Labs 1-5

Student ID: 23796349

Student Name: Adharsh Sundaram Soudakar

Lab 1

AWS Account and Log in

[1] Log into an IAM user account created for you on AWS

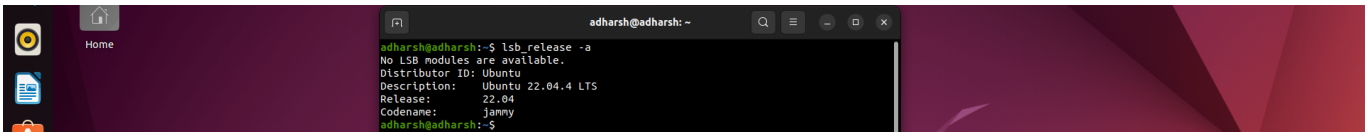
- Navigate to the AWS login page AWS Console.
- Sign in with the provided username ('student_ID'@student.uwa.edu.au) and password.
- Once signed in, a **change your password** prompt will appear.
- On successfully changing the password, you will redirected to the AWS dashboard.

[2] Search and open Identity Access Management

- Click on your user profile and click on **Security Credentials**.
- Scroll down to the **Access Keys** section and click on **Create access key** button.
- Choose the **Other** option and click **Next**.
- Set a description tag(optional) and click on **Create access key**.
- Make note of the **Access ID key** and **Secret Access key** or download the **csv** file that contains these information.

Set up recent Linux OSes

- I already have Ubuntu OS 22.04.4 LTS set up on Oracle VM VirtualBox.

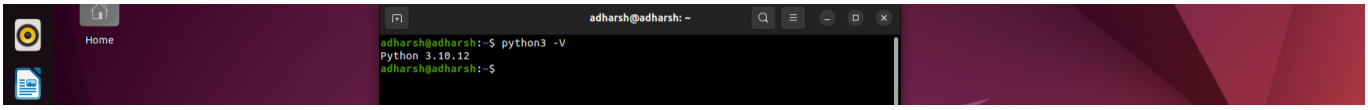


- This was done when I was enrolled in the units **CITS1003 - Introduction to Cybersecurity** and **CITS4407 - Open Source Tools and Scripting**.

Install Linux packages

[1] Install Python 3.8.x

- I already have the latest version of Python installed.



- Again, this was done when I was enrolled in the unit **CITS1003 - Introduction to Cybersecurity**.

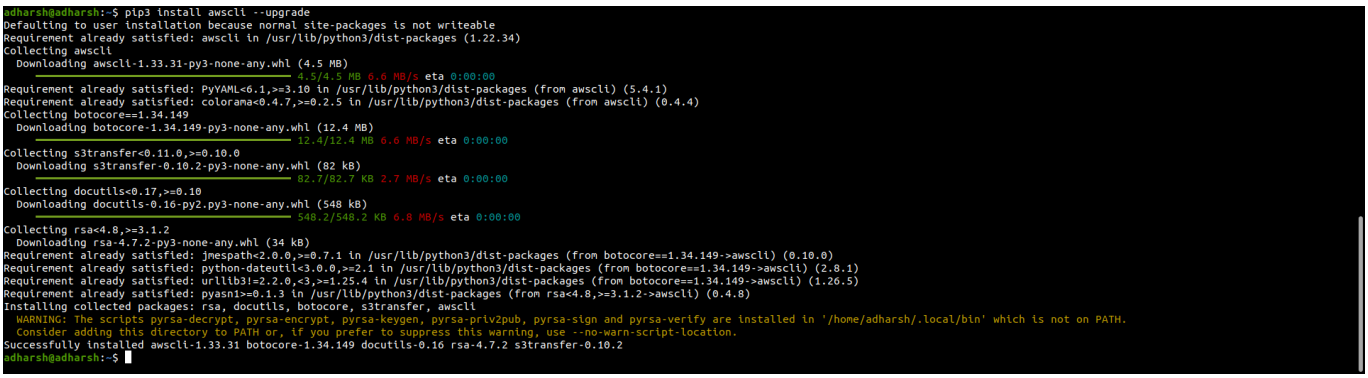
[2] Install awscli

- Open a terminal and enter the following commands:

```
sudo apt install awscli
```

Followed by,

```
pip3 install awscli --upgrade` or `sudo snap install aws-cli --classic
```



- To ensure correct installation, try checking the version using the following command:

```
aws --version
```

[3] Configure AWS

- To configure AWS, enter the following command:

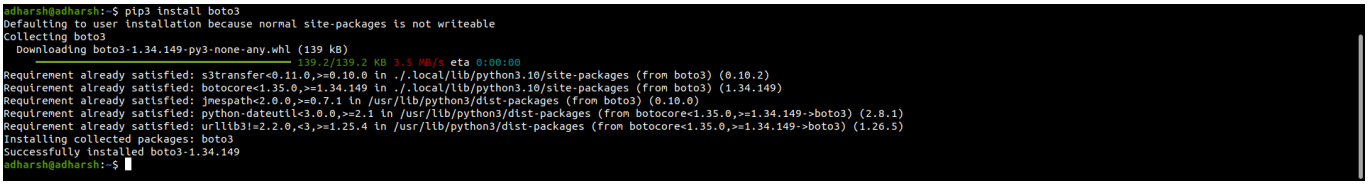
```
aws configure
```

- When prompted, enter your **AWS Access Key ID** and **AWS Secret Access Key**.
- Enter your **Default region name**, in my case it is **ap-northeast-2**.
- Set the **Default output format"** to **json**.

[4] Install boto3

- To install the **Boto3** library, enter the following command:

```
pip3 install boto3
```

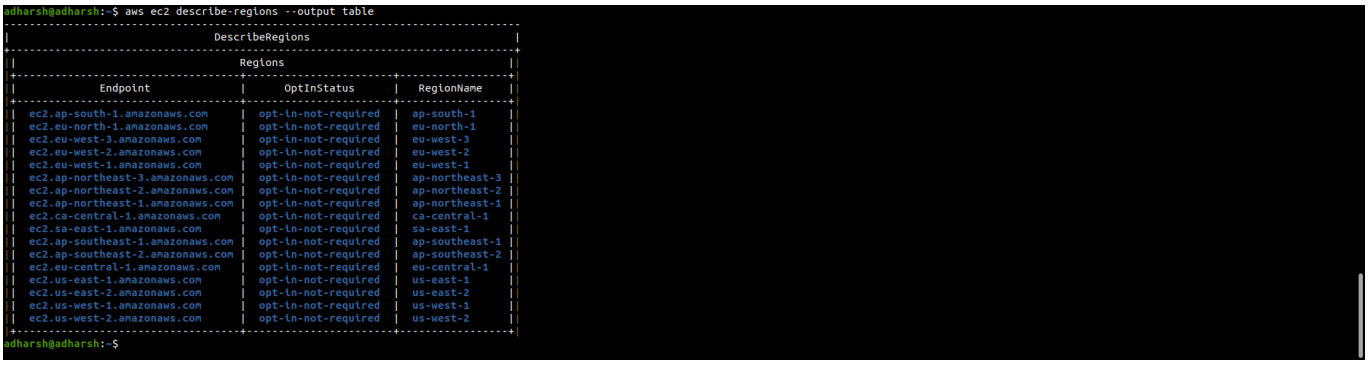


Test the installed environment

[1] Test the AWS environment

- To test the AWS environment, for example, to display the available regions with their respective endpoint names , enter the following command:

```
aws ec2 describe-regions --output table
```



[2] Test the Python environment

- To test the python environment, first we need to open the python interpreter:

```
python3
```

- Again to display the available regions with their respective endpoint names (un-tabulated), enter the following python code:

```
>>> import boto3
>>> ec2 = boto3.client('ec2')
>>> response = ec2.describe_regions()
>>> print(response)
```

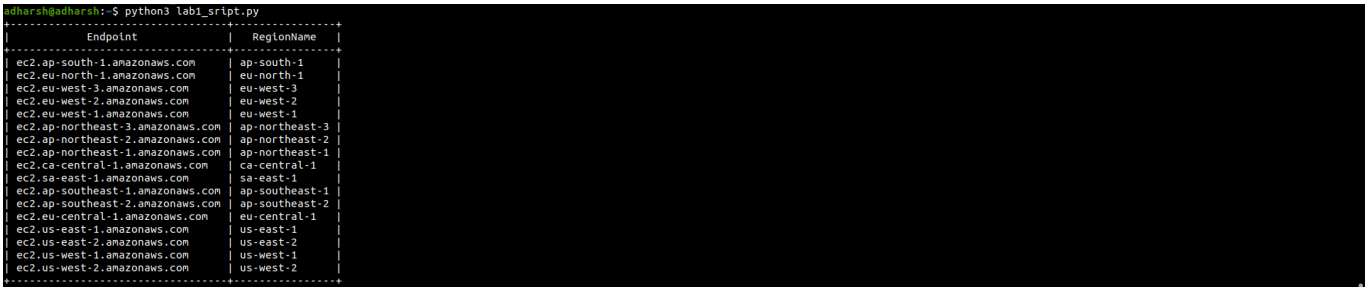
[3] Write a Python script

- To get a tabulated response of the above output with only **Endpoint** and **RegionName** columns, follow the steps below:
- Create a Python file eg: **lab1_script.py** and open it with a text editor (Vim) or an IDE (VScode).
- Type the following code and save the file:

```
import boto3
ec2 = boto3.client('ec2')
response = ec2.describe_regions()
print("+-------------------------------------+-----------------+")
print("|             Endpoint                |   RegionName    |")
print("+-------------------------------------+-----------------+")
for item in response["Regions"]:
    print("| {:<32} | {:<14} |".format(item["Endpoint"],item["RegionName"]))
print("+-------------------------------------+-----------------+")
```

- Run the script using the command:

```
python3 lab1_script.py
```



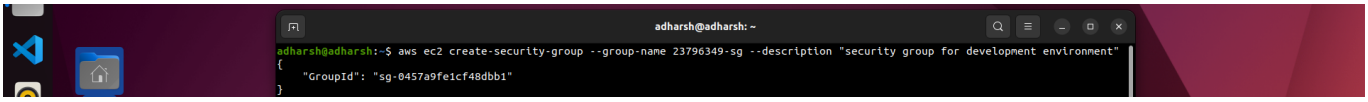
Lab 2

Create an EC2 instance using awscli

[1] Create a security group

- To create a **security group**, open a terminal and type the following command, use your student ID followed by sg:

```
aws ec2 create-security-group --group-name 23796349-sg --description "security
group for development environment"
```

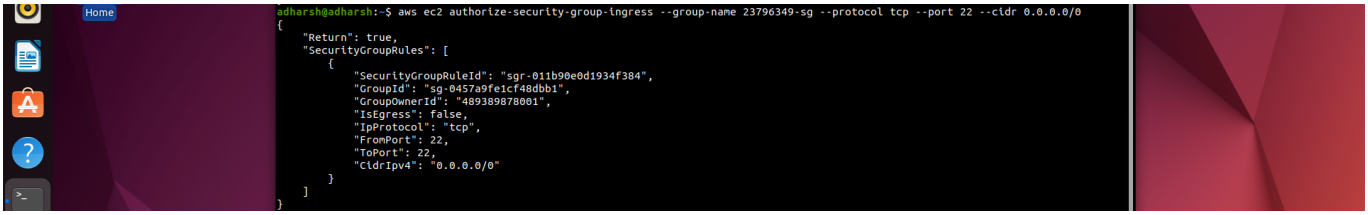


- Take a note of the security group id that is created.

[2] Authorise inbound traffic for ssh

- Type the following command to set **ssh authorisation**:

```
aws ec2 authorize-security-group-ingress --group-name 23796349-sg --protocol tcp -
-port 22 --cidr 0.0.0.0/0
```



[3] Create a key pair

- Type the following command to create a **key pair** and save it in a file:

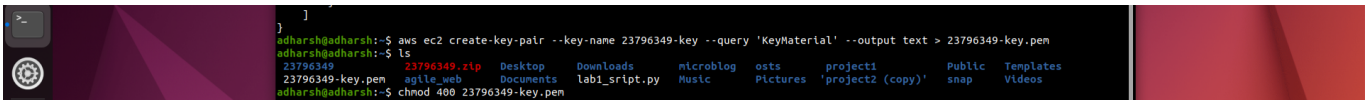
```
aws ec2 create-key-pair --key-name 23796349-key --query 'KeyMaterial' --output
text > 23796349-key.pem
```

- To use this key on Linux, copy the file to a directory ~/.ssh using the following command:

```
mv 23796349-key.pem ~/.ssh
```

and change its permissions to **read-only** using the following command:

```
chmod 400 23796349-key.pem
```



[4] Create the instance

- Enter the following command to create an instance and get its ID.

