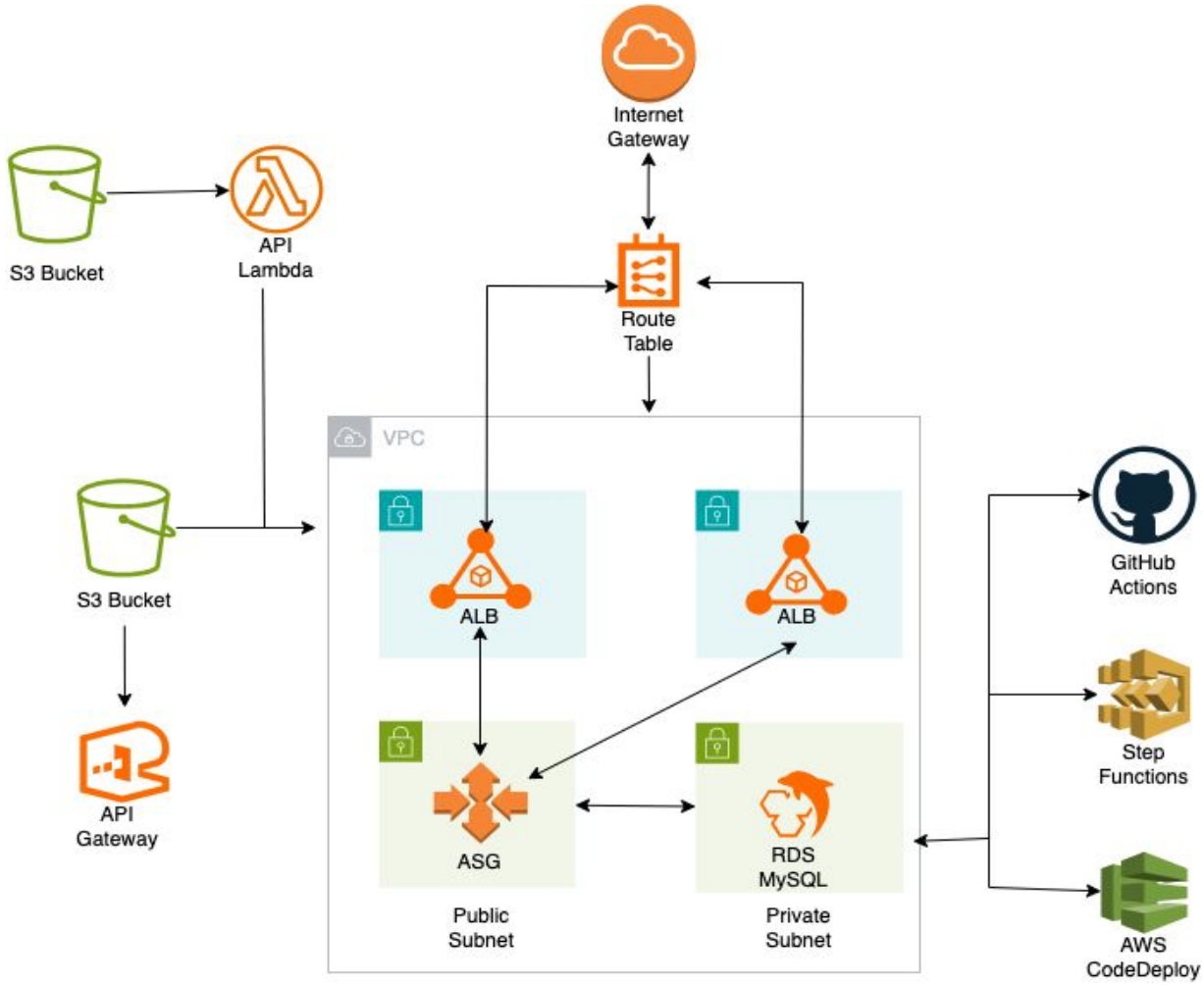
IS 698 Special Topics in Information Systems
Project

Name- Akanksha Patel
Id- Ri79216

Deploying a Scalable AWS Architecture with Infrastructure as Code

1. Design an AWS Architecture

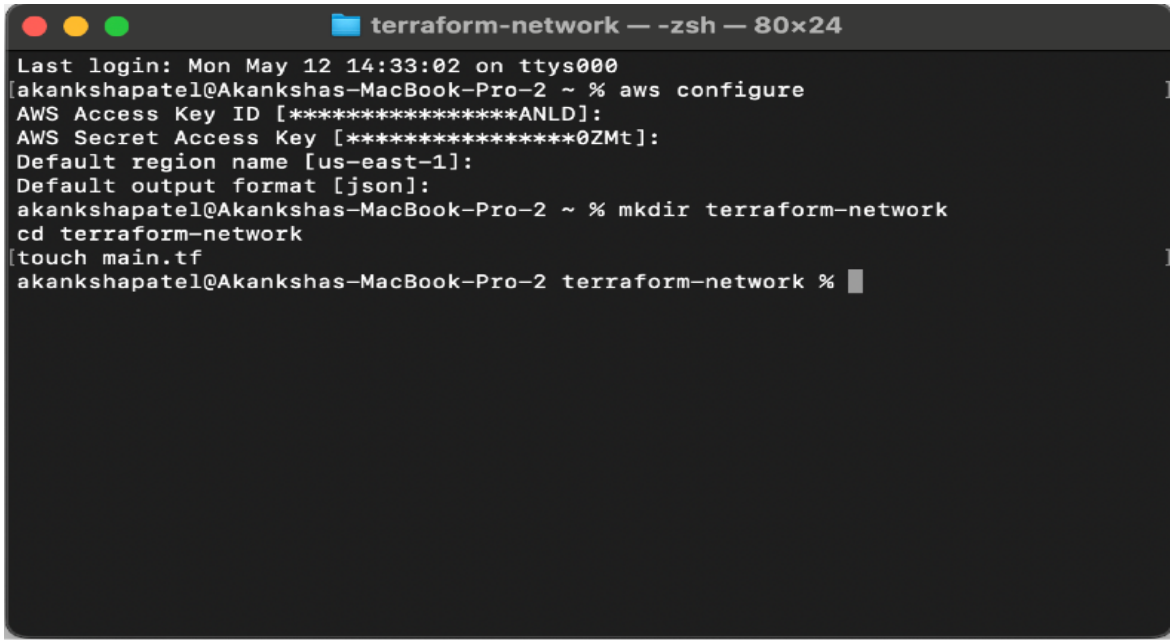


2. Implementation

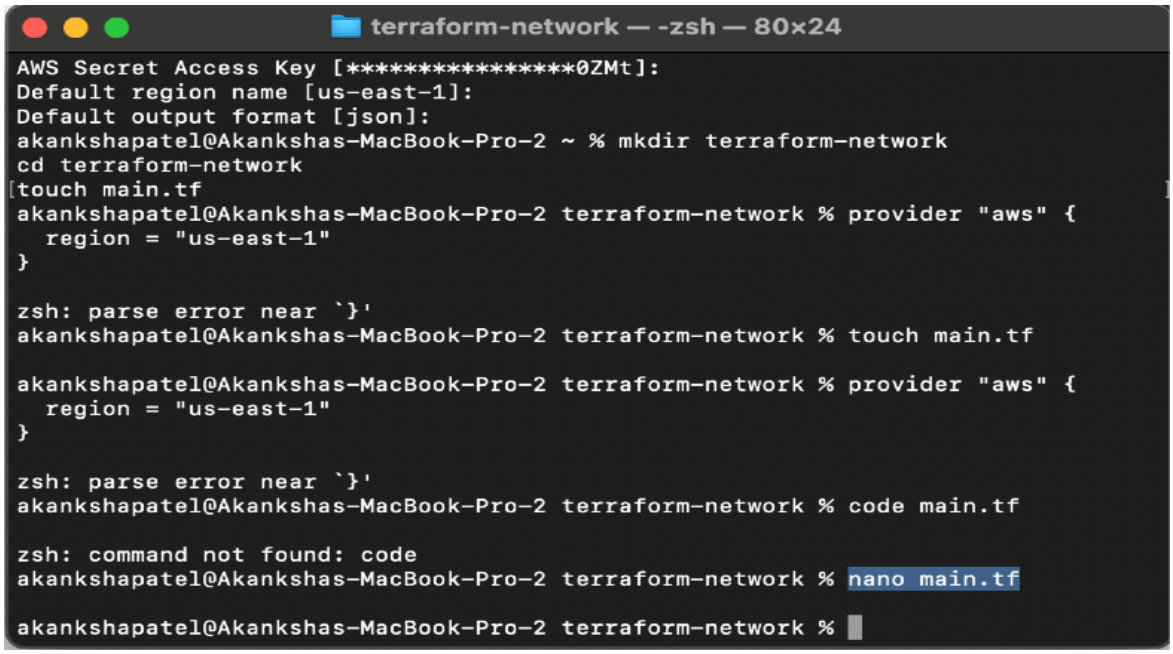
A. Infrastructure Deployment

Create AWS Networking Using Terraform

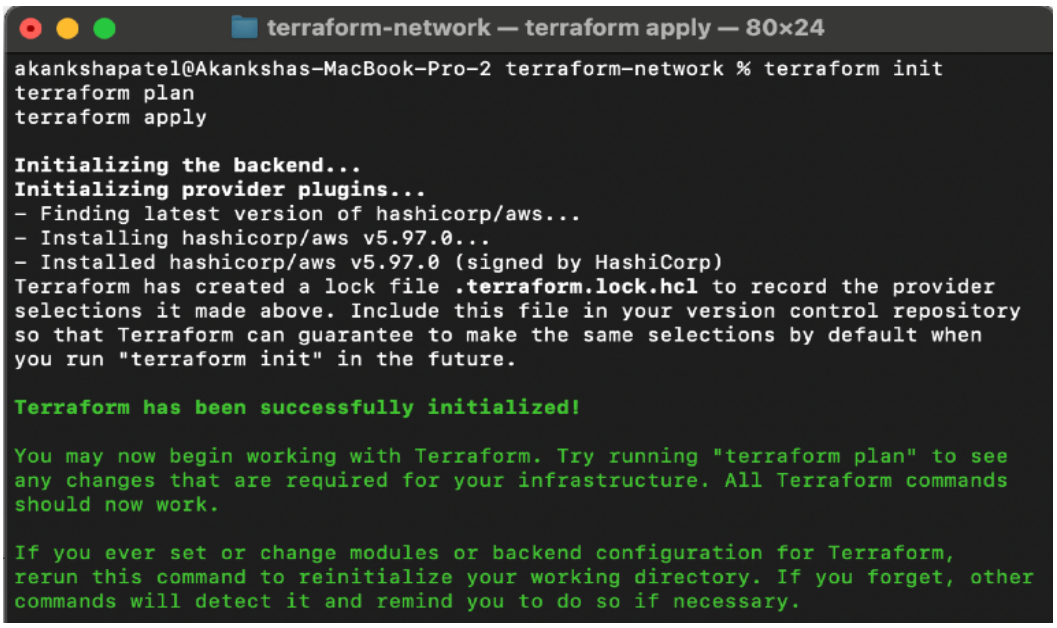
STEP 1: Set Up the Project Folder

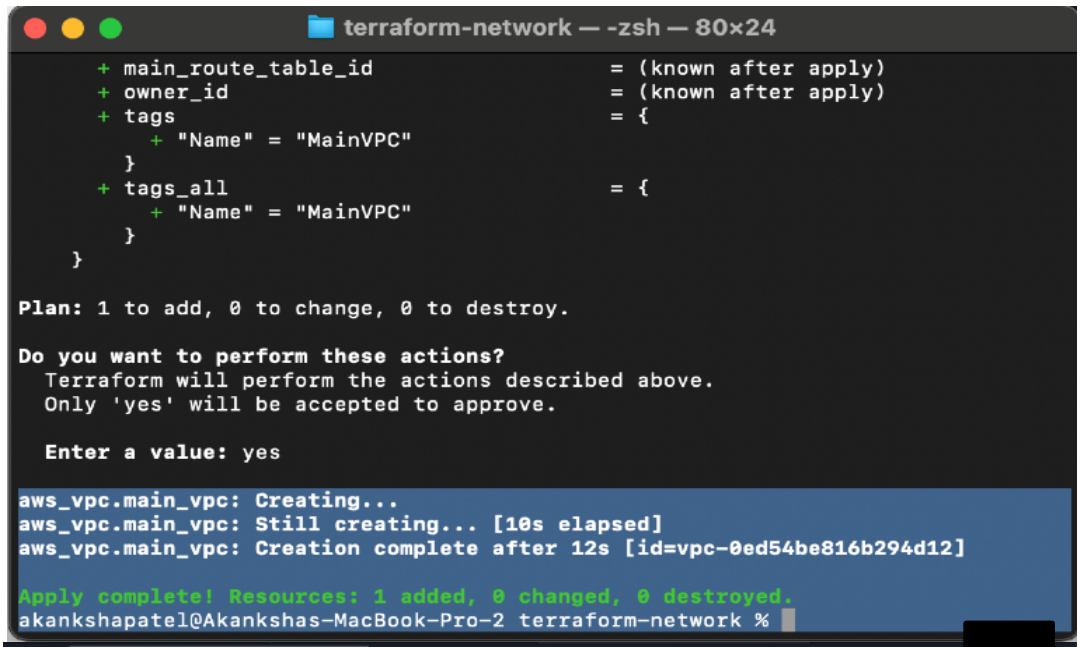


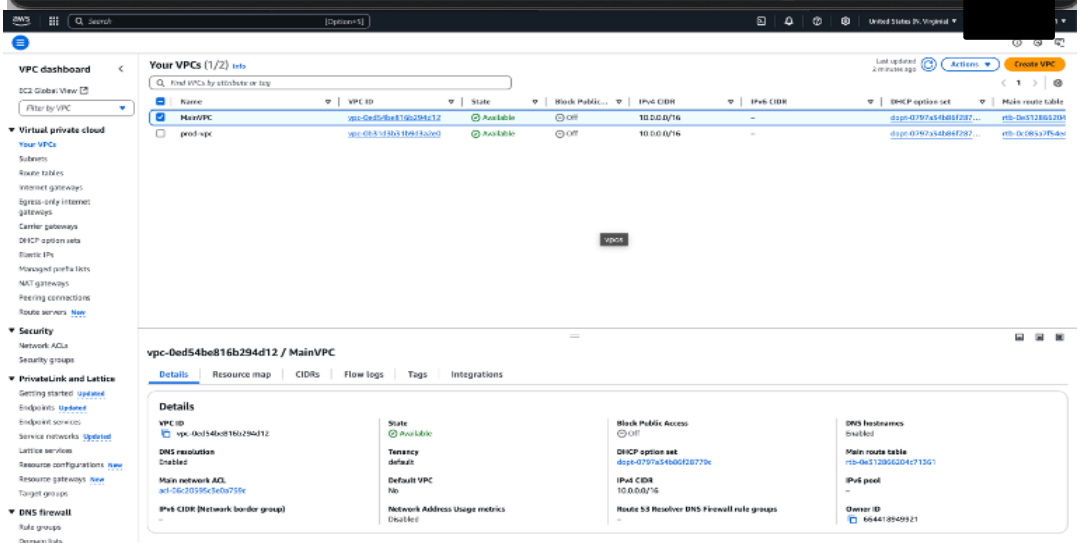
STEP 2: Configure the AWS Provider



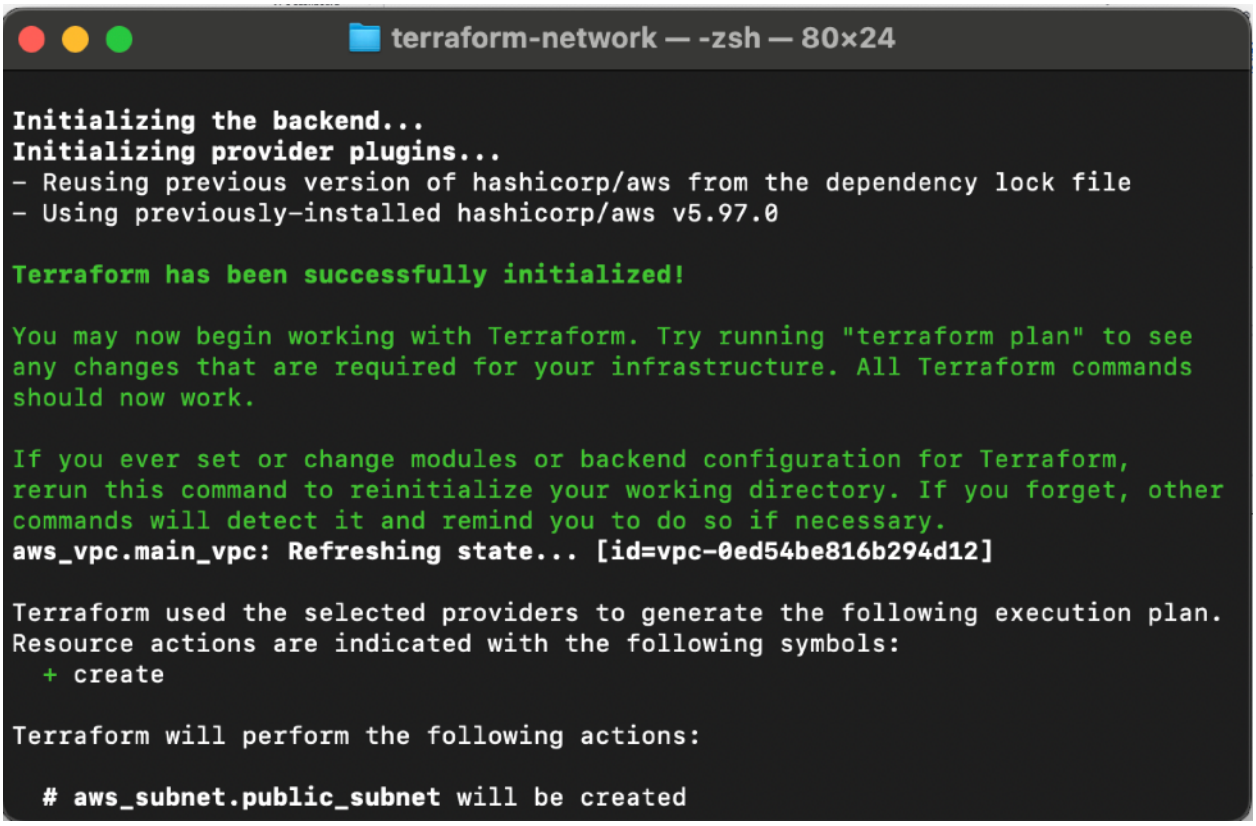
STEP 3: Create a VPC

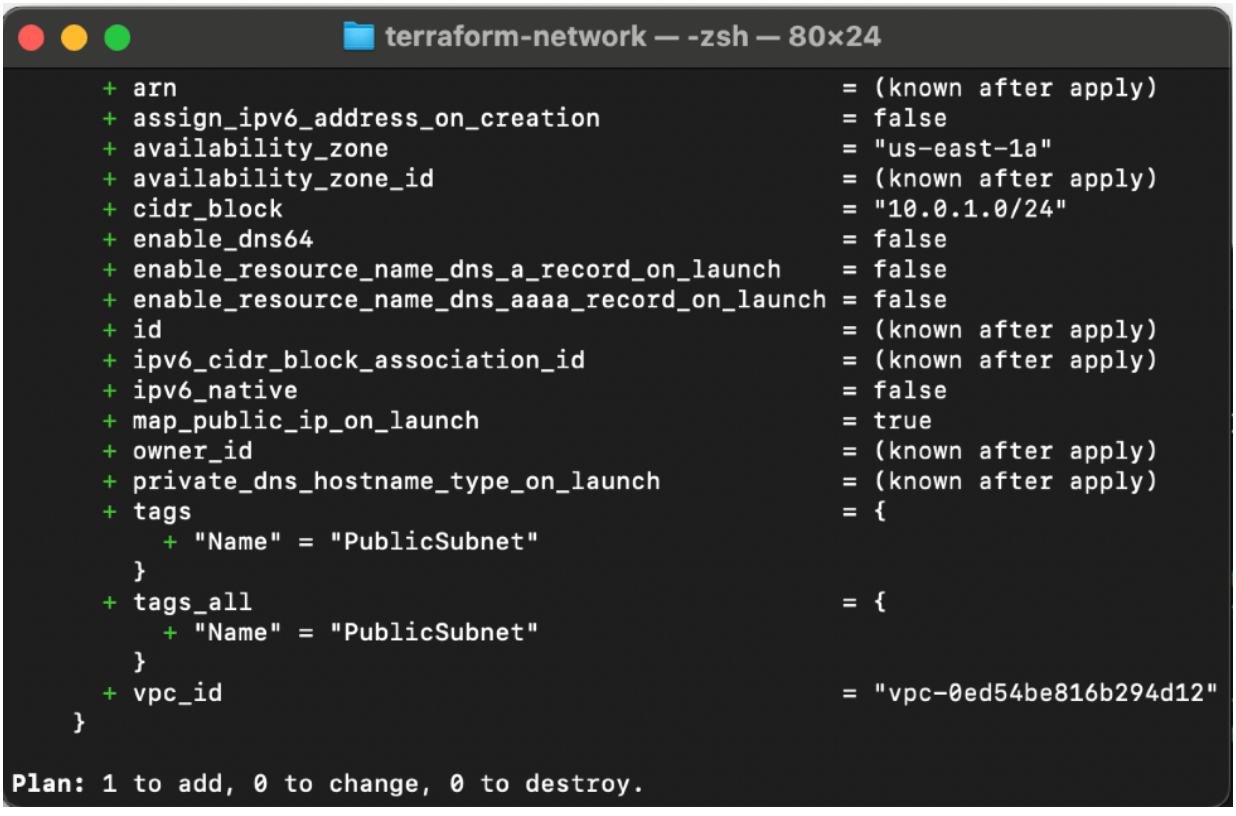






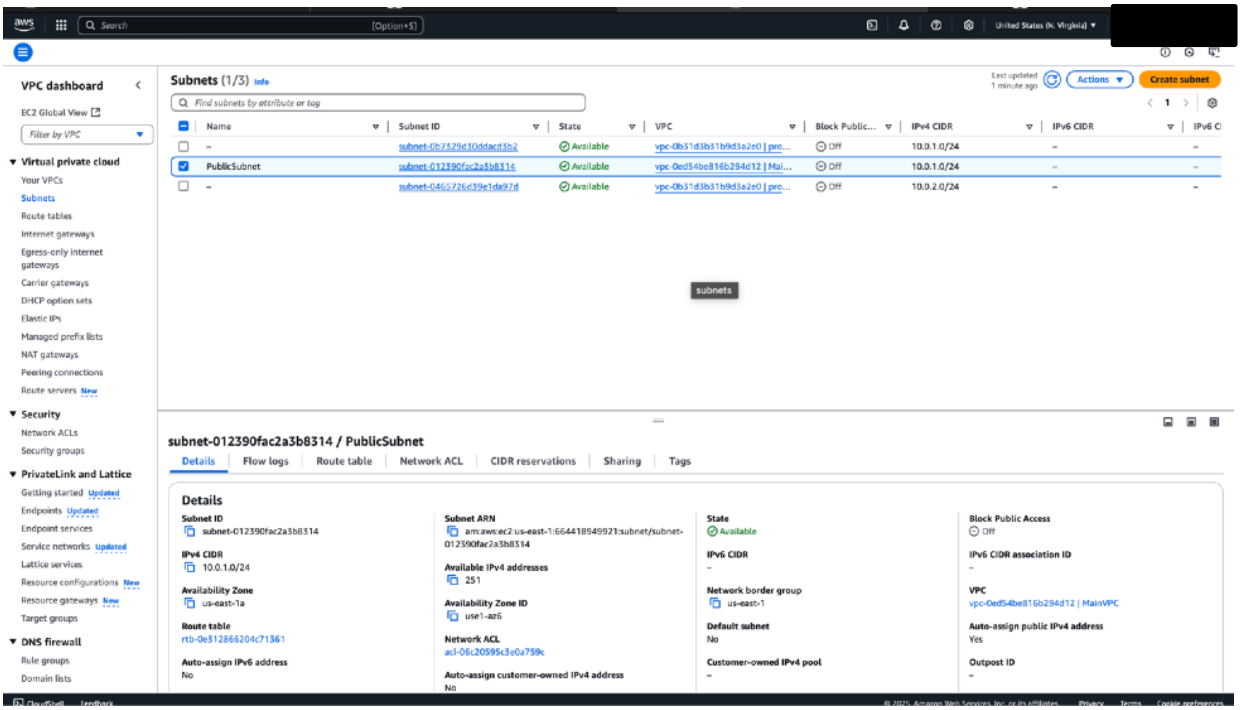
STEP 4: Create Public and Private Subnets
Public Subnet



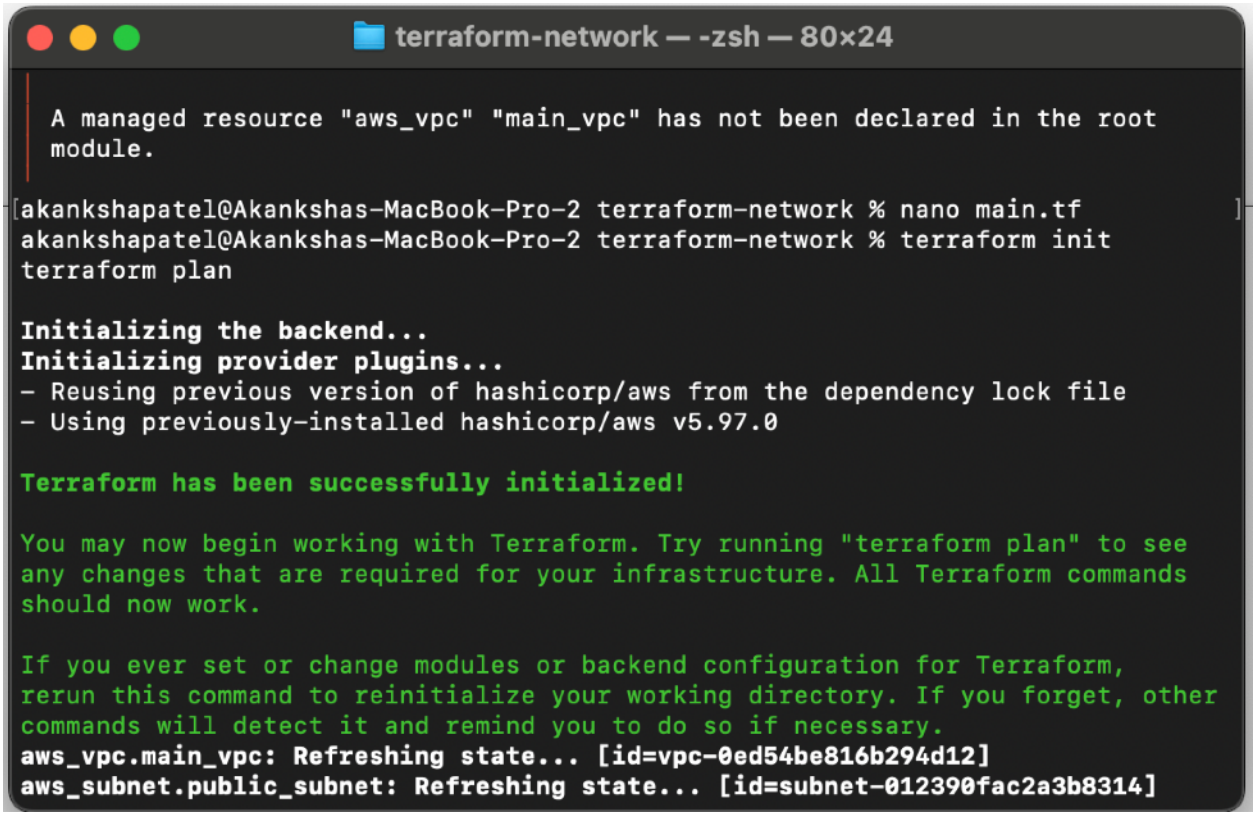


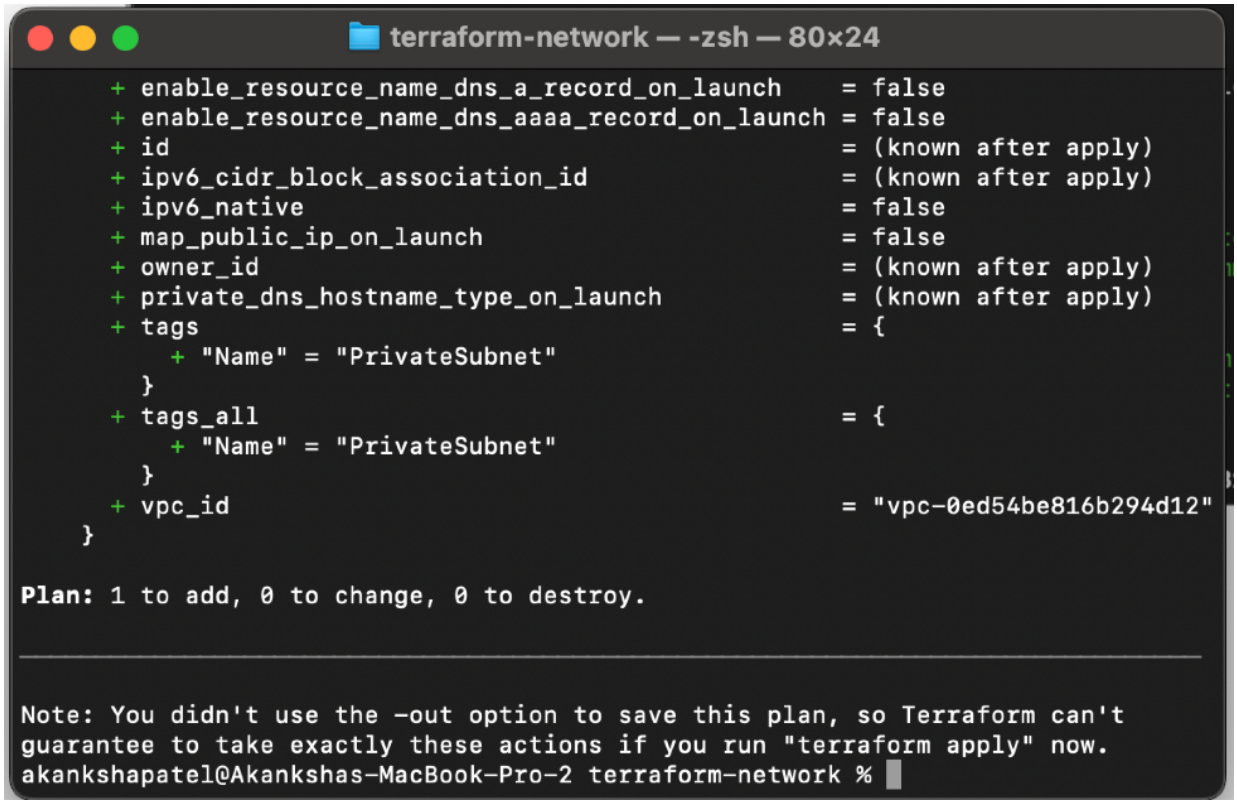
terraform-network — -zsh — 80×24

```
      + tags                                                 = {
          + "Name" = "PublicSubnet"
        }
      + tags_all                                             = {
          + "Name" = "PublicSubnet"
        }
      + vpc_id                                               = "vpc-0ed54be816b294d12"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_subnet.public_subnet: Creating...
aws_subnet.public_subnet: Still creating... [10s elapsed]
aws_subnet.public_subnet: Creation complete after 11s [id=subnet-012390fac2a3b83
14]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
akankshapatel@Akankshas-MacBook-Pro-2 terraform-network %
```

