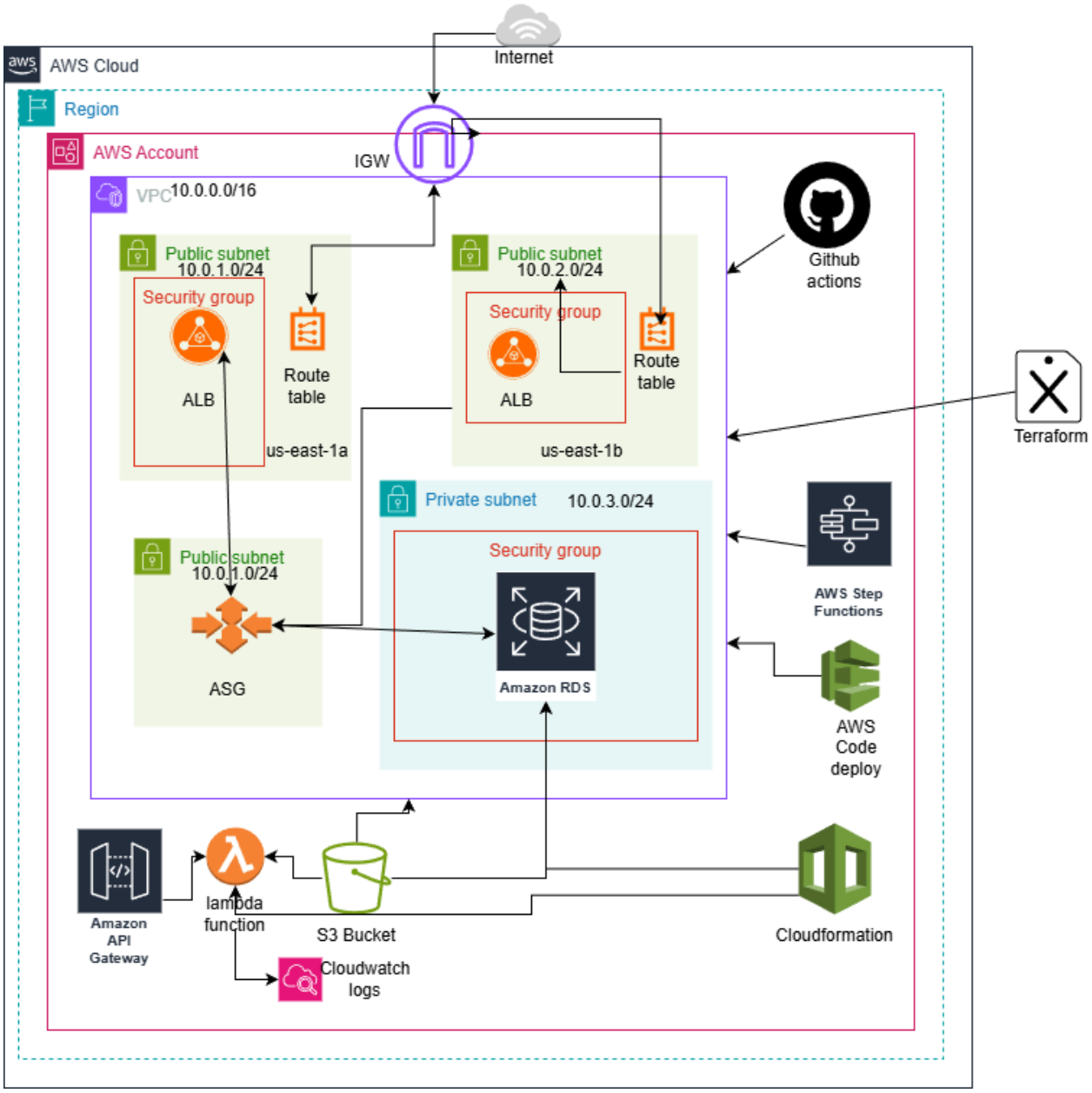
IS698 - Final Project

Deploying a Scalable AWS Architecture with Infrastructure as Code

Submitted By-
Sai Sumanth Reddy Kachi
Campus ID: DC93457

Project Requirements

1. Architecture Design



With this architecture, various AWS capabilities come together to form a system that is both reliable and easy to scale. It all starts with the construction of the VPC. In this instance, Terraform is employed to implement this layer so that consistency is achieved each time it is created. In the structure created by the construction of the VPC, subnets and internet gateways exist within the network design for enhanced organization.