NOTE: Use your allocated image-id, in my case it is **ami-056a29f2eddc40520**.

```
 aws ec2 run-instances --image-id ami-056a29f2eddc40520 --security-group-ids
23796349-sg --count 1 --instance-type t2.micro --key-name 23796349-key --query
'Instances[0].InstanceId'
```

[5] Add a tag to your Instance

- To easily identify the instance, add a tag to it using the following command with the instance ID obtained from the previous step:

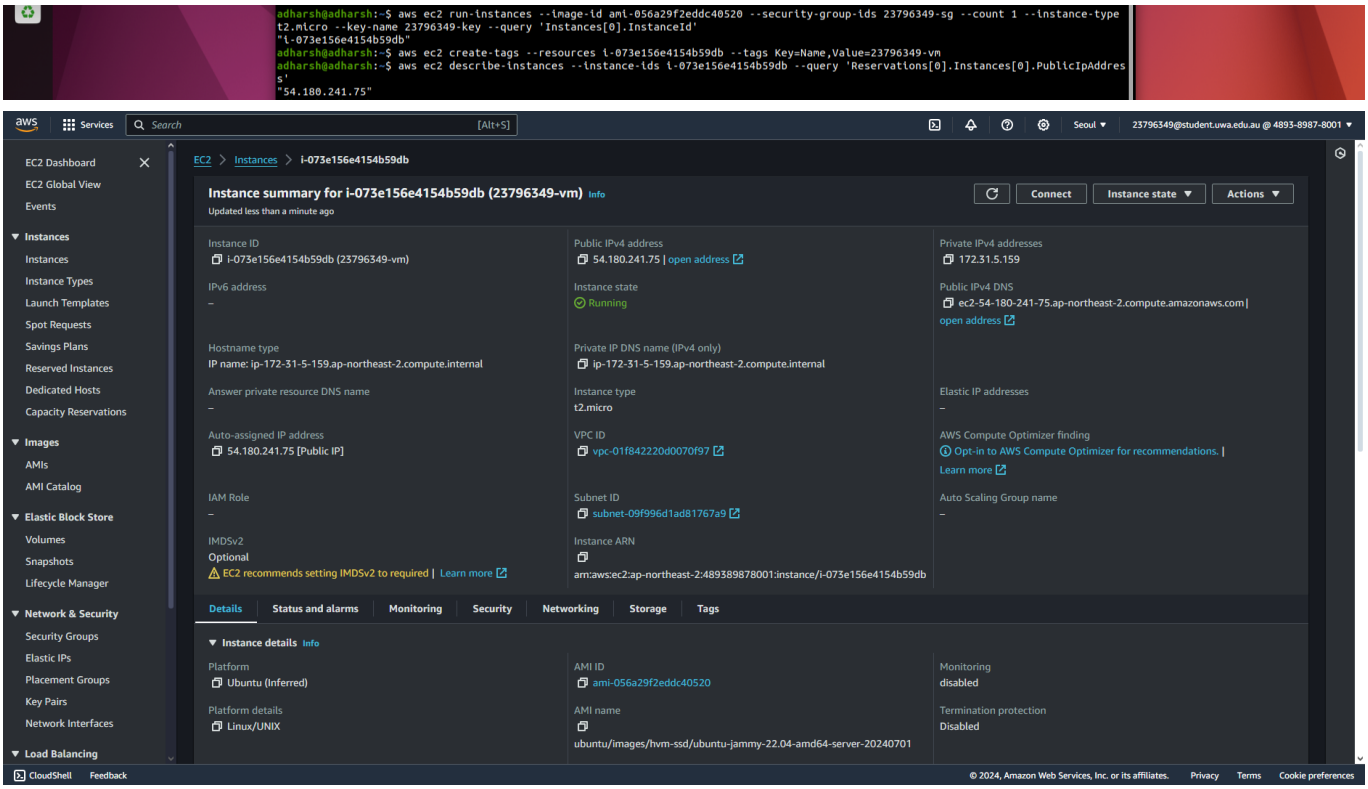
```
 aws ec2 create-tags --resources i-073e156e4154b59db --tags
Key=Name,Value=23796349-vm
```

[6] Get the public IP address

- To get the public IP address of the instance, use the following command with the instance ID obtained from `Step 4`:

```
aws ec2 describe-instances --instance-ids i-073e156e4154b59db --query
'Reservations[0].Instances[0].PublicIpAddress'
```

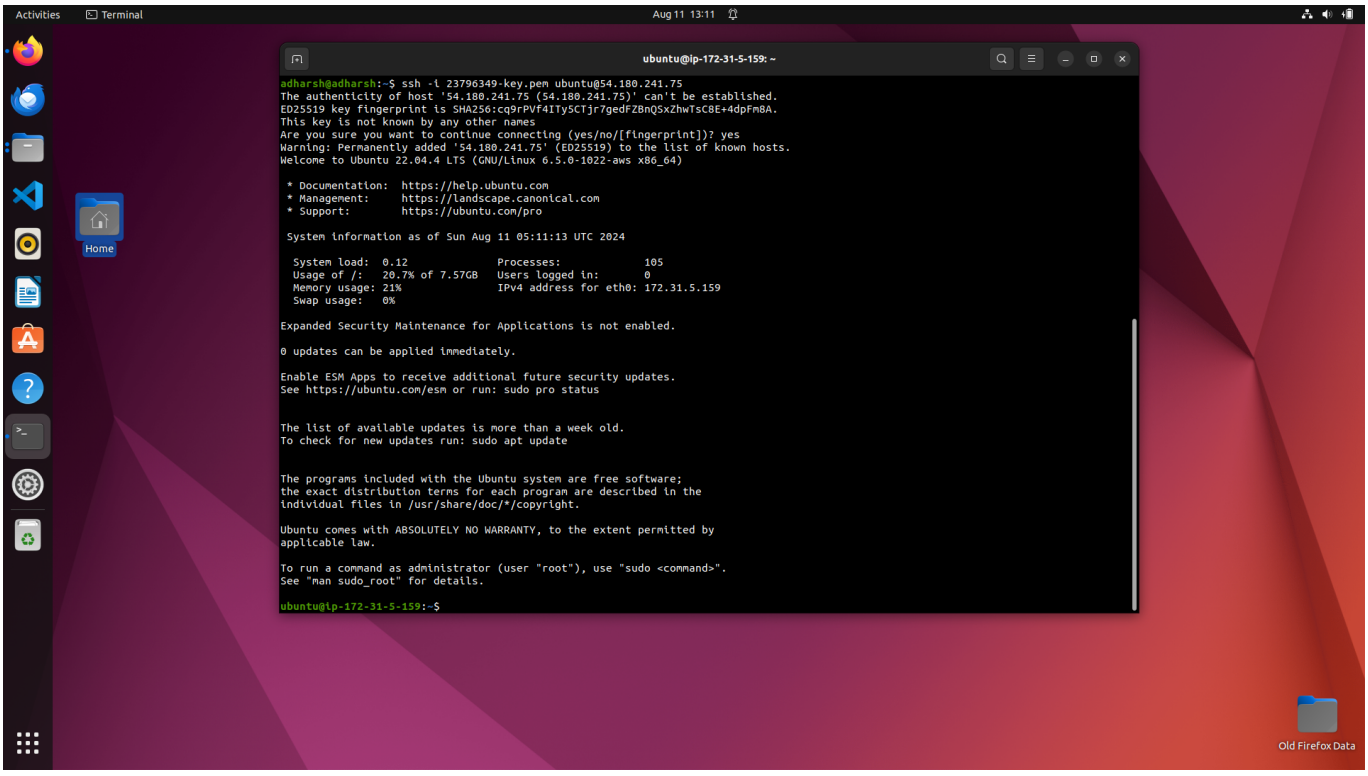
- Screenshots below show `Steps 4,5,6`:



[7] Connect to the instance via ssh

- To connect to the instance from the terminal, use the following command with the public IP obtained from the previous step:

```
ssh -i 23796349-key.pem ubuntu@54.180.241.75
```



[8] List the created instance using the AWS console

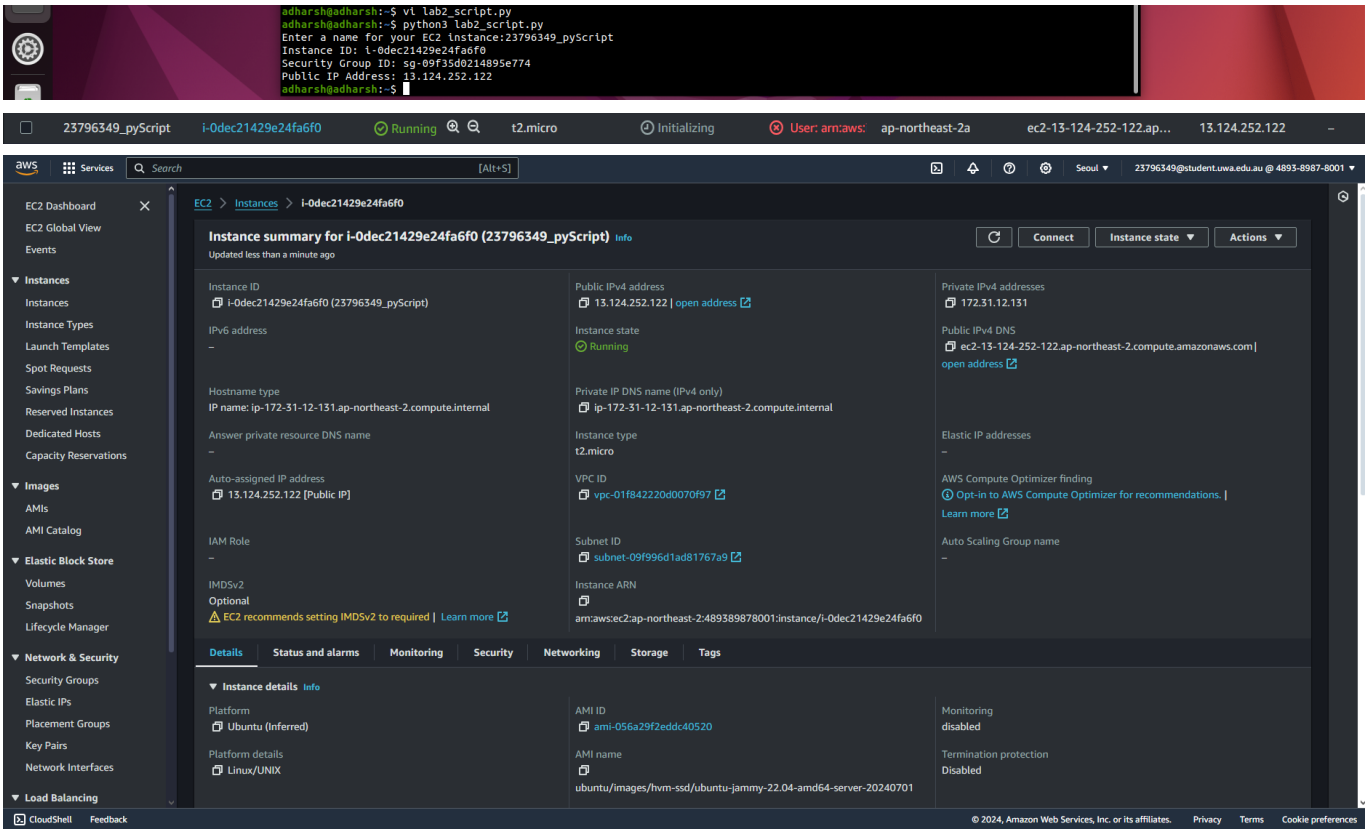
Create an EC2 instance with Python Boto3

- To create an EC2 instance using python script, use the Boto3 library, refer the code below:

```
import boto3

# Start a session
ec2 = boto3.resource('ec2')
ec2_client = boto3.client('ec2')

# Naming the EC2 instance
name = input("Enter a name for your EC2 instance:")

#Check if security group exists else creating one
try:
    sg = ec2_client.describe_security_groups(GroupNames=[f'{name}-sg'])
    security_group_id = sg['SecurityGroups'][0]['GroupId']
except:
    security_group = ec2.create_security_group(GroupName=f'{name}-sg',
Description='Security group for development environment using pythonScript')
    security_group.authorize_ingress(
            IpProtocol='tcp',
            FromPort=22,
            ToPort=22,
            Cidrlp='0.0.0.0/0'
    )
    security_group_id = security_group.id

#Check if key pair exists else creating one
try:
    kp = ec2_client.describe_key_pairs(KeyNames=[f'{name}-key'])
except:
    key_pair = ec2.create_key_pair(KeyName=f'{name}-key')
    with open(f'{name}-key.pem','w') as file:
        file.write(key_pair.key_material)

# Create EC2 instance
instance = ec2.create_instances(
        ImageId='ami-056a29f2eddc40520',
        MinCount=1,
        MaxCount=1,
        InstanceType='t2.micro',
        KeyName=f'{name}-key',
        SecurityGroupIds=[security_group_id],
        TagSpecifications=[
            {
                'ResourceType':'instance',
                'Tags':[
                    {
                        'Key':'Name',
                        'Value':name
                    },
                ]
```

```
            },
        ]
    )[0]

# Wait and load instance details
instance.wait_until_running()
instance.load()

#Printing instance details
print(f'Instance ID: {instance.id}')
print(f'Security Group ID: {security_group_id}')
print(f'Public IP Address: {instance.public_ip_address}')
```

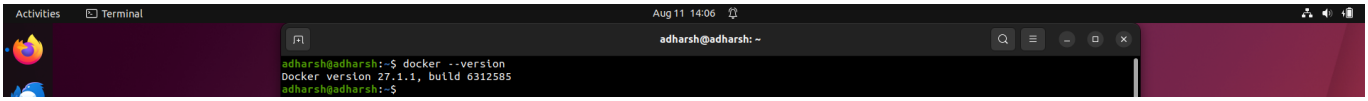
NOTE: Code referenced from here.



Use Docker inside a Linux OS

[1] Install Docker

- I have already set up the latest version of docker on my system.



- This was done when I was enrolled in **CITS1003 - Introduction to Cybersecurity** and **CITS4407 - Open Source Tools and Scripting**.