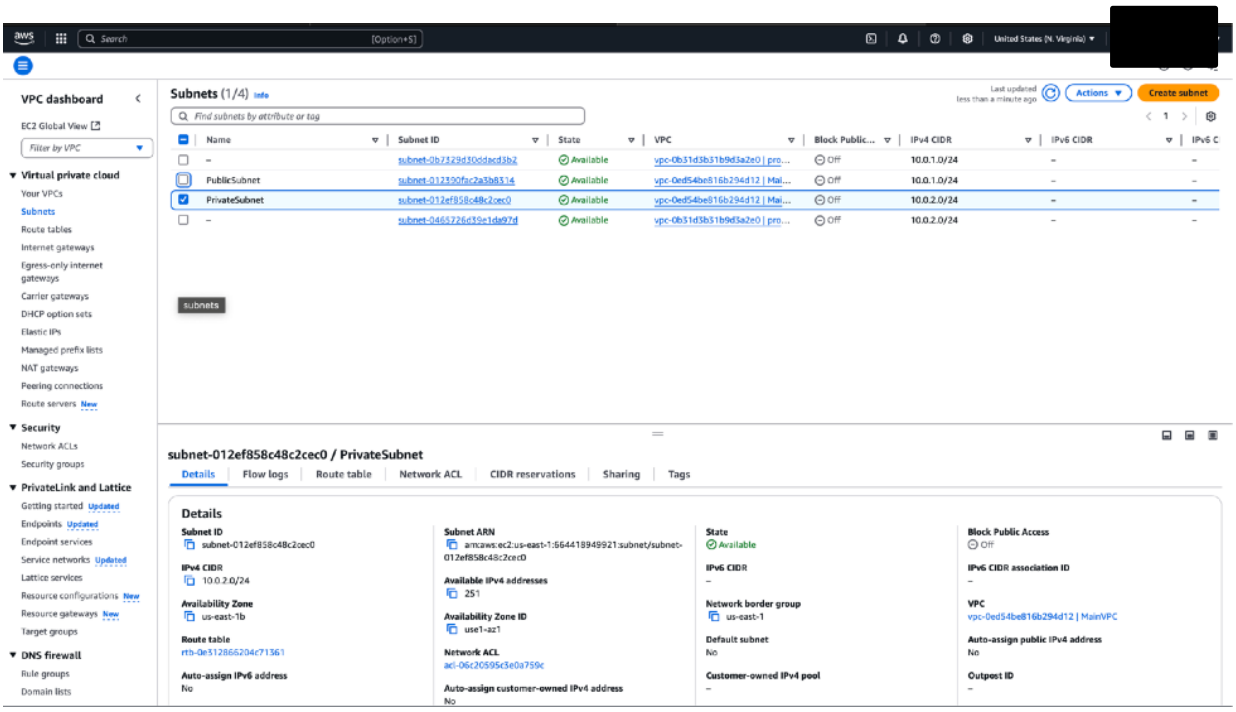


Private Subnet

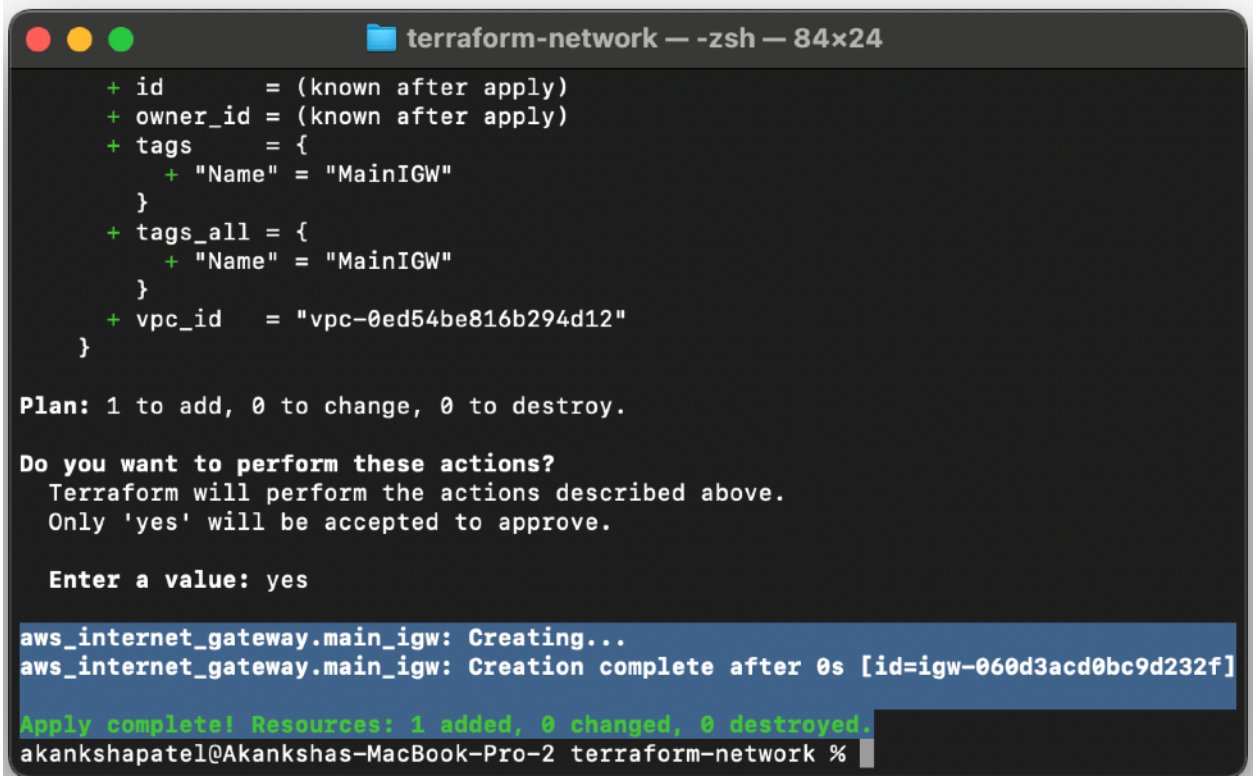


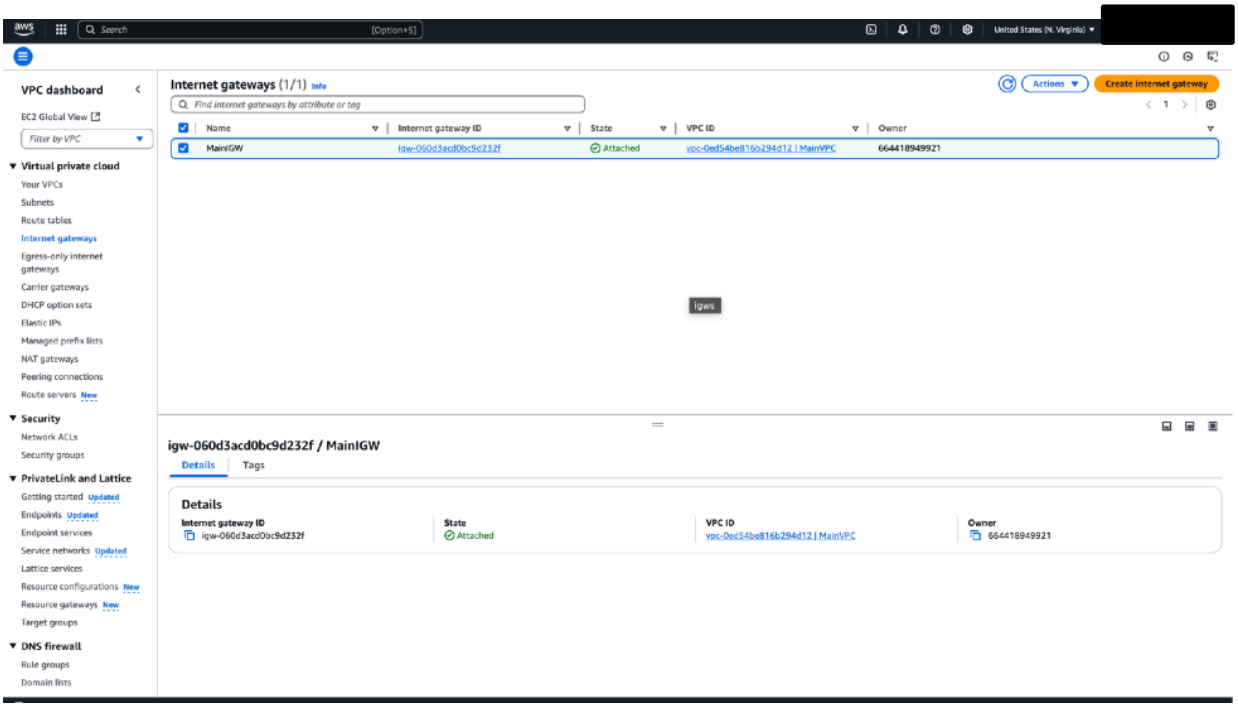


```
terraform-network — -zsh — 84×24

      + owner_id                                     = (known after apply)
      + private_dns_hostname_type_on_launch          = (known after apply)
      + tags                                         = {
          + "Name" = "PrivateSubnet"
        }
      + tags_all                                     = {
          + "Name" = "PrivateSubnet"
        }
      + vpc_id                                       = "vpc-0ed54be816b294d12"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_subnet.private_subnet: Creating...
aws_subnet.private_subnet: Creation complete after 1s [id=subnet-012ef858c48c2cec0]

akankshapatel@Akankshas-MacBook-Pro-2 terraform-network %
```

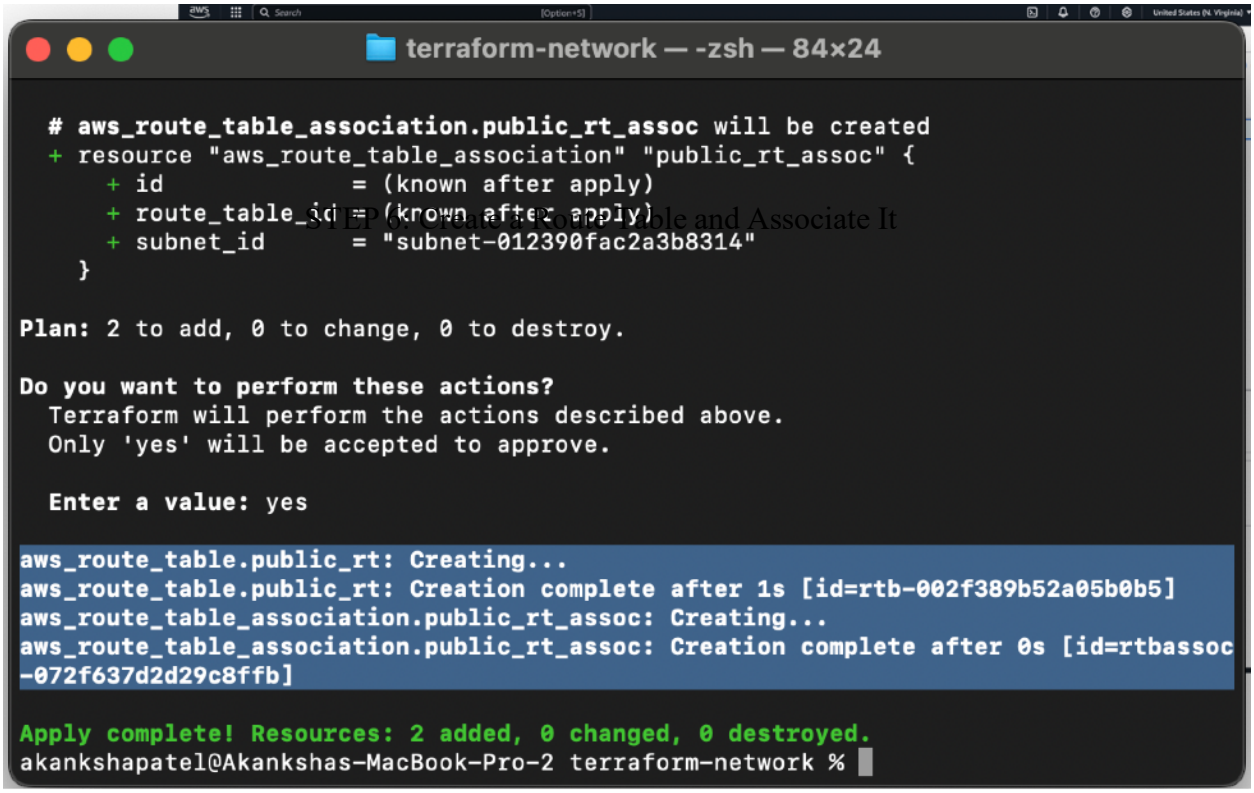


STEP 5: Create an Internet Gateway

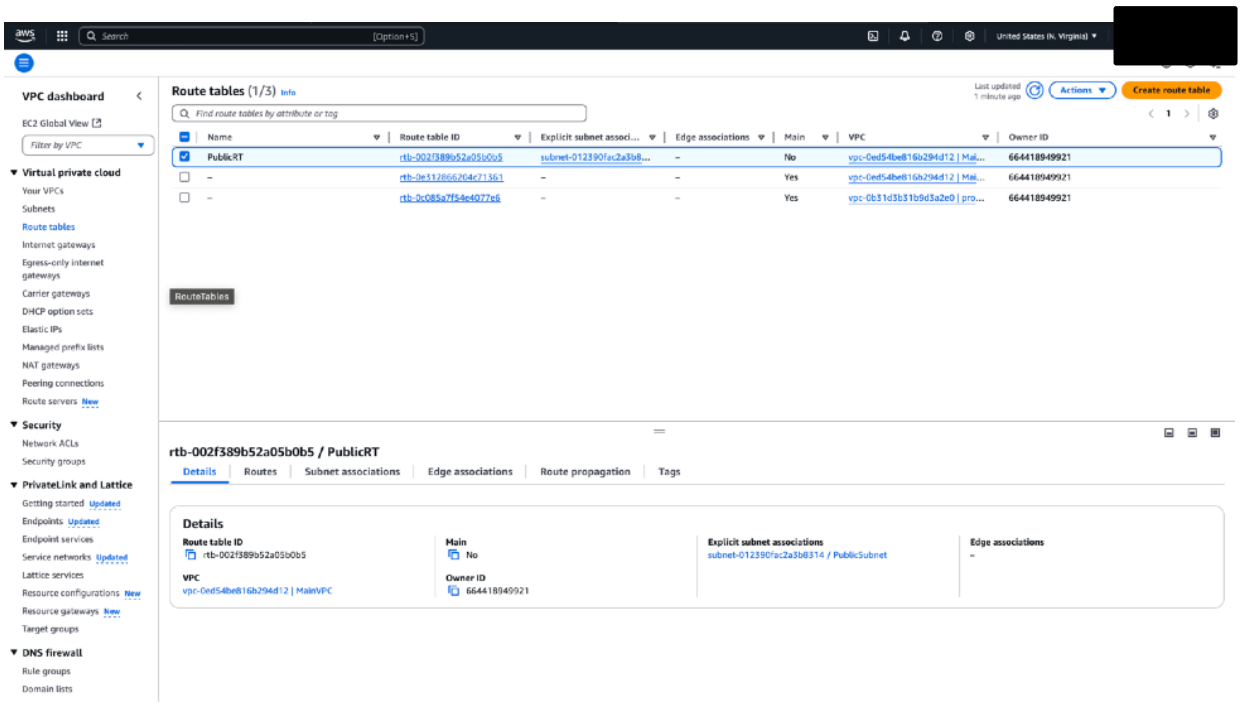




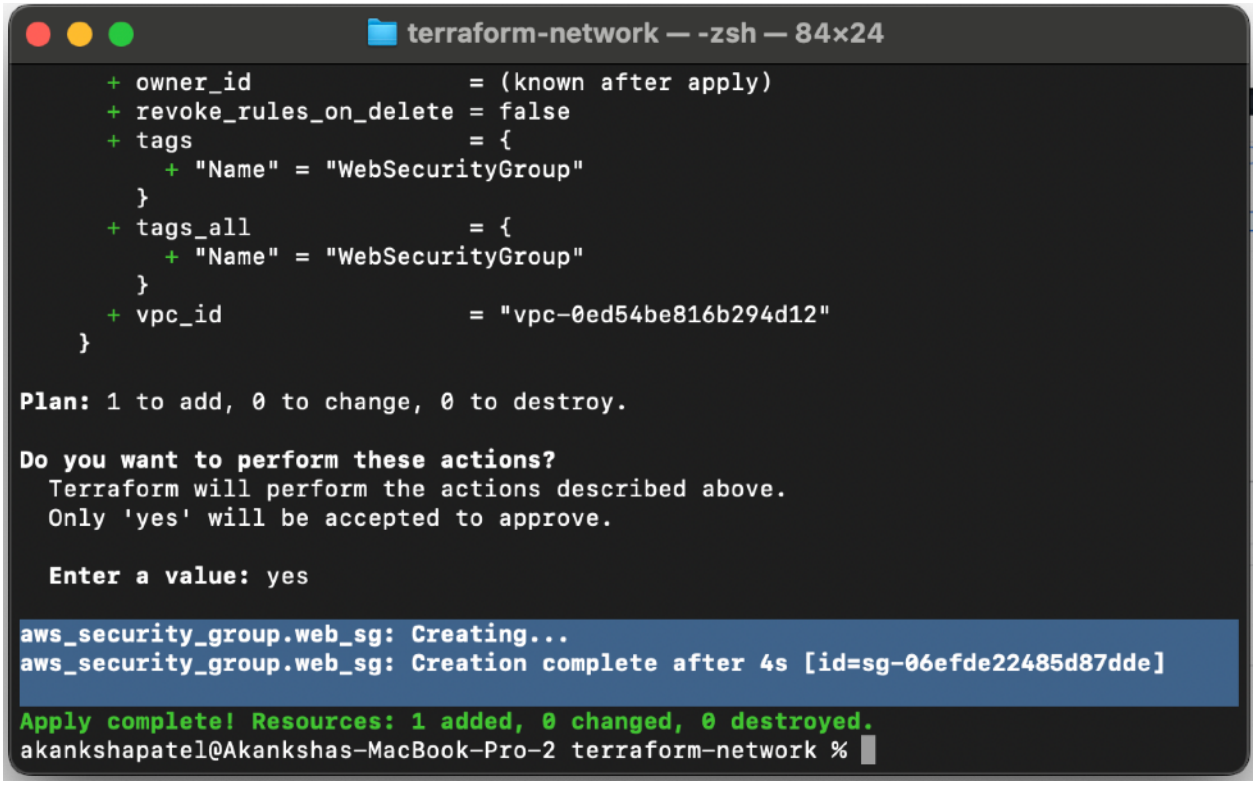
STEP 6: Create a Route Table

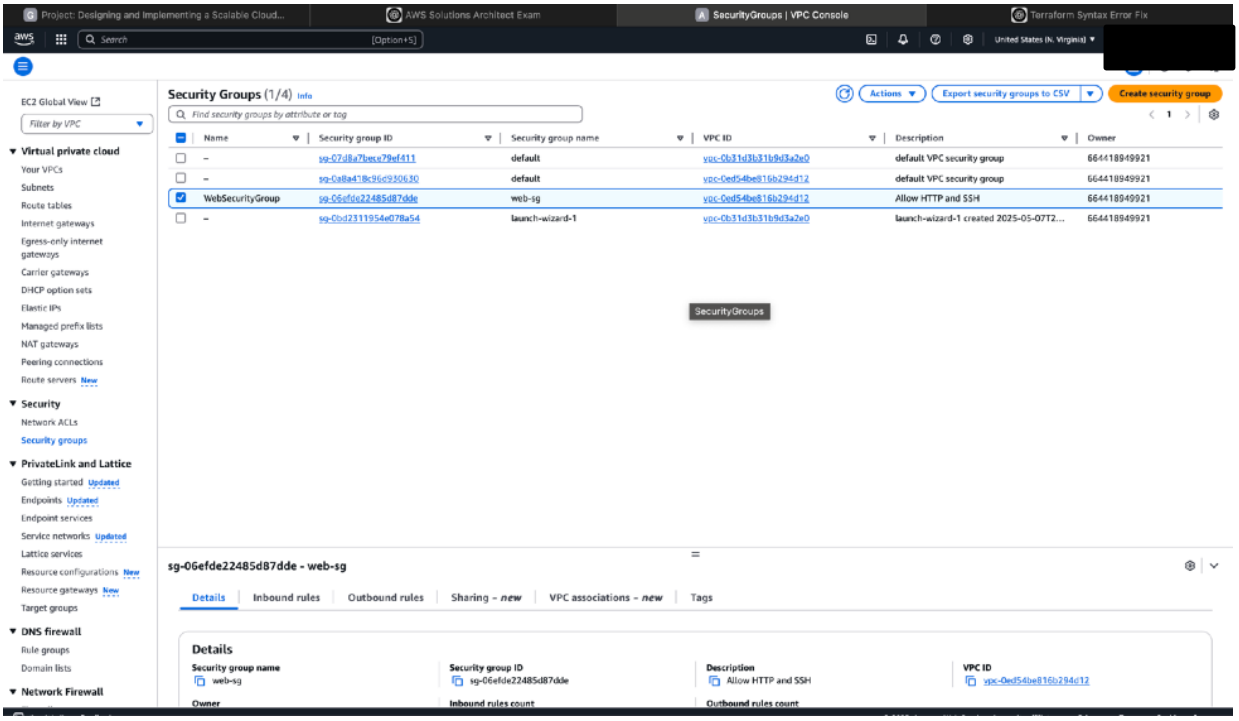


STEP 6: Create a Route Table and Associate It



STEP 7: Create a Security Group

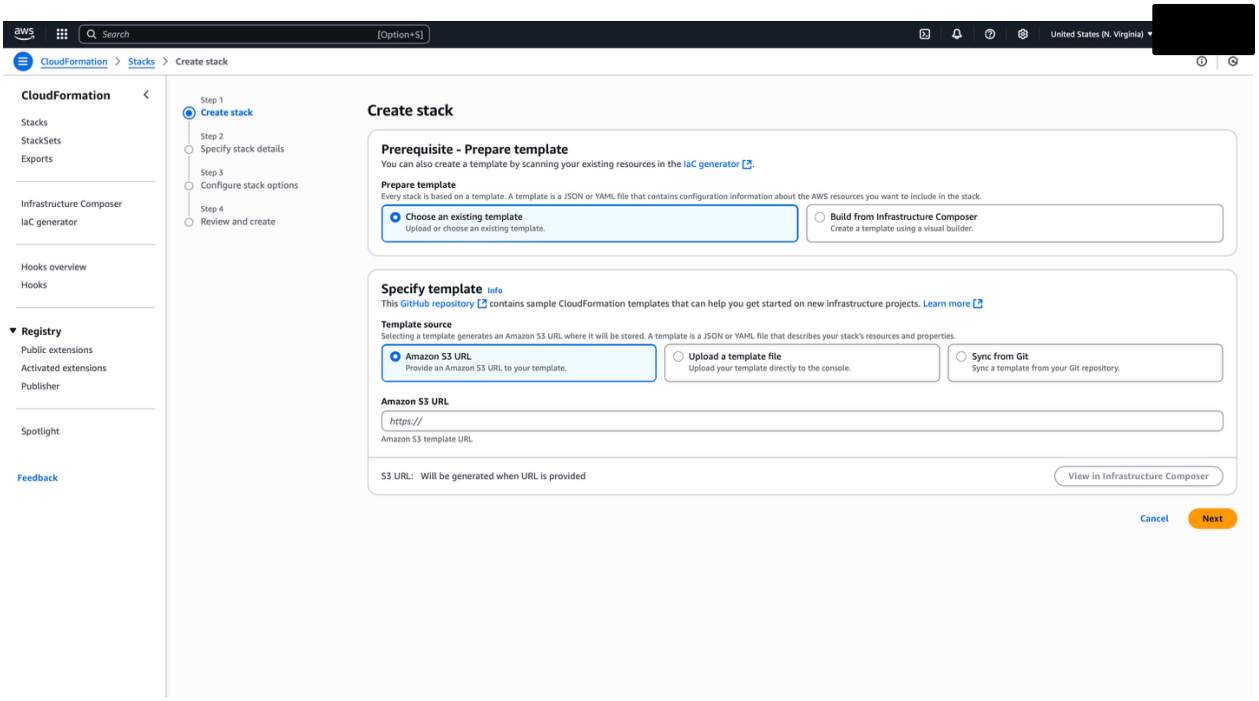




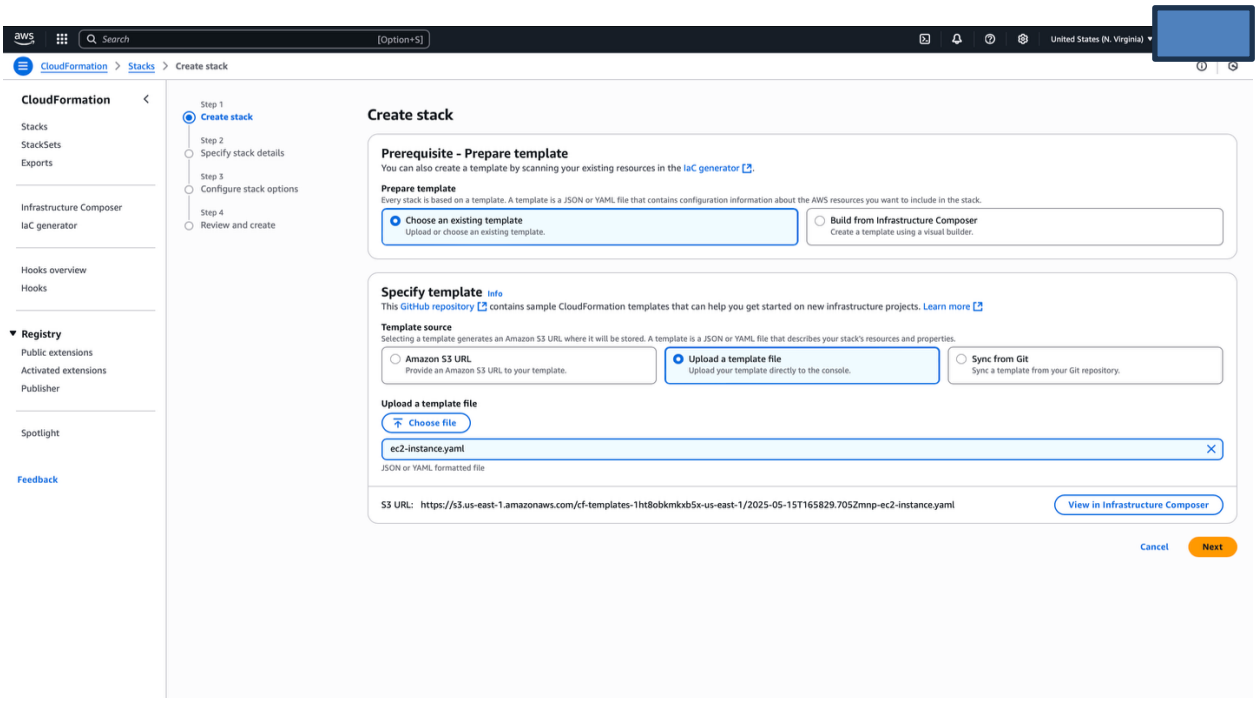
Use CloudFormation to deploy EC2 instances, RDS, and Lambda functions