The public subnet handles the web-facing part. An Auto Scaling Group launches EC2 instances running the application, and then an Application Load Balancer is placed on top to manage traffic. These components function together to ensure that the application is still responsive even with increased traffic. Security groups are layered around each element to filter incoming and outgoing traffic appropriately.

In the private subnet environment, the database on Amazon RDS is maintained out of reach from the internet and is allowed limited connectivity from the application layer. This provides a significant safeguard for the data.

It is further supplemented by event-driven processing with the use of an S3 bucket and a Lambda function. Every time a file is uploaded, Lambda records the events on AWS CloudWatch. AWS CloudFormation facilitates the deployment process for core elements, while services including GitHub Actions and CodeDeploy simplify the process for updates and automation.

A. Infrastructure Deployment

1. Terraform: Networking and Infrastructure

Terraform was used to provision the entire network layer of the architecture. This included the VPC, subnets, route tables, internet gateway and security groups. Using Infrastructure-as-Code ensured that networking resources were consistent and repeatable across deployments.

A Terraform configuration was created to define:

- VPC (10.0.0.0/16)
- Public subnets for EC2 and autoscaling
- Private subnet for RDS
- Internet Gateway for outbound access
- Route tables for public routing
- Security groups for EC2 and RDS traffic control

E: > Sumanth UMBC > IS698-Project > terraform-698 > main.tf

```
#########################
# Terraform and Provider
#########################

terraform {
  required_version = ">= 1.5.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

#########################
# VPC
#########################

resource "aws_vpc" "main" {
  cidr_block           = var.vpc_cidr
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name = "${var.project_name}-vpc"
  }
}

#########################
# Internet Gateway
#########################

```

main.tf

E: > Sumanth UMBC > IS698-Project > variables.tf

```
variable "aws_region" {
  description = "AWS region to deploy into"
  type        = string
  default     = "us-east-1"
}

variable "project_name" {
  description = "Base name for tagging resources"
  type        = string
  default     = "cloud-arch-project"
}

variable "vpc_cidr" {
  description = "CIDR block for the VPC"
  type        = string
  default     = "10.0.0.0/16"
}

variable "public_subnets" {
  description = "CIDR blocks for public subnets"
  type        = list(string)
  default     = ["10.0.1.0/24", "10.0.2.0/24"]
}

variable "private_subnets" {
  description = "CIDR blocks for private subnets"
  type        = list(string)
  default     = ["10.0.11.0/24", "10.0.12.0/24"]
}

variable "availability_zones" {
  description = "AZs for subnets"
  type        = list(string)
  default     = ["us-east-1a", "us-east-1b"]
}
```

variables.tf

E: > Sumanth UMBC > IS698-Project > terraform-698 > outputs.tf

```
output "vpc_id" {
  value       = aws_vpc.main.id
  description = "ID of the main VPC"
}

output "public_subnet_ids" {
  value       = [for s in aws_subnet.public : s.id]
  description = "IDs of public subnets"
}

output "private_subnet_ids" {
  value       = [for s in aws_subnet.private : s.id]
  description = "IDs of private subnets"
}

output "alb_sg_id" {
  value       = aws_security_group.alb_sg.id
  description = "ALB security group ID"
}

output "web_sg_id" {
  value       = aws_security_group.web_sg.id
  description = "Web server security group ID"
}

output "rds_sg_id" {
  value       = aws_security_group.rds_sg.id
  description = "RDS security group ID"
}
```

outputs.tf

Terraform files are then executed using the following commands:

terraform init
terraform plan
terraform apply

After terraform apply, the AWS console confirms that all networking components were created successfully.

