Experiment 1

Aim: To launch a Windows instance using AWS EC2 & to understand the concept of IaaS (Infrastructure as a Service)

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Windows**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow RDP traffic & HTTP traffic**.
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **RDP client**, download the **remote desktop file**.
9. Get the password by uploading the private key file & obtain the decrypted password. Copy the password.
10. Open the **RDP file downloaded**, allow all permissions & connect to the instance. Paste the password to access the instance.
11. After accessing the windows, open any browser and check for the IP address of the instance.

Experiment 2

Aim: To launch an Ubuntu instance using AWS EC2 & to understand the concept of IaaS (Infrastructure as a Service) with the help of MobaXterm.

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Ubuntu**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow SSH traffic & HTTP traffic.**
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session. Using terminal, check the IP address of the machine.

Experiment 3

Aim: To learn Amazon S3 Service for creating buckets, storing objects & deploying of a static website on AWS S3.

Steps:

1. Sign in to AWS console and go to S3 service
2. Create a new bucket for general purpose, enter a **unique name**.
3. Under block public access, **uncheck all checkboxes** & check the acknowledgement to make bucket objects public.
4. Create the bucket. Select the created bucket and upload objects to it.
5. Add files and select the **index.html** file to **upload it as a bucket object**.
6. Create a **new bucket policy**. Go to the bucket, then go to the **permissions tab**. In the bucket policy section, **edit the policy** and in the **given text area paste** the following code snippet.

```
{
     "Version": "2008-10-17",
     "Id": "PolicyForPublicWebsiteContent",
     "Statement": [
          {
               "Sid": "PublicReadGetObject",
               "Effect": "Allow",
               "Principal": {
                    "AWS": "*"
               },
               "Action": "s3:GetObject",
               "Resource": "arn:aws:s3:::<bucket-name>/*"
          }
     ]
}
```

7. Replace **<bucket-name>** with the **actual name** of the bucket created.
8. Save changes to the policy.
9. Go the **properties tab** of the bucket. In the **static website hosting section edit the properties**. Set the **static website hosting radio button to enable**. Choose **hosting type as "Host a static website"**. In **Index document**, specify the **name of the object i.e. `index.html`**.
10. Save all the changes. A **bucket website endpoint URL** will be generated. Click on the URL to open the static hosted website in a new tab.

Experiment 4

Aim: To create a Lambda function which will log a message once an object is added to a specific bucket in Amazon S3.

Steps:

1. Sign in to AWS console and go to S3 service
2. Create a new bucket for general purpose, enter a **unique name**.
3. Under block public access, **uncheck all checkboxes** & check the acknowledgement to make bucket objects public.
4. Create the bucket.
5. Go to AWS Lambda service and create a function.
6. Use a blueprint, select **blueprint name** as **"Get S3 object"** (**python** or any suitable language) & give a **function name**. **Under execution role**, select "**Create a new role from AWS policy templates**" & provide a role name.
7. Under **S3 trigger**, select the **bucket created earlier** & **acknowledge** the before creating the function.
8. After creating the function, navigate to **"Code" section**. In the given code, **add a print statement in the "try" block**. Once changes are done, **deploy** the code.
9. Go to S3 service, **add object to the bucket** created earlier to **trigger the lambda** service.
10. Go to Lambda service, select the function created. Go to **"Monitor" tab** and click **View CloudWatch logs**. Find the **recent log under log streams** & **check the log events**. The whole event log must be seen & the print statement that was added to the code. This marks the execution of function triggered when an object was uploaded to S3.

Experiment 5

Aim: To create and configure an AWS Lambda function (using Python/Java/Node.js) that is triggered by Amazon S3 events, and interacts with a DynamoDB table using appropriate IAM role permissions.