EC2 Instance:

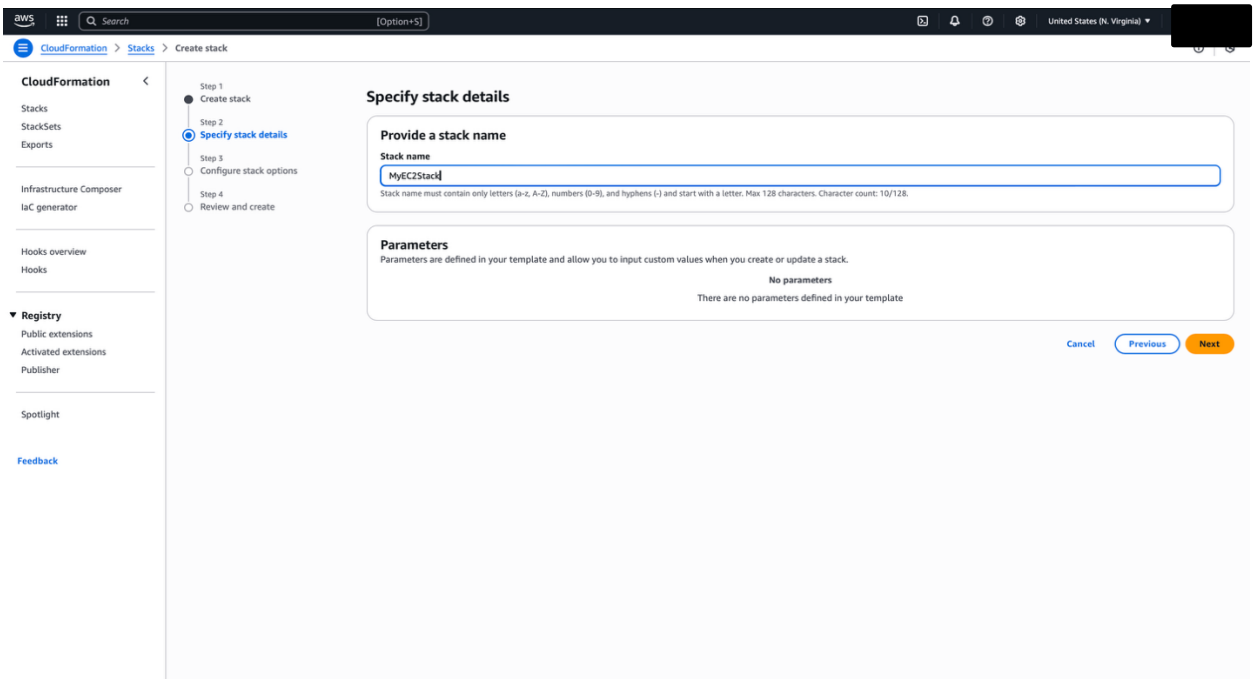
CloudFormation



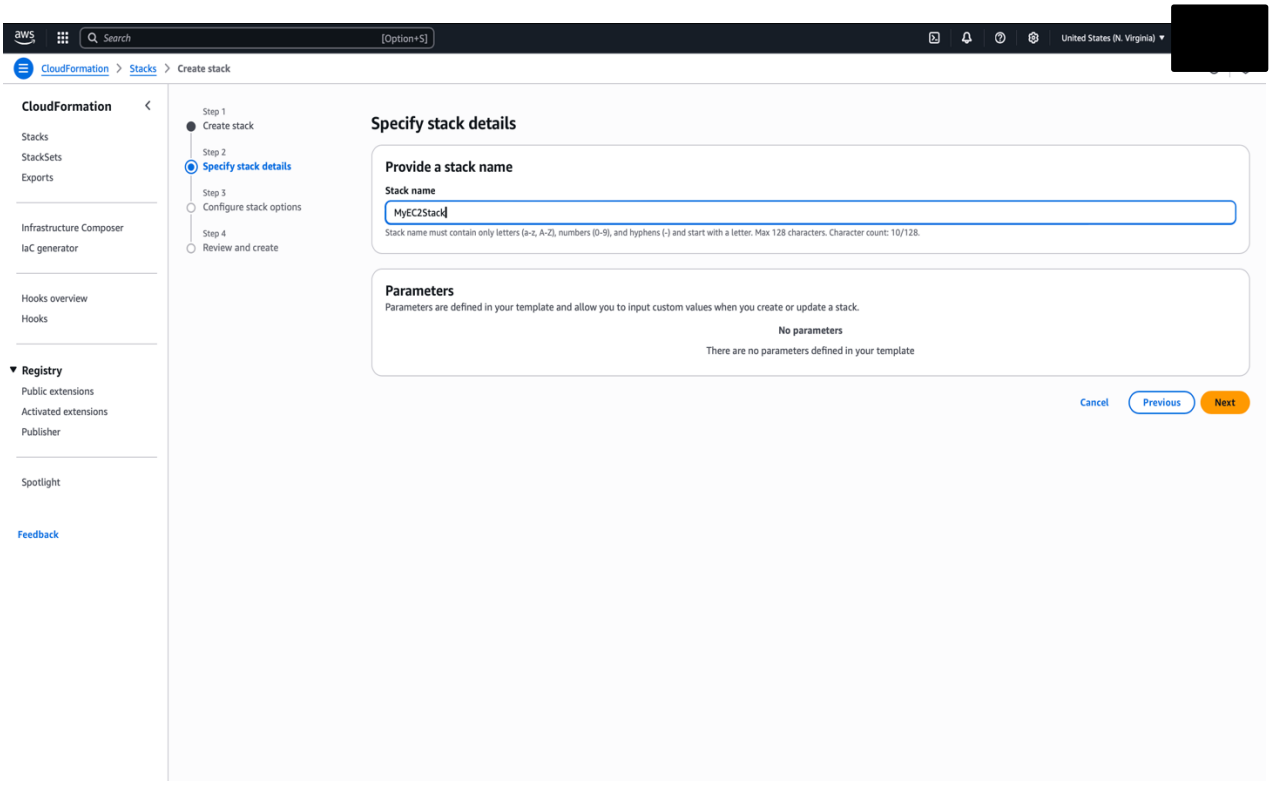
Upload the Template



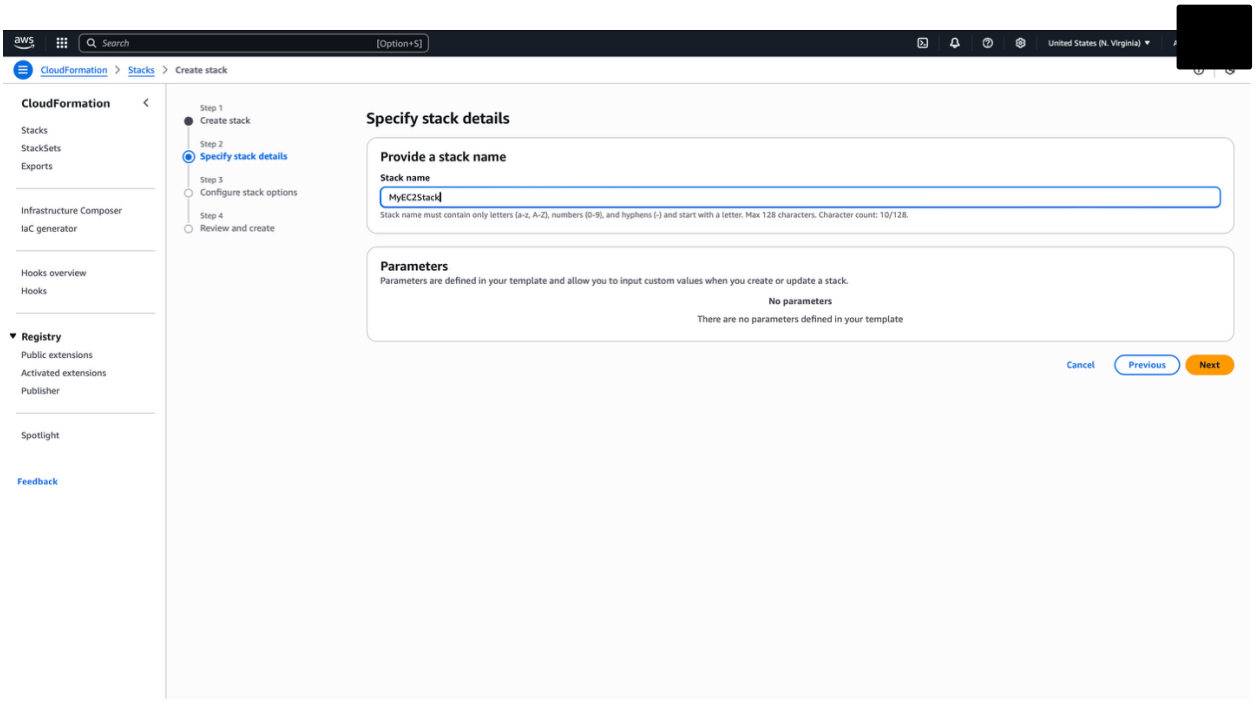
Create Stack



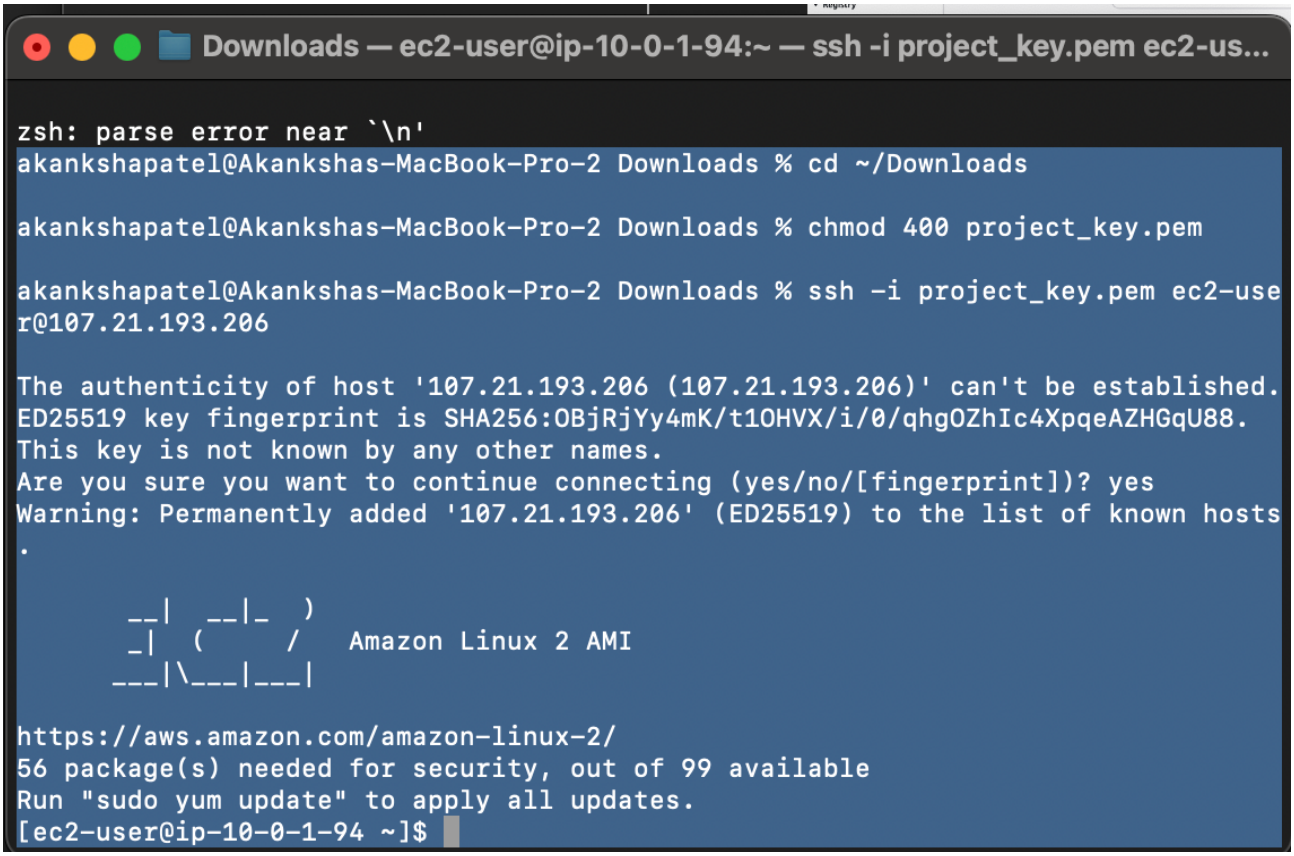
Monitor Progress



Verify EC2 Instance

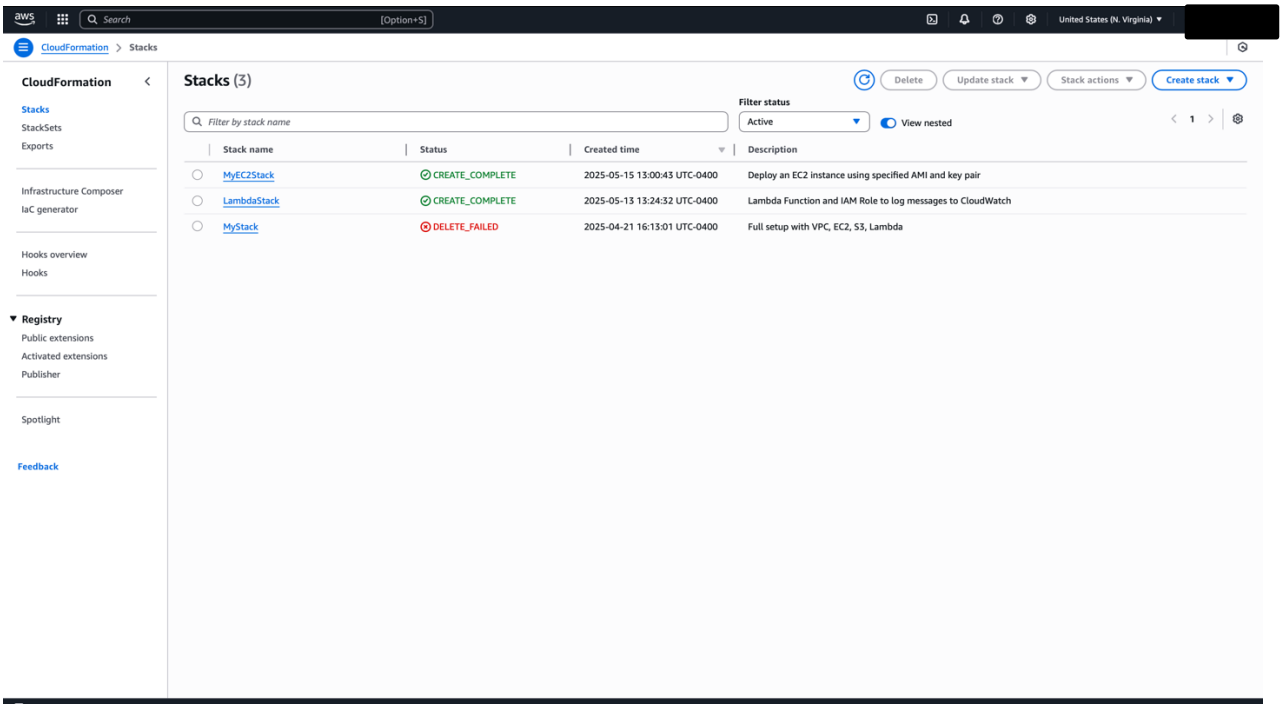


Connect via SSH

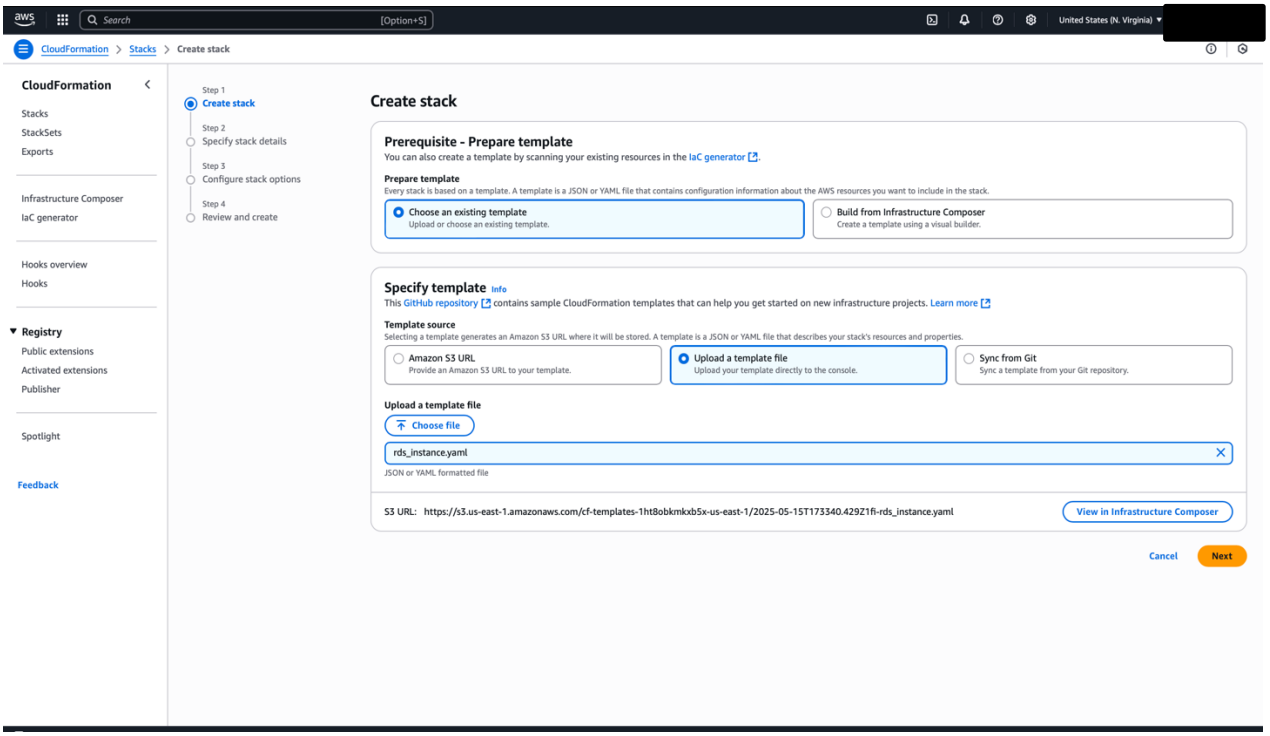


RDS Instances

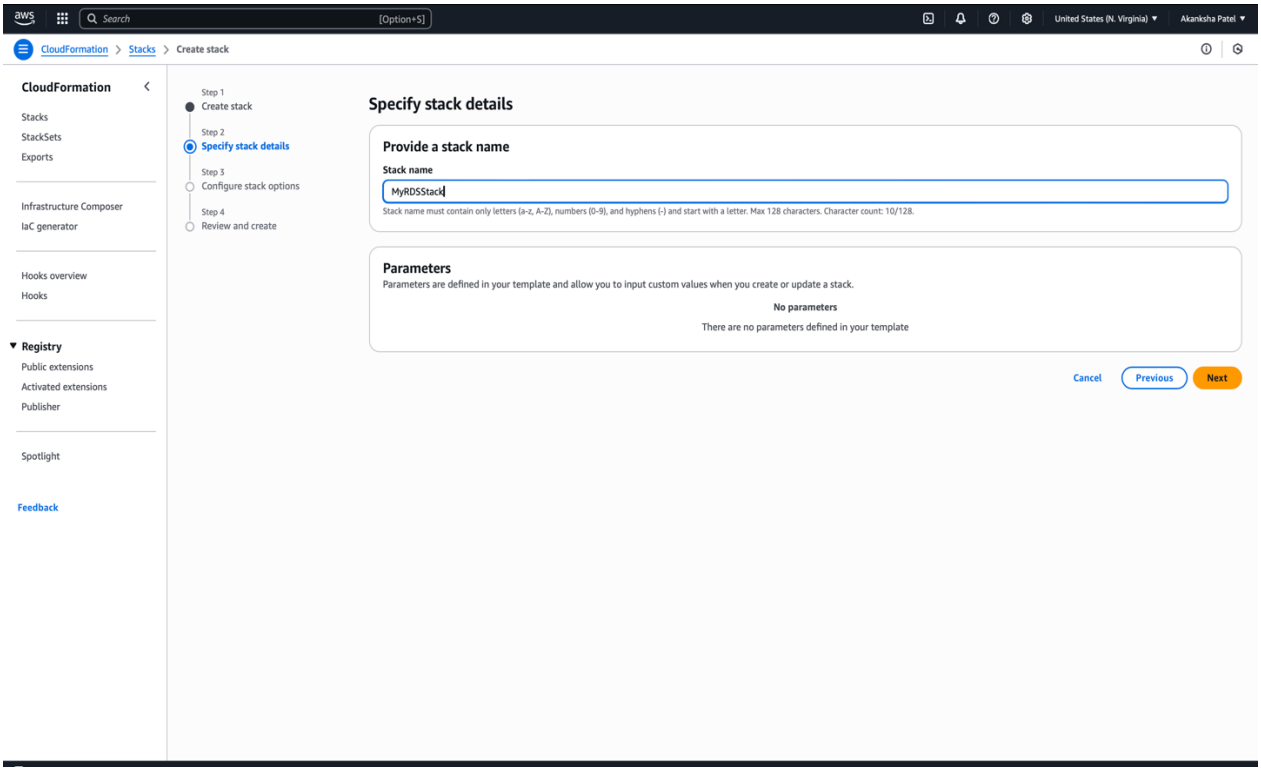
CloudFormation



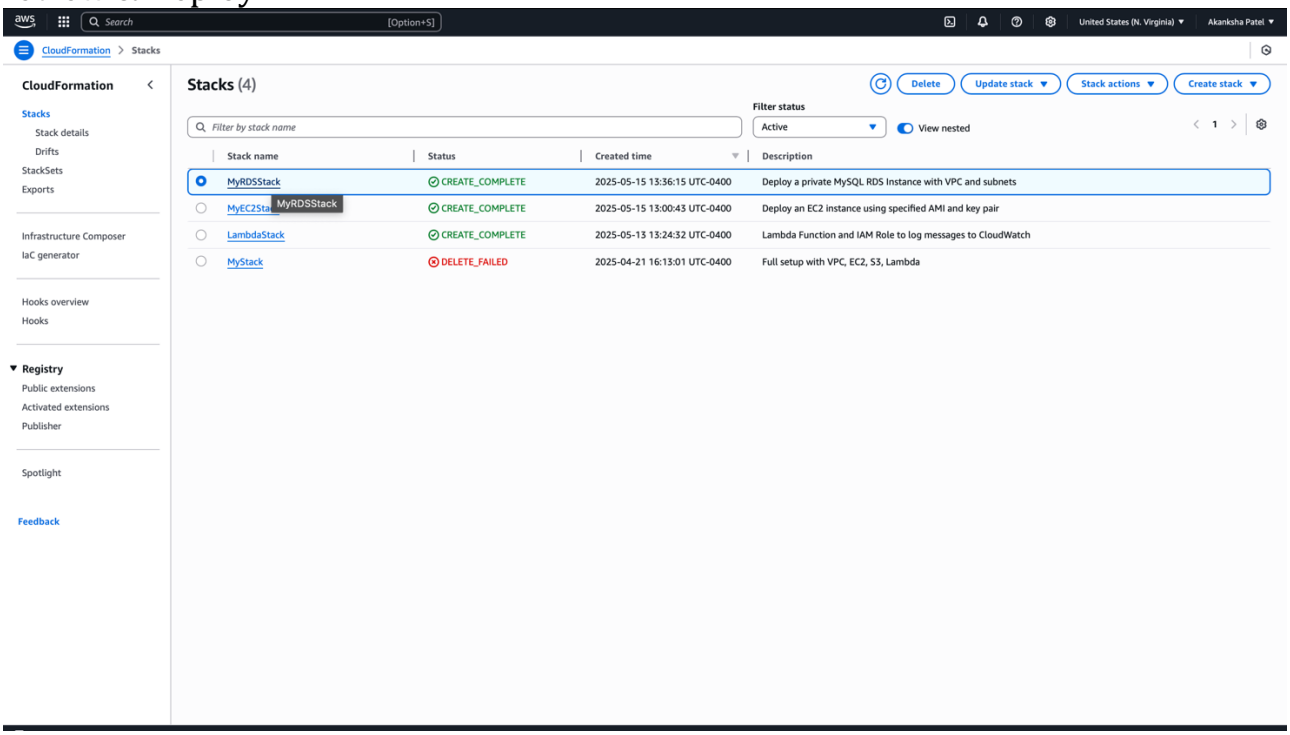
Upload the YAML File



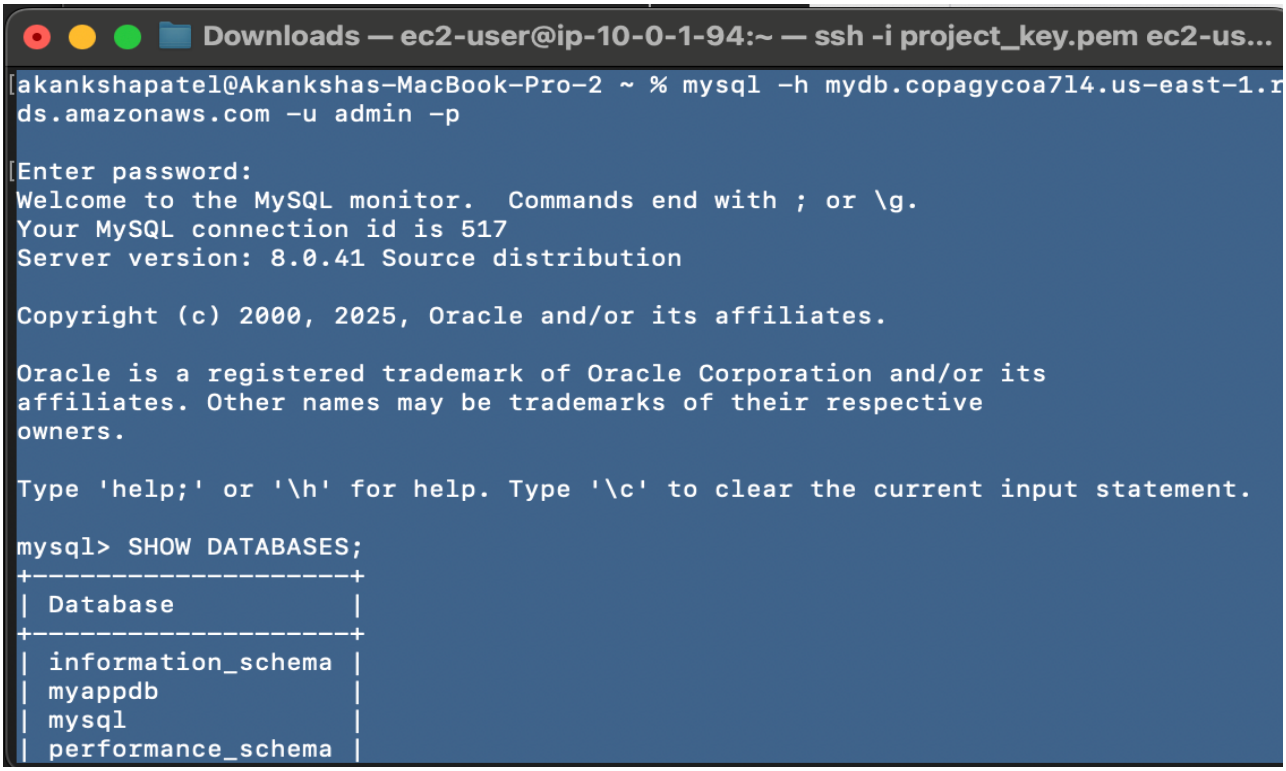
Configure Stack Details