2. Cloudformation: EC2 and RDS with Static Web Page Deployment

CloudFormation was used to automate the creation of EC2 instances and an RDS MySQL database.

1. Two YAML templates were created:

- **EC2 template** for launching the web server

- **RDS template** for provisioning the MySQL database

E: > Sumanth UMBC > IS698-Project > cloudformation > ! ec2-asg-alb.yaml

```yaml
Resources:
  WebServerSecurityGroup:

  WebServerInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds:
        - !Ref WebServerSecurityGroup
      Tags:
        - Key: Name
          Value: simple-ec2-public
      UserData:
        Fn::Base64: |
          #!/bin/bash
          yum update -y
          yum install -y httpd
          systemctl enable httpd
          systemctl start httpd

          cat > /var/www/html/index.html << 'EOF'
          <html>
          <head>
            <title>AWS Simple EC2</title>
          </head>
          <body>
            <h1>Hello from a simple EC2 instance!</h1>
            <p>This instance was created using a basic CloudFormation template
            in a public subnet.</p>
          </body>
          </html>
          EOF

Outputs:
```

Ec2-alb.yaml

E: > Sumanth UMBC > IS698-Project > cloudformation > ! rds-mysql.yaml

```yaml
Parameters:

Resources:
  DBSubnetGroup:
    Type: AWS::RDS::DBSubnetGroup
    Properties:
      DBSubnetGroupDescription: "Subnet group for RDS in private subnets"
      SubnetIds:
        - !Ref PrivateSubnet1Id
        - !Ref PrivateSubnet2Id
      # Name omitted so AWS generates a unique one

  MyDBInstance:
    Type: AWS::RDS::DBInstance
    Properties:
      # Let AWS pick a unique identifier automatically by omitting DBInstanceIdentifier
      AllocatedStorage: 20
      DBInstanceClass: db.t3.micro
      Engine: mysql
      DBName: !Ref DBName
      MasterUsername: !Ref DBUsername
      MasterUserPassword: !Ref DBPassword
      VPCSecurityGroups:
        - !Ref RdsSecurityGroupId
      DBSubnetGroupName: !Ref DBSubnetGroup
      MultiAZ: false
      PubliclyAccessible: false
      DeletionProtection: false
      BackupRetentionPeriod: 0

Outputs:
  DatabaseEndpoint:
    Description: RDS database endpoint address
    Value: !GetAtt MyDBInstance.Endpoint.Address
```

Rds-mysql.yaml

2. After creating these YAML files, stacks were launched through the AWS console by uploading the existing templates.

3. For the EC2 instance a static web page was deployed

4. For the RDS, a database was created with the specified characteristics.

3. Database Backend Configuration

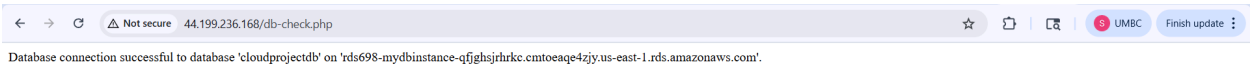
The EC2 server was configured to use the RDS MySQL database as its backend.