[2] Build and run an httpd container

- Create a directory called **html** using the following command:

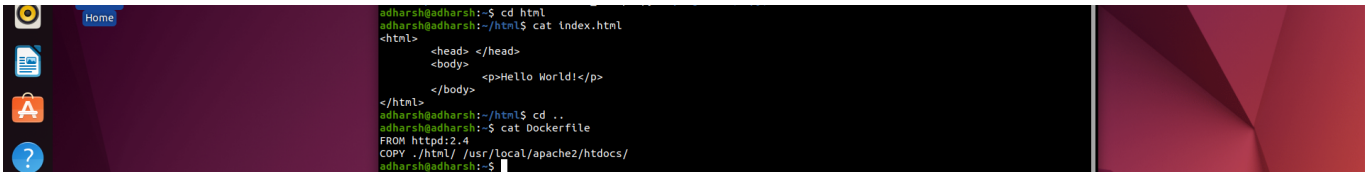
```
mkdir html
```

- Create a file **index.html** inside the html directory (use commands `vim or nano or touch` to create the file) and add the following content:

```
<html>
  <head> </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

- Create a file called **Dockerfile** outside the html directory with the following content:

```
FROM httpd:2.4
COPY ./html/ /usr/local/apache2/htdocs/
```

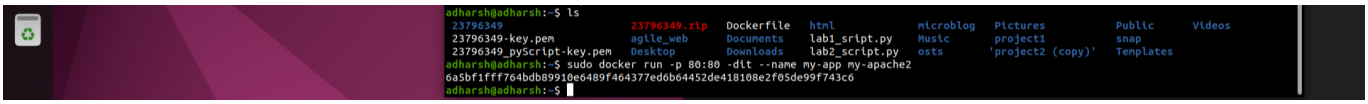


- Use the following command to build a docker image:

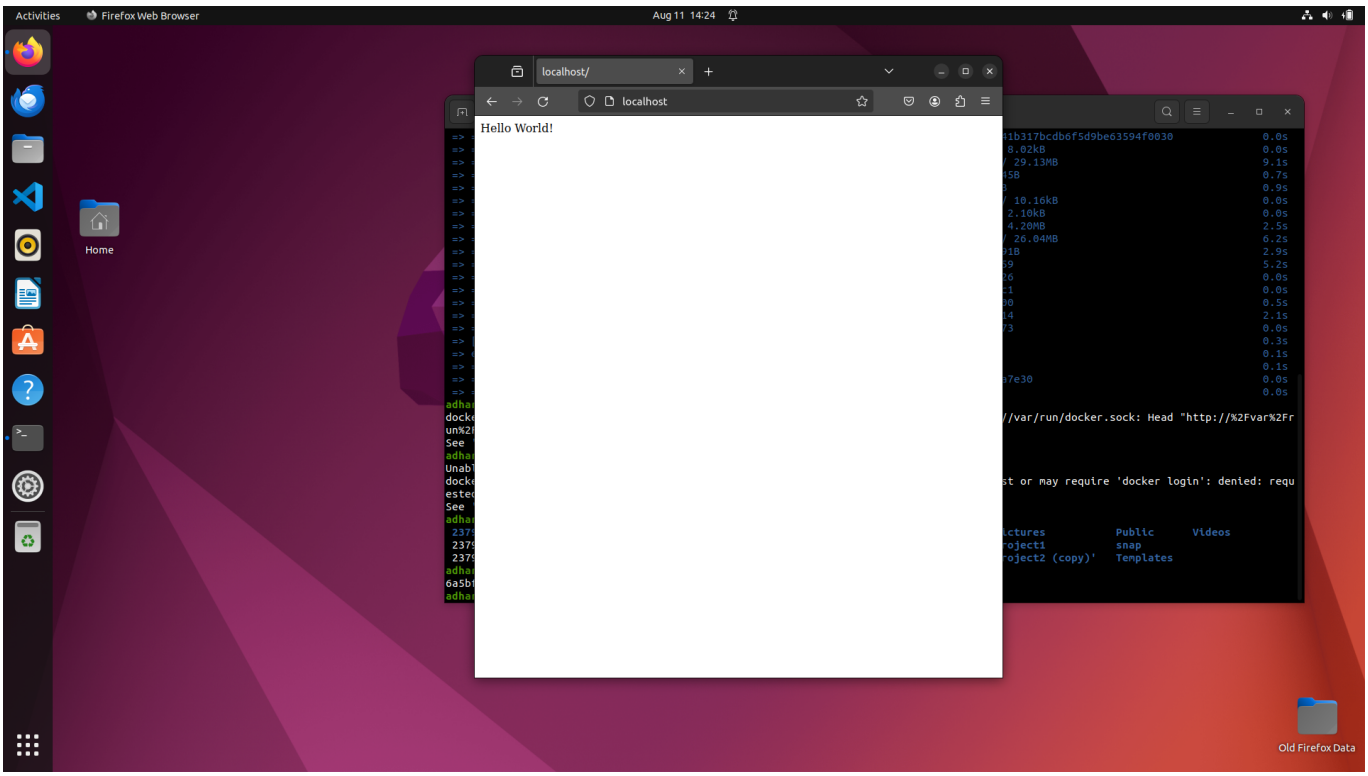
```
docker build -t my-apache2 .
```

- To run the docker image created before, use the following command:

```
docker run -p 80:80 -dit --name my-app my-apache2
```



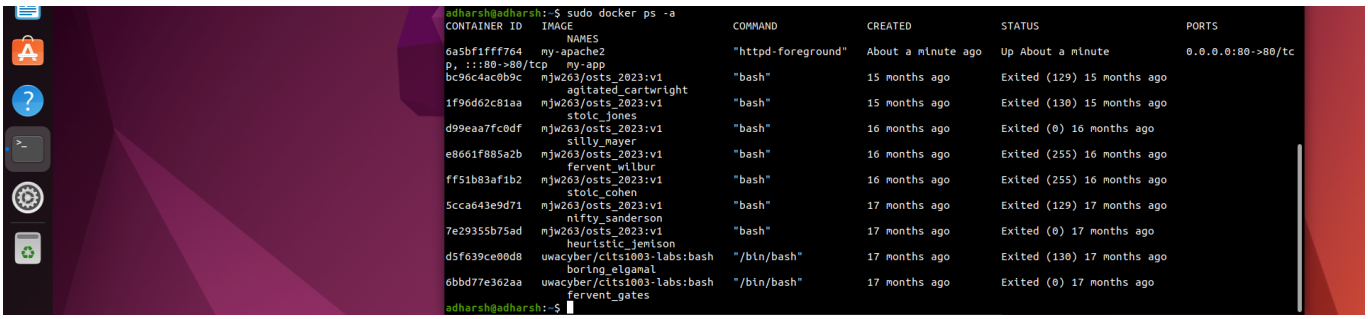
and proper execution of the previous steps can be verified by opening a browser and accessing address: http://localhost or http://127.0.0.1.



[3] Other docker commands

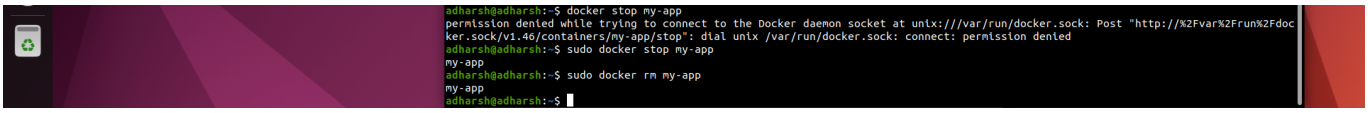
- To check the containers that are running, use the following command:

```
docker ps -a
```



- To stop and remove any of the running containers, use the following command:
  NOTE: Here the container name is **my-app**.

```
docker stop my-app
docker rm my-app
```



- The ec2 instances that were created for this lab were deleted through the **AWS Console** after completion.

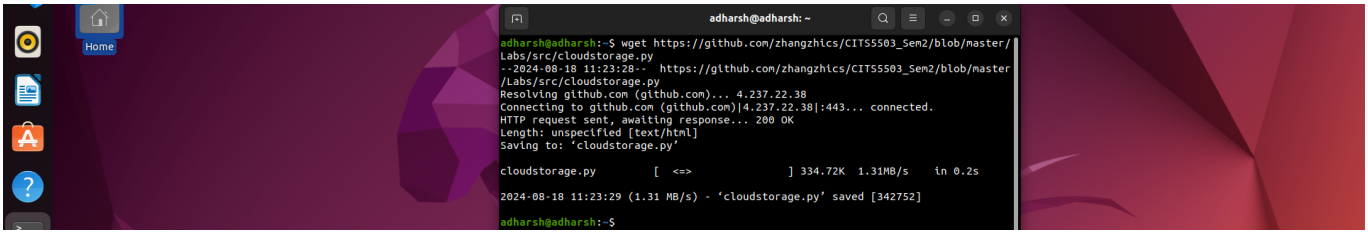
Lab 3

Program

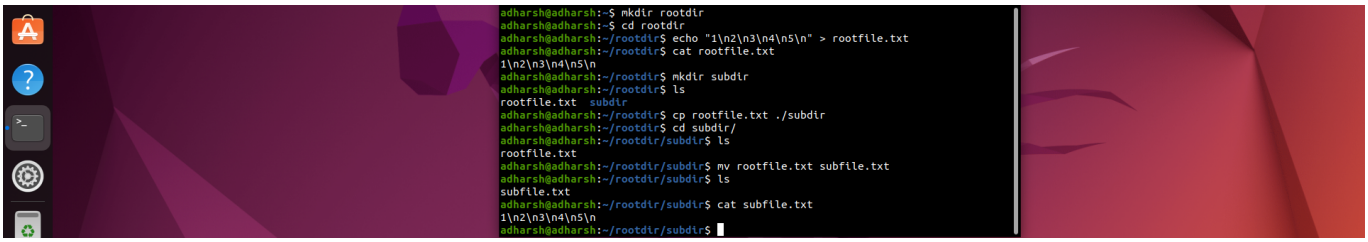
[1] Preparation

- To download the **cloudstorage.py** from the src, use the command

```
wget
https://github.com/zhangzhics/CITS5503_Sem2/blob/master/Labs/src/cloudstorage.py
```



- The necessary directories and files were created as shown in the screenshot below:

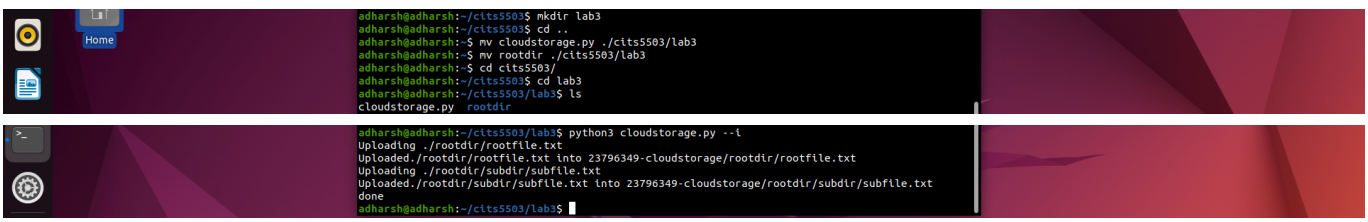


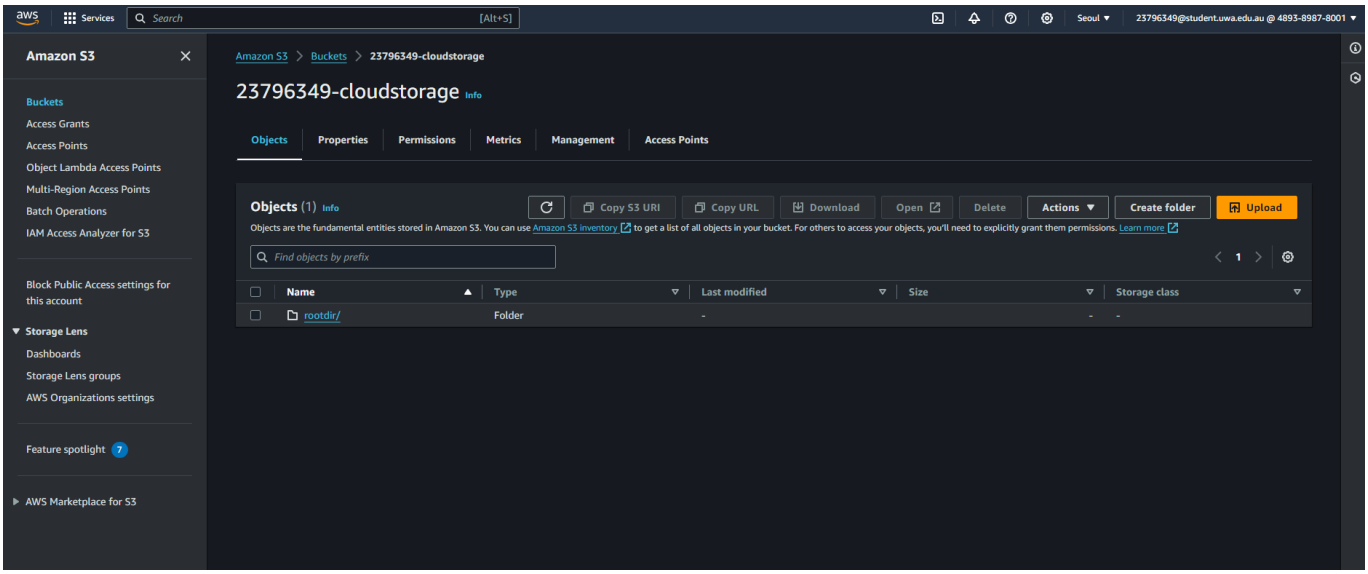
[2] Save to S3 by updating `cloudstorage.py`

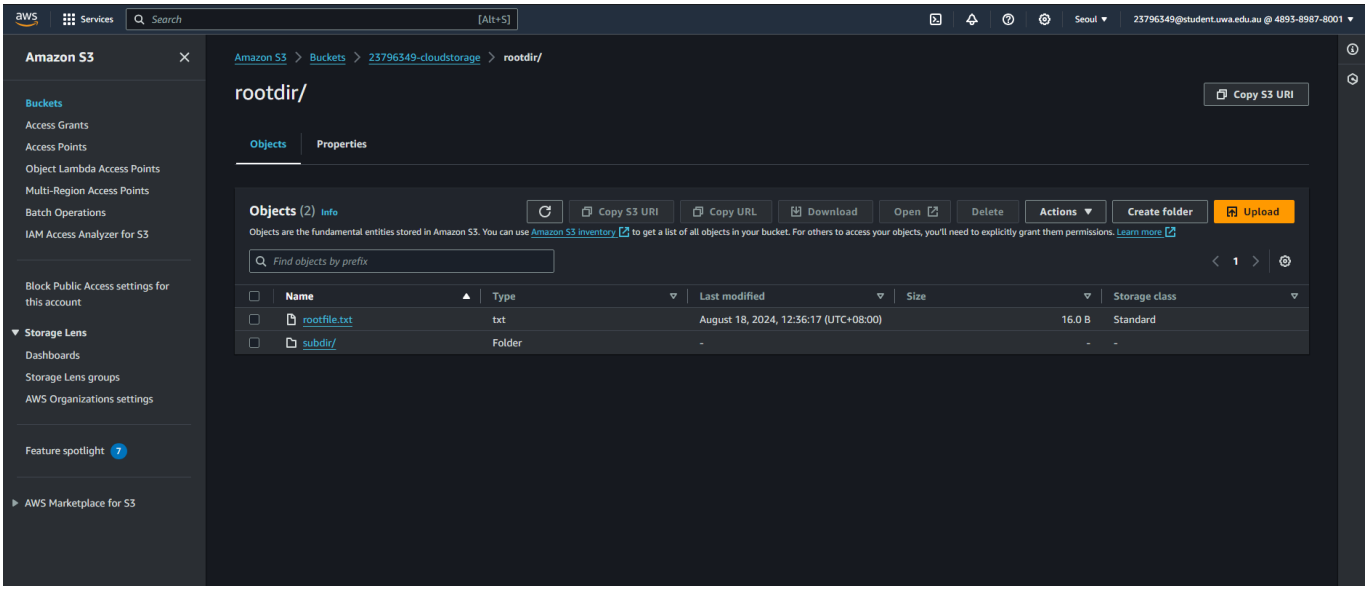
- Modified code referenced from here :

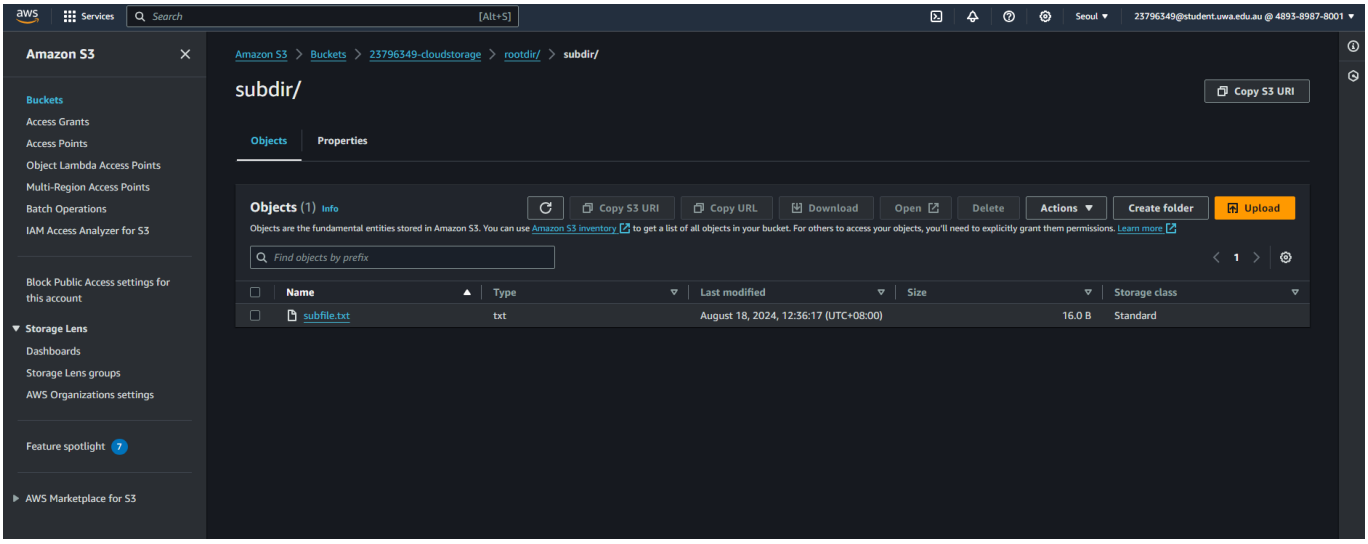
```
import os
import boto3
import base64