Review & Deploy

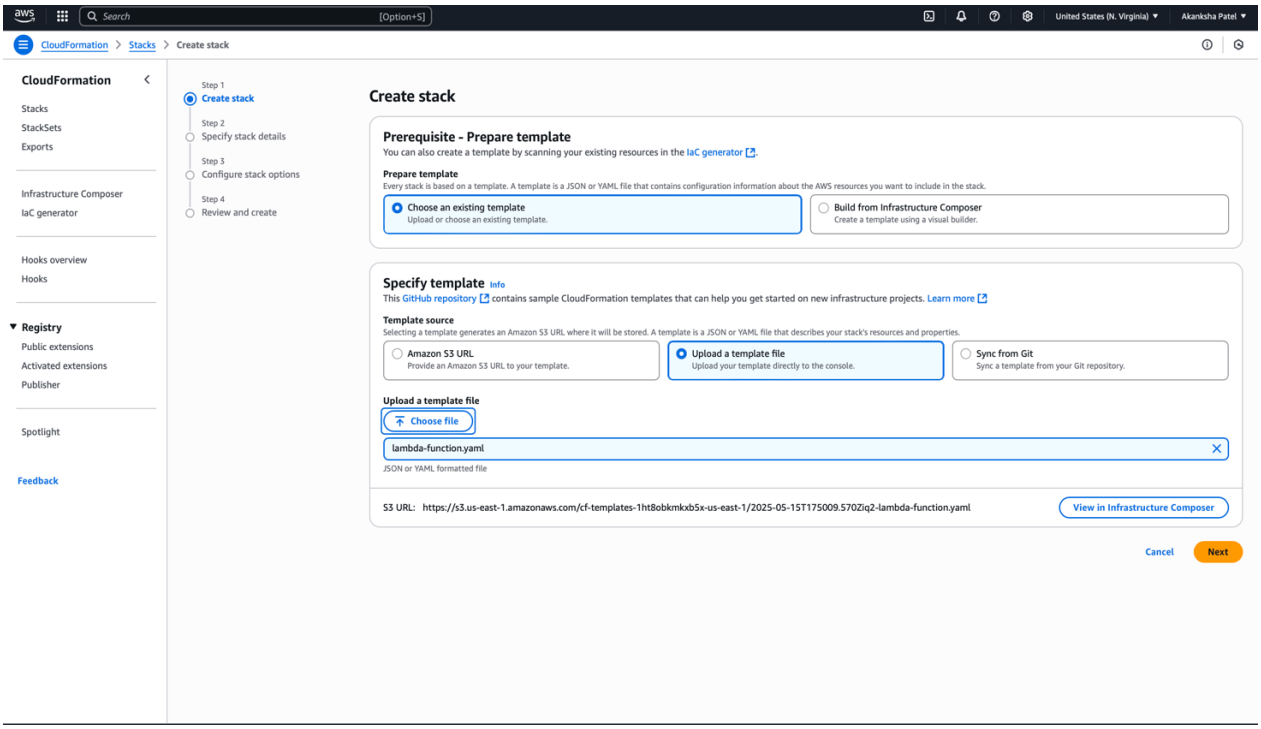


Verify RDS Instance

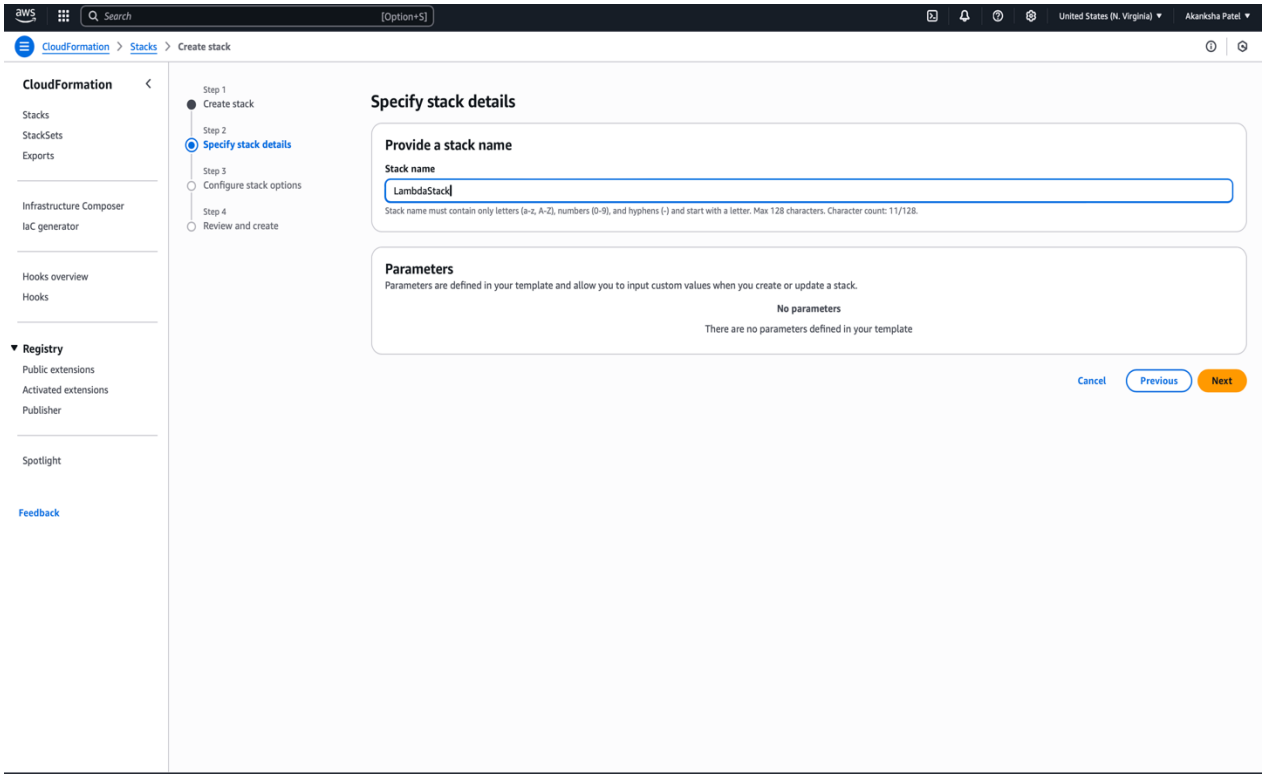


Lambda functions

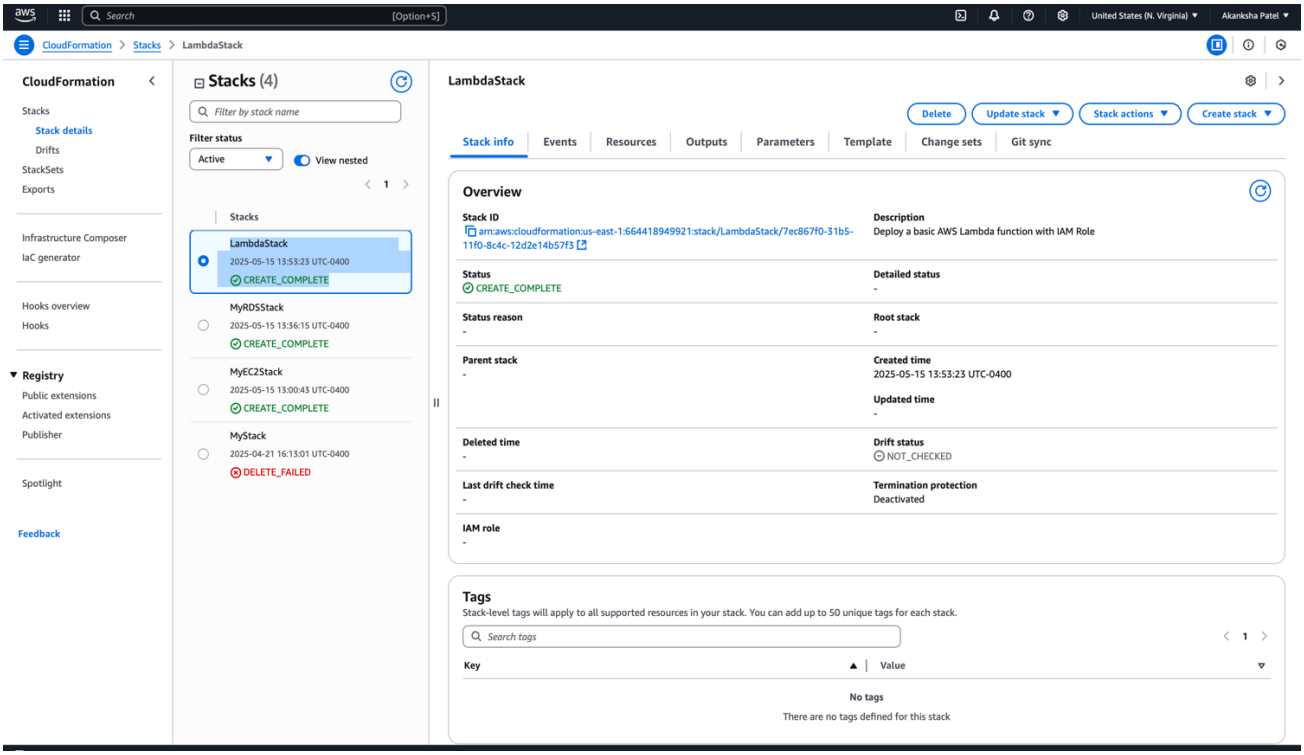
CloudFormation and uploading the Template



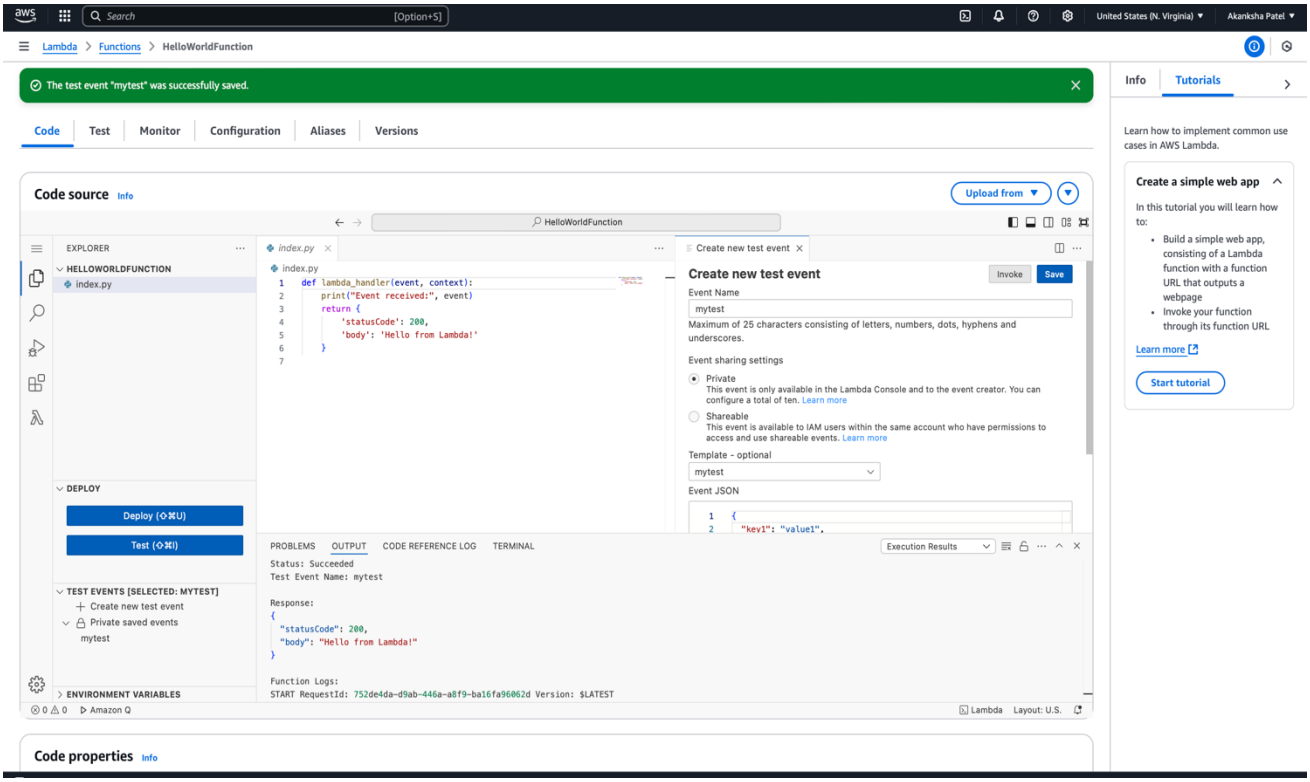
Create Stack

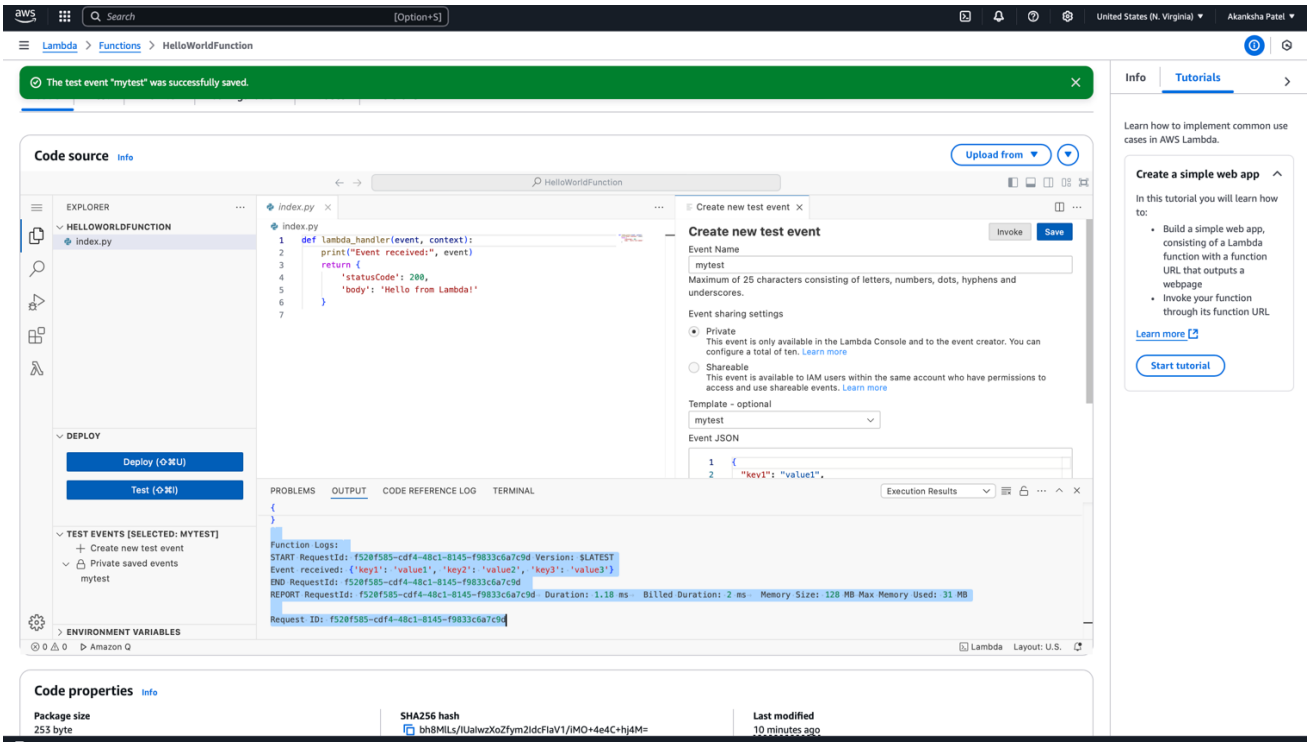


Configure Stack



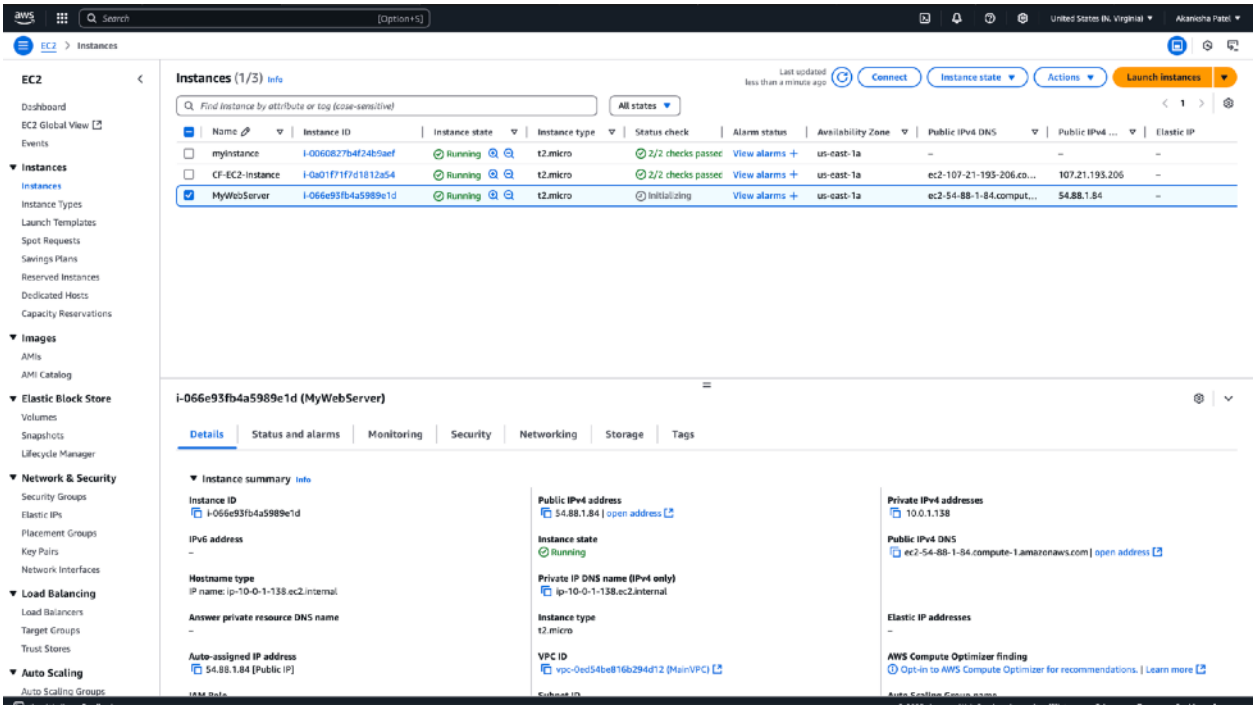
Verify Lambda Function Created



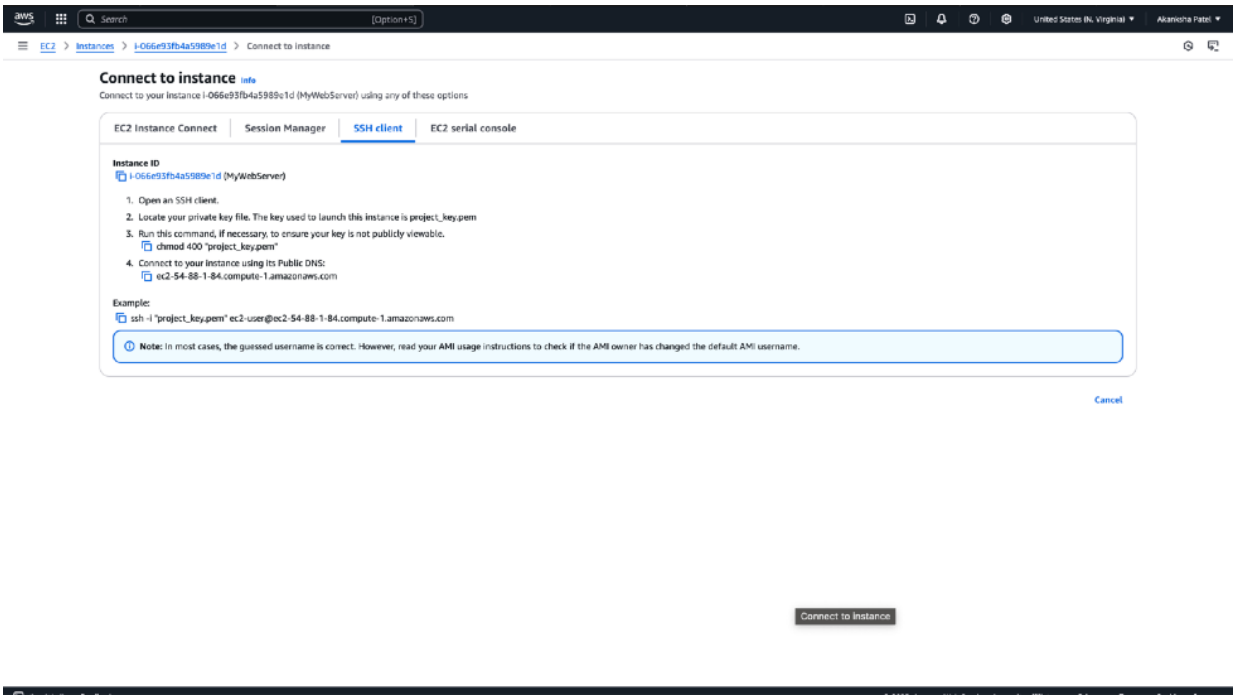


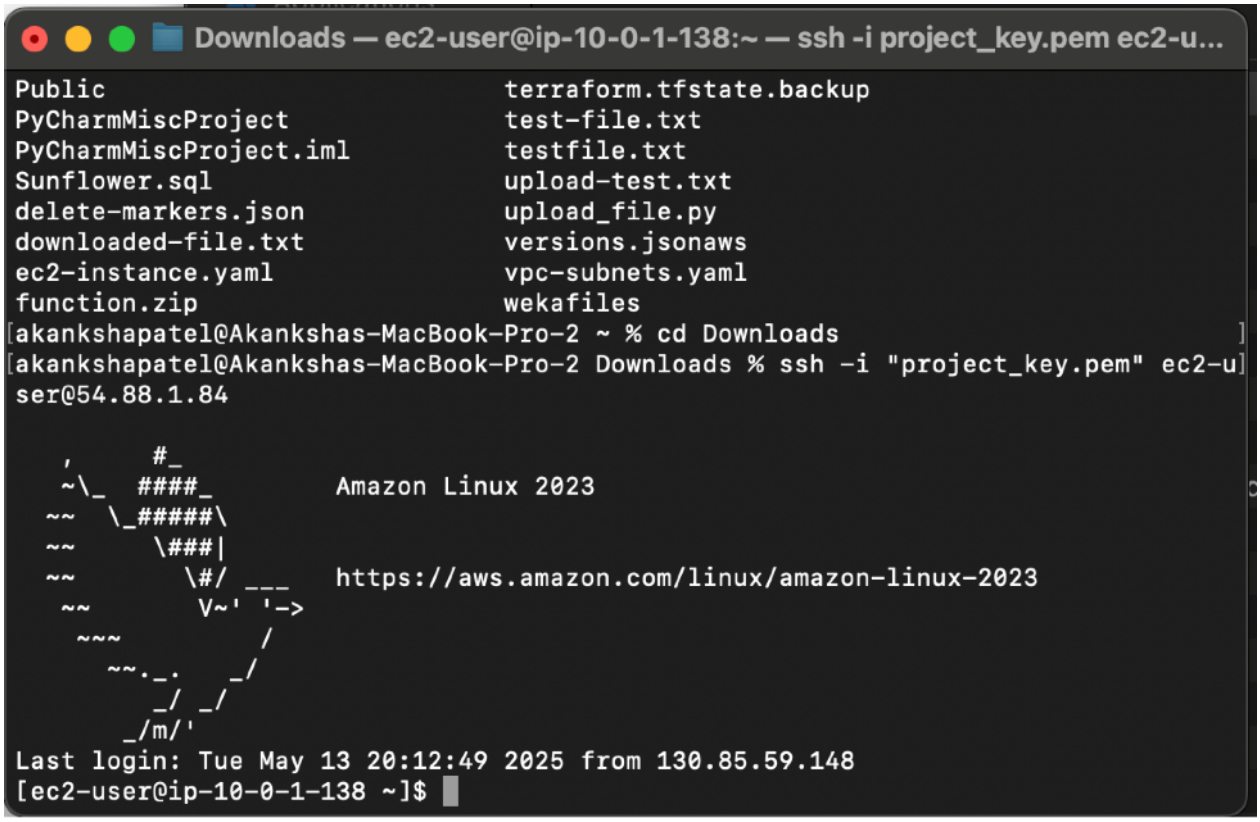
Deploy a web application on EC2 (can be a simple HTML page).