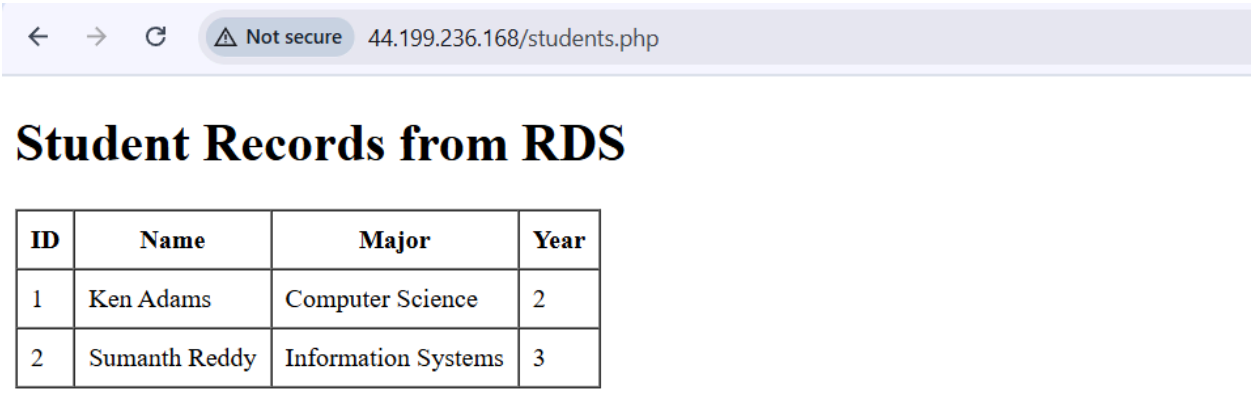
1. Connected to the EC2 instance via SSH using **putty.exe**
2. Installed Apache Web Servers with the following code:
   sudo yum install httpd -y
   sudo systemctl start httpd
   sudo systemctl enable httpd
3. Uploaded the index.html file to /var/www/html
4. Created the database script file (db-check.php)
   sudo nano /var/www/html/db-check.php

E: > Sumanth UMBC > IS698-Project > db_check.php

```php
<?php
error_reporting(E_ALL);
ini_set('display_errors', 1);

$db_host = 'rds698-mydbinstance-qfjghsjrhrkc.cmtoeaqe4zjy.us-east-1.rds.amazonaws.com';
$db_name = 'cloudprojectdb';
$db_user = 'admin';
$db_pass = '*********';

$conn = new mysqli($db_host, $db_user, $db_pass, $db_name);

if ($conn->connect_error) {
    echo "Database connection failed: " . $conn->connect_error;
} else {
    echo "Database connection successful to database '$db_name' on '$db_host'.";
}

$conn->close();
?>
```

5. Inserted RDS endpoint, DB name, DB username and DB password to the script.
6. After checking the databse connection, the Students data is added and a new students.php file is created.
7. After testing the script, the browser confirmed that EC2 and RDS are connected.