Steps:

1. Sign in to AWS console and go to IAM service
2. Create a **new role**, select **entity type as AWS service** & **use case as Lambda**. In the next step, **search for DynamoDB** & **select AmazonDynamoDBFullAccess**. Next **provide name** for the **role** & create a role.
3. Go to Lambda service, create a function. Provide function name, select the **runtime** (**Python** or whichever suitable). Under **execution role**, select "**Use an existing role**" & select the **IAM role created earlier**. Create the function.
4. Go to S3 service, **create a new bucket with unique name**. **Uncheck** all the **checkboxes to avoid blocking public access**.
5. Go to the Lambda function created earlier. **Add a trigger to the function**. Select the **trigger as S3 bucket** & **select the newly created bucket**. Finally add the trigger.
6. Go to DynamoDB service. Create a **new table**, provide table name, set partition key & create the table.
7. Go to Lambda function created earlier. Under the **code tab** of Lambda function, paste the following code.

```
import boto3
from uuid import uuid4
def lambda_handler(event, context):
    s3 = boto3.client("s3")
    dynamodb = boto3.resource('dynamodb')
    for record in event['Records']:
        bucket_name = record['s3']['bucket']['name']
        object_key = record['s3']['object']['key']
        size = record['s3']['object'].get('size', -1)
        event_name = record ['eventName']
        event_time = record['eventTime']
        dynamoTable = dynamodb.Table('<table_name>')
        dynamoTable.put_item(
            Item={'unique': str(uuid4()), 'Bucket':
bucket_name, 'Object': object_key,'Size': size, 'Event':
event_name, 'EventTime': event_time})
```

8. Replace **<table_name>** with the **actual table name created in DynamoDB** service. After making changes **deploy the code**.
9. **Add one or more objects** to S3 bucket.
10. Go to DynamoDB service, **explore the table items** of the table created earlier. The **uploaded objects in S3 must reflect** in the DynamoDB table.

Experiment 7

Aim: To understand the concept of NOSQL Databases, AWS DynamoDB creating of tables, adding table items & querying the table.

Steps:

1. Sign in to AWS console and go to DynamoDB service
2. Create a **new table** in DynamoDB
3. Provide a **table name**, set the **partition key** & **optionally the sort key**. Create the table.
4. Browse to the newly created table & **explore table items**.
5. **Manually** create a **new table item**.
6. Use the **form fields** to **add attribute names** & **attribute values**.
7. To **add multiple attributes**, use the "**Add new attribute**" button.
8. Table items can also be **added using JSON view**.
9. Add **at least 10 items** in the DynamoDB table.
10. Use the **query feature** to perform **different queries** on the DynamoDB table & retrieve information.

Experiment 8

Aim: Install & configure Jenkins on AWS EC2 Ubuntu instance

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Ubuntu**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, edit the settings, add “All traffic” type & keep source type as “Anywhere”.
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session.
11. Update the packages.

```
sudo apt update
```

12. Upgrade the packages.

```
sudo apt upgrade
```

13. Download Java JDK version 11 (or 17 or 21).

```
sudo apt install openjdk-11-jdk
```

14. Fetch the Jenkins package.

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key |
sudo apt-key add -
```

15. Add the Jenkins package fetched using wget to list of sources under apt

```
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable
binary/ > /etc/apt/sources.list.d/jenkins.list'
```

16. Add the keys to the keyserver

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-
keys 5BA31D57EF5975CA
```

17. Update again to fetch the latest Jenkins package

```
sudo apt update
```

18. Install Jenkins

```
sudo apt install jenkins
```

19. Enable the Jenkins service

```
sudo systemctl enable jenkins
```

20. Start the jenkins service

```
sudo systemctl start jenkins
```

21. Check the status of Jenkins whether it is running on port 8080 or not

```
sudo systemctl status jenkins
```

22. Go to EC2 instances, select the instance and note down its public IPv4 address or Public DNS, copy it and paste it in a new tab.
23. Make sure to use http protocol & at the end of the IP address or DNS, add the port 8080 on which Jenkins is running.

```
http://<IPv4_address>:8080 or http://<public_DNS>:8080
```

24. Jenkins will require a secret password to unlock. Fetch password by running the following command on the CLI (read the initialAdminPassword file) & copy it.

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

25. Paste the password in Jenkins & continue. Install suggested plugins, configure the Jenkins URL (let it be default as it appears).

Experiment 9

Aim: Install Docker & deploy a containerized web application using Nginx on AWS EC2 Linux.