# ------------------------------
# CITS5503
#
# cloudstorage.py
#
# skeleton application to copy local files to S3
#
# Given a root local directory, will return files in each level and
# copy to same path on S3
#
# ------------------------------


ROOT_DIR = '.'
```

```
ROOT_S3_DIR = '23796349-cloudstorage'


s3 = boto3.client("s3")

bucket_config = {'LocationConstraint': 'ap-northeast-2'} #Replace the region with
your allocated region name.

def upload_file(bucket,folder_name, file, file_name):
    try:
        path = f"{folder_name}{file_name}"
        print("Uploading %s" % file)
        s3.upload_file(file,bucket,path)
        print(f"Uploaded{file} into {bucket}/{path}")
    except Exception as error:
        pass

# Main program
# Insert code to create bucket if not there

try:
    s3.create_bucket(Bucket=ROOT_S3_DIR,CreateBucketConfiguration=bucket_config)
#create_bucket function is idempotent.
    response = s3.head_bucket(Bucket=ROOT_S3_DIR)
    print(f"Bucket '{ROOT_S3_DIR}' created and exists.")
    print(response)
except Exception as error:
    pass


# parse directory and upload files

for dir_name, subdir_list, file_list in os.walk(ROOT_DIR, topdown=True):
    if dir_name != ROOT_DIR:
        for fname in file_list:
            upload_file(ROOT_S3_DIR,"%s/" % dir_name[2:], "%s/%s" % (dir_name,
fname), fname)


print("done")
```









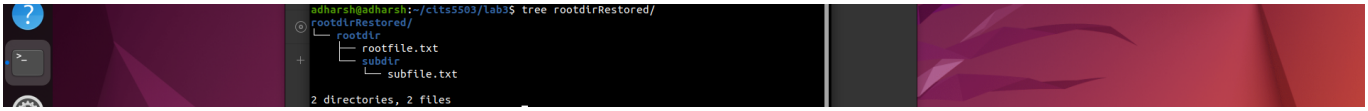
[3] Restore from S3

- Code to retrieve data from the same S3 bucket (Reference : link):

```
import os
import boto3

def download_file(bucket, s3_path, local_path):
```

```
        try:
            s3.download_file(bucket, s3_path, local_path)
            print(f"Downloaded {s3_path} to {local_path}")
        except Exception as e:
            print(f"An error occurred while downloading {s3_path}: {e}")

    # Constants
    ROOT_DIR = './rootdirRestored' # The directory where files will be restored
    s3 = boto3.client("s3")
    bucket = '23796349-cloudstorage'

    # Create a "rootdir" for restored data
    if not os.path.exists(ROOT_DIR):
        os.makedirs(ROOT_DIR)

    # Fetching list of all objects from the bucket
    try:
        response = s3.list_objects(Bucket=bucket)
    except Exception as e:
        print(f"An error occurred: {e}")
        exit(1)

    # Download each file to the appropriate directory
    for item in response['Contents']:
        s3_path = item['Key']
        local_path = os.path.join(ROOT_DIR, s3_path)

        # Create directory if it doesn't exist
        local_dir = os.path.dirname(local_path)
        if not os.path.exists(local_dir):
            os.makedirs(local_dir)

        # Download the file
        download_file(bucket, s3_path, local_path)

    print("Restore completed.")
```

- Screenshot below shows the file structure of the restored data from s3:



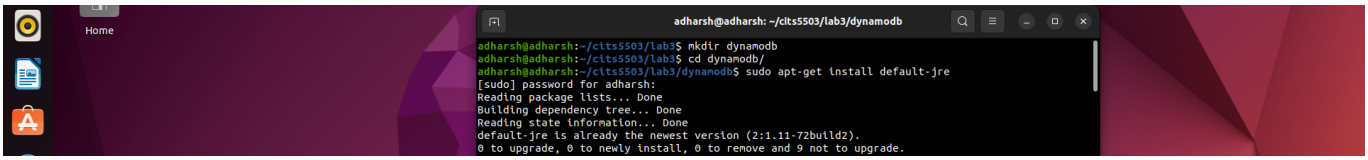
[4] Write information about files to DynamoDB

- The required directory was created and the current directory was changed using the following commands:

```
```

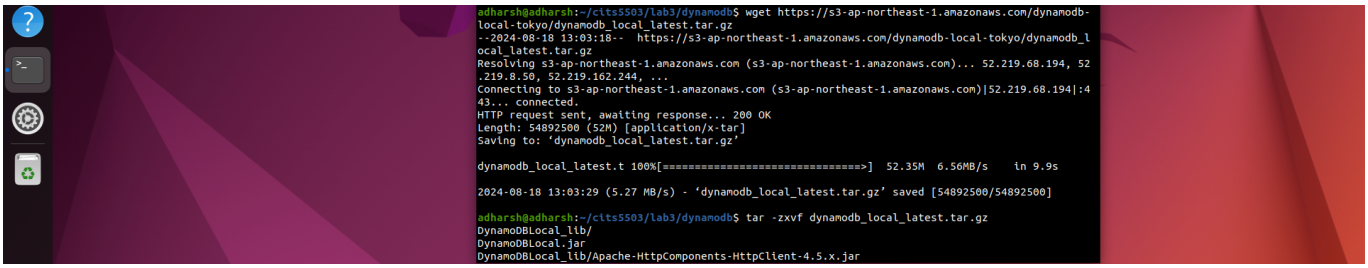
```
mkdir dynamodb
cd dynamodb
```

- The screenshot below shows that I have the updated version of jre already installed:



- The required file was downloaded and its contents were extracted using the following commands:

```
wget https://s3-ap-northeast-1.amazonaws.com/dynamodb-local-
tokyo/dynamodb_local_latest.tar.gz

tar -zxvf dynamodb_local_latest.tar.gz
```

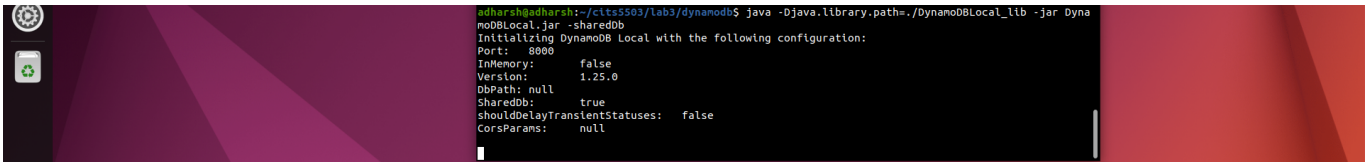


- Local instance of DynamboDB can be started using the following commands:

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
```

or

```
docker run -p 8000:8000 amazon/dynamodb-local -jar DynamoDBLocal.jar -inMemory -
sharedDb
```



***NOTE**: All the steps below were done on a new terminal, keeping the old one open.*

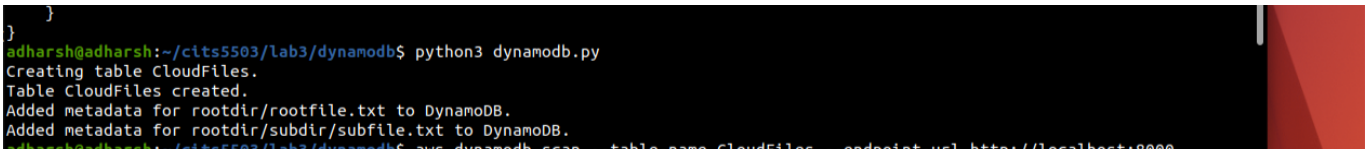
- Python code to create a DynamoDB table named **CloudFiles** and write metadata of each file that was uploaded to s3 (References: 1, 2, 3):

```
```

```python
import boto3
from datetime import datetime

# Initialize DynamoDB resource pointing to local instance
dynamodb = boto3.resource('dynamodb', endpoint_url='http://localhost:8000')
table_name = 'CloudFiles'

# Check if table already exists
try:
    table = dynamodb.create_table(
        TableName=table_name,
        KeySchema=[
            {
                'AttributeName': 'userId',
                'KeyType': 'HASH' # Primary key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'userId',
                'AttributeType': 'S' # String
            }
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 1,
            'WriteCapacityUnits': 1
        }
    )
    print(f"Creating table {table_name}.")
    table.meta.client.get_waiter('table_exists').wait(TableName=table_name)
    print(f"Table {table_name} created.")
except dynamodb.meta.client.exceptions.ResourceInUseException:
    # Table already exists, retrieve it
    table = dynamodb.Table(table_name)
    print(f"Table {table_name} already exists.")
# Initialize S3 client
s3 = boto3.client('s3')
bucket_name = '23796349-cloudstorage'

try:
    # Get list of all objects from the bucket
    objects = s3.list_objects_v2(Bucket=bucket_name)['Contents']
    for obj in objects:
        file_name = obj['Key']
        last_modified = obj['LastModified'].strftime("%Y-%m-%d %H:%M:%S")
        # Skip the folder itself
        if file_name.endswith('/'):
            continue
        # Get Object ACL
        acl = s3.get_object_acl(Bucket=bucket_name, Key=file_name)
        #print(acl)
        owner = acl['Owner']['ID']
        permissions = [grant['Permission'] for grant in acl['Grants']]
        # Insert metadata into DynamoDB
```

```
            table.put_item(
                Item={
                    'userId': '23796349', #Since I am the user of this bucket
                    'fileName': file_name,
                    'path': f"s3://{bucket_name}/{file_name}",
                    'lastUpdated': last_modified,
                    'owner': owner,
                    'permissions': ','.join(permissions)
                }
            )
            print(f"Added metadata for {file_name} to DynamoDB.")
    except Exception as e:
        print(f"An error occurred: {e}")
```

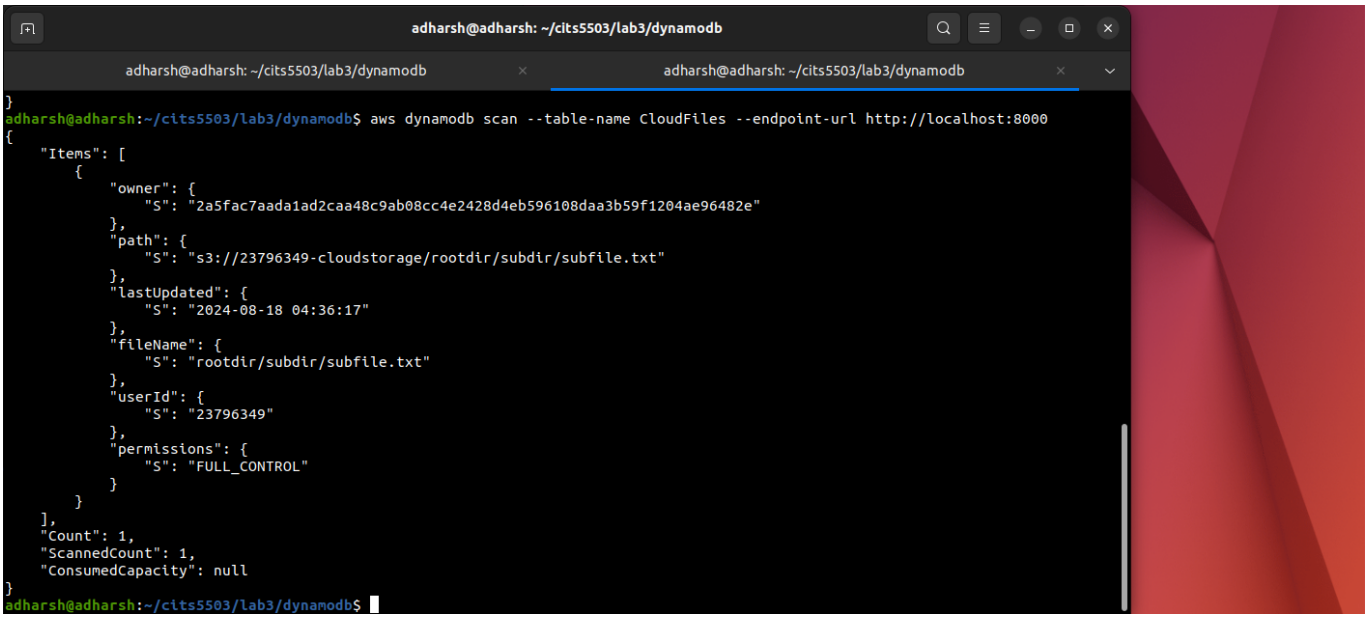
- *NOTE:* I have used **Owner's ID**, as my region is not within the listed regions.