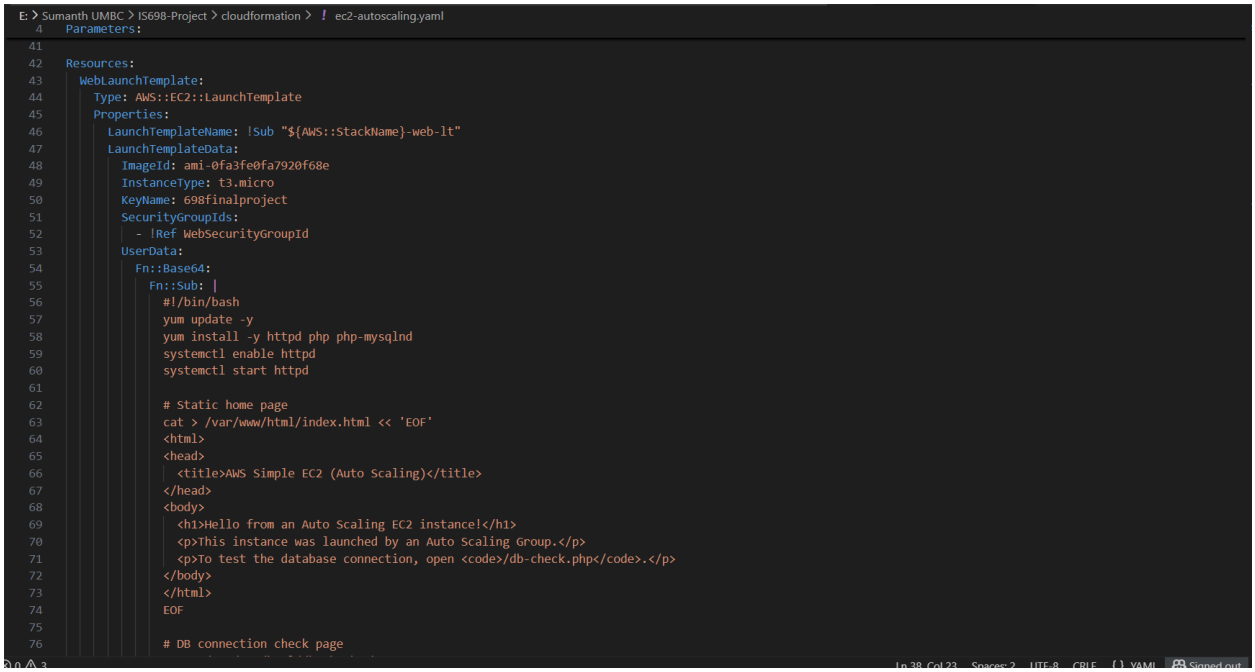




4. Autoscaling Implementation

An Auto Scaling Group (ASG) was created to automatically add or remove EC2 instances based on CPU load. This allowed the architecture to scale dynamically.

1. A Launch Template is created using:
    - The same AMI
    - The same EC2 configuration
    - The same user data script used for the manual EC2 instance



2. The ASG is configured across two public subnets for high availability.
3. CPU-based scaling policies are applied.
4. To increase the load, CPU stress is generated using the SSH command:

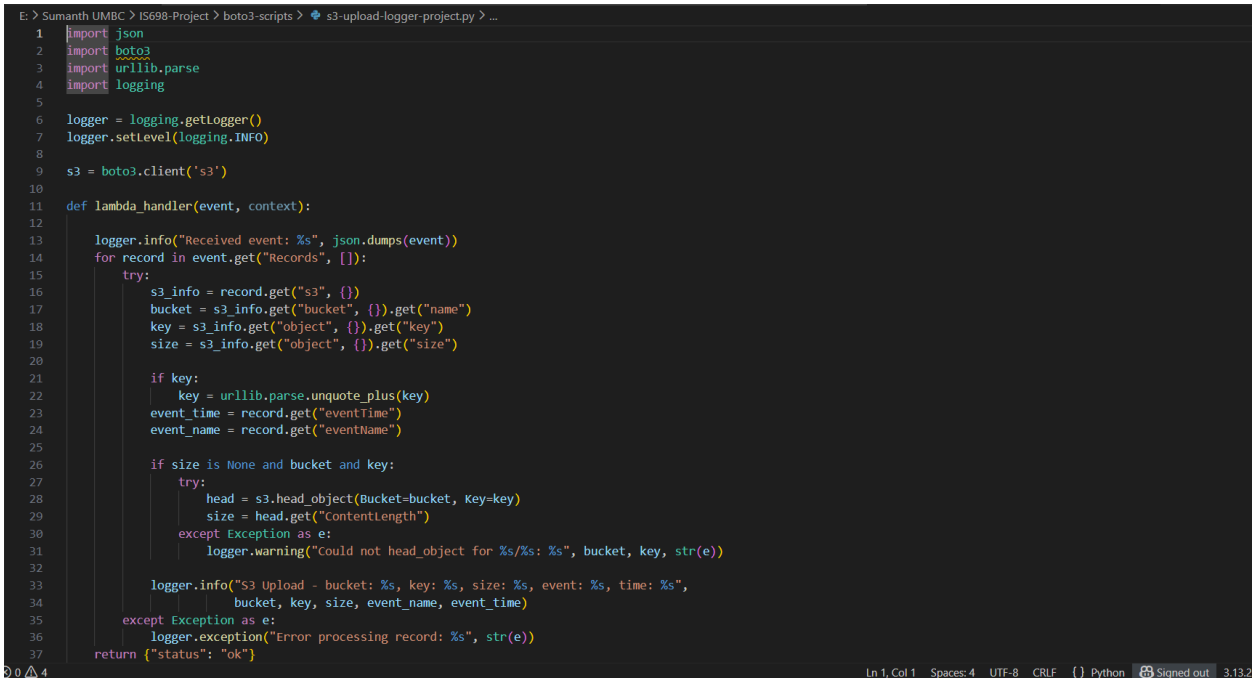
   stress −cpu 4 −timeout 120

5. When CPU usage crossed the threshold, a new EC2 instance was launched automatically.

B. AWS Lambda for Logging S3 Uploads

A Lambda function was created to automatically log S3 uploads. Whenever a file is uploaded to a S3 bucket, Lambda function is triggered and it writes the information about the upload to CloudWatch Logs.

1. Created an S3 bucket.
2. Wrote a Python Lambda function that reads S3 event metadata and prints it.
3. Linked S3 to Lambda using a bucket event trigger for operations.



4. Uploaded test files to S3, and CloudWatch Logs confirmed the event was processed.

C. AWS Interaction via AWS CLI

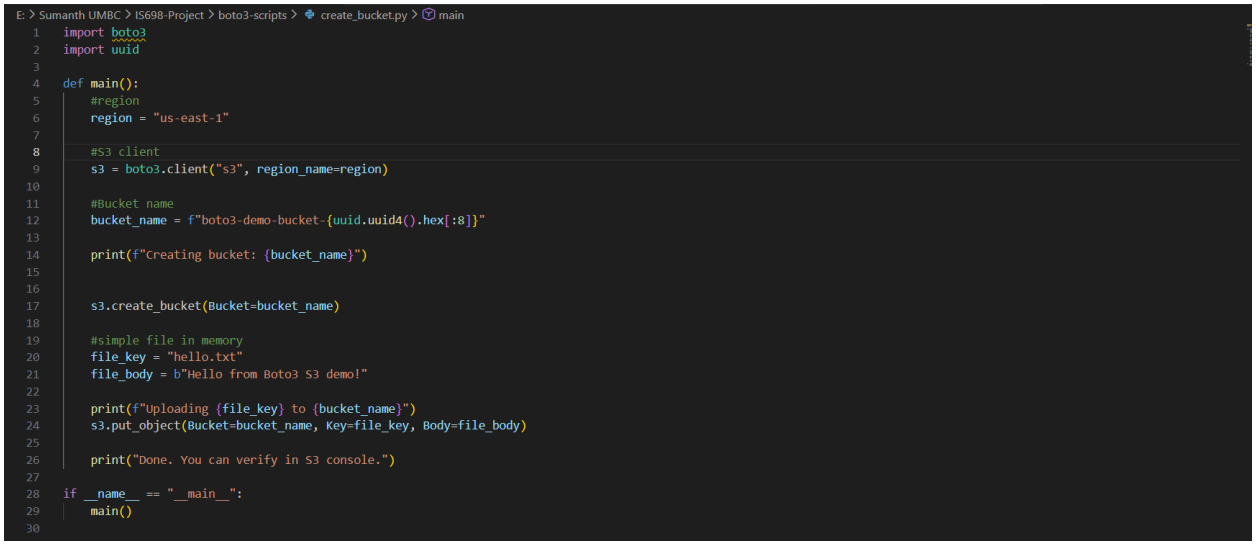
AWS CLI commands were used to manage multiple AWS resources directly through terminal.
Commands:-

1. List EC2 Instances: aws ec2 describe-instances
2. List S3 buckets: aws s3 ls
3. Invoking Lambda Function: aws lambda invoke

D. Python Boto3 Interaction

To interact with AWS programmatically, python scripts were written.