Launched EC2 instance: MyWebServer

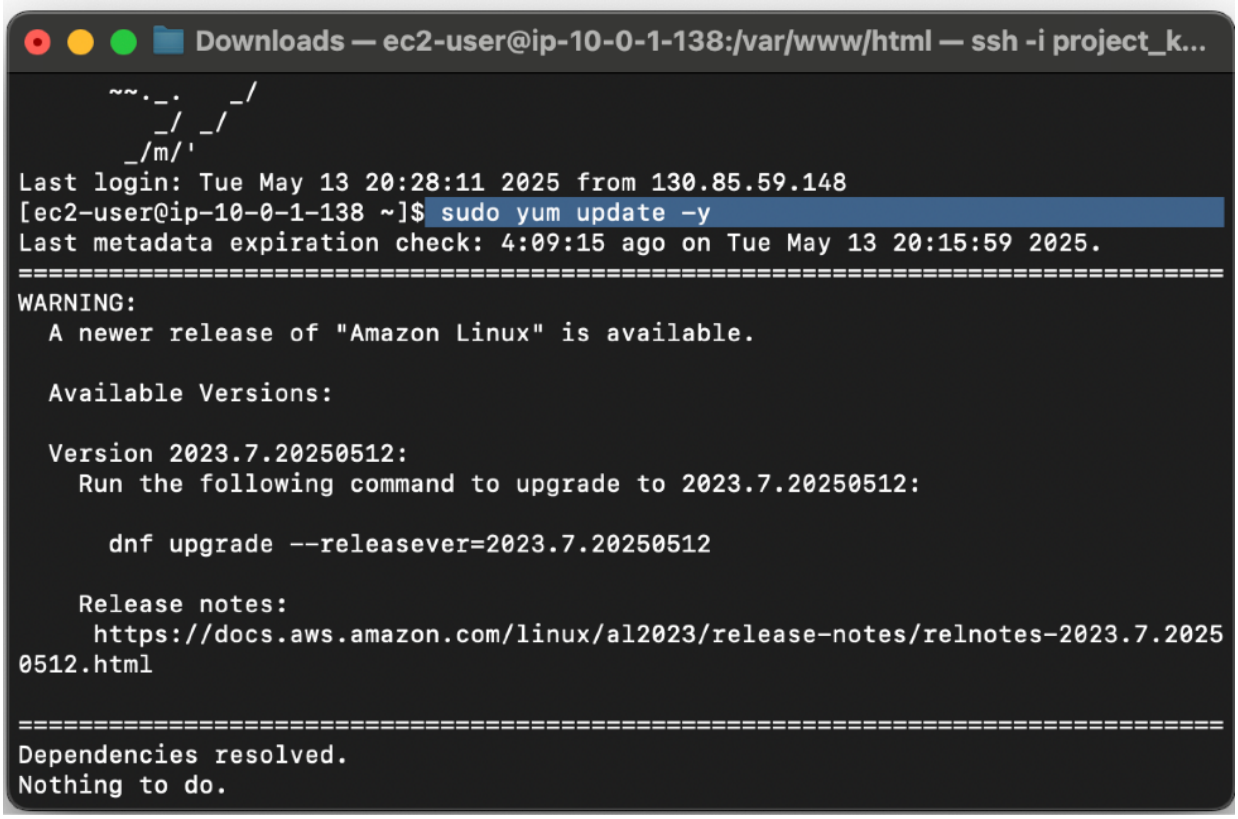


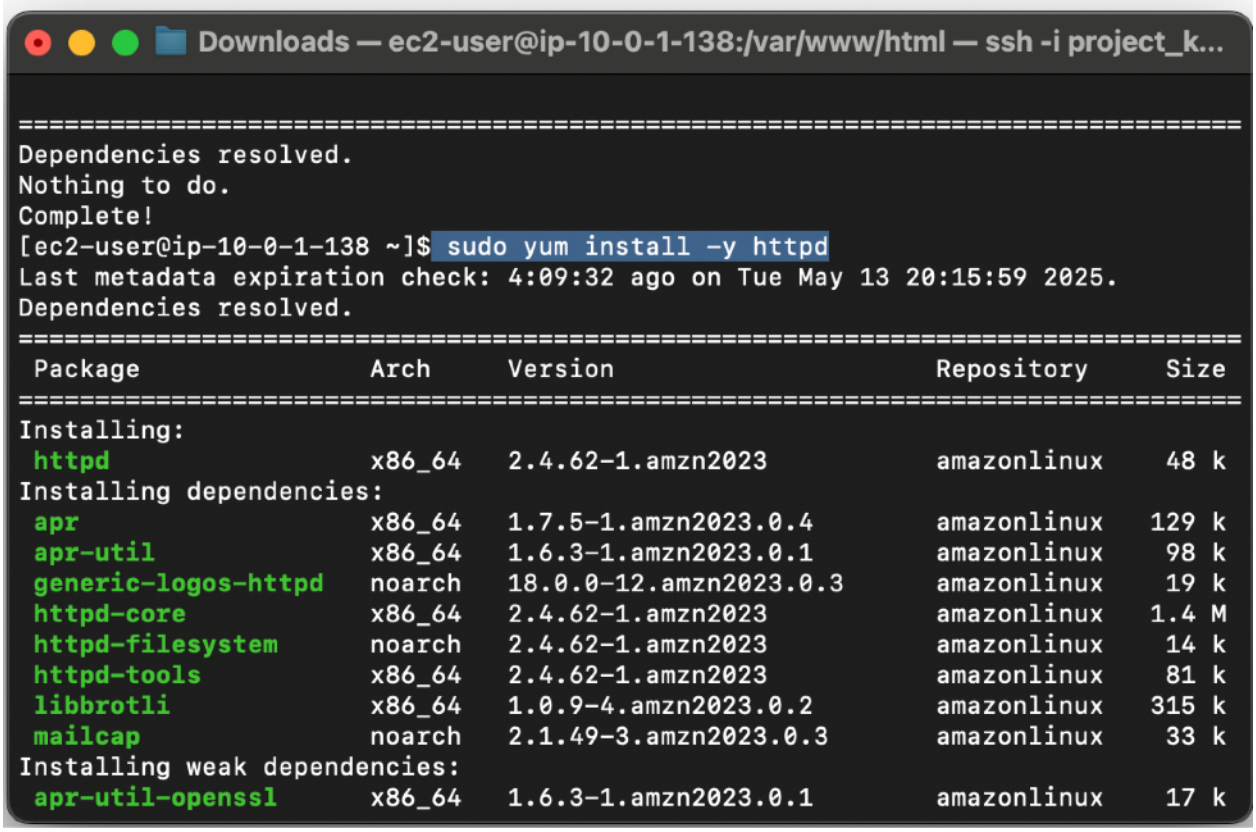
Connect to the EC2 Instance (via SSH):



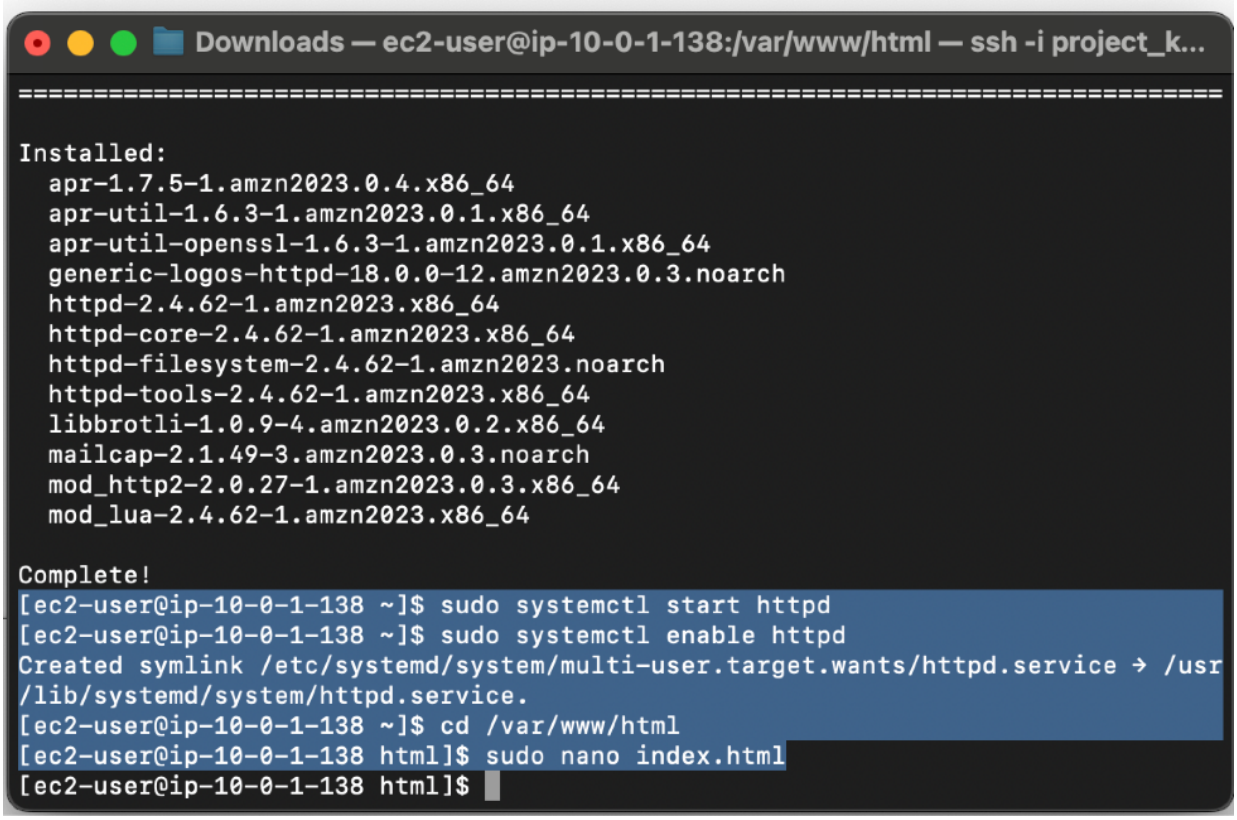


Install Apache Web Server on EC2

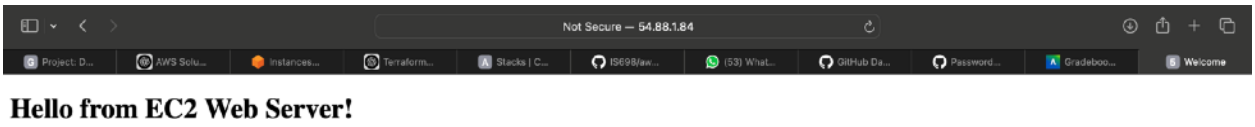




Deploy a Simple HTML Page to Apache



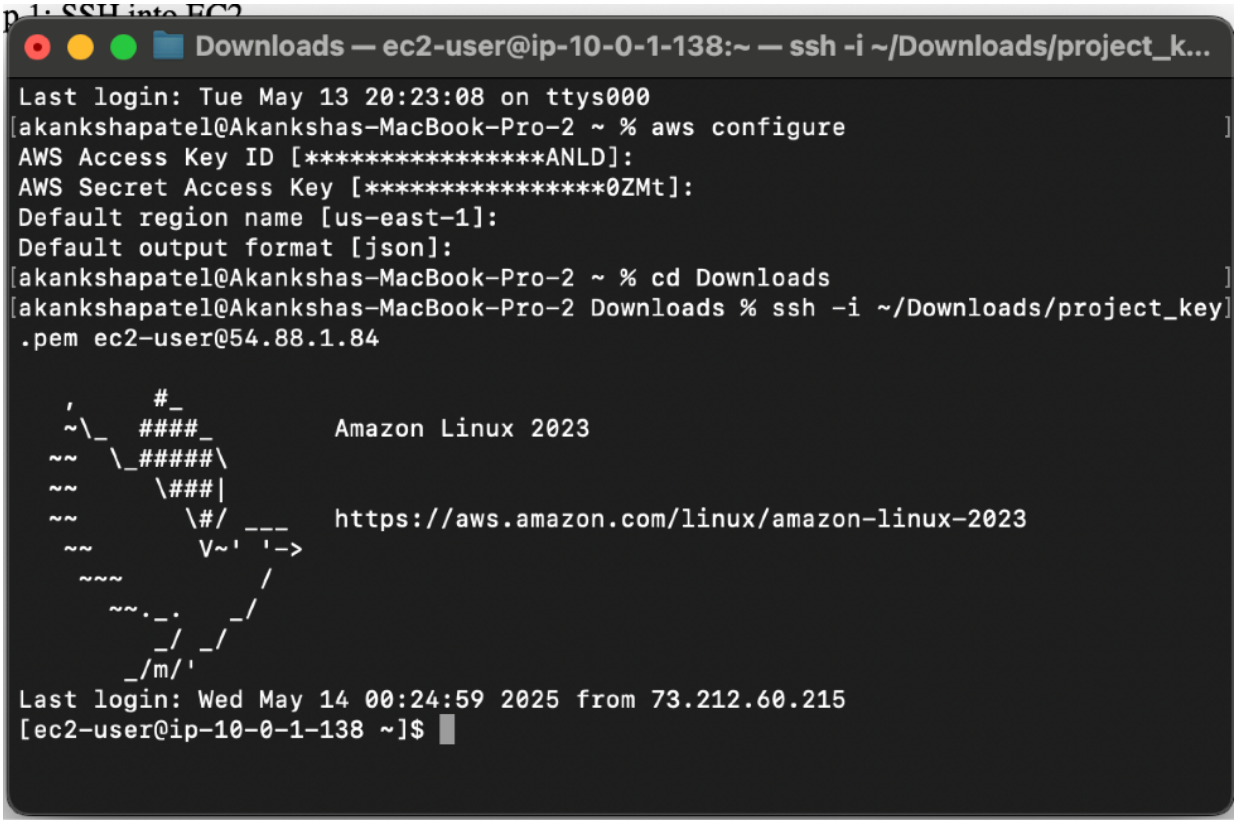
Result



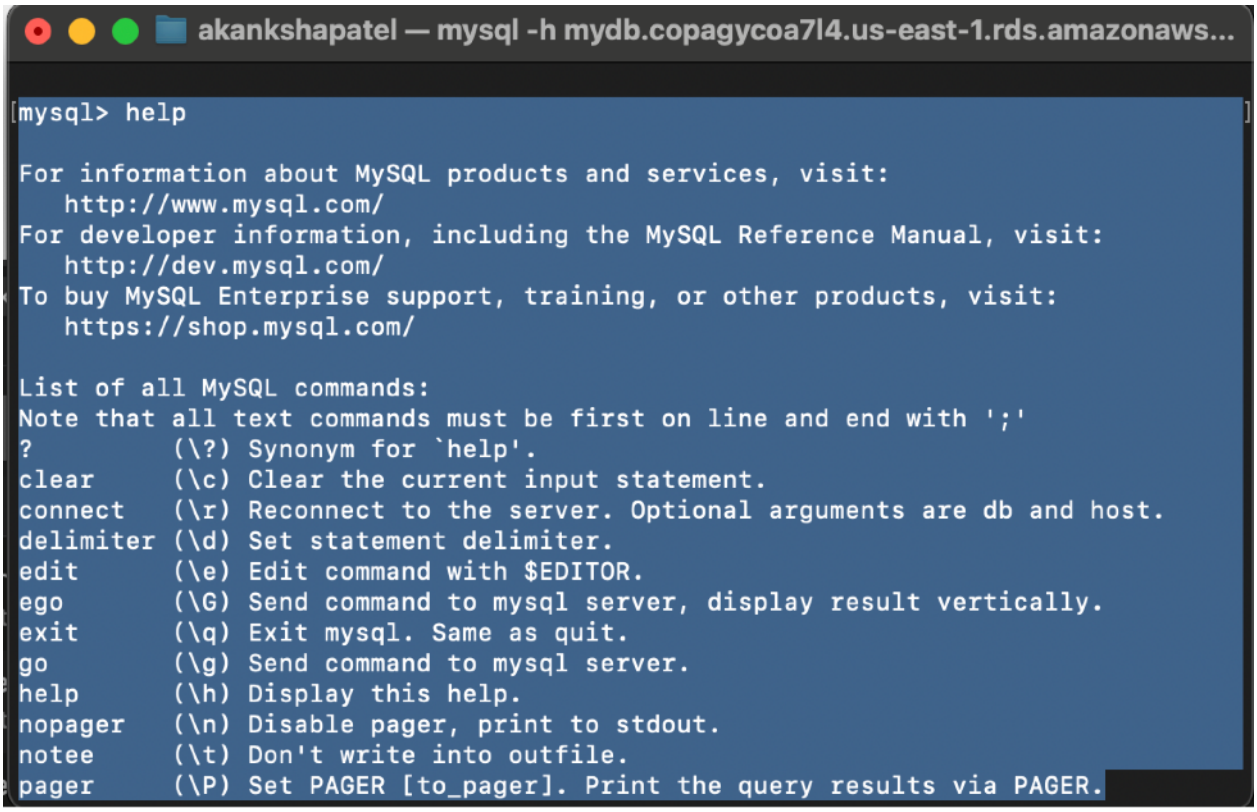
Configure the database for the web application.