[5] Scan the table

- The following command can be used to read the contents of the **CloudFiles** table:

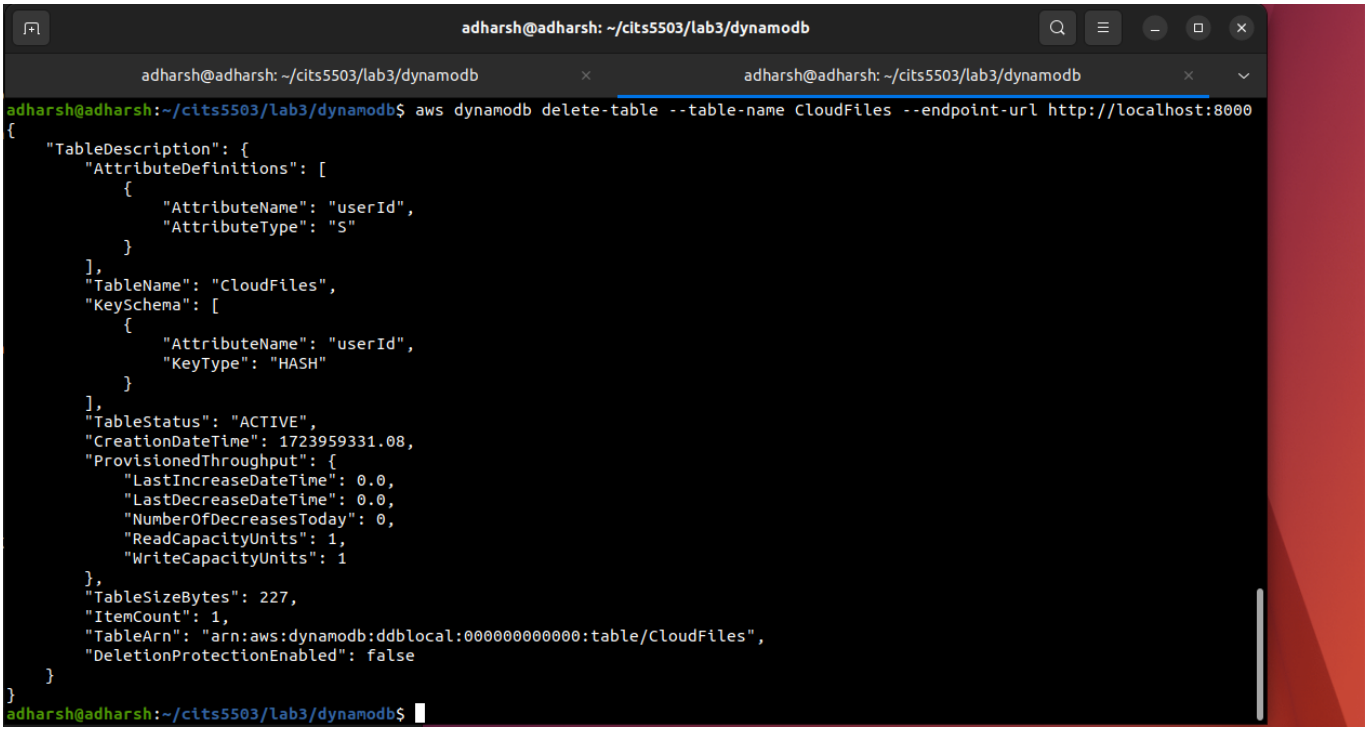
```
aws dynamodb scan --table-name CloudFiles --endpoint-url http://localhost:8000
```



[6] Delete the table

- The following command can be used to delete the **CloudFiles** table:

```
aws dynamodb delete-table --table-name CloudFiles --endpoint-url
http://localhost:8000
```



- All the resources created for this lab were deleted via the AWS Console after completion.

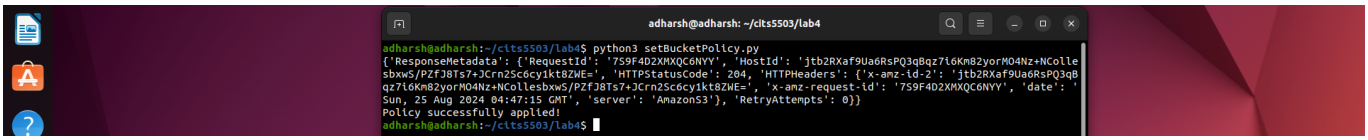
Lab 4

Apply a policy to restrict permissions on bucket

[1] Write a Python script

- Before enforcing a policy, the appropriate resource needs to be created.
- Although the lab instructions ask us to use the S3 bucket created in **lab3**, the instructions in the same lab ask us to **delete** the resource after completion.
- Hence a new S3 bucket with the same name and objects was created using the AWS console.
- The following script applies the specified policy to the specified S3 bucket (Reference: link):

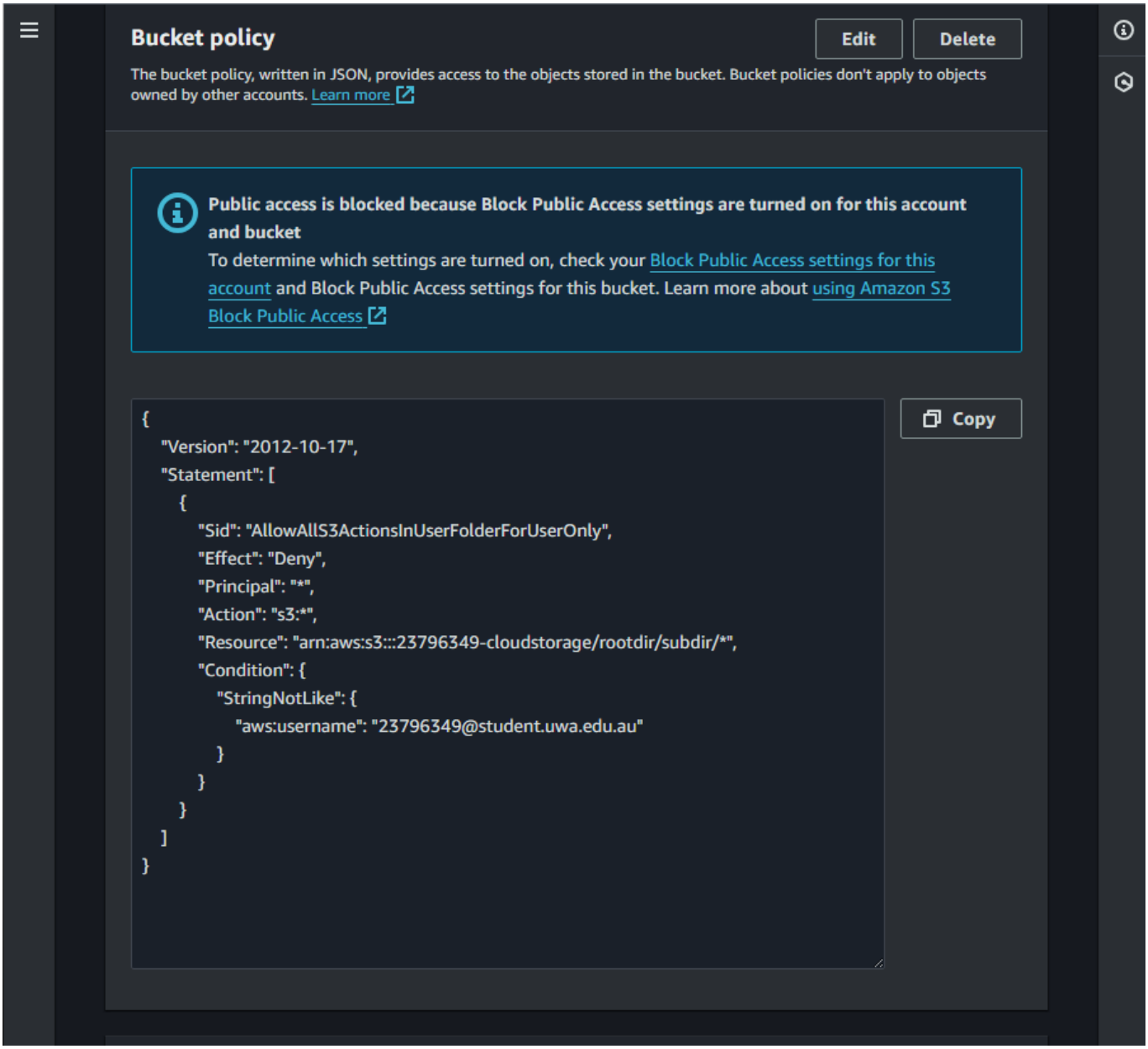
```
import boto3
import json

# Initialize a session using Amazon S3
s3 = boto3.client('s3')

# Variables
bucket_name = '23796349-cloudstorage'
student_number = '23796349'

# Defining policy
policy = {
    "Version": "2012-10-17",
    "Statement": {
        "Sid": "AllowAllS3ActionsInUserFolderForUserOnly",
        "Effect": "DENY",
        "Principal": "*",
        "Action": "s3:*",
        "Resource": f"arn:aws:s3:::{bucket_name}/rootdir/subdir/*",
        "Condition": {
            "StringNotLike": {
                "aws:username": f"{student_number}@student.uwa.edu.au"
            }
        }
    }
}

# Convert the policy to a JSON string
policy_string = json.dumps(policy)

# Apply the bucket policy
response = s3.put_bucket_policy(
    Bucket=bucket_name,
    Policy=policy_string
)

# Printing response
print(response)
print("Policy successfully applied!")
```



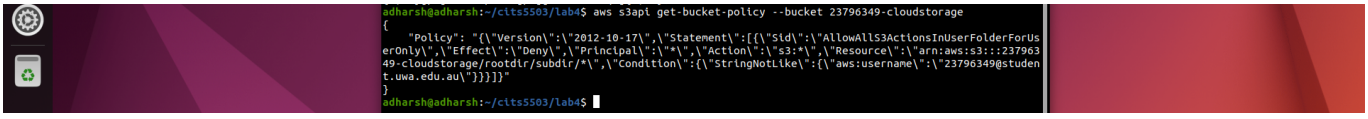
[2] Check if the script works

- The screenshots below prove that the script works:

The policy on the console



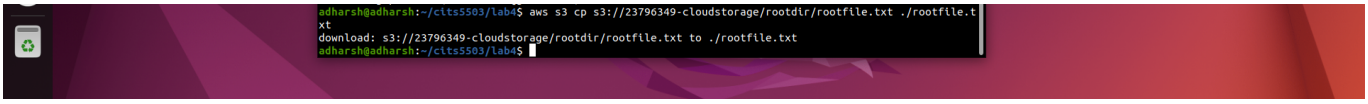
The policy using aws cli



Trying to download the rootfile.txt using an username without access

```
adharsh@adharsh:~/cits5503/lab4$ aws s3 cp s3://23796349-cloudstorage/rootdir/rootfile.txt ./rootfile.t
xt --profile adharsh136@gmail.com
fatal error: An error occurred (403) when calling the HeadObject operation: Forbidden
```

Trying to download the rootfile.txt using the username with access



AES Encryption using KMS

[1] Create a KMS key

- The code below creates a KMS key with my student number as its alias (Reference: link):

```
import boto3
import json

# Initialize a session using Amazon KMS
session = boto3.Session(region_name='ap-northeast-2')
kms_client = session.client('kms')

# Create the KMS key
try:
    response = kms_client.create_key(
    Description='KMS key for lab4',
    KeyUsage='ENCRYPT_DECRYPT',
    Origin='AWS_KMS',
    BypassPolicyLockoutSafetyCheck=False
    )
    key_id = response['KeyMetadata']['KeyId']
    print(f"Created KMS key with id: {key_id}")
except Exception as e:
    print(f"Failed to create key: {e}")


key_alias = "alias/23796349_KMS"
# Add the alias
try:
    kms_client.create_alias(
    AliasName=key_alias,
    TargetKeyId=key_id
    )
    print(f"Successfully attached alias {key_alias} to KMS key.")
except kms_client.exceptions.AlreadyExistsException:
    print(f"Alias {key_alias} already exists. Deleting existing alias.")

    # Delete the existing alias
    try:
        kms_client.delete_alias(
        AliasName=key_alias
        )
        print(f"Successfully deleted existing alias {key_alias}.")
```

```
    # Create the new alias
        kms_client.create_alias(
        AliasName=key_alias,
        TargetKeyId=key_id
        )

        print(f"Successfully attached new alias {key_alias} to KMS key.")
    except Exception as e:
        print(f"Failed to replace alias: {e}")
except Exception as e:
    print(f"Failed to attach alias: {e}")
```

[2] Attach a policy to the created KMS key

- The code in the previous step is updated to define a KMS key policy and attach the same to the KMS key (Reference: link):