1. Creating an S3 bucket:

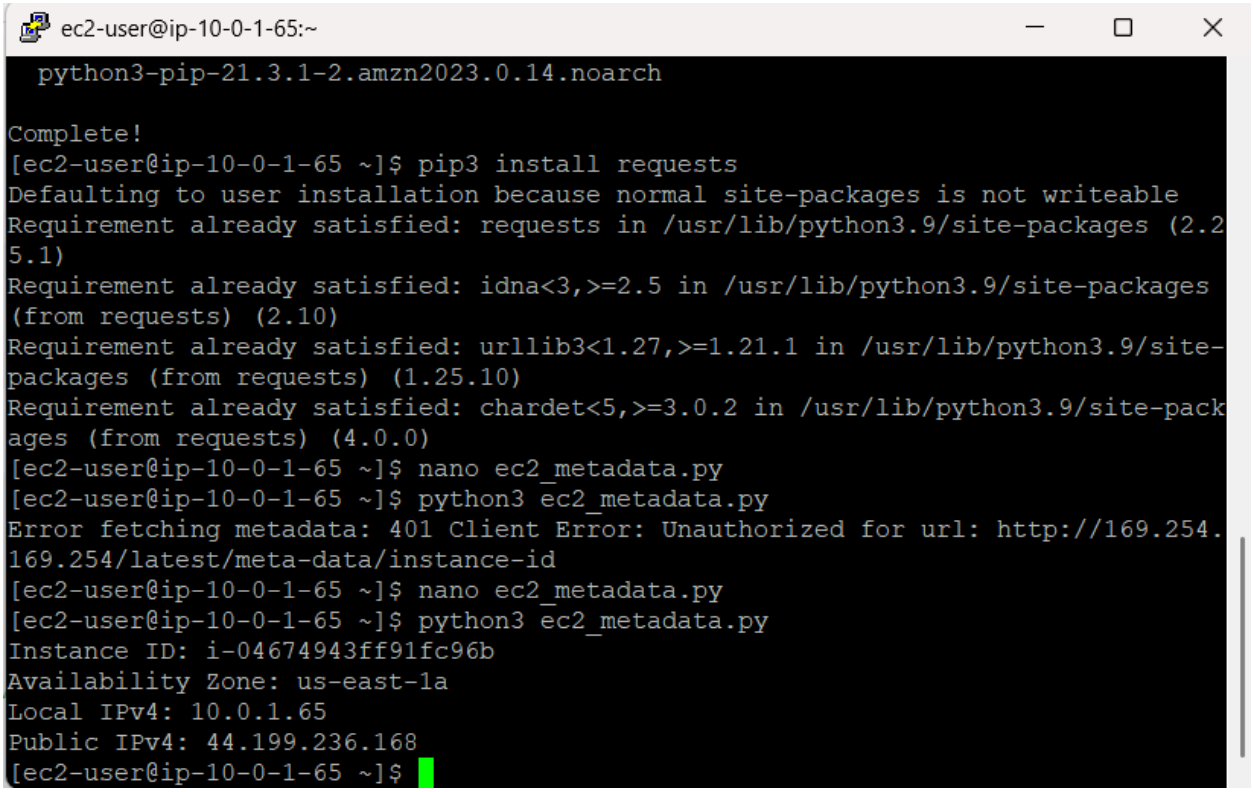


```python
import boto3
import uuid

def main():
    #region
    region = "us-east-1"

    #S3 client
    s3 = boto3.client("s3", region_name=region)

    #Bucket name
    bucket_name = f"boto3-demo-bucket-{uuid.uuid4().hex[:8]}"

    print(f"Creating bucket: {bucket_name}")


    s3.create_bucket(Bucket=bucket_name)

    #simple file in memory
    file_key = "hello.txt"
    file_body = b"Hello from Boto3 S3 demo!"

    print(f"Uploading {file_key} to {bucket_name}")
    s3.put_object(Bucket=bucket_name, Key=file_key, Body=file_body)

    print("Done. You can verify in S3 console.")

if __name__ == "__main__":
    main()
```