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Ubuntu**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow SSH traffic, HTTPS traffic & HTTP traffic.**
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session.
11. Update & upgrade the latest packages using following command.

```
sudo apt update && sudo apt upgrade -y
```

12. Install docker using curl

```
curl -fsSL https://get.docker.com -o get-docker.sh && sh get-docker.sh
```

13. Check docker installation & version

```
docker -v
```

14. Pull nginx using docker

```
sudo docker pull nginx
```

15. Run nginx in a docker container & list the running containers in docker

```
sudo docker run -d -p 80:80 --name nginx-container nginx
```

16. Go to AWS EC2 instances, select the instance, copy the Public DNS or the Public IPv4 address and paste it in a new tab. Make sure to use HTTP protocol
17. Pull python image using docker

```
sudo docker pull python
```

18. Pull python:3.9-slim image using docker

```
sudo docker pull python:3.9-slim
```

19. Pull python:3.8-alpine image using docker

```
sudo docker pull python:3.8-alpine
```

20. List all the images in docker

```
sudo docker images
```

Experiment 10

Aim: To install Docker on Ubuntu instance using curl, running hello-world from docker hub, pulling images from docker & executing docker commands.

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Ubuntu**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow SSH traffic, HTTPS traffic & HTTP traffic.**
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session.
11. Get root access. Using curl, install docker.

```
sudo su
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh
```

12. Check docker installation & version

```
docker -v
```

13. Run ‘hello-world’ using docker

```
docker run hello-world
```

14. Pull some images using docker

```
docker pull python
docker pull python:3.9-slim
docker pull python:3.8-alpine
docker pull ubuntu
```

15. Execute following docker commands

```
# Run python print statement
docker run python:3.10 python -c “print(‘hello’)”
```

```
# Run ubuntu
docker run -it ubuntu

# List all running containers
docker ps

# List all containers
docker ps -a

# List all images
docker images

# Stop a running container
docker stop <container_id>

# Start a container
docker start <container_id>

# Restart a container
docker restart <container_id>

# Remove a container
docker rm <container_id>

# Remove an image
docker rmi <image _id>

# Fetch logs of a container
docker logs <container_id>

# List network
docker network ls

# List volume
docker volume ls

# Inspect a container
docker inspect <container_id>
```

Experiment 11

Aim: To run a containerized web application using Nginx server in Docker

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Ubuntu**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow SSH traffic, HTTPS traffic & HTTP traffic.**
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session.
11. Get root access. Use curl to install docker.

```
sudo su
curl -fsSL https://get.docker.com -o get-docker.sh
```

12. Run bash of nginx server using docker.

```
docker run -it -p 80:80 nginx bash
```

13. Start the nginx service on the nginx image (machine). Change the directory to where html file is stored of Nginx server.

```
service nginx start
cd usr/share/nginx/html
```

14. In a new tab, paste the public IPv4 address or public DNS of the EC2 instance & make sure to use the HTTP protocol. Nginx server must now be running and following page must be shown.

15. In the Nginx bash, install nano editor.

```
apt install nano
```

16. Using nano editor, modify the contents of the Nginx server html file. Add your own HTML code.

```
nano index.html
```

17. Save the file (`Ctrl + S`) and close (`Ctrl + X`) the editor. Now go to the browser and reload the page.

Experiment 12

Aim: Install docker on AWS EC2 using curl & run a flask application inside a docker container.

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Ubuntu**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow SSH traffic, HTTPS traffic & HTTP traffic.**
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session.
11. Get root access. Use curl to install docker.

```
sudo su
curl -fsSL https://get.docker.com -o get-docker.sh
```

12. Verify docker installation & its version.

```
docker --version
```

13. Create a new directory for flask application. Change directory to flask application directory. Create a new app directory and change directory to the app directory. In the app directory create app.py (flask application) file using nano editor.

```
Ubuntu
|
|-- myflaskapp
    |
    |-- app
        |
        |-- app.py
```

```
mkdir myflaskapp
cd myflaskapp
mkdir app
cd app
```

```
nano app.py
```

14. In the app.py file, paste the following contents.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
     return "This is a sample flask application"

if __name__ == '__main__':
     app.run(host="0.0.0.0", port=80)
```

15. Create a requirements.txt file

```
nano requirements.txt
```

16. Mention the required python libraries

```
Flask
```

17. Change directory and come out of the app directory. Create a Docker file in the directory.

