Python Shashank-puppala-python-s3-create-upload-project.py
Bucket created: shashank-puppala-test-project-bucket-2c30d8
File uploaded: sample.txt
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698>
```

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)

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. On the left is a sidebar with icons for file operations like Open, Save, Find, and Delete. The main area has three tabs at the top: 'Shashank-puppala-python-running-ec2-instances-project.py', 'Shashank-puppala-python-s3-create-upload-project.py', and 'Shashank-puppala-python-metadata-ec2-project.py'. The bottom tab bar includes 'PROBLEMS', 'OUTPUT', 'TERMINAL', 'PORTS', 'SQL HISTORY', 'TASK MONITOR', and 'DEBUG CONSOLE'. The 'TERMINAL' tab is active, showing a PowerShell prompt and the output of running the Python script:

```
C:\> Users > puppa > Desktop > Shashank > Shashank > Shashank_Masters > UMBC > IS_698 > Shashank-puppala-python-metadata-ec2-project.py
PS C:\> C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> Python Shashank-puppala-python-metadata-ec2-project.py
Instance ID: i-0ca785c90daabf6f1
Type: t3.micro
State: running
AZ: us-east-1a
Private IP: 10.0.1.38
Public IP: 3.92.84.188
Name: shashank-puppala-EC2-instance-project
PS C:\> C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698>
```

3. List running EC2 instances.

This screenshot shows the same VS Code environment as the previous one, but the code in the editor has been modified to filter for running EC2 instances. The terminal output remains the same, showing the details of the single running instance.

```
C:\> Users > puppa > Desktop > Shashank > Shashank > Shashank_Masters > UMBC > IS_698 > Shashank-puppala-python-running-ec2-instances-project.py > ...
1 import boto3
2
3 ec2 = boto3.client('ec2')
4
5 response = ec2.describe_instances(
6     Filters=[
7         {"Name": "instance-state-name", "Values": ["running", "stopped", "pending", "stopping"]}
8     ]
9 )
10
11 if not response["Reservations"]:
12     print("No EC2 instances found.")
13 else:
14     print("EC2 Instances:\n")
15     for reservation in response["Reservations"]:
16         for instance in reservation["Instances"]:
17             # Get Name tag if exists
18             name = "N/A"
19             if "Tags" in instance:
20                 for tag in instance["Tags"]:
21                     if tag["Key"] == "Name":
22                         name = tag["Value"]
23             print("Name:", name)
24             print("Instance ID:", instance["InstanceId"])
25             print("State:", instance["State"]["Name"])

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

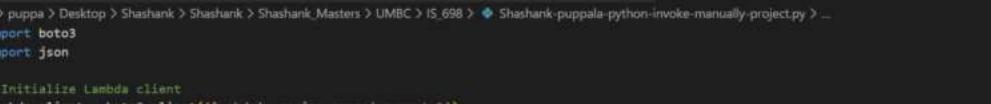
```
Shashank-puppala-python-running-ec2-instances-project.py
C:\Users\puppa\Desktop\Shashank\Shashank\Masters\UMBC\IS_698> Shashank-puppala-python-running-ec2-instances-project.py > ...
15     for reservation in response['Reservations']:
PROBLEMS OUTPUT TERMINAL PORTS SQL HISTORY TASK MONITOR DEBUG CONSOLE
PS C:\Users\puppa\Desktop\Shashank\Shashank\Masters\UMBC\IS_698> python Shashank-puppala-python-running-ec2-instances-project.py
EC2 Instances:
Name: shashank-puppala-EC2-instance-project
Instance ID: i-8ca785c90daabf6f1
State: running
Type: t3.micro
AZ: us-east-1a
Private IP: 10.0.1.88
Public IP: 3.92.84.188

Name: WebAppInstance
Instance ID: i-0459f28e28887f1a8
State: running
Type: t3.micro
AZ: us-east-1a
Private IP: 10.0.1.81
Public IP: 3.84.243.26

Name: Shashank-puppala-ec2-scaling-instance
Instance ID: i-0826b9ef30584211a
State: running
Type: t3.micro
AZ: us-east-1a
Private IP: 10.0.1.132
Public IP: 13.220.95.72

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

4. Invoke Lambda Function manually



The screenshot shows a Windows desktop environment with a taskbar at the bottom. The taskbar icons include File Explorer, Task View, Start, Task Manager, and several pinned applications. The main window is a Microsoft Visual Studio Code editor. The title bar says "File Edit Selection View Go Run ...". The search bar contains "Search". The code editor has tabs for "Shashank-puppala-python-s3-create-upload-project.py", "Shashank-puppala-python-metadata-ec2-project.py", and "Shashank-puppala-python-invoker-manually-project.py X". The current file is "Shashank-puppala-python-invoker-manually-project.py". The code itself is as follows:

```
1 import boto3
2 import json
3
4 # Initialize Lambda client
5 lambda_client = boto3.client('lambda', region_name='us-east-1')
6
7 # Name of your Lambda function
8 function_name = "MyHelloLambda"
9
10 # Optional: Payload to send to Lambda
11 payload = {
```

Below the code editor, there are tabs for PROBLEMS, OUTPUT, TERMINAL, PORTS, SQL HISTORY, TASK MONITOR, and DEBUG CONSOLE. The TERMINAL tab is selected, showing command-line output:

```
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698> Python Shashank-puppala-python-invoker-manually-project.py
Lambda Response: {'statusCode': 200, 'body': 'Hello from Lambda - Shashank Puppala!'}
PS C:\Users\puppa\Desktop\Shashank\Shashank_Masters\UMBC\IS_698>
```

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.

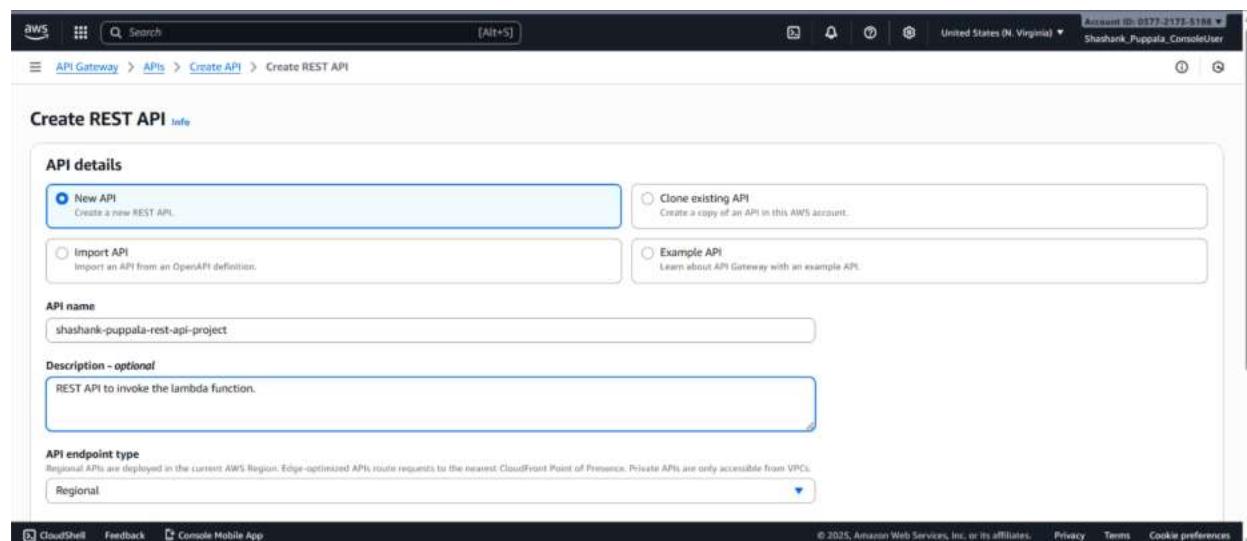
Future Work: -

- 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

The screenshot shows the AWS API Gateway Stages page. On the left, there's a sidebar with 'APIs' and 'Stages' sections. Under 'Stages', it shows an expanded 'prod' stage with a single resource '/hello' and a 'GET' method. To the right, there's a 'Method overrides' section with a note that the method inherits settings from the 'prod' stage. Below that is an 'Invoke URL' field containing the URL: <https://mal2tfg8cf.execute-api.us-east-1.amazonaws.com/prod/hello>. At the top right, there are 'Stage actions' and 'Create stage' buttons.

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

The screenshot shows a web browser window with the URL <https://mal2tfg8cf.execute-api.us-east-1.amazonaws.com/prod/hello>. The response is displayed in a JSON format under 'Pretty-print' mode. The response body is:

```
{
  "statusCode": 200,
  "body": "Hello from Lambda - Shashank Puppala!"
}
```