2. Listing EC2 instances

```python
import boto3

def main():
    region = "us-east-1"
    ec2 = boto3.client("ec2", region_name=region)

    response = ec2.describe_instances(
        Filters=[{"Name": "instance-state-name", "Values": ["running"]}]
    )

    print("Running EC2 instances:")
    for reservation in response["Reservations"]:
        for inst in reservation["Instances"]:
            instance_id = inst["InstanceId"]
            state = inst["State"]["Name"]
            public_ip = inst.get("PublicIpAddress")
            name_tag = next(
                (t["Value"] for t in inst.get("Tags", []) if t["Key"] == "Name"),
                None,
            )

            print(f"- ID: {instance_id}, State: {state}, Name: {name_tag}, Public IP: {public_ip}")

if __name__ == "__main__":
    main()
```

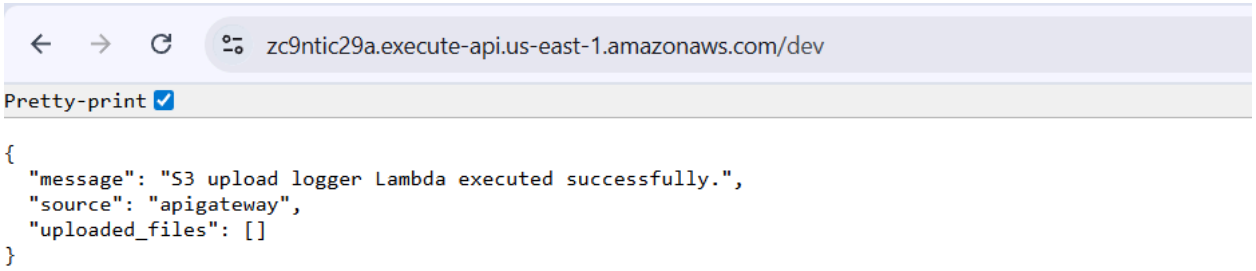
3. Retrieving EC2 metadata



4. Invoking lambda

E. Bonus Challenge

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



A REST API was added using API Gateway to trigger the existing Lambda function through an HTTP request. The Lambda code was updated to recognize API Gateway events and return a simple JSON message. After creating a new resource and GET method in API Gateway, the method was linked to the Lambda function using proxy integration and deployed to a stage. The generated URL allowed the Lambda function to be invoked from a browser, confirming that the function works for both S3 uploads and manual API calls.

F. Challenges Encountered and Solutions Applied

During the course of this project, various difficulties were encountered while working with Terraform, CloudFormation, EC2, RDS, and Lambda functions. The first difficulty was encountered while creating the RDS instance. The stack was failing consistently due to improper DB parameters and the security group configuration. This problem was fixed by correcting the DB name, checking the credentials, and allowing MySQL access from the EC2 security group on the RDS security group. After this was done, the stack was created successfully.

Another issue at this stage was ensuring the connection between the EC2 instance and RDS was established. Despite updating the PHP script, the issue persisted. It was necessary to allow outbound traffic on port 3306.

Test cases for auto-scaling also included a wait for newly created EC2 instances. Initially, there was assumed to be a problem with the auto-scaling configuration, though the actual issue was the time for image initialization. Once realizing that image deployment does take about 2-3 minutes, the auto-scaling function was tested successfully.

At first, the script for EC2 metadata failed because the instance was set to use IMDSv2. This requires the use of a session token. Incorporating this token logic into the script solved the issue with authentication.

G. Future improvements and Recommendations

In the future, this architecture could be improved in a few practical ways:

- Enabling RDS Multi-AZ and automated backups would protect the database from Availability Zone failures and make recovery from issues much easier. Performance Insights could also help tune queries if the workload grows.
- CloudWatch dashboards and alarms can be added for CPU usage, scaling events, RDS performance, and Lambda errors. Connecting these alarms to SNS notifications would make it easier to react quickly when something goes wrong.