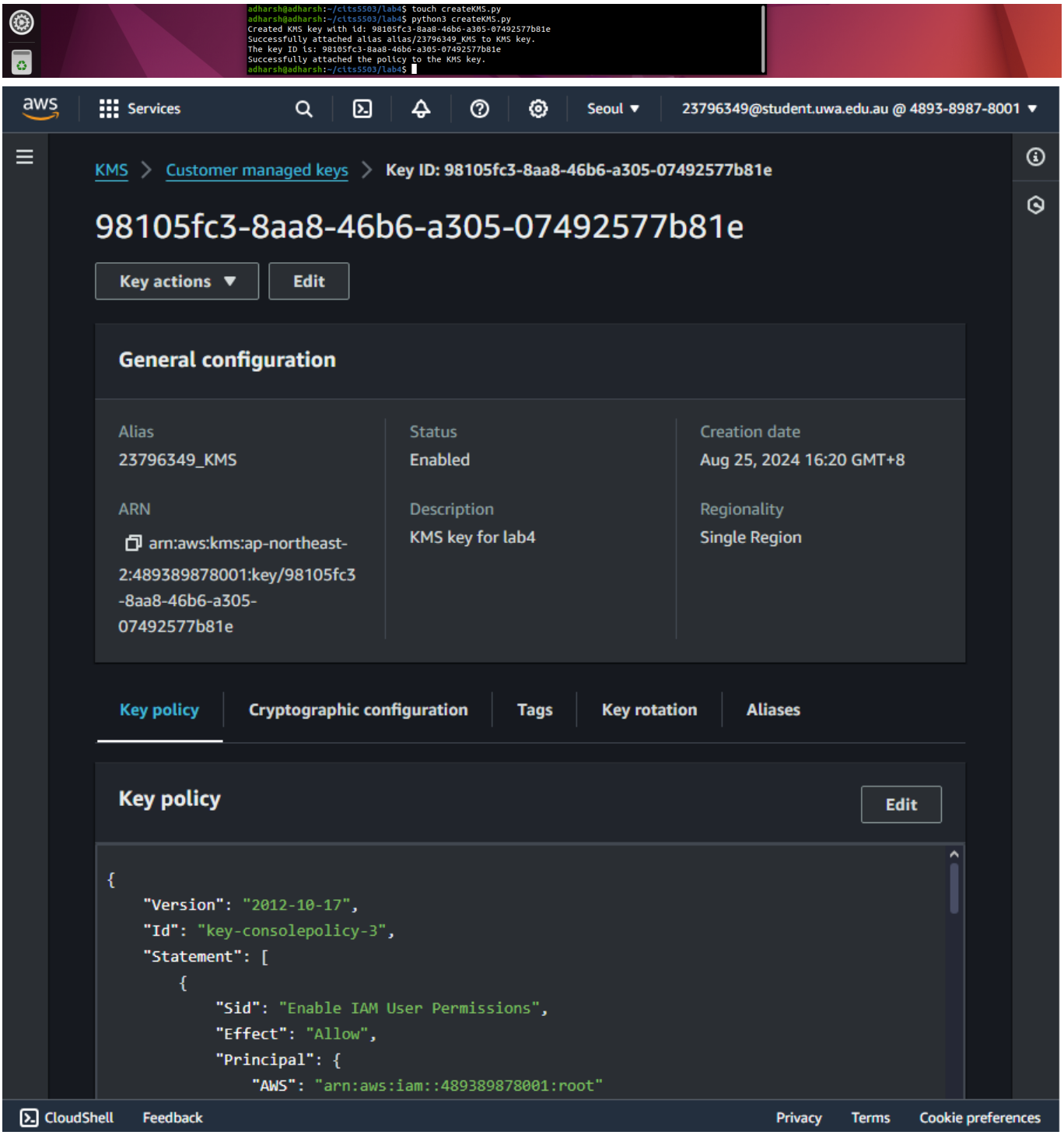
```
import boto3
import json

# Initialize a session using Amazon KMS
session = boto3.Session(region_name='ap-northeast-2')
kms_client = session.client('kms')

# Create the KMS key
try:
    response = kms_client.create_key(
    Description='KMS key for lab4',
    KeyUsage='ENCRYPT_DECRYPT',
    Origin='AWS_KMS',
    BypassPolicyLockoutSafetyCheck=False
    )
    key_id = response['KeyMetadata']['KeyId']
    print(f"Created KMS key with id: {key_id}")
except Exception as e:
    print(f"Failed to create key: {e}")


key_alias = "alias/23796349_KMS"
# Add the alias
try:
    kms_client.create_alias(
    AliasName=key_alias,
    TargetKeyId=key_id
    )
    print(f"Successfully attached alias {key_alias} to KMS key.")
except kms_client.exceptions.AlreadyExistsException:
    print(f"Alias {key_alias} already exists. Deleting existing alias.")

    # Delete the existing alias
    try:
```

```
            kms_client.delete_alias(
            AliasName=key_alias
            )
            print(f"Successfully deleted existing alias {key_alias}.")

        # Create the new alias
            kms_client.create_alias(
            AliasName=key_alias,
            TargetKeyId=key_id
            )

            print(f"Successfully attached new alias {key_alias} to KMS key.")
        except Exception as e:
            print(f"Failed to replace alias: {e}")
    except Exception as e:
        print(f"Failed to attach alias: {e}")

    #-------------------------UPDATED SECTION----------------------------------
    #Defining key policy

    key_policy={
      "Version": "2012-10-17",
      "Id": "key-consolepolicy-3",
      "Statement": [
        {
          "Sid": "Enable IAM User Permissions",
          "Effect": "Allow",
          "Principal": {
            "AWS": "arn:aws:iam::489389878001:root"
          },
          "Action": "kms:*",
          "Resource": "*"
        },
        {
          "Sid": "Allow access for Key Administrators",
          "Effect": "Allow",
          "Principal": {
            "AWS": "arn:aws:iam::489389878001:user/23796349@student.uwa.edu.au"
          },
          "Action": [
            "kms:Create*",
            "kms:Describe*",
            "kms:Enable*",
            "kms:List*",
            "kms:Put*",
            "kms:Update*",
            "kms:Revoke*",
            "kms:Disable*",
            "kms:Get*",
            "kms:Delete*",
            "kms:TagResource",
            "kms:UntagResource",
            "kms:ScheduleKeyDeletion",
            "kms:CancelKeyDeletion"
          ],
```

```
        "Resource": "*"
      },
      {
        "Sid": "Allow use of the key",
        "Effect": "Allow",
        "Principal": {
          "AWS": "arn:aws:iam::489389878001:user/23796349@student.uwa.edu.au"
        },
        "Action": [
          "kms:Encrypt",
          "kms:Decrypt",
          "kms:ReEncrypt*",
          "kms:GenerateDataKey*",
          "kms:DescribeKey"
        ],
        "Resource": "*"
      },
      {
        "Sid": "Allow attachment of persistent resources",
        "Effect": "Allow",
        "Principal": {
          "AWS": "arn:aws:iam::489389878001:user/23796349@student.uwa.edu.au"
        },
        "Action": [
          "kms:CreateGrant",
          "kms:ListGrants",
          "kms:RevokeGrant"
        ],
        "Resource": "*",
        "Condition": {
          "Bool": {
            "kms:GrantIsForAWSResource": "true"
          }
        }
      }
    ]
}
# Put key policy
try:
    kms_client.put_key_policy(
    KeyId=key_id,
    PolicyName='default',
    Policy=json.dumps(key_policy, indent=4),
    BypassPolicyLockoutSafetyCheck=False
    )
    print(f"The key ID is: {key_id}")
    print("Successfully attached the policy to the KMS key.")
except Exception as e:
    print(f"Failed to attach policy: {e}")
```

[3] Check whether the script works

- Screenshots below show the policy attached to the KMS key seen in the AWS Console:



***NOTE**: Here for both the key administrator and the key user, the ARN is the same, arn:aws:iam::489389878001:user/23796349@student.uwa.edu.au*

[4] Use the created KMS key for encryption/decryption

- The code below uses the created KMS key to encrypt/decrypt every file in the bucket (Reference: 1, 2)
- I have used argument parser to have the choice between encrypting and decrypting files (options **-e** to encrypt and **-d** to decrypt files).

```
import os
import boto3
import argparse
from botocore.exceptions import NoCredentialsError
```

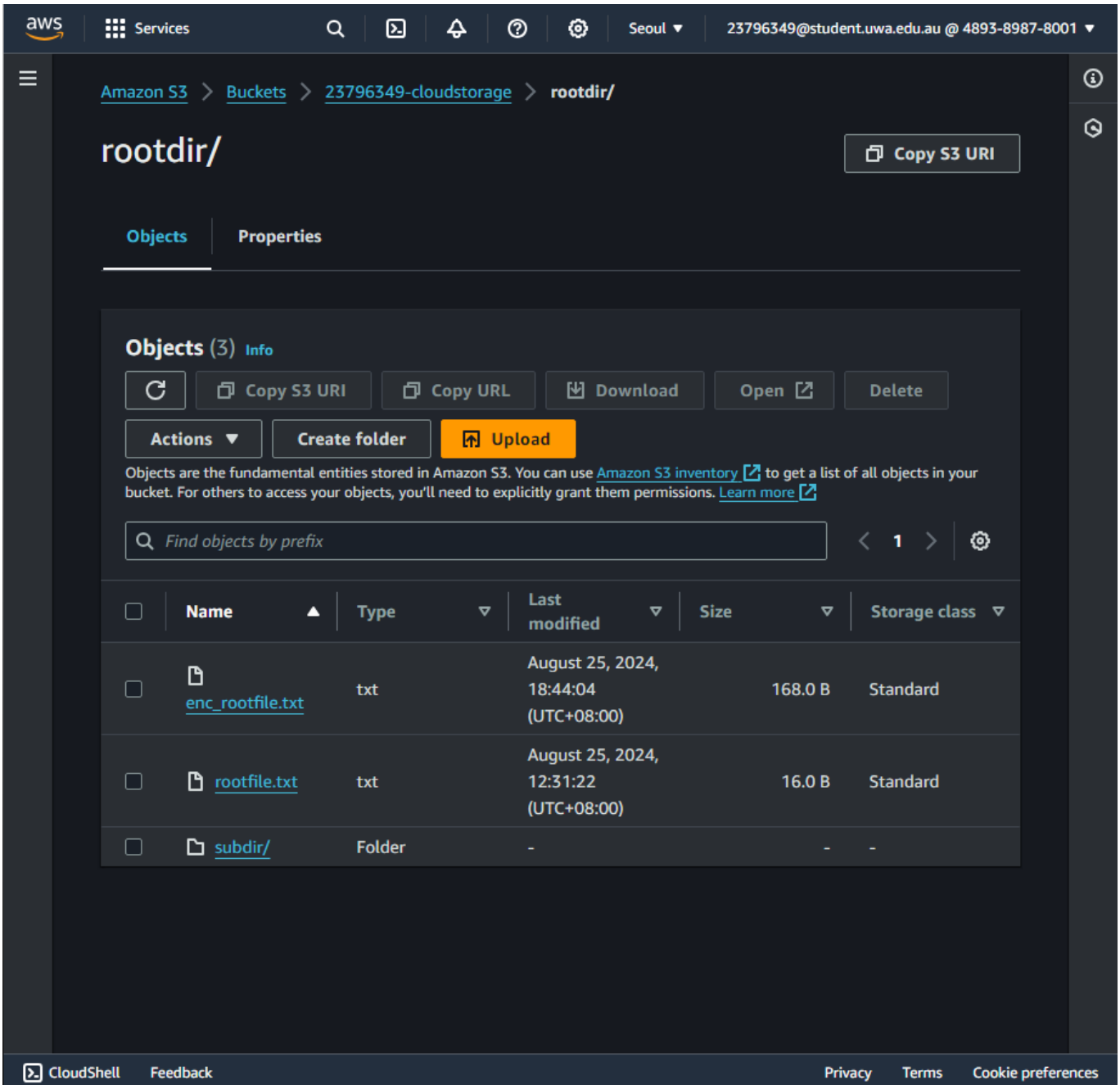
```python
# Initialize KMS client
kms = boto3.client('kms')
# Function to encrypt data using KMS key
def encrypt_file(file_path, kms_key_id):
    with open(file_path, 'rb') as f:
        plaintext = f.read()
    response = kms.encrypt(KeyId=kms_key_id, Plaintext=plaintext)
    ciphertext = response['CiphertextBlob']
    return ciphertext

# Function to decrypt data using KMS key
def decrypt_file(ciphertext, kms_key_id):
    response = kms.decrypt(KeyId=kms_key_id, CiphertextBlob=ciphertext)
    plaintext = response['Plaintext']
    return plaintext

# Argument Parsing
parser = argparse.ArgumentParser(description='KMS_Encrypt_Decrypt')
parser.add_argument('--encrypt', '-e', help='KMS key ID for encrypting files')
parser.add_argument('--decrypt', '-d', help='KMS key ID for decrypting files')
args = parser.parse_args()

# Constants and Initialization
ROOT_DIR = './rootdir'
DECRYPT_DIR = './rootdir_de'
s3 = boto3.client("s3")
bucket_name = '23796349-cloudstorage'

# Function to upload files
def upload_file(bucket_name, folder_name, file_data, file_name):
    s3_path = f"{folder_name}/enc_{file_name}"
    s3.put_object(Body=file_data, Bucket=bucket_name, Key=s3_path)
    print(f"Uploaded encrypted {file_name} to {bucket_name}/{s3_path}")

# Function to download and decrypt files
def download_and_decrypt_file(bucket_name, s3_path, local_path, kms_key_id):
    response = s3.get_object(Bucket=bucket_name, Key=s3_path)
    encrypted_data = response['Body'].read()
    decrypted_data = decrypt_file(encrypted_data, kms_key_id)
    with open(local_path, 'wb') as f:
        f.write(decrypted_data)
    print(f"Downloaded and decrypted {s3_path} to {local_path}")

# Walk through the directory and upload files
if args.encrypt:
    for dir_name, _, file_list in os.walk(ROOT_DIR):
        relative_folder = os.path.relpath(dir_name, ROOT_DIR)
        if relative_folder == '.':
            relative_folder = 'rootdir'
        else:
            relative_folder = 'rootdir/' + relative_folder
        for fname in file_list:
            full_path = os.path.join(dir_name, fname)
            encrypted_data = encrypt_file(full_path, args.encrypt)
            upload_file(bucket_name, relative_folder, encrypted_data, fname)
```