```
cd ..
nano Dockerfile
```

18. Paste the following in the Dockerfile

```
#Specify the base image of the Docker image we are building
FROM python

#Set the working directory inside the Docker image
WORKDIR /opt/demo/

#Copy the content of the /app directory on host machine into
current working directory
COPY /app .

#Run a command inside the image during build phase
RUN pip install -r requirements.txt

#Define a default command that will run when a container is
started from the image
ENTRYPOINT python app.py
```

19. Build the image for the Flask application. Replace <img_name> with any name

```
docker build -t <img_name>:latest .
```

20. Run the Flask image using docker on port 80. Replace <img_name> with the image built in previous step.

```
docker run -it -p 80:80 <img_name>
```

21. Select the launched instance. Copy the public IPv4 address or public DNS, open a new tab & paste it. Make sure to use HTTP protocol.

Experiment 13

Aim: To understand continuous monitoring, installation & configuration of Nagios core & plugins on Linux machine.

Steps:

1. Sign in to AWS console and go to EC2 service
2. Launch an instance, provide a name to the instance.
3. Select AMI image as **Amazon Linux**.
4. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
5. In the network settings, **allow SSH traffic, HTTPS traffic & HTTP traffic.**
6. Once all configuration is done, launch the instance.
7. The instance state must be running & all status checks must pass.
8. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
9. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
10. Start the SSH session.
11. Get root access. Install the httpd & php package using yum.

```
sudo su
yum install httpd php
```

12. Install the gcc packages

```
yum install gcc glibc glibc-common
```

13. Install gd & gd-devel package

```
yum install gd gd-devel
```

14. Add a user named nagios & set password for the user.

```
adduser -m nagios
passwd nagios
```

15. Create a new group, and attach users to it

```
groupadd nagioscmd
usermod -a -G nagioscmd nagios
usermod -a -G nagioscmd apache
```

16. Download nagios using wget.

```
wget     http://prdownloads.sourceforge.net/sourceforge/nagios/nagios-4.4.14.tar.gz
```

17. Download nagios plugins using wget

```
wget     http://nagios-plugins.org/download/nagios-plugins-2.4.6.tar.gz
```

18. Unzip the nagios zip file and change directory to the file

```
tar zxvf nagios-4.4.14.tar.gz
cd nagios-4.4.14
```

19. Install openssl-devel package.

```
yum install openssl-devel
```

20. Configure nagios, with the command group created earlier.

```
./configure --with-command-group=nagioscmd
```

21. Compile all components of nagios

```
make all
```

22. Build and install software from source code.

```
make install
```

23. Install initialization scripts for nagios

```
make install-init
```

24. Install default configuration files for nagios

```
make install-config
```

25. Set proper permissions & ownerships for Nagios external command file

```
make install-commandmode
```

26. Install the web configurations needed to run Nagios web interface

```
make install-webconf
```

27. Create a password file for authenticating

```
htpasswd -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

28. Restart the httpd service

```
service httpd restart
```

29. Change directory to where nagions-plugins zip file was installed & unzip it.

```
cd ..
tar zxvf nagios-plugins-2.4.6.tar.gz
```

30. Change directory to nagios-plugins directory & configure the nagios plugins.

```
cd nagios-plugins-2.4.6
./configure --with-nagios-user=nagios --with-nagios-group=nagios
```

31. Build & compile the nagios-plugins

```
make
```

32. Build & install software from the source code.

```
make install
```

33. Enable nagios service to start automatically

```
chkconfig nagios on
```

34. Verify the Nagios configurations

```
/usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

35. Start the nagios service & restart the httpd service