Step 1: SSH into EC2

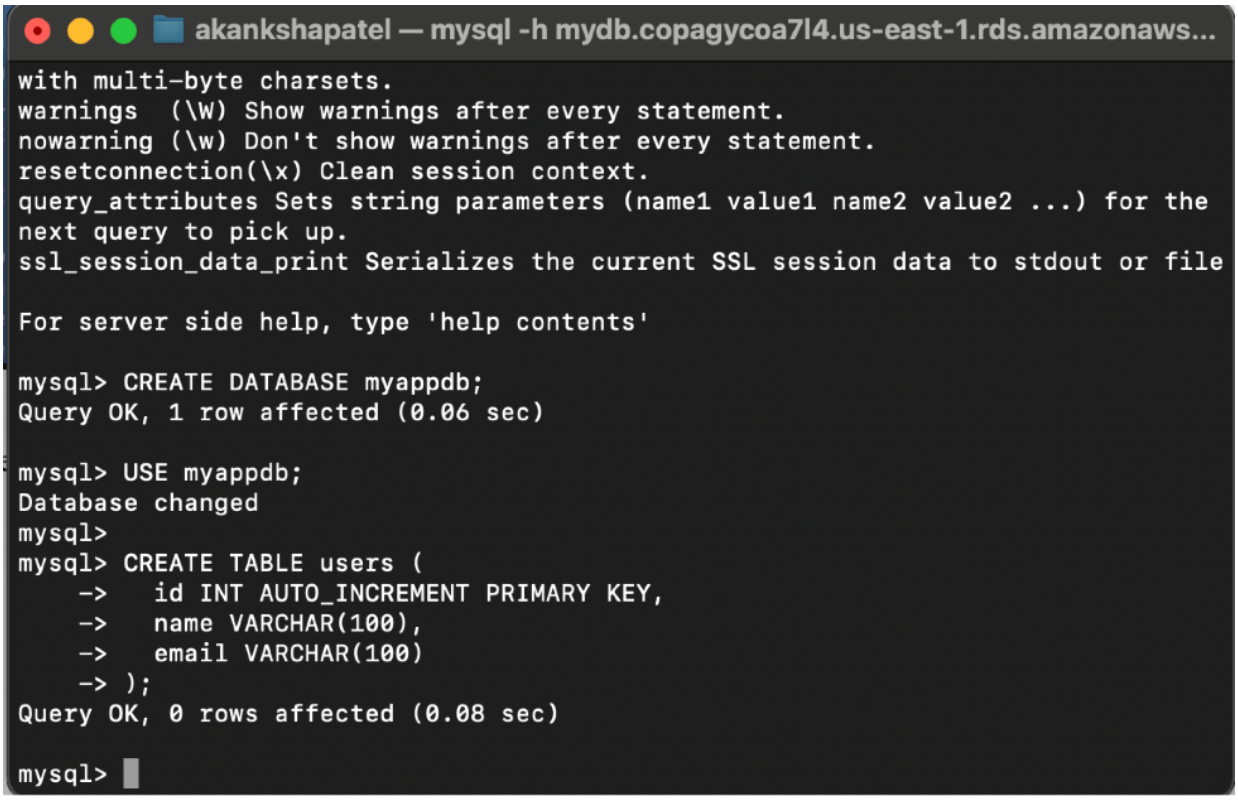
p 1: SSH into EC2



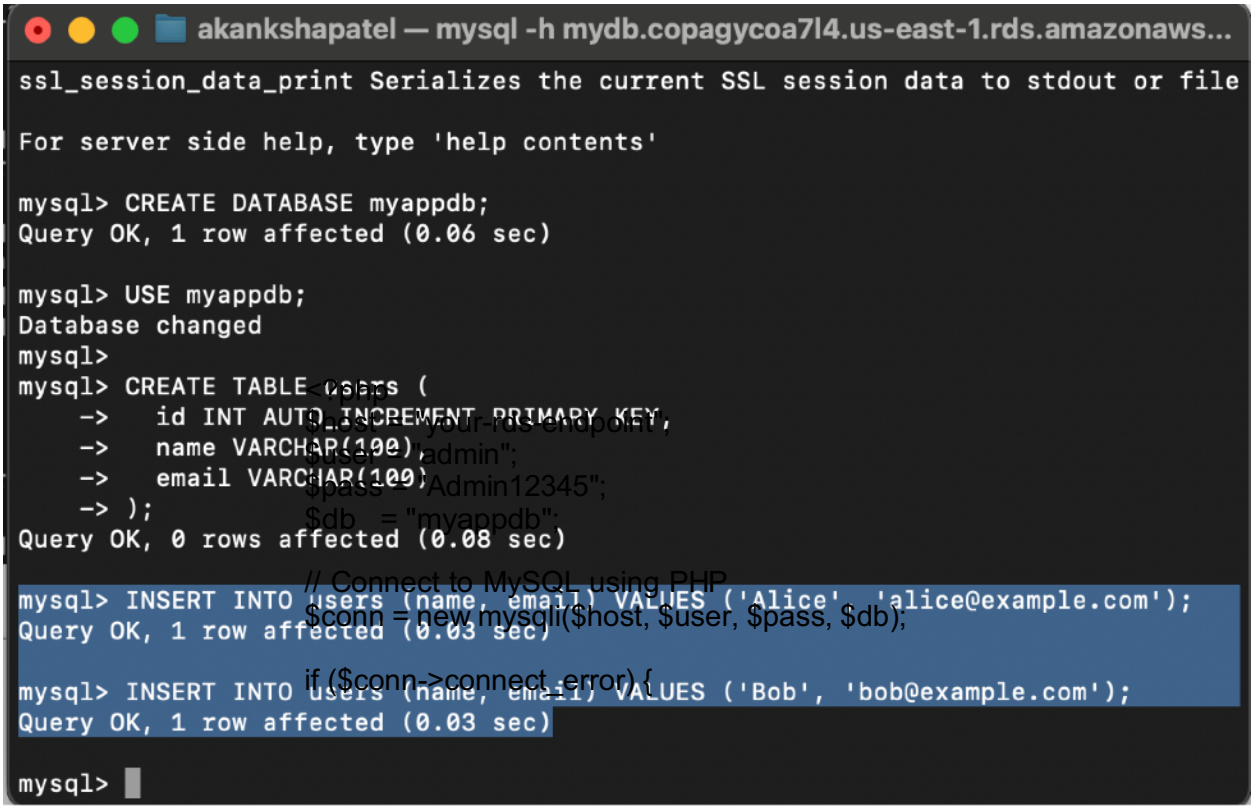
Step 2: Install MySQL client on EC2



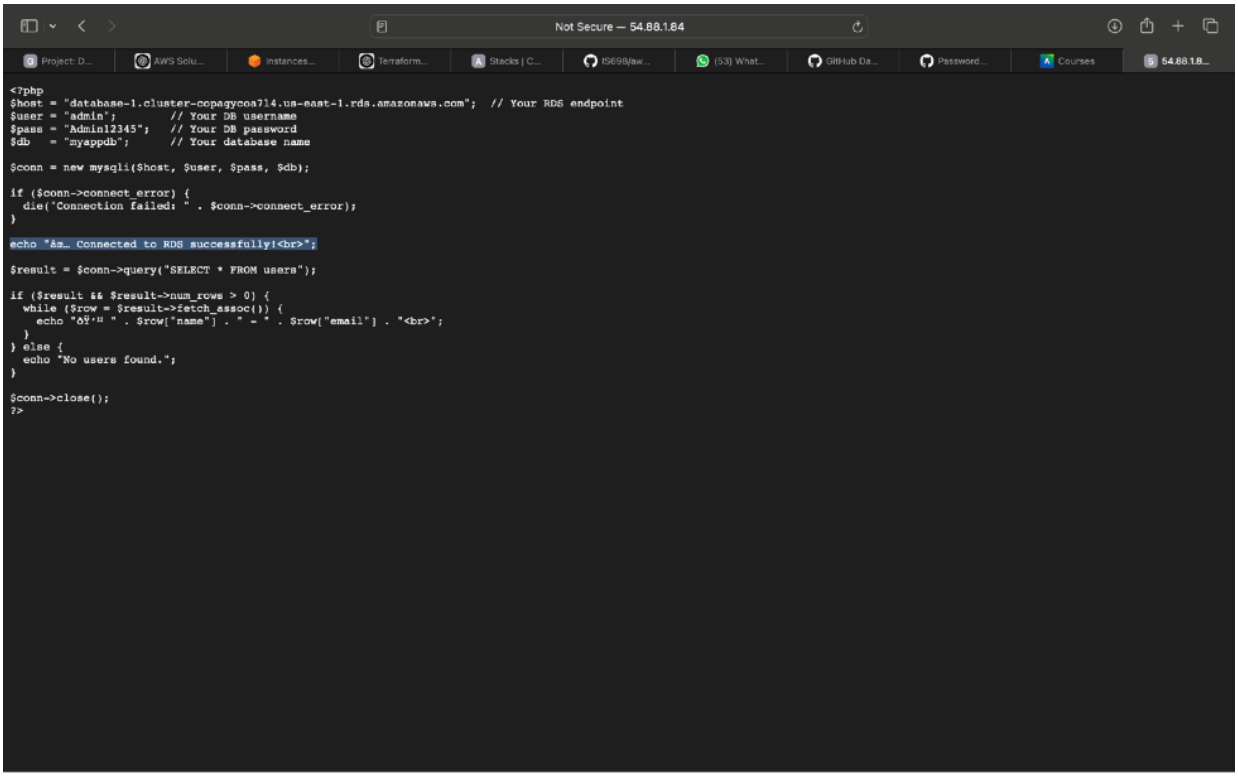
Create the Database and Table



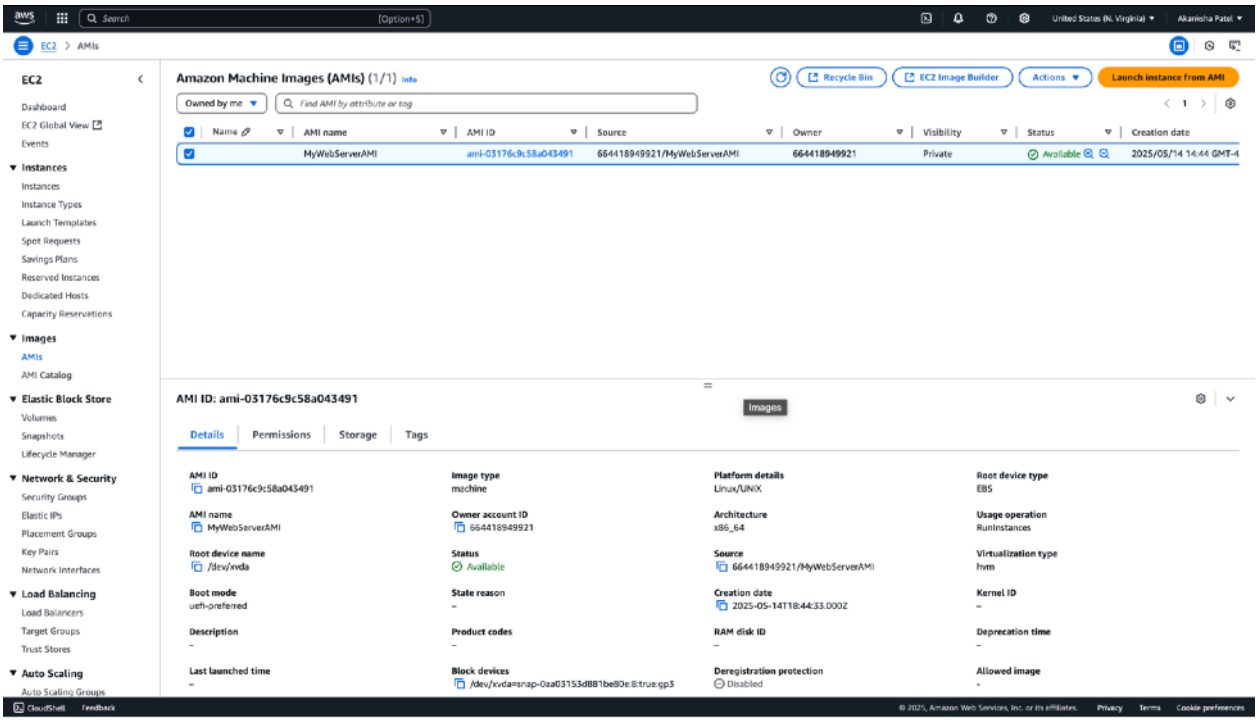
Insert Sample Data



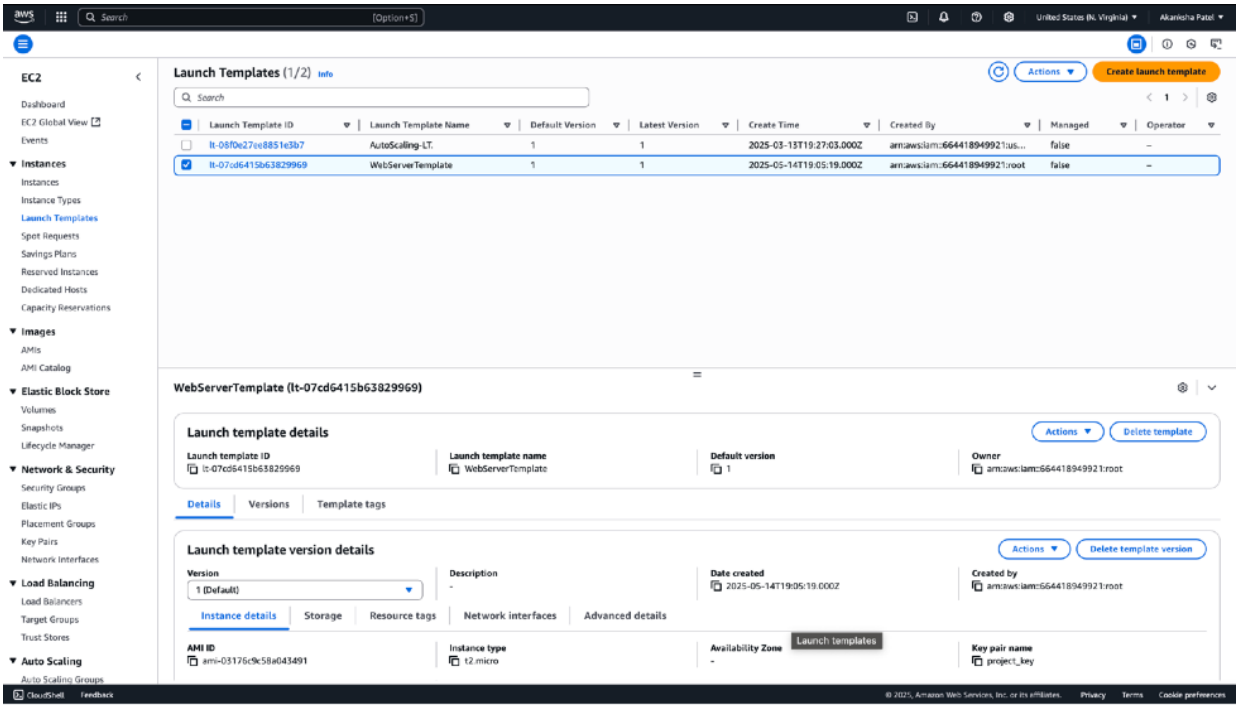
Configure Your EC2 App



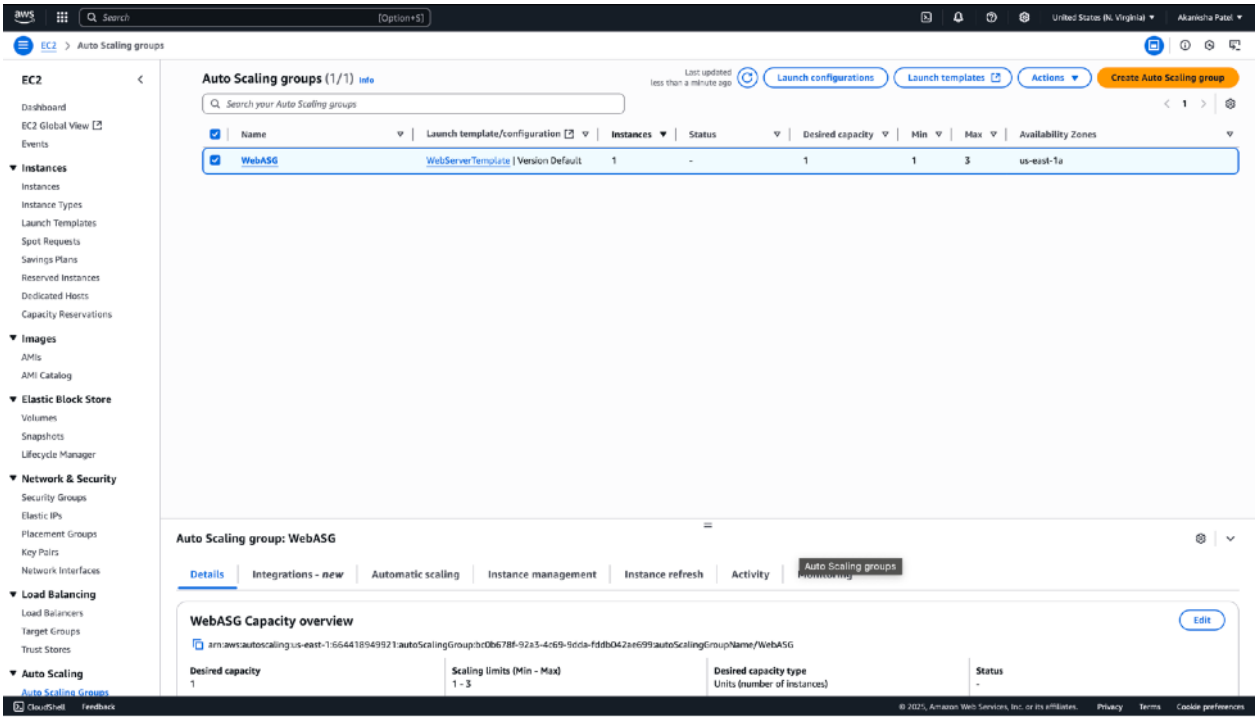
Implement autoscaling to manage web server load.
STEP 1: Create an AMI from Existing EC2



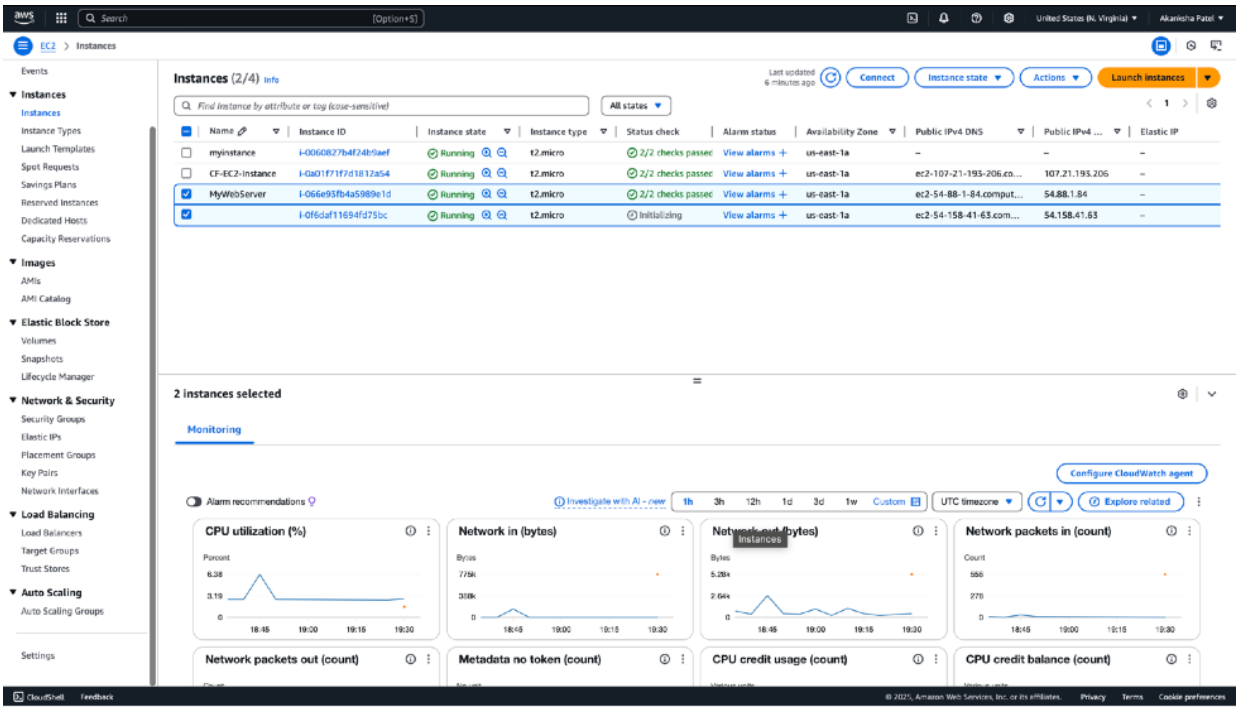
STEP 2: Create a Launch Template



STEP 3: Create an Auto Scaling Group



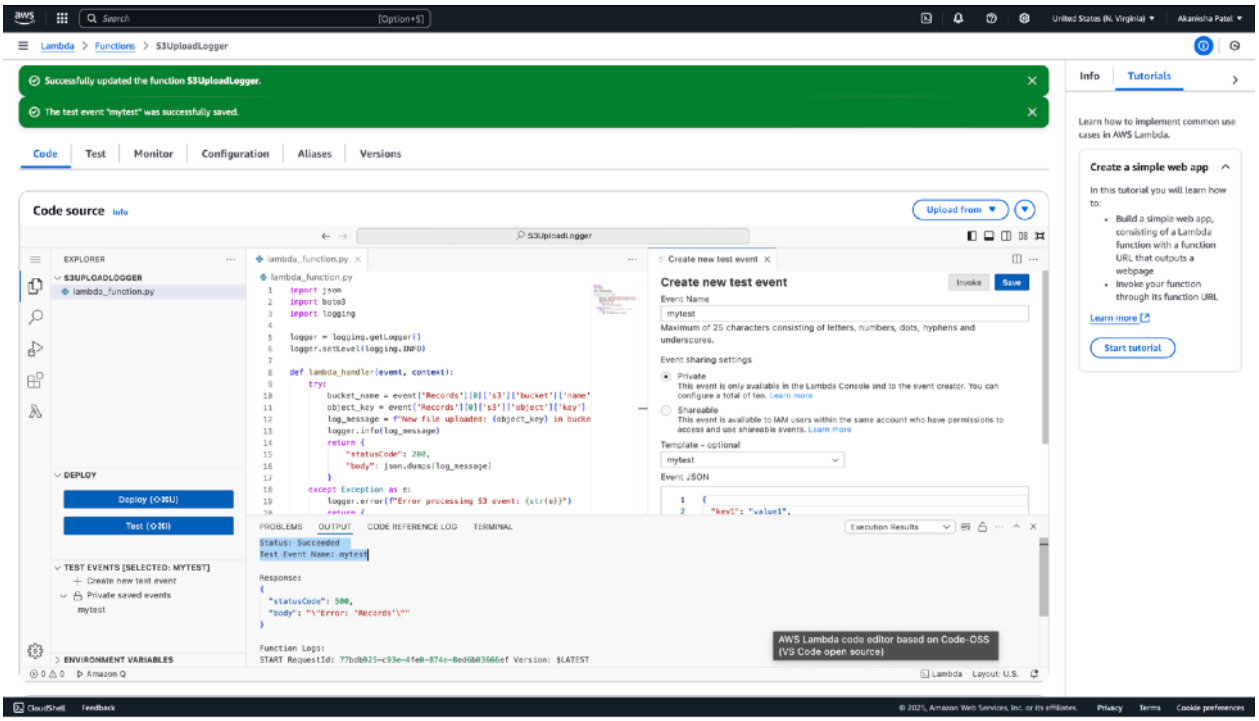
Test



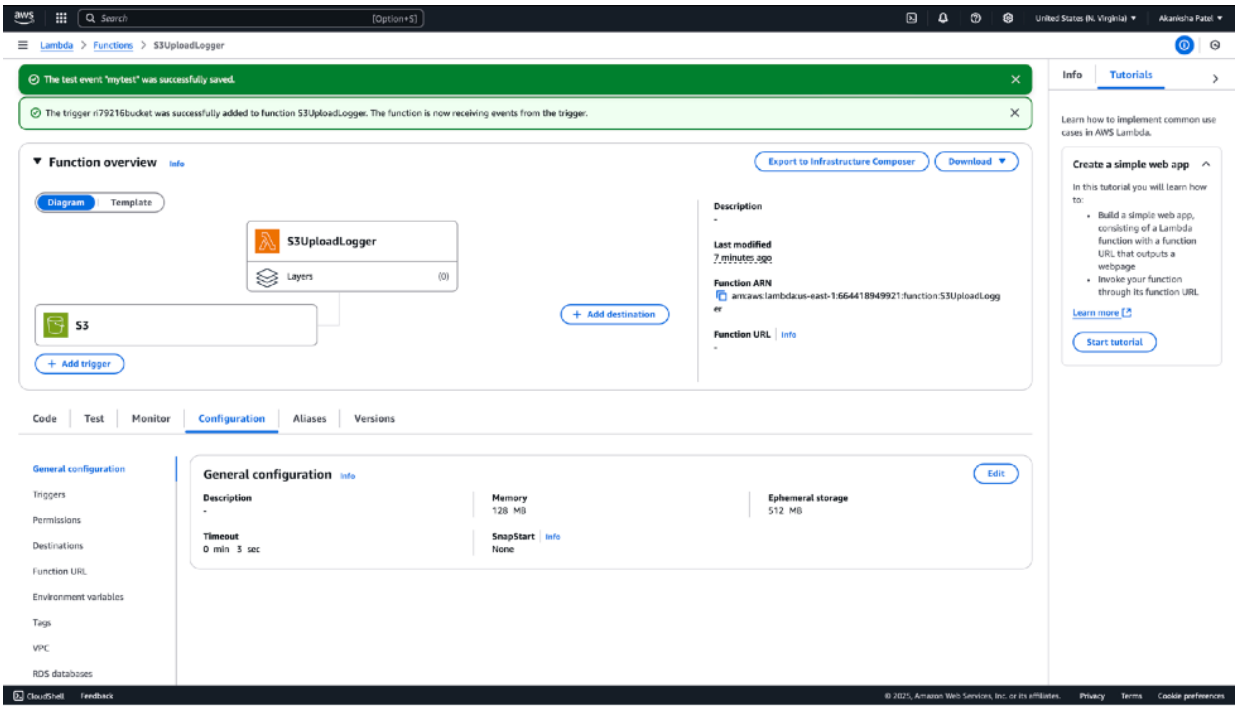
B. AWS Lambda for Logging S3 Uploads

Create an AWS Lambda function that logs S3 file uploads to CloudWatch Logs.

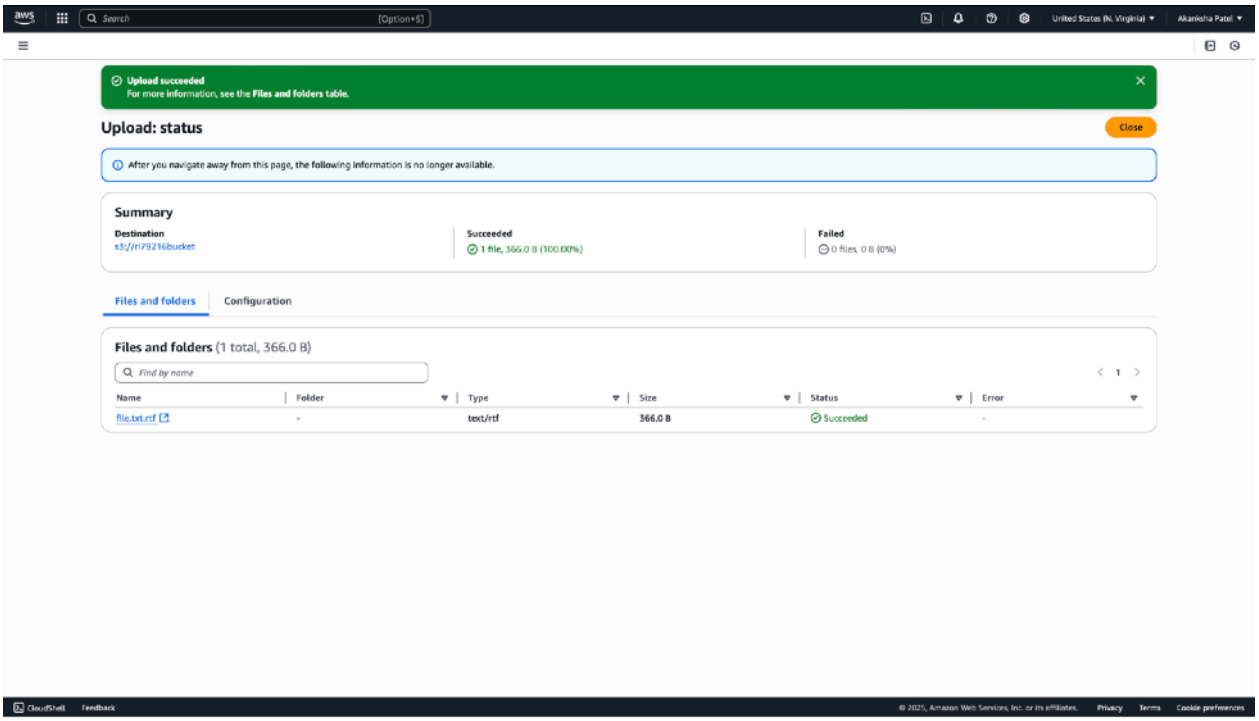
Step 1: Create the Lambda Function



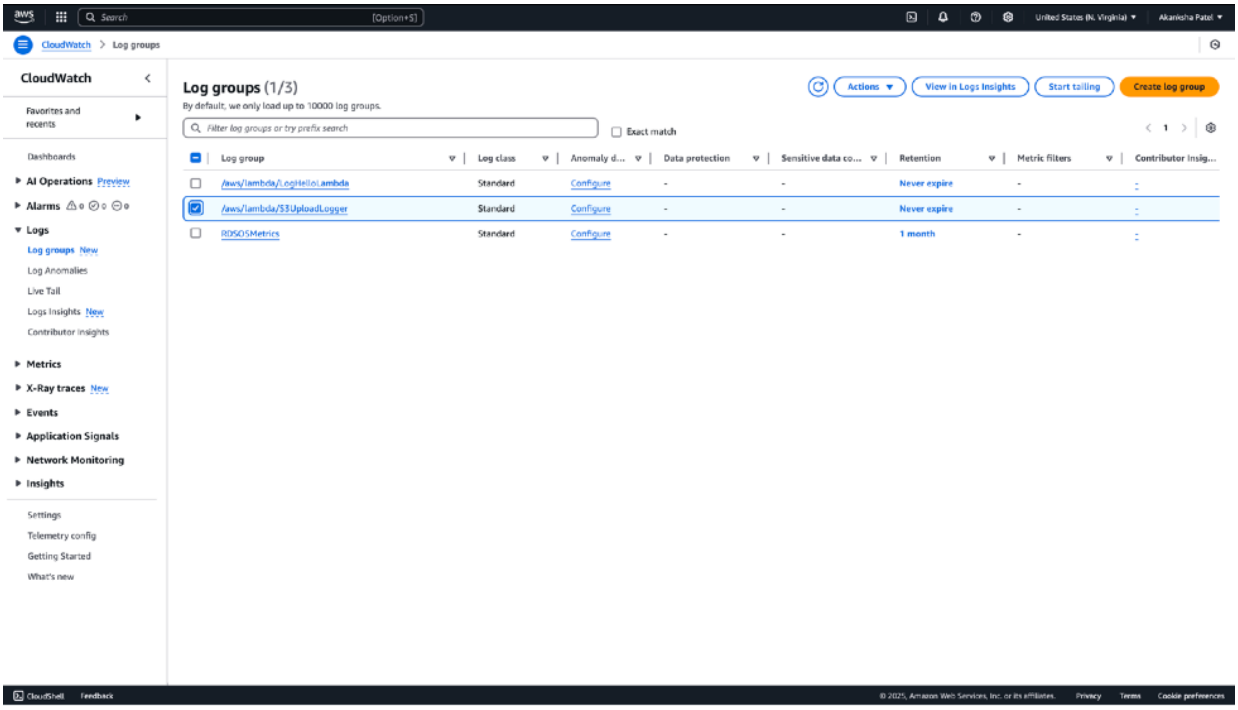
Add S3 as a Trigger

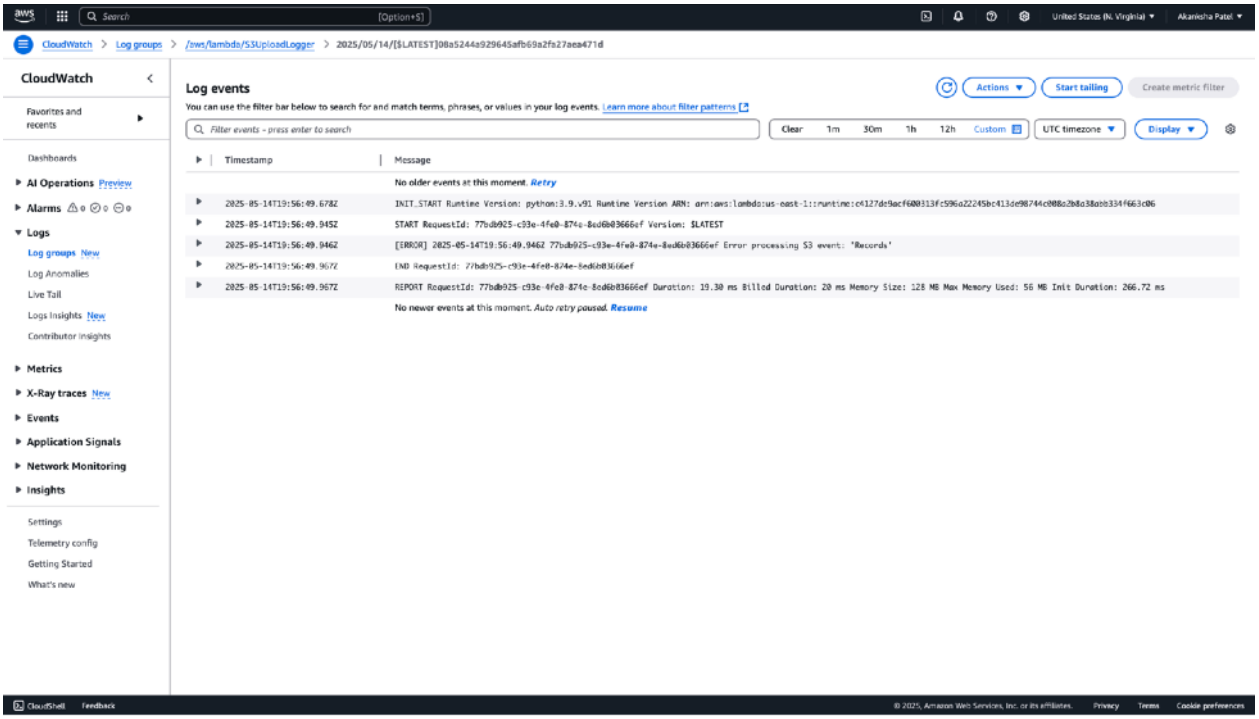


Upload a File to S3



View the Logs in CloudWatch

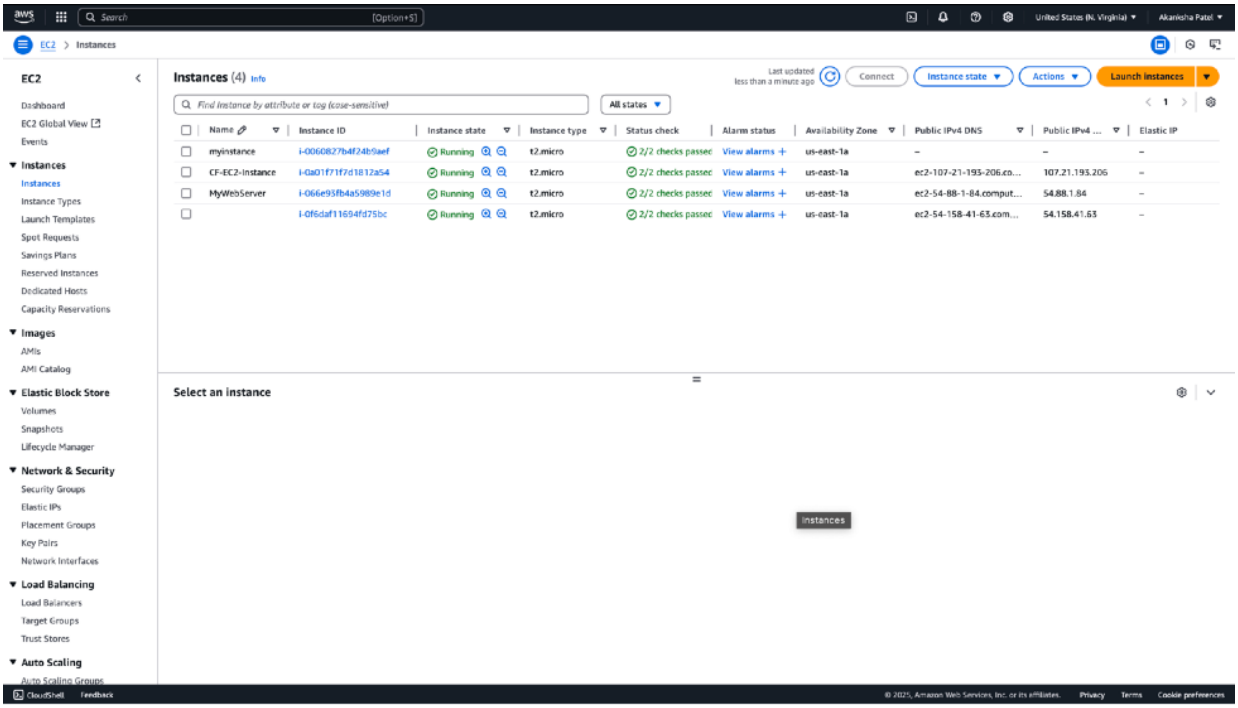




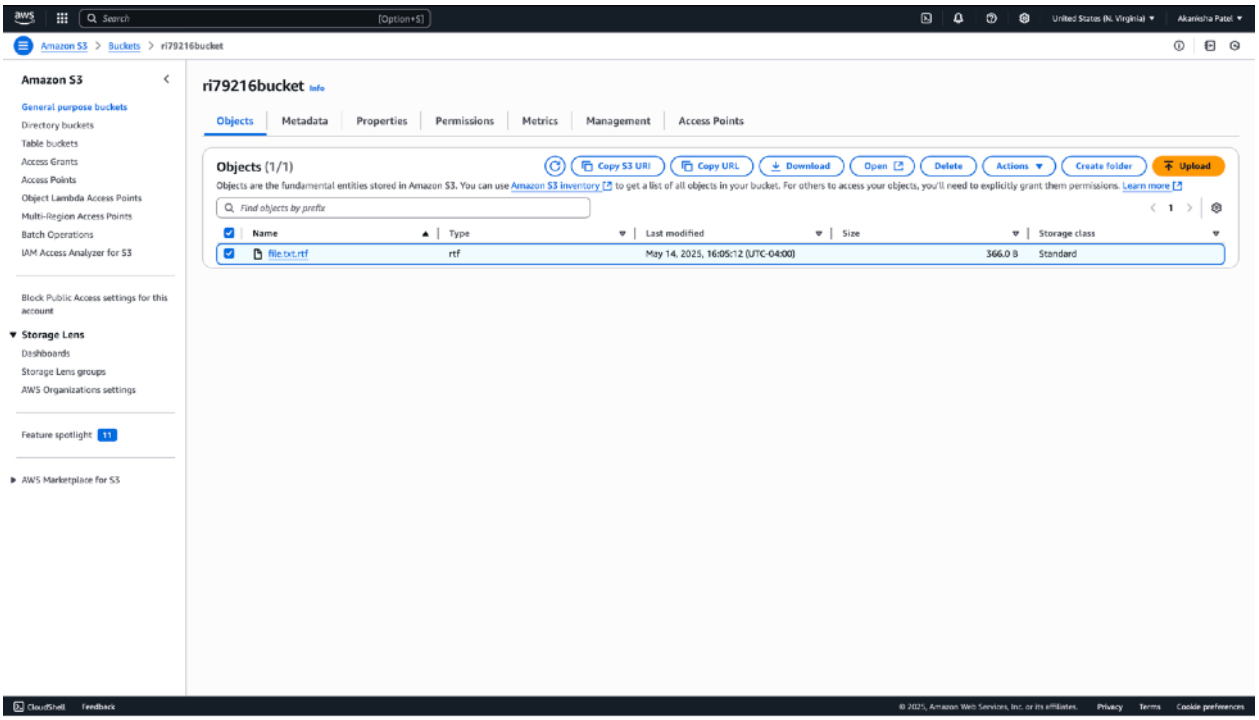
C. Interaction with AWS

Use AWS Console to verify infrastructure deployment.