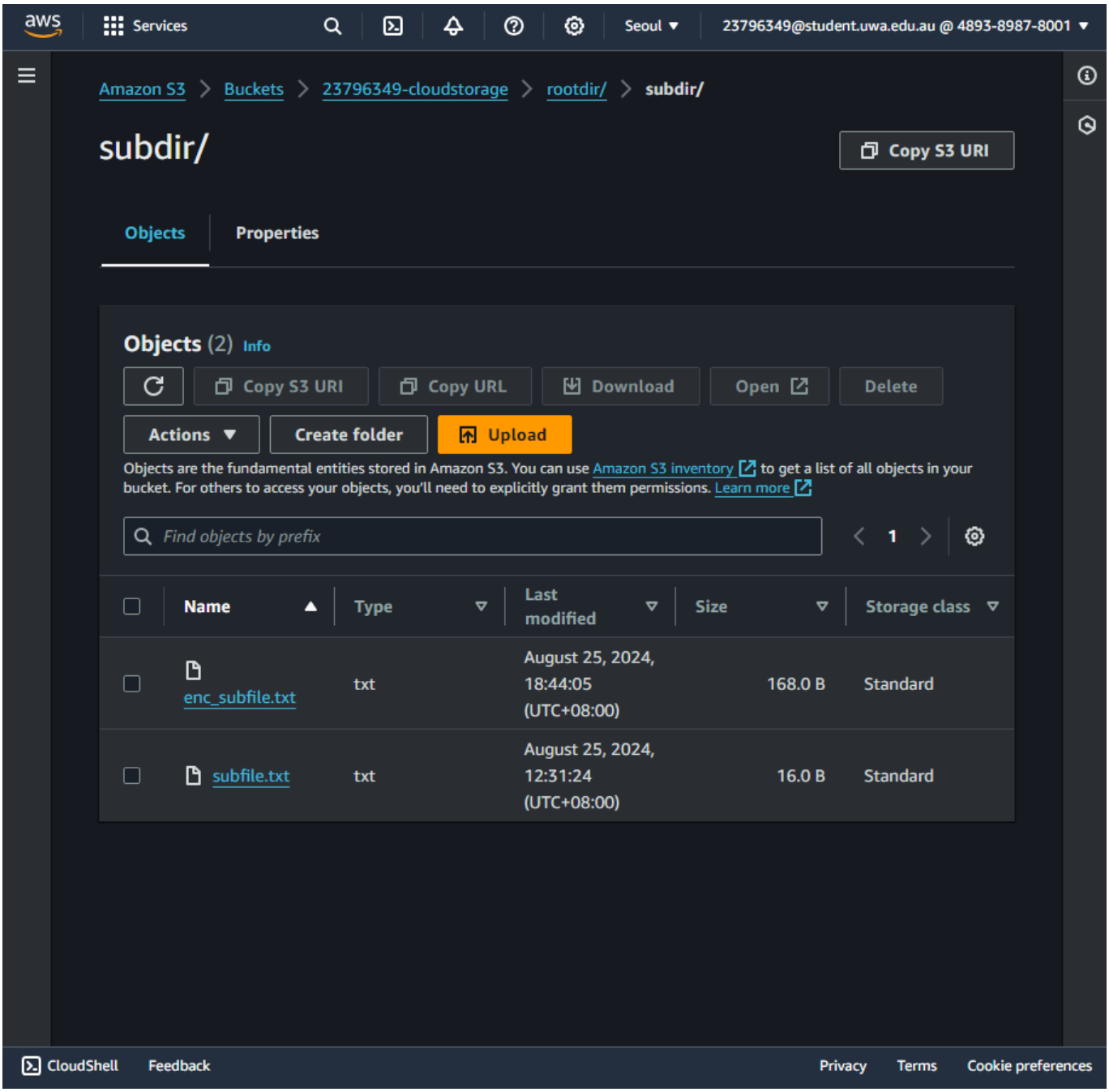
```
# Download and decrypt files
if args.decrypt:
    os.makedirs(DECRYPT_DIR, exist_ok=True)
    s3_resource = boto3.resource('s3')
    my_bucket = s3_resource.Bucket(bucket_name)
    for obj in my_bucket.objects.all():
        s3_path = obj.key
        local_path = os.path.join(DECRYPT_DIR, '/'.join(s3_path.split('/')[1:]))
        local_dir = os.path.dirname(local_path)
        os.makedirs(local_dir, exist_ok=True)
        print(s3_path)
        if 'enc_' in s3_path:
            download_and_decrypt_file(bucket_name, s3_path,
local_path.replace('enc_','dec_'), args.decrypt)
print("Done")
```

- Screenshots below show the encryption and decryption operations:

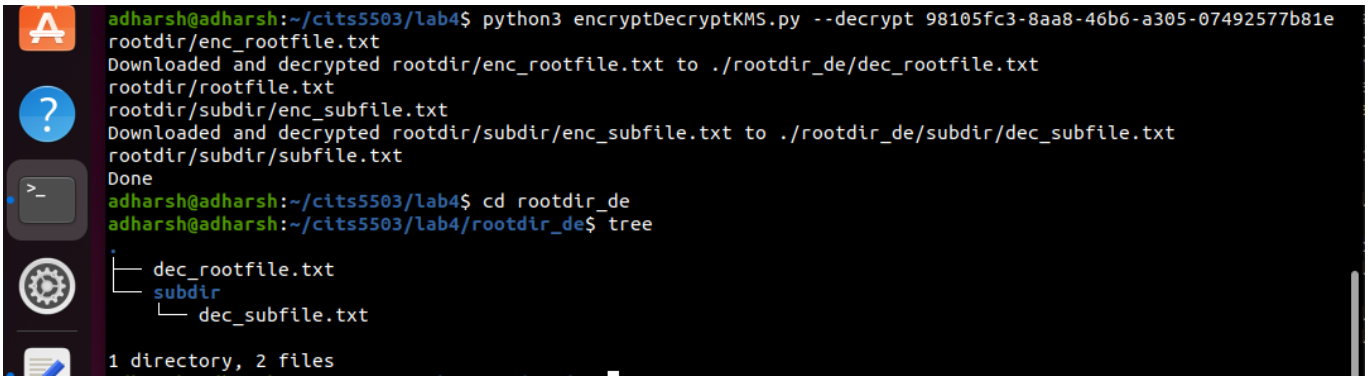
Encrypt

```
adharsh@adharsh:~/cits5503/lab4$ python3 encryptDecryptKMS.py --encrypt 98105fc3
-8aa8-46b6-a305-07492577b81e
Uploaded encrypted rootfile.txt to 23796349-cloudstorage/rootdir/enc_rootfile.tx
t
Uploaded encrypted subfile.txt to 23796349-cloudstorage/rootdir/subdir/enc_subfi
le.txt
Done
```





Decrypt



[5] Apply `pycryptodome` for encryption/decryption

- The code below uses the pycryptodome library to encrypt/decrypt every file in the bucket (Adopted from: link).

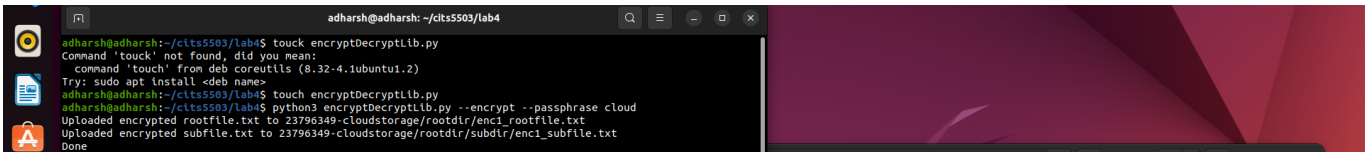
- Again, I have used argument parser to choice between encryption and decryption of files (options **-e** to encrypt and **-d** to decrypt files).
- Additionally, as the ecryption/decryption function requires a passphrase, option **-p** was introduced to facilitate this.

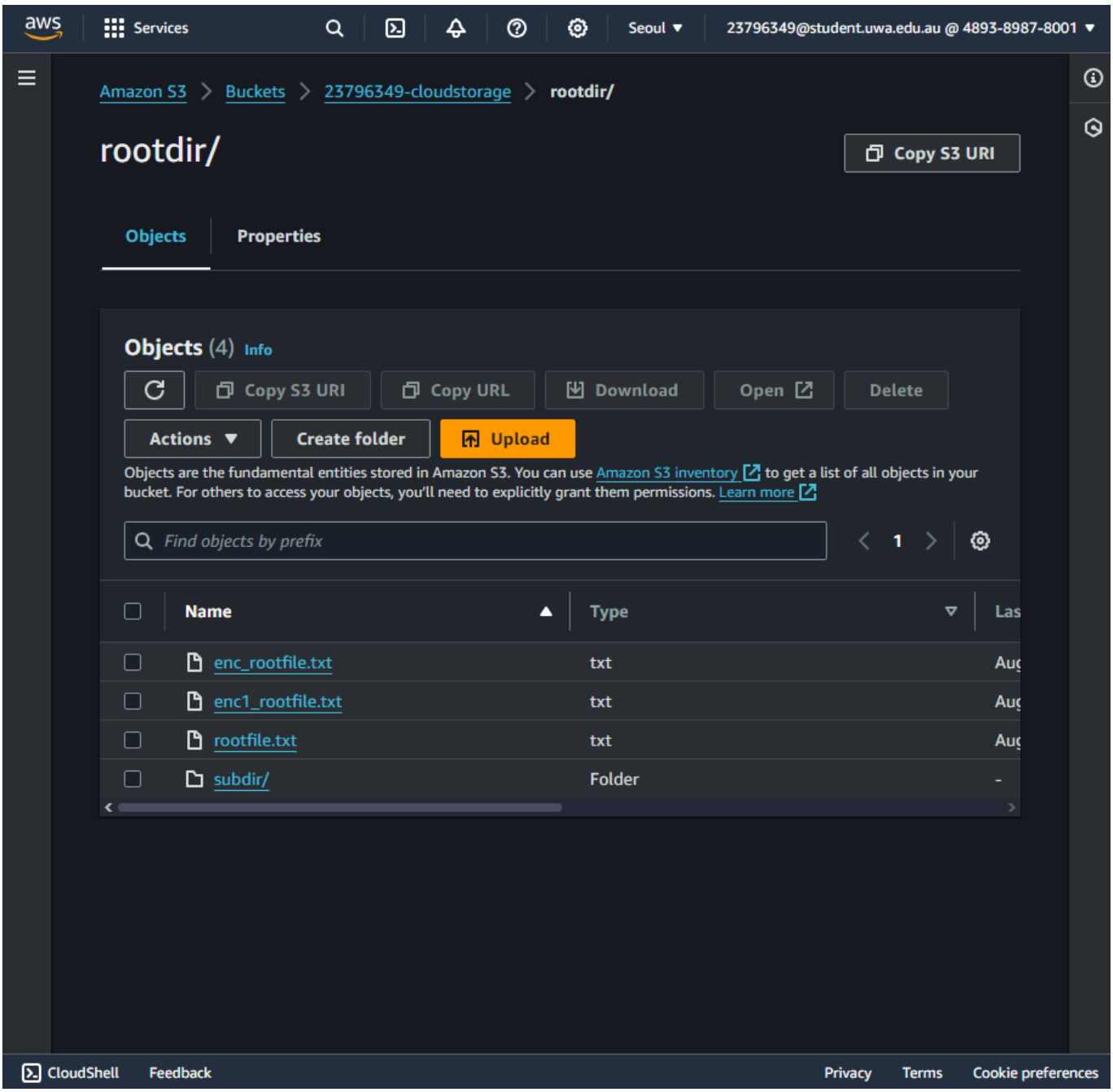
```
import os, struct, hashlib, argparse
import boto3
from botocore.exceptions import NoCredentialsError
from Crypto.Cipher import AES
from Crypto import Random

CHUNK_SIZE = 64 * 1024

# Argument Parsing
parser = argparse.ArgumentParser(description='Encrypt_decrypt_pycryptodome')
parser.add_argument('--encrypt', '-e', action='store_true', help='Encrypt files')
parser.add_argument('--decrypt', '-d', action='store_true', help='Decrypt files')
parser.add_argument('--passphrase', '-p', required=True, help='Passphrase for
encryption/decryption')
args = parser.parse_args()

# Constants and Initialization
ROOT_DIR = './rootdir'
DECRYPT_DIR = './rootdir_de2'
s3 = boto3.client("s3")
bucket_name = '23796349-cloudstorage'

# Function to encrypt data using passphrase
def encrypt_file(file_path, passphrase):
    key = hashlib.sha256(passphrase.encode("utf-8")).digest()
    iv = Random.new().read(AES.block_size)
    encryptor = AES.new(key, AES.MODE_CBC, iv)
    filesize = os.path.getsize(file_path)
    with open(file_path, 'rb') as infile:
        encrypted_data = struct.pack('<Q', filesize) + iv
        while True:
            chunk = infile.read(CHUNK_SIZE)
            if len(chunk) == 0:
                break
            elif len(chunk) % 16 != 0:
                chunk += ' '.encode("utf-8") * (16 - len(chunk) % 16)
            encrypted_data += encryptor.encrypt(chunk)
    return encrypted_data

# Function to decrypt data using passphrase
def decrypt_file(encrypted_data, passphrase):
    key = hashlib.sha256(passphrase.encode("utf-8")).digest()
    origsize = struct.unpack('<Q', encrypted_data[:8])[0]
    iv = encrypted_data[8:24]
    decryptor = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = encrypted_data[24:]

    plaintext = b''
```