```
service nagios start
service httpd restart
```

36. Go to AWS EC2 instance, select the running instance, copy the public DNS or the public IPv4 address.
37. Open a new tab, paste the public DNS or public IPv4 address and at the end add nagios endpoint. Make sure to use HTTP protocol.

Experiment 14

Aim: To understand Terraform lifecycle & implement IaC (Infrastructure as a Code) to launch AWS EC2 instances

Steps:

1. Sign in to AWS console and go to IAM service
2. Create a new role, select entity type as AWS service & use case as EC2. In the next step, search for EC2Full & select AmazonEC2FullAccess. Next provide name for the role & create a role.
3. Go to EC2 service & launch an instance, provide a name to the instance.
4. Select AMI image as **Amazon Linux**.
5. Create a new key pair, provide a name, select key pair type as **RSA** & select file format as **.pem**
6. In the network settings, **allow SSH traffic, HTTPS traffic & HTTP traffic.**
7. Under advanced details, select IAM instance profile & attach the role created earlier.
8. Once all configuration is done, launch the instance.
9. The instance state must be running & all status checks must pass.
10. Connect to the instance, select **SSH client**, **copy the public DNS address** and the **username**.
11. Open MobaXterm, create a new session, session type must be **SSH**. In the **remote host field, paste the public DNS copied** earlier. **Specify the username**. Under advanced settings, **select the private key file downloaded** while launching the instance.
12. Start the SSH session.
13. Access root account. Create a directory to install terraform. Create a variables.tf file which will store variables for terraform.

```
sudo su
mkdir project-terraform
cd project-terraform
nano variables.tf
```

14. Paste the following contents in the variables.tf file

```
variable "aws_region" {
     description = "The AWS region to create things in."
     default = "eu-north-1"
}

variable "key_name" {
     description = "SSH keys to connect to ec2 instance"
     default = "ab_sec"
}
```

```
variable "instance_type" {
     description = "instance type for ec2"
     default = "t3.micro"
}

variable "security_group" {
     description = "Name of security group"
     default = "ab-jenkins-sg"
}

variable "tag_name" {
     description = "Tag Name for EC2 instance"
     default = "ab-terraform-started"
}

variable "ami_id" {
     description = "AMI for Ubuntu EC2 instance"
     default = "ami-0a716d3f3b16d290c"
}
```

15. Save (`Ctrl + S`) the variables.tf file & close (`Ctrl + X`) the nano editor. Create main.tf file

```
nano main.tf
```

16. Paste the following content in main.tf file

```
provider "aws" {
     region = var.aws_region
}
#Create security group with firewall rules
resource                                  "aws_security_group"
"ab_terra_security_jenkins_grp"
{
     name = var.security_group
     description = "security group for jenkins"

     ingress {
          from_port = 8080
          to_port = 8080
          protocol = "tcp"
          cidr_blocks = ["0.0.0.0/0"]
     }

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

```
     egress {
          from_port = 0
          to_port = 65535
          protocol = "tcp"
          cidr_blocks = ["0.0.0.0/0"]
     }

     tags {
          Name = var.security_group
     }
}

resource "aws_instance" "abTerraInstance" {
     ami = var.ami_id
     key_name = var.key_name
     instance_type = var.instance_type
     security_groups =
     [aws_security_group.ab_terra_security_jenkins_grp.nam
     e]
     tags = {
          Name = var.tag_name
     }
}

resource "aws_eip" "abTerraElastIP" {
     instance = aws_instance.abTerraInstance.id
     tags = {
          Name = "jenkins_elastic_ip"
     }
}
```

17. Save (`Ctrl + S`) the main.tf file & close (`Ctrl + X`) the nano editor.
18. Download terraform using wget

```
wget
https://releases.hashicorp.com/terraform/1.0.9/terraform_1
.0.9_linux_amd64.zip
```

19. Initialize terraform

```
# Initialize terraform working directory
terraform init
```

20. Preview the changes before applying them.

```
# Preview changes that terraform will make to infrastructure
before applying
terraform plan
```

21. Apply the changes to infrastructure

```
# Executes the changes needed to create, update or delete infrastructure
terraform apply
```

Type “yes” to confirm the changes

22. Go to AWS EC2 instances, a new instance will be initializing. Under security group, verify the security group attached to the EC2 instance as per main.tf file.
23. Verify the elastic IP address of the new instance, it will be set as per our main.tf file.

(Optional)

24. Destroy the Infrastructure resources created using terraform.

```
# Destroy all infrastructure managed by terraform
terraform destroy
```

25. Terraform will automatically delete the EC2 instance which was managed by Terraform.
26. The security group managed by terraform has also been deleted.
27. The elastic IP addresses created by terraform has also been destroyed.