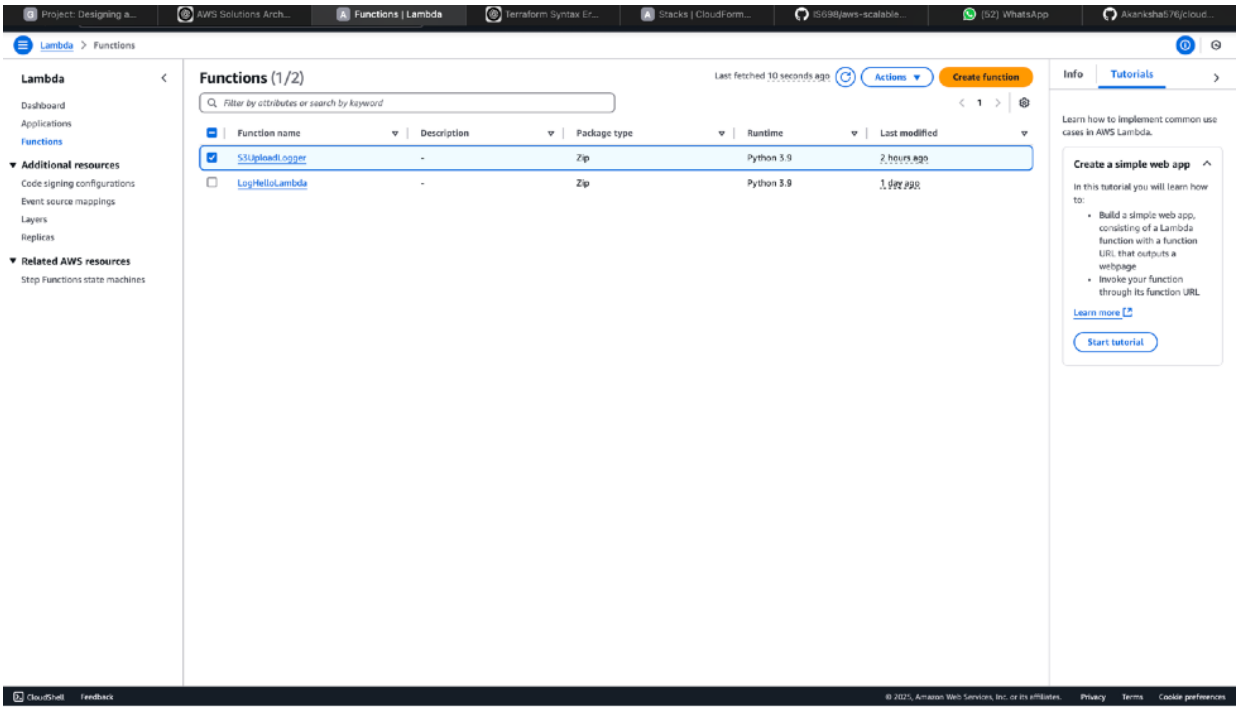
STEP 1: Verify EC2 Web Server



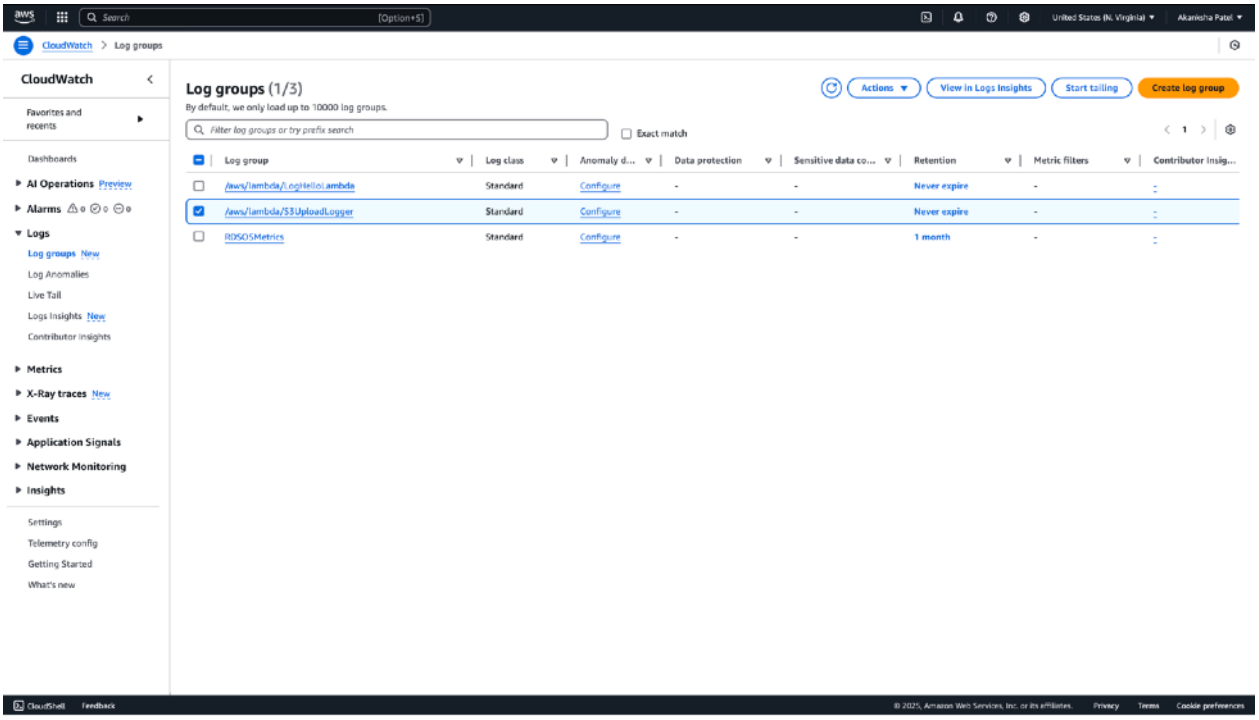
STEP 2: Verify S3 Bucket and File Uploads



STEP 3: Verify Lambda Function and Logs

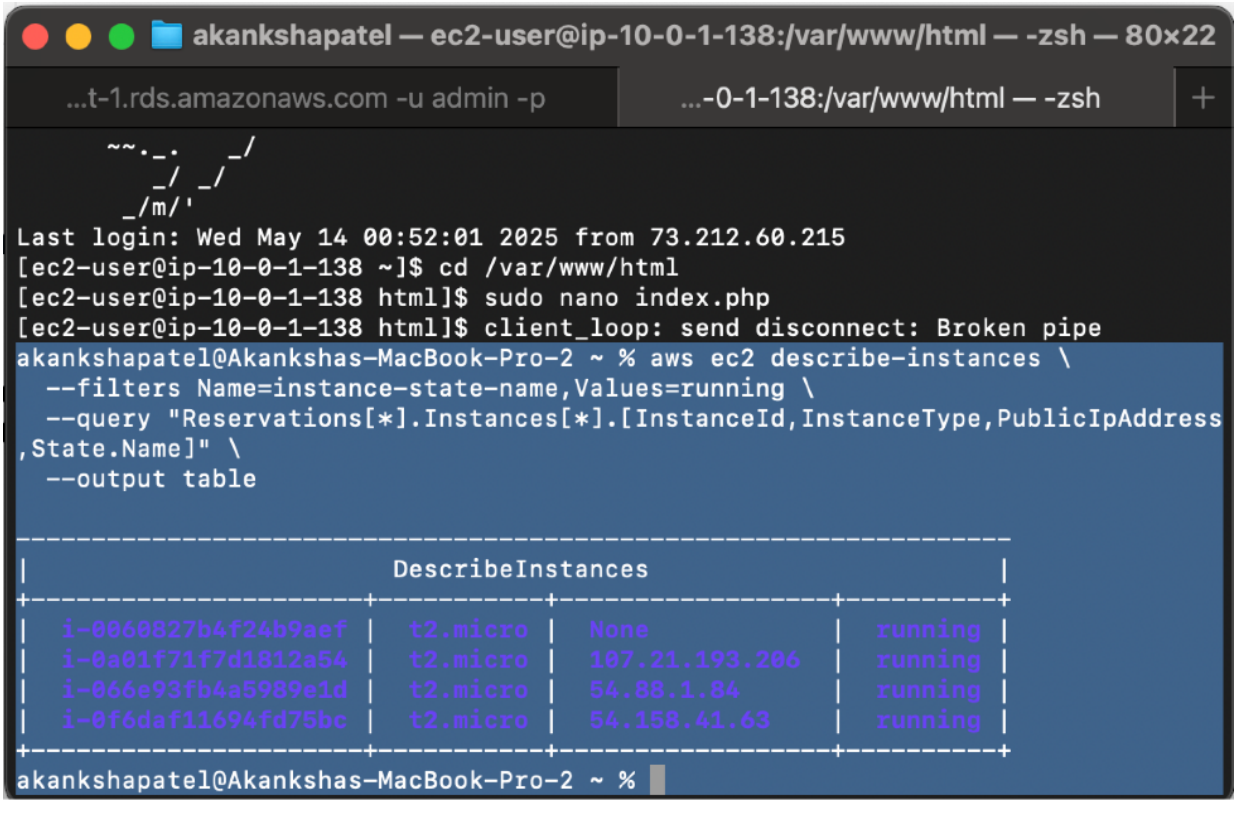


STEP 4: Verify Logs in CloudWatch

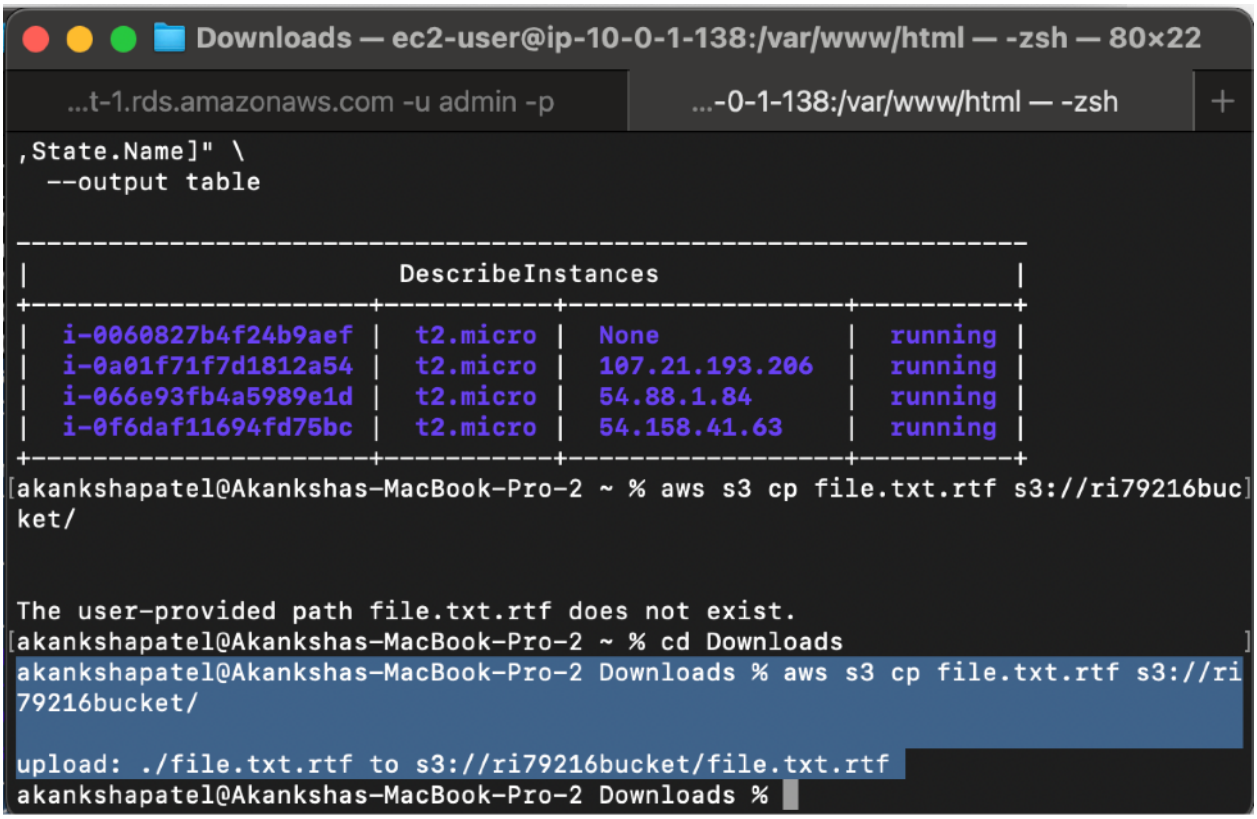


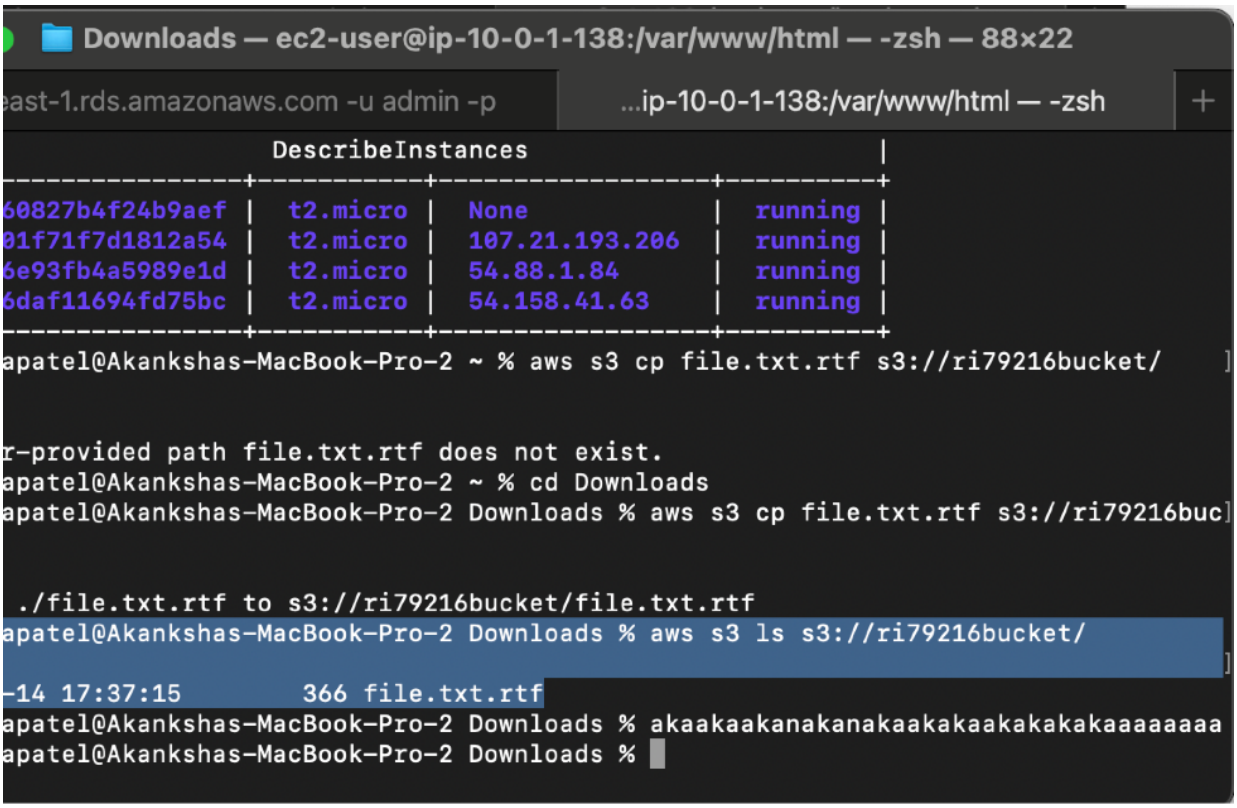
Use AWS CLI to interact with EC2, S3, and Lambda.

Step 1: EC2: List Running Instances



Step2: S3: Upload a File to a Bucket



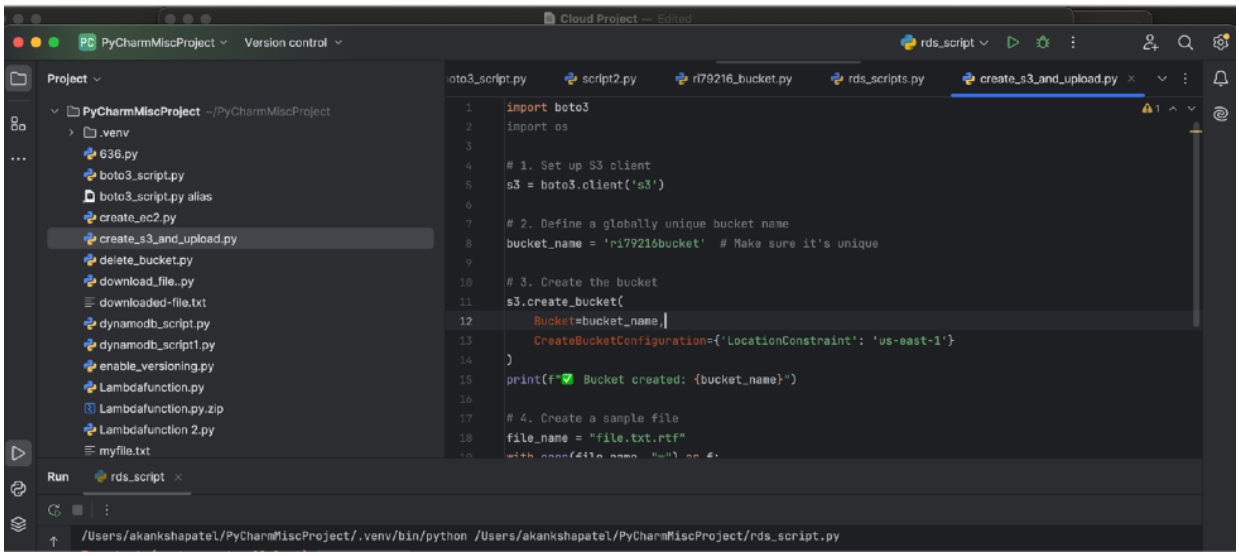


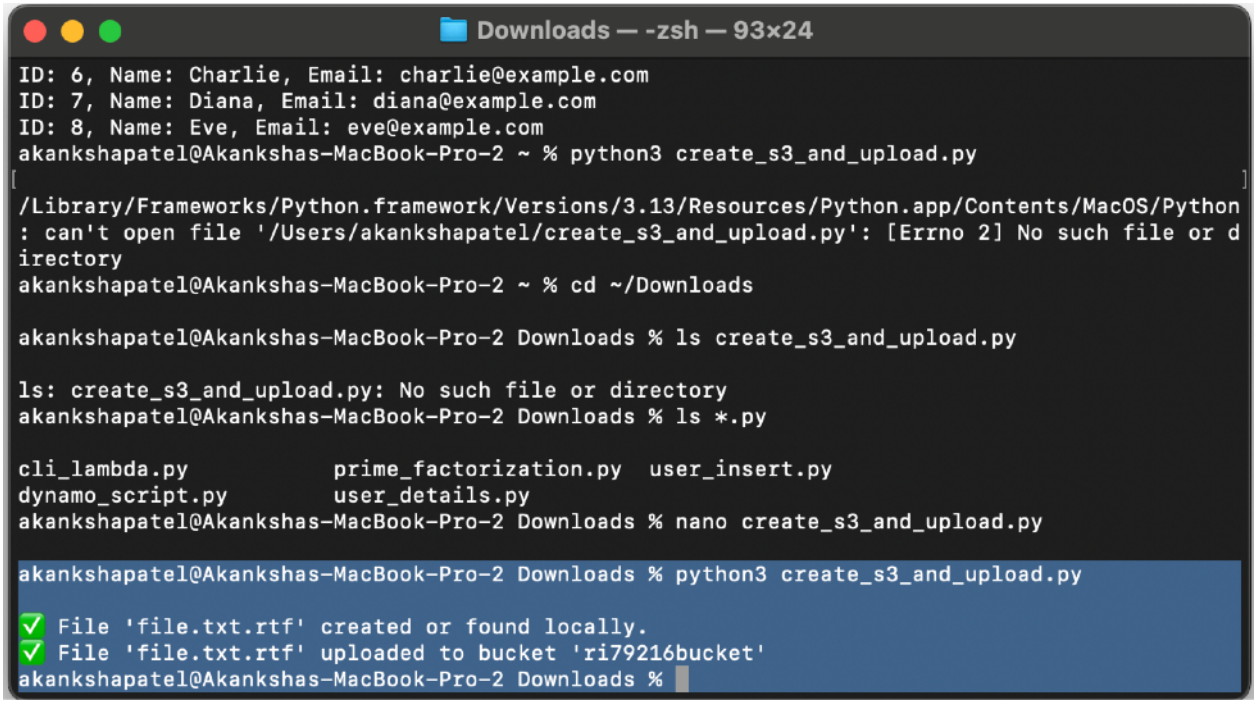
Step 3: Lambda

Downloads — ec2-user@ip-10-0-1-138:~ — ssh -i ~/Downloads/project_key.pem ec2-user@54.8...

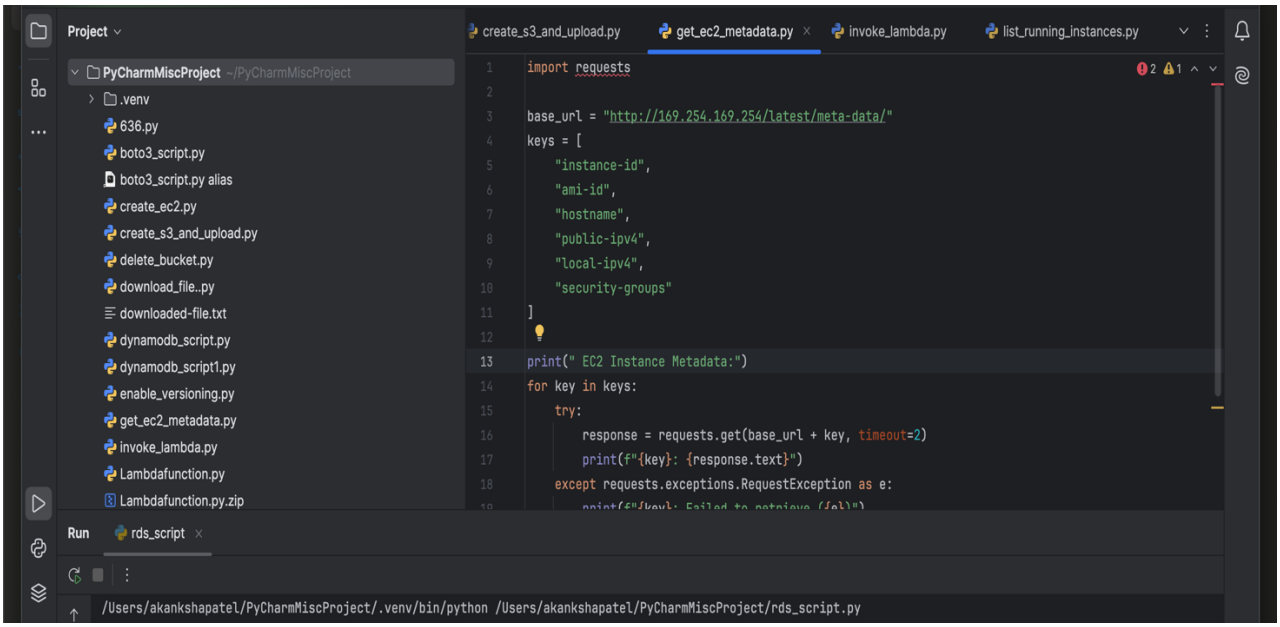
```
[ec2-user@ip-10-0-1-138 ~]$ aws lambda list-functions
{
    "Functions": [
        {
            "FunctionName": "S3UploadLogger",
            "FunctionArn": "arn:aws:lambda:us-east-1:664418949921:function:S3UploadLogger",
            "Runtime": "python3.9",
            "Role": "arn:aws:iam::664418949921:role/service-role/S3UploadLogger-role-xims4k68",
            "Handler": "lambda_function.lambda_handler",
            "CodeSize": 454,
            "Description": "",
            "Timeout": 3,
            "MemorySize": 128,
            "LastModified": "2025-05-14T19:52:06.000+0000",
            "CodeSha256": "QUJ5FXLQcda8Pynjx3WTxj+YPOzT/XJvmp9YE9xRuZY=",
            "Version": "$LATEST",
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "RevisionId": "8474f9fd-b537-47a0-bb65-4661cf43436d",
            "PackageType": "Zip",
            "Architectures": [
                "x86_64"
            ],
```

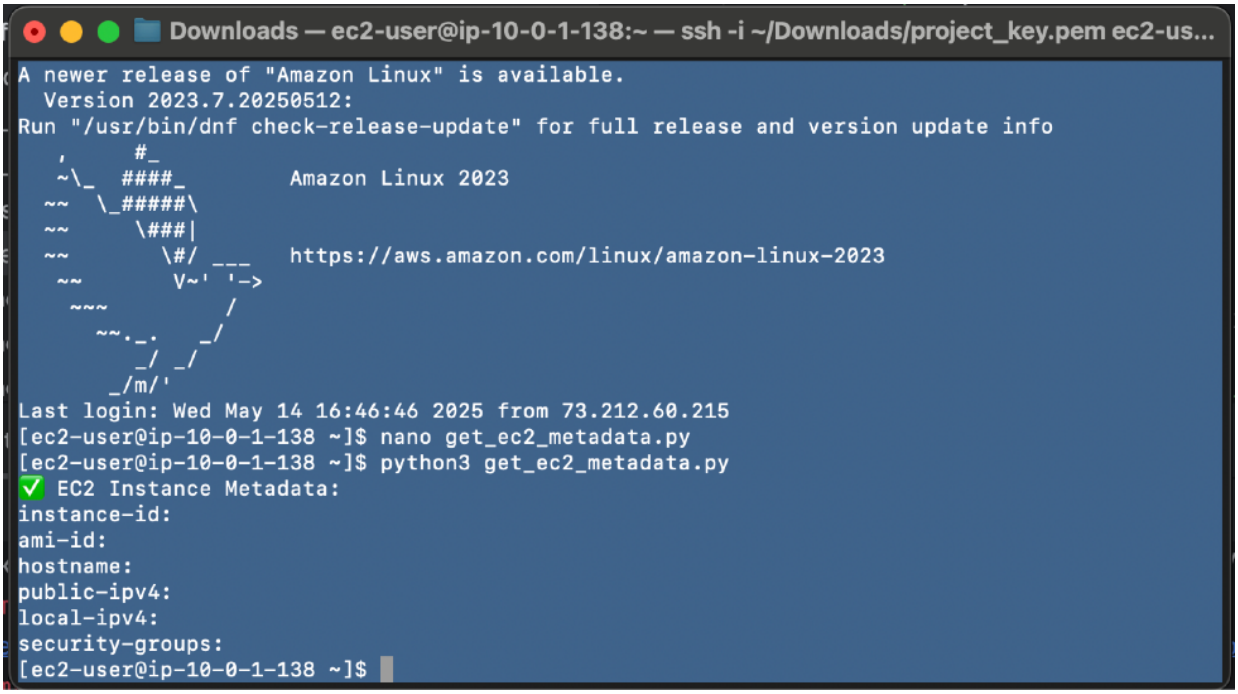
Write Python scripts using Boto3 to:
1. Create an S3 bucket and upload a file.



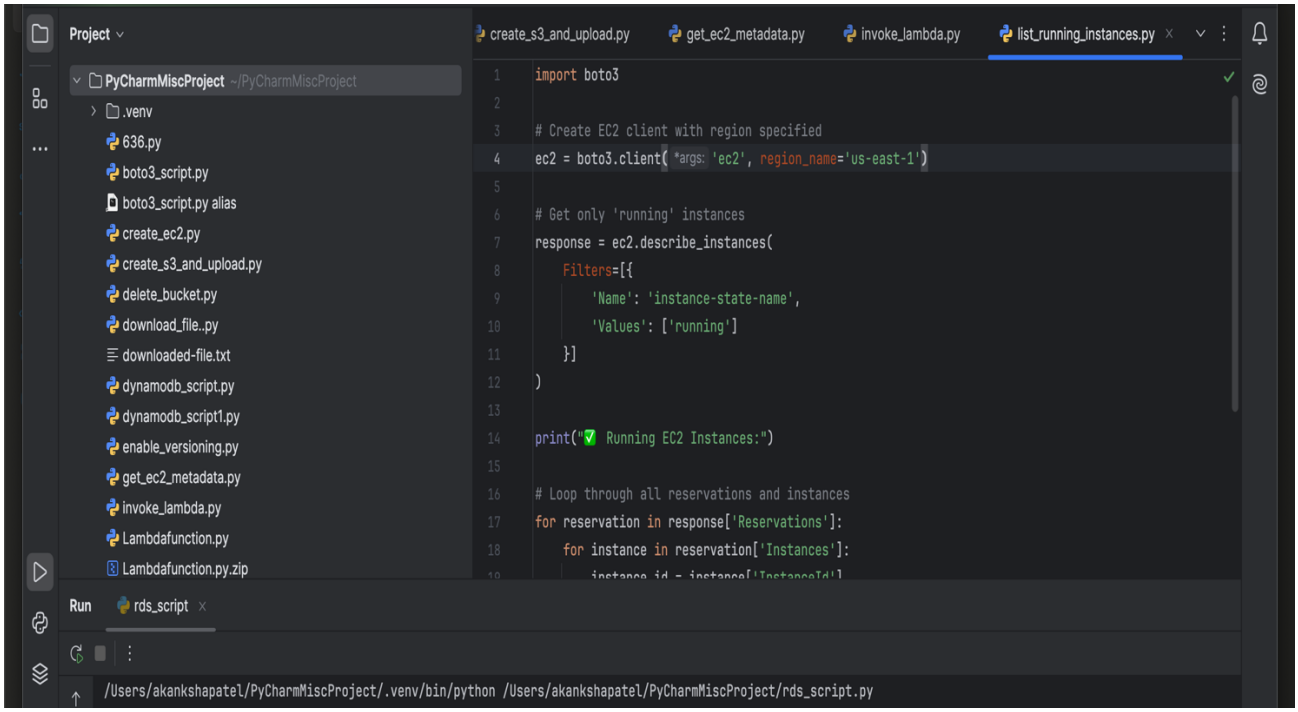


2. Retrieve EC2 instance metadata.



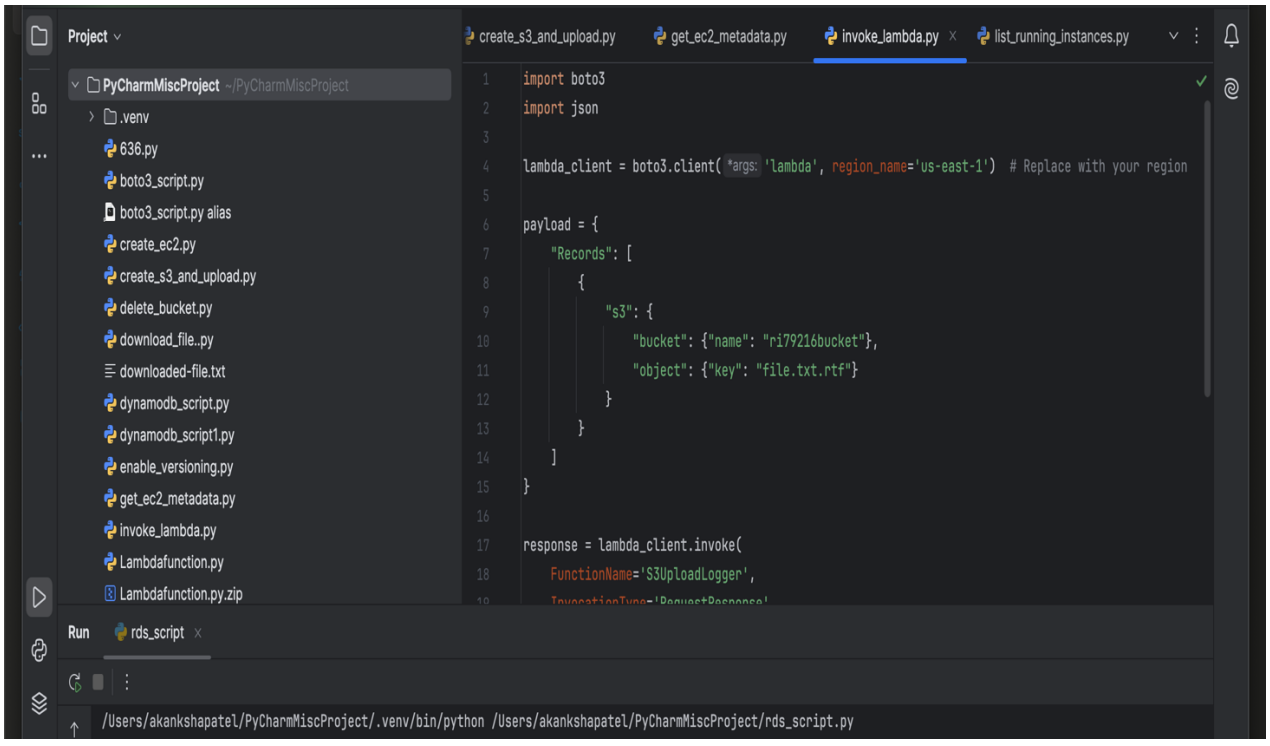


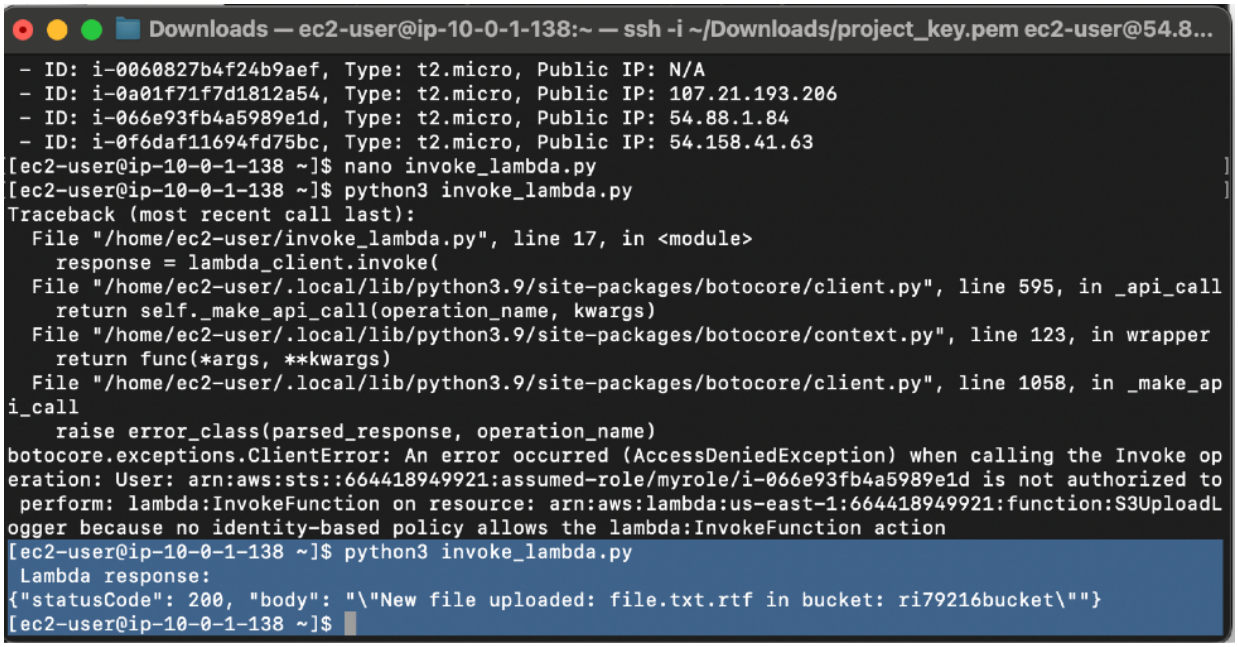
3. List running EC2 instance



```
Downloads — ec2-user@ip-10-0-1-138:~ — ssh -i ~/Downloads/project_key.pem ec2-user@54.8...
dpoint_config
    return self._resolve_endpoint(**resolve_endpoint_kwargs)
  File "/home/ec2-user/.local/lib/python3.9/site-packages/botocore/args.py", line 537, in _resolve_en
dpoint
    return endpoint_bridge.resolve(
  File "/home/ec2-user/.local/lib/python3.9/site-packages/botocore/client.py", line 655, in resolve
    resolved = self.endpoint_resolver.construct_endpoint(
  File "/home/ec2-user/.local/lib/python3.9/site-packages/botocore/regions.py", line 233, in construc
t_endpoint
    result = self._endpoint_for_partition(
  File "/home/ec2-user/.local/lib/python3.9/site-packages/botocore/regions.py", line 281, in _endpoin
t_for_partition
    raise NoRegionError()
botocore.exceptions.NoRegionError: You must specify a region.
[ec2-user@ip-10-0-1-138 ~]$ nano list_running_instances.py
[ec2-user@ip-10-0-1-138 ~]$ python3 list_running_instances.py
 Running EC2 Instances:
 - ID: i-0060827b4f24b9aef, Type: t2.micro, Public IP: N/A
 - ID: i-0a01f71f7d1812a54, Type: t2.micro, Public IP: 107.21.193.206
 - ID: i-066e93fb4a5989e1d, Type: t2.micro, Public IP: 54.88.1.84
 - ID: i-0f6daf11694fd75bc, Type: t2.micro, Public IP: 54.158.41.63
[ec2-user@ip-10-0-1-138 ~]$ nano invoke_lambda.py
[ec2-user@ip-10-0-1-138 ~]$ python3 invoke_lambda.py
Traceback (most recent call last):
```

4. Invoke the AWS Lambda function manually.





Report and Documentation

1. Architecture Diagram explaining the setup.

1. VPC (Virtual Private Cloud)

- A custom isolated network where all AWS resources live.
- Divided into:
  - Public Subnets – Can access internet
  - Private Subnets – No direct internet access

All EC2s, ALBs, RDS, and NAT routing happen inside this VPC.

2. Internet Gateway + Route Table

- Internet Gateway (IGW):
  - Connects your VPC to the internet
  - Required for any public-facing resources (like ALB, EC2)
- Route Table:
  - Routes traffic:
    - 0.0.0.0/0 → IGW for public subnet (Internet access)
    - Private subnet route does not include IGW

3. Public Subnet

a. ALB (Application Load Balancer)

- Distributes incoming HTTP/HTTPS traffic to:
  - Auto Scaling Group (EC2 app servers)
  - Lambda (in ALB-Lambda integration)

b. ASG (Auto Scaling Group)

- Maintains minimum and maximum EC2 instances based on demand
- Launches EC2s in multiple AZs
- Registers EC2s behind the ALB

4. Private Subnet

a. RDS MySQL

- Fully managed relational database
- Deployed in a private subnet (no internet access)
- Only EC2s or Lambdas in VPC can connect

This ensures high security — RDS can't be attacked directly from the internet.

5. Amazon S3 Buckets

a. Frontend Hosting S3 Bucket

- Stores static frontend files
- Optionally connected to CloudFront + API Gateway for public delivery

b. Data Upload S3 Bucket

- File uploads
- Trigger Lambda function for processing

6. Lambda Functions

a. Triggered by S3

- When a new object is uploaded, S3 sends an event to Lambda
- Lambda:
  - Reads file

    - Parses or processes it
    - Stores result in RDS or another S3 bucket

b. API Lambda

- Also invoked through API Gateway (explained next)

7. API Gateway

- Fully managed API layer
- Exposes REST or HTTP endpoints to the public
- Routes traffic to:
    - Lambda functions
    - ALB/EC2 backend
    - S3 static websites

8. Automation & DevOps Integration

a. GitHub Actions

- CI/CD pipeline:
    - On git push, GitHub runs tests
    - Builds code, zips artifacts
    - Deploys via Code Deploy or updates Lambda

b. AWS Code Deploy

- Handles automated deployments:
    - EC2 code (via install scripts)
    - Lambda function versioning
    - Blue/green and canary deployments

c. AWS Step Functions

- Orchestrates Lambda and API tasks
- Example: A data pipeline triggered by S3 upload, processed in 3 steps:
    - Parse → Validate → Store
- Can retry failed steps and define workflows

2. **Use Terraform to create networking components (VPC, subnets, security groups).**

Step 1: Create Project Directory
mkdir aws-networking
cd aws-networking

Step 2: Create main.tf with the following Terraform code

```
provider "aws" {
  region = "us-east-1"
}

# VPC
resource "aws_vpc" "main" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "main-vpc"
  }
}

# Internet Gateway
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id
  tags = {
    Name = "main-igw"
  }
}

# Public Subnet
resource "aws_subnet" "public" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone       = "us-east-1a"
  tags = {
    Name = "public-subnet"
  }
}

# Private Subnet
```

```
resource "aws_subnet" "private" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "private-subnet"
  }
}

# Public Route Table and Route
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
  tags = {
    Name = "public-rt"
  }
}

resource "aws_route" "public_internet" {
  route_table_id         = aws_route_table.public.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.igw.id
}

resource "aws_route_table_association" "public_assoc" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}

# NAT Gateway
resource "aws_eip" "nat" {
  vpc = true
}

resource "aws_nat_gateway" "natgw" {
  allocation_id = aws_eip.nat.id
  subnet_id     = aws_subnet.public.id
  depends_on    = [aws_internet_gateway.igw]
  tags = {
    Name = "nat-gateway"
  }
}
```

```
# Private Route Table and Route
resource "aws_route_table" "private" {
  vpc_id = aws_vpc.main.id
  tags = {
    Name = "private-rt"
  }
}

resource "aws_route" "private_nat" {
  route_table_id         = aws_route_table.private.id
  destination_cidr_block = "0.0.0.0/0"
  nat_gateway_id         = aws_nat_gateway.natgw.id
}

resource "aws_route_table_association" "private_assoc" {
  subnet_id      = aws_subnet.private.id
  route_table_id = aws_route_table.private.id
}

# Security Group for EC2 (allow SSH)
resource "aws_security_group" "ec2_sg" {
  name        = "ec2-sg"
  description = "Allow SSH"
  vpc_id      = aws_vpc.main.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "ec2-sg"
  }
```

}

Step 3: Initialize and Deploy
A. Initialize Terraform
terraform init

B. See the plan
terraform plan

C. Apply the configuration
terraform apply

Confirm with yes when prompted.

Step 4: Output and Verify

- Go to AWS Console → VPC
  - Verify:
    - VPC
    - Subnets (public/private)
    - Internet Gateway
    - NAT Gateway
    - Route Tables
    - Security Group

3. Use CloudFormation to deploy EC2 instances, RDS, and Lambda functions

PART 1: Deploy EC2 Instance

Step 1: Create a CloudFormation Template ec2-instance.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Deploy an EC2 instance using specified AMI and key pair
Resources:
  MyVPC:
 Type: AWS::EC2::VPC
  Properties:
 CidrBlock: 10.0.0.0/16
      Tags:
  - Key: Name
    Value: MyVPC
```

```
MySubnet:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.1.0/24
    MapPublicIpOnLaunch: true
    AvailabilityZone: !Select [0, !GetAZs '']
    Tags:
      - Key: Name
        Value: MySubnet

MyInternetGateway:
  Type: AWS::EC2::InternetGateway

AttachGateway:
  Type: AWS::EC2::VPCGatewayAttachment
  Properties:
    VpcId: !Ref MyVPC
    InternetGatewayId: !Ref MyInternetGateway

MyRouteTable:
  Type: AWS::EC2::RouteTable
  Properties:
    VpcId: !Ref MyVPC

DefaultRoute:
  Type: AWS::EC2::Route
  DependsOn: AttachGateway
  Properties:
    RouteTableId: !Ref MyRouteTable
    DestinationCidrBlock: 0.0.0.0/0
    GatewayId: !Ref MyInternetGateway

SubnetRouteTableAssociation:
  Type: AWS::EC2::SubnetRouteTableAssociation
  Properties:
    SubnetId: !Ref MySubnet
    RouteTableId: !Ref MyRouteTable
```

```
  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Allow SSH access
      VpcId: !Ref MyVPC
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 22
          ToPort: 22
          CidrIp: 0.0.0.0/0

  MyEC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      InstanceType: t2.micro
      ImageId: ami-0953476d60561c955
      KeyName: project_key
      NetworkInterfaces:
        - AssociatePublicIpAddress: true
          SubnetId: !Ref MySubnet
          DeviceIndex: 0
          GroupSet:
            - !Ref MySecurityGroupStep 2: Deploy via Console
```

1. Go to AWS CloudFormation → **Create Stack**
2. Choose Upload a template file
3. Upload ec2-instance.yaml
4. Click **Next**, name the stack (e.g., EC2Stack)
5. Click Next → Next → Acknowledge → Create Stack

Output:

- A running EC2 instance in your public subnet
- Accessible via SSH:

  ssh -i project_key.pem ec2-user@<public-ip>

- HTML page available at http://<public-ip>

PART 2: Deploy RDS MySQL Instance

Step 1: Create CloudFormation Template rds-mysql.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Description: RDS MySQL Instance

Resources:
  MyDBSubnetGroup:
    Type: AWS::RDS::DBSubnetGroup
    Properties:
      DBSubnetGroupDescription: Subnets for RDS
      SubnetIds:
        - subnet-private-1   # From Terraform
        - subnet-private-2   # Optional second AZ

  MyRDSInstance:
    Type: AWS::RDS::DBInstance
    Properties:
      DBInstanceIdentifier: mymysql-db
      DBInstanceClass: db.t3.micro
      Engine: mysql
      EngineVersion: 8.0.35
      AllocatedStorage: 20
      MasterUsername: admin
      MasterUserPassword: Anchal#10
      VPCSecurityGroups:
        - sg-rds-id   # Only accessible from EC2 SG
      DBSubnetGroupName: !Ref MyDBSubnetGroup
      PubliclyAccessible: false
```

Step 2: Deploy via Console

1. Go to CloudFormation → Create Stack
2. Upload rds-mysql.yaml
3. Click through wizard → Name stack RDSStack
4. Confirm → Create Stack

Output:

- RDS instance deployed in private subnet
- Accessible only from EC2 inside VPC:

mysql -h <endpoint> -u admin -p

PART 3: Deploy Lambda Function

Step 1: Create lambda-function.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda Hello World

Resources:
  LambdaExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: LambdaExecutionRole
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

  HelloWorldFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: HelloWorldFunction
      Runtime: python3.9
      Handler: index.lambda_handler
      Timeout: 10
      Role: !GetAtt LambdaExecutionRole.Arn
      Code:
        ZipFile: |
          def lambda_handler(event, context):
              return {
                  'statusCode': 200,
                  'body': 'Hello from Lambda!'
              }
```

Step 2: Deploy

1. Go to CloudFormation → Create Stack
2. Upload lambda-function.yaml
3. Name it LambdaStack

4. Click Create Stack

Step 3: Test Lambda

1. Go to **AWS Lambda**
2. Select HelloWorldFunction
3. Click **Test** → use default test event
4. Click Test again

Output:

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

4. Deploy a web application on EC2 (can be a simple HTML page).

Step 1: Prerequisites

- A VPC and public subnet were already created using Terraform
- A security group was configured to allow:
  - SSH access (TCP port 22)
  - HTTP access (TCP port 80)
- A Key Pair (project_key.pem) was available for SSH

Step 2: Create CloudFormation Template

A CloudFormation template ec2-webapp.yaml was created with:

- An EC2 instance resource using the Amazon Linux 2 AMI
- Apache Web Server installed and started via a User Data script
- An HTML file (index.html) created automatically in /var/www/html/

Key Code Snippet:

```
UserData:
  Fn::Base64: |
    #!/bin/bash
    yum update -y
    yum install -y httpd
    systemctl start httpd
```

systemctl enable httpd
echo "<h1>Hello from EC2 Web Server</h1>" > /var/www/html/index.html

Step 3: Deploy the Stack

1. Opened AWS Console → CloudFormation
2. Selected **Create Stack** → With new resources
3. Uploaded ec2-webapp.yaml
4. Provided parameter: project_key as Key Pair
5. Clicked **Create Stack** and monitored the stack status
6. Stack status changed to: CREATE_COMPLETE

Step 4: Verify the EC2 Instance

- Navigated to **EC2 > Instances**
- Located the instance deployed by CloudFormation
- Copied its Public IPv4 address

Step 5: Test the Web Application

- Opened a browser and visited:

  http://<public-ip>

Output:

pgsql
CopyEdit
Hello from EC2 Web Server

5. **Configure the database for the web application.**

Step 1: Prerequisites

- **A** RDS MySQL instance was deployed using CloudFormation (rds-mysql.yaml)
- The RDS instance was:
  - Placed in a private subnet
  - Attached to a security group that allows access from the EC2 security group
- The EC2 web server is already running in the **public subnet** and is configured to allow outbound access to RDS

Step 2: Connect to RDS from EC2

1. SSH into EC2:
   ssh -i project_key.pem ec2-user@<ec2-public-ip>

2. Install MySQL client on EC2:
   sudo yum install mysql -y

3. Connect to RDS:
   mysql -h <rds-endpoint> -u admin -p
   - Password: *********

Connection successful confirms that networking, security groups, and routing are correctly configured.

Step 3: Create Database and Table

Once connected to MySQL shell:

```
CREATE DATABASE myappdb;
USE myappdb;

CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

The schema is now ready for use by the web app.

🔷 Step 5: Validate Data Operations

- Insert data:

  INSERT INTO users (name, email) VALUES ('Akanksha', 'akanksha@example.com');

- Query data:

  SELECT * FROM users;

6. Implement autoscaling to manage web server load.

Step 1: Prerequisites

- A VPC, public subnet, internet gateway, and security group were already provisioned using Terraform
- A key pair (project_key.pem) is available
- A simple web app (index.html) is already tested on a standalone EC2 instance

Step 2: Create a Launch Template

1. Go to EC2 Console → Launch Templates → Create launch template
2. Set values:
    - Name: WebServerTemplate
    - AMI: ami-0953476d60561c955
    - Instance type: t2.micro
    - Key pair: project_key
    - Security group: Allow HTTP (port 80) and SSH (port 22)
3. Under Advanced details → User Data, add:

```
#!/bin/bash
yum update -y
yum install -y httpd
systemctl start httpd
echo "<h1>Hello from Auto Scaling EC2</h1>" > /var/www/html/index.html
```

4. Click Create launch template

The launch template defines the EC2 configuration that will be used by the Auto Scaling Group.

Step 3: Create an Application Load Balancer (ALB)

1. Go to EC2 → Load Balancers → Create Load Balancer
2. Choose Application Load Balancer
3. Name: WebALB
4. Scheme: Internet-facing

5. Listener: HTTP on port 80
6. Availability Zones: select at least 2 public subnets
7. Security group: allow HTTP (port 80)
8. Target group:
    - Name: WebTargetGroup
    - Target type: instance
    - Protocol: HTTP
    - Port: 80
    - Health check path: /
9. Click Create

The ALB will distribute traffic across EC2 instances and handle health checks.

Step 4: Create an Auto Scaling Group (ASG)

1. Go to EC2 → Auto Scaling Groups → Create Auto Scaling Group
2. Name: WebASG
3. Launch template: WebServerTemplate
4. VPC: Select your VPC
5. Subnets: Select public subnets
6. Attach to existing ALB:
    - Target Group: WebTargetGroup
7. Set desired capacity:
    - Desired: 1
    - Min: 1
    - Max: 3
8. Scaling policy:
    - Choose Target tracking scaling policy
    - Metric: Average CPU utilization
    - Target value: 60%
9. Click Create Auto Scaling Group

ASG automatically launches new EC2 instances based on CPU utilization.

Step 5: Test Autoscaling

1. Get the DNS name of the ALB
2. Open in browser http://<ALB-DNS-name>
3. We should see: Hello from Auto Scaling.

B. AWS Lambda for Logging S3 Uploads

Step 1: Prerequisites

- An Amazon S3 bucket was already created
- We have AWS IAM permissions to create Lambda functions and attach event triggers
- CloudWatch Logs permissions are included in the Lambda role

Step 2: Write the Lambda Function (Python 3.9)

Function name: LogS3Uploads

Code:

```
import json
import boto3
import logging

# Set up logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    try:
        bucket_name = event['Records'][0]['s3']['bucket']['name']
        object_key = event['Records'][0]['s3']['object']['key']

        log_message = f"New file uploaded: {object_key} in bucket: {bucket_name}"
        logger.info(log_message)

        return {
            "statusCode": 200,
            "body": json.dumps(log_message)
        }

    except Exception as e:
        logger.error(f"Error processing S3 event: {str(e)}")
        return {
            "statusCode": 500,
            "body": json.dumps(f"Error: {str(e)}")
        }
```

Step 3: Create IAM Role for Lambda

1. Go to IAM > Roles → Create role
2. Choose Lambda as the trusted entity
3. Attach these policies:
    - AWSLambdaBasicExecutionRole (CloudWatch Logs)
4. Name it: LambdaS3LoggingRole

Step 4: Create Lambda Function

1. Go to AWS Lambda → Create function
2. Choose:
    - Author from scratch
    - Function name: LogS3Uploads
    - Runtime: Python 3.9
    - Execution Role: Use an existing role → LambdaS3LoggingRole
3. Paste the function code into the editor
4. Click Deploy

Step 5: Add S3 Trigger to Lambda

1. Under Lambda → Triggers, click Add trigger
2. Choose S3
3. Bucket: my-upload-logs-bucket (your existing bucket)
4. Event type: PUT (All object creates events)
5. Prefix/suffix: *(optional)*
6. Check the box to acknowledge permission changes
7. Click Add

Step 6: Test the Setup

1. Go to S3 → my-upload-logs-bucket
2. Click Upload → upload a test file (e.g., sample.txt)
3. Wait a few seconds

Step 7: Verify CloudWatch Logs

1. Go to CloudWatch → Logs → Log groups
2. Find: /aws/lambda/LogS3Uploads

3. Open the latest log stream
4. Output: INFO:root:New file uploaded: sample.txt in bucket: my-upload-logs-bucket

C. Interaction with AWS

Use AWS CLI for Basic Operations

1. List EC2 Instances

aws ec2 describe-instances --query "Reservations[*].Instances[*].InstanceId" --output text

Output: List of EC2 Instance IDs currently running

2. List S3 Buckets

aws s3 ls

Output: List of existing S3 buckets

3. Invoke Lambda Function Manually

```
aws lambda invoke \
  --function-name LogS3Uploads \
  --payload '{}' \
  output.json
```

Output saved to output.json, viewable with:

C. Python Scripts Using Boto3

All scripts were written using boto3 to programmatically manage AWS resources.

1. Create S3 Bucket & Upload a File

```python
import boto3
import os

# 1. Set up S3 client
s3 = boto3.client('s3')

# 2. Define a globally unique bucket name
bucket_name = 'ri79216bucket'  # Make sure it's unique

# 3. Create the bucket
s3.create_bucket(
    Bucket=bucket_name,
    CreateBucketConfiguration={'LocationConstraint': 'us-east-1'}
```

```
)
print(f" Bucket created: {bucket_name}")

# 4. Create a sample file
file_name = "file.txt.rtf"
with open(file_name, "w") as f:
    f.write("Hello from Boto3!")

# 5. Upload file to S3
s3.upload_file(file_name, bucket_name, file_name)
print(f" File '{file_name}' uploaded to bucket '{bucket_name}'")
```

2. Retrieve EC2 Metadata from Instance (runs on EC2)
import requests

```
import requests

base_url = "http://169.254.169.254/latest/meta-data/"
keys = [
    "instance-id",
    "ami-id",
    "hostname",
    "public-ipv4",
    "local-ipv4",
    "security-groups"
]

print(" EC2 Instance Metadata:")
for key in keys:
    try:
        response = requests.get(base_url + key, timeout=2)
        print(f"{key}: {response.text}")
    except requests.exceptions.RequestException as e:
        print(f"{key}: Failed to retrieve ({e})")
5
```

3. List Running EC2 Instances
import boto3

```
import boto3

# Create EC2 client with region specified
ec2 = boto3.client('ec2', region_name='us-east-1')

# Get only 'running' instances
response = ec2.describe_instances(
    Filters=[{
        'Name': 'instance-state-name',
        'Values': ['running']
    }]
)

print(" Running EC2 Instances:")

# Loop through all reservations and instances
for reservation in response['Reservations']:
```

```
    for instance in reservation['Instances']:
        instance_id = instance['InstanceId']
        instance_type = instance['InstanceType']
        public_ip = instance.get('PublicIpAddress', 'N/A')
        print(f" - ID: {instance_id}, Type: {instance_type}, Public IP: {public_ip}")
```

4. Manually Invoke Lambda Function

```
import boto3
import json

lambda_client = boto3.client('lambda', region_name='us-east-1')  # Replace with your
region

payload = {
    "Records": [
        {
            "s3": {
                "bucket": {"name": "ri79216bucket"},
                "object": {"key": "file.txt.rtf"}
            }
        }
    ]
}

response = lambda_client.invoke(
    FunctionName='S3UploadLogger',
    InvocationType='RequestResponse',
    Payload=json.dumps(payload),
)

response_payload = response['Payload'].read().decode()
print(" Lambda response:")
print(response_payload)
```

. No Challenges Faced

. Github link - https://github.com/Akanksha576/cloud-project