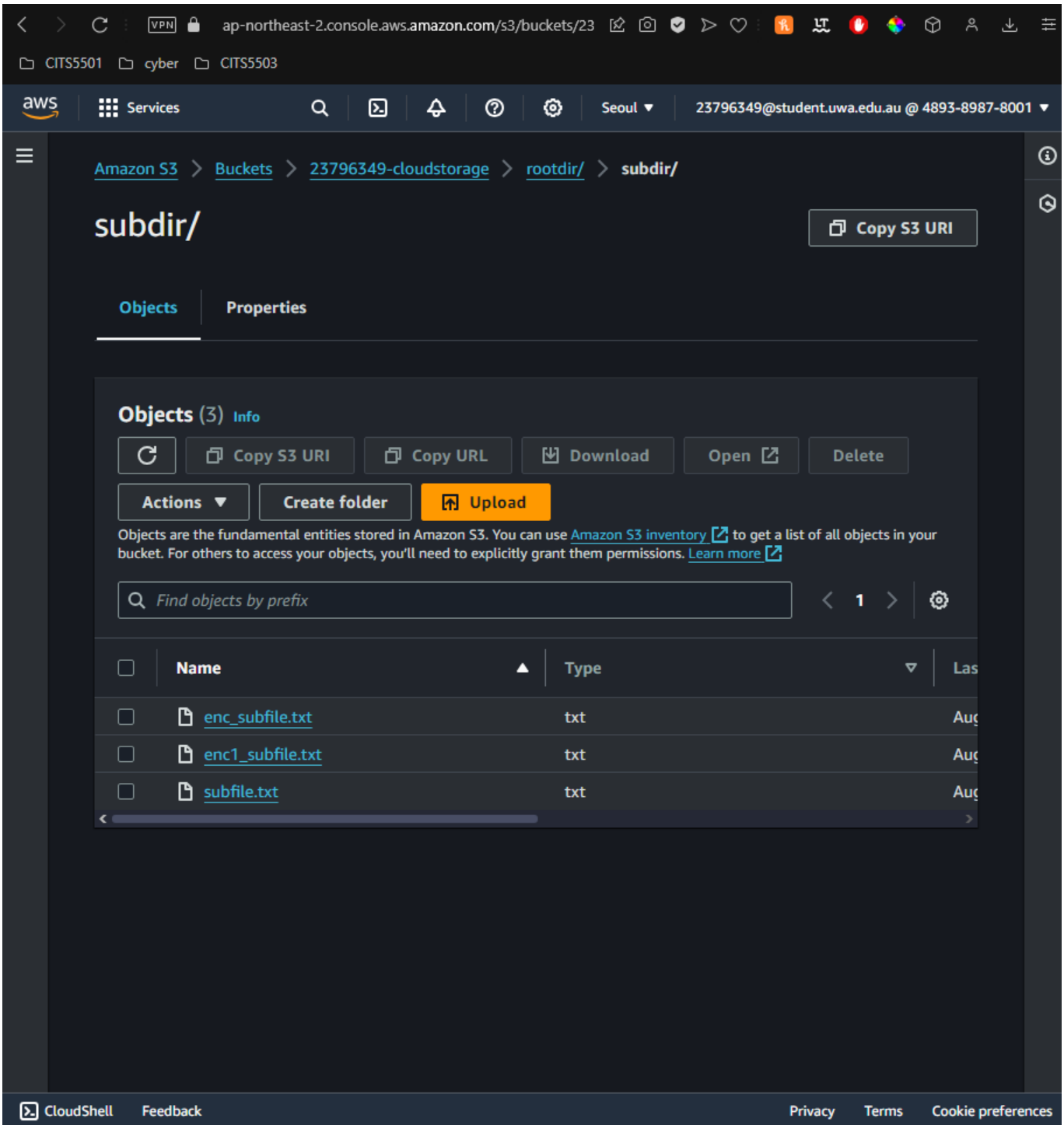
```
    for i in range(0, len(ciphertext), CHUNK_SIZE):
        chunk = ciphertext[i:i + CHUNK_SIZE]
        plaintext += decryptor.decrypt(chunk)
    return plaintext[:origsize]

# Function to upload files
def upload_file(bucket_name, folder_name, file_data, file_name):
    s3_path = f"{folder_name}/enc1_{file_name}"
    s3.put_object(Body=file_data, Bucket=bucket_name, Key=s3_path)
    print(f"Uploaded encrypted {file_name} to {bucket_name}/{s3_path}")

# Function to download and decrypt files
def download_and_decrypt_file(bucket_name, s3_path, local_path, passphrase):
    response = s3.get_object(Bucket=bucket_name, Key=s3_path)
    encrypted_data = response['Body'].read()
    decrypted_data = decrypt_file(encrypted_data, passphrase)
    with open(local_path, 'wb') as f:
        f.write(decrypted_data)
    print(f"Downloaded and decrypted {s3_path} to {local_path}")

# Walk through the directory and upload files
if args.encrypt:
    for dir_name, _, file_list in os.walk(ROOT_DIR):
        relative_folder = os.path.relpath(dir_name, ROOT_DIR)
        if relative_folder == '.':
            relative_folder = 'rootdir'
        else:
            relative_folder = 'rootdir/' + relative_folder
        for fname in file_list:
            full_path = os.path.join(dir_name, fname)
            encrypted_data = encrypt_file(full_path, args.passphrase)
            upload_file(bucket_name, relative_folder, encrypted_data, fname)

# Download and decrypt files
if args.decrypt:
    os.makedirs(DECRYPT_DIR, exist_ok=True)
    s3_resource = boto3.resource('s3')
    my_bucket = s3_resource.Bucket(bucket_name)
    for obj in my_bucket.objects.all():
        s3_path = obj.key
        local_path = os.path.join(DECRYPT_DIR, '/'.join(s3_path.split('/')[1:]))
        local_dir = os.path.dirname(local_path)
        os.makedirs(local_dir, exist_ok=True)
        if 'enc1_' in s3_path:
            download_and_decrypt_file(bucket_name, s3_path,
local_path.replace('enc1_','dec1_'), args.passphrase)
print("Done")
```

- Screenshots below show the encryption and decryption operations:

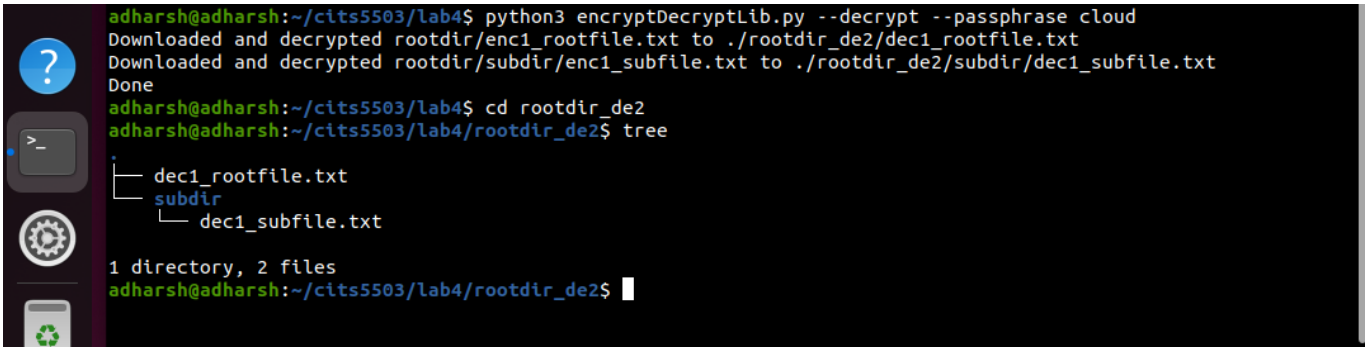
Encrypt


adharsh@adharsh: ~/cits5503/lab4
adharsh@adharsh:~/cits5503/lab4$ touck encryptDecryptLib.py
Command 'touck' not found, did you mean:
command 'touch' from deb coreutils (8.32-4.1ubuntu1.2)
Try: sudo apt install <deb name>
adharsh@adharsh:~/cits5503/lab4$ touch encryptDecryptLib.py
adharsh@adharsh:~/cits5503/lab4$ python3 encryptDecryptLib.py --encrypt --passphrase cloud
Uploaded encrypted rootfile.txt to 23796349-cloudstorage/rootdir/enc1_rootfile.txt
Uploaded encrypted subfile.txt to 23796349-cloudstorage/rootdir/subdir/enc1_subfile.txt
Done


aws
Services
Seoul
23796349@student.uwa.edu.au @ 4893-8987-8001
Amazon S3 > Buckets > 23796349-cloudstorage > rootdir/
rootdir/
Copy S3 URI
Objects
Properties
Objects (4) Info
Copy S3 URI
Copy URL
Download
Open
Delete
Actions
Create folder
Upload
Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more
Find objects by prefix
1
Name
Type
Las
enc_rootfile.txt
txt
Aug
enc1_rootfile.txt
txt
Aug
rootfile.txt
txt
Aug
subdir/
Folder
-
CloudShell
Feedback
Privacy
Terms
Cookie preferences



Decrypt



Answer the following question

What is the performance difference between using KMS and using the custom solution?

AWS KMS:

- Since KMS is a managed service by AWS, it's easier to implement and scale to all resources.
- The Encryption/Decryption operations are offloaded to KMS, so overhead on the user's or application's resources is reduced.
- Additionally, features such as key rotation, key versioning, etc. are available.

Custom Solution:

- Custom solution are important if there is specific or unique encryption/decryption need.
- The performance can vary depending on various factors such as algorithm, code quality, implentation technique, etc and there's a greater control over the cryptographic operations.
- There could be a performance hit if the encryption/decryption operation is performed directly by the user's or application's resources.

At the end of the day, its over ease of use, cost-effectiveness and seamless integration or greater control, complexity and specific need.

All resources that were created for this lab were deleted using AWS Console after completion.

Lab 5

Application Load Balancer

[1] Create 2 EC2 instances

- Initially, I had to find the **VPC ID** of my allocated region and also the respective **subnet IDs**.
- The following AWS CLI command was used to find the VPC ID:

```
aws ec2 describe-vpcs --filters "Name=isDefault,Values=true"
```

- To get the subnets IDs specific to a VPC, the following AWS CLI command:

```
aws ec2 describe-subnets --filters "Name=vpc-id,Values=vpc-01f842220d0070f97"
```

NOTE: I have used the VPC ID in my allocated region in the **Values** field.

- The code below creates **two EC2** instances and a security group that authorise inbound traffic for **HTTP** and **SSH** (Reference: Lab 2).

NOTE: The **key-pair** created in Lab 2 was used here. Also, I chose availability zones A and C since I am creating instances of the type **t2.micro**.

```
import boto3
import time

# Constants

#Using the same key-pair created for Lab2
KEY_NAME = '23796349-key'

#The AMI ID for my region
AMI_ID = 'ami-056a29f2eddc40520'

#The default subnets a and c IDs respectively
SUBNETS = [ 'subnet-09f996d1ad81767a9', 'subnet-0b3601189181ee48e']

#The default VPC ID
VPC_ID = 'vpc-01f842220d0070f97'

#The two availablility zones in the region
AVAILABILITY_ZONES = ['ap-northeast-2a', 'ap-northeast-2c']

# Initialize the EC2 client
ec2 = boto3.resource('ec2')

# Create a security group and 2 EC2 instances in two different availability zones
```

```
try:
    sg = ec2.describe_security_groups(GroupNames='23796349-sg')
    sec_grp_id = sg['SecurityGroups'][0]['GroupId']
except:
    security_group = ec2.create_security_group(GroupName='23796349-sg',
Description='Security group for ssh and http inbound traffic')
    security_group.authorize_ingress(
        IpPermissions=[
            {
                'IpProtocol': 'tcp',
                'FromPort': 22,
                'ToPort': 22,
                'IpRanges':[{'CidrIp': '0.0.0.0/0'}]
            },
            {
                'IpProtocol': 'tcp',
                'FromPort': 80,
                'ToPort': 80,
                'IpRanges':[{'CidrIp': '0.0.0.0/0'}]
            }
        ]
    )
    sec_grp_id = security_group.id

instances = []
for i, zone in enumerate(AVAILABILITY_ZONES):
   try:
       instance = ec2.create_instances(
           ImageId=AMI_ID,
           MinCount=1,
           MaxCount=1,
           KeyName=KEY_NAME,
           InstanceType='t2.micro',
           SecurityGroupIds=[sec_grp_id],
           Placement={
               'AvailabilityZone': zone
           },
           SubnetId=SUBNETS[i],
           TagSpecifications=[
               {
                   'ResourceType': 'instance',
                   'Tags': [
                       {
                           'Key': 'Name',
                           'Value': '23796349-vm' + str(i+1)
                       },
                   ]
               },
           ]
       )[0]
       instances.append(instance)
   except Exception as e:
       print(f"Error creating instance in {zone}: {e}")

# Wait for instances to be in running state
```

```
waiter = ec2.meta.client.get_waiter('instance_running')
waiter.wait(InstanceIds=[instance.id for instance in instances])

# Fetch and print the public IP addresses of the instances
for instance in instances:
   instance.reload()  # Refresh instance details
   print(f"Instance {instance.id} Public IP: {instance.public_ip_address}")
```

[2] Create an Application Load Balancer

- The code above was updated to create an application load balancer and target group with the two instances previously created as the targets and finally a listener. (Reference: link)
- The implementation uses the ELB V2 interface.

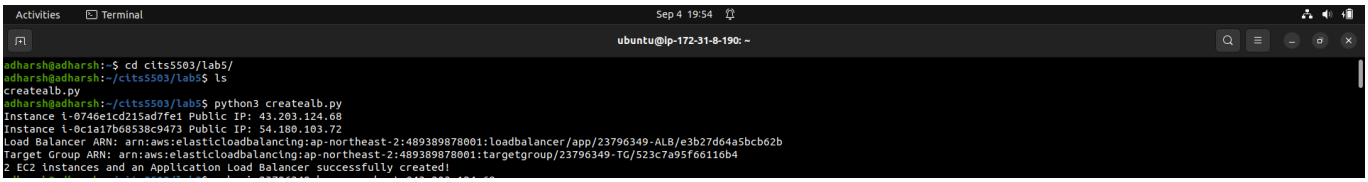
```
import boto3
import time

# Constants

#Using the same key-pair created for Lab2
KEY_NAME = '23796349-key'

#The AMI ID for my region
AMI_ID = 'ami-056a29f2eddc40520'

#The default subnets a and c IDs respectively
SUBNETS = [ 'subnet-09f996d1ad81767a9', 'subnet-0b3601189181ee48e']

#The default VPC ID
VPC_ID = 'vpc-01f842220d0070f97'

#The two availablility zones in the region
AVAILABILITY_ZONES = ['ap-northeast-2a', 'ap-northeast-2c']

# Initialize the EC2 client
ec2 = boto3.resource('ec2')

# Create a security group and 2 EC2 instances in two different availability zones
try:
    sg = ec2.describe_security_groups(GroupNames='23796349-sg')
    sec_grp_id = sg['SecurityGroups'][0]['GroupId']
except:
    security_group = ec2.create_security_group(GroupName='23796349-sg',
Description='Security group for ssh and http inbound traffic')
    security_group.authorize_ingress(
        IpPermissions=[
            {
                'IpProtocol': 'tcp',
                'FromPort': 22,
                'ToPort': 22,
                'IpRanges':[{'CidrIp': '0.0.0.0/0'}]
```

```
            },
            {
                'IpProtocol': 'tcp',
                'FromPort': 80,
                'ToPort': 80,
                'IpRanges':[{'CidrIp': '0.0.0.0/0'}]
            }
        ]
    )
    sec_grp_id = security_group.id

instances = []
for i, zone in enumerate(AVAILABILITY_ZONES):
   try:
       instance = ec2.create_instances(
           ImageId=AMI_ID,
           MinCount=1,
           MaxCount=1,
           KeyName=KEY_NAME,
           InstanceType='t2.micro',
           SecurityGroupIds=[sec_grp_id],
           Placement={
               'AvailabilityZone': zone
           },
           SubnetId=SUBNETS[i],
           TagSpecifications=[
               {
                   'ResourceType': 'instance',
                   'Tags': [
                       {
                           'Key': 'Name',
                           'Value': '23796349-vm' + str(i+1)
                       },
                   ]
               },
           ]
       )[0]
       instances.append(instance)
   except Exception as e:
       print(f"Error creating instance in {zone}: {e}")

# Wait for instances to be in running state
waiter = ec2.meta.client.get_waiter('instance_running')
waiter.wait(InstanceIds=[instance.id for instance in instances])

# Fetch and print the public IP addresses of the instances
for instance in instances:
   instance.reload()  # Refresh instance details
   print(f"Instance {instance.id} Public IP: {instance.public_ip_address}")

# Initialize the ELBV2 client
client = boto3.client('elbv2')

# Create the Application Load Balancer
response = client.create_load_balancer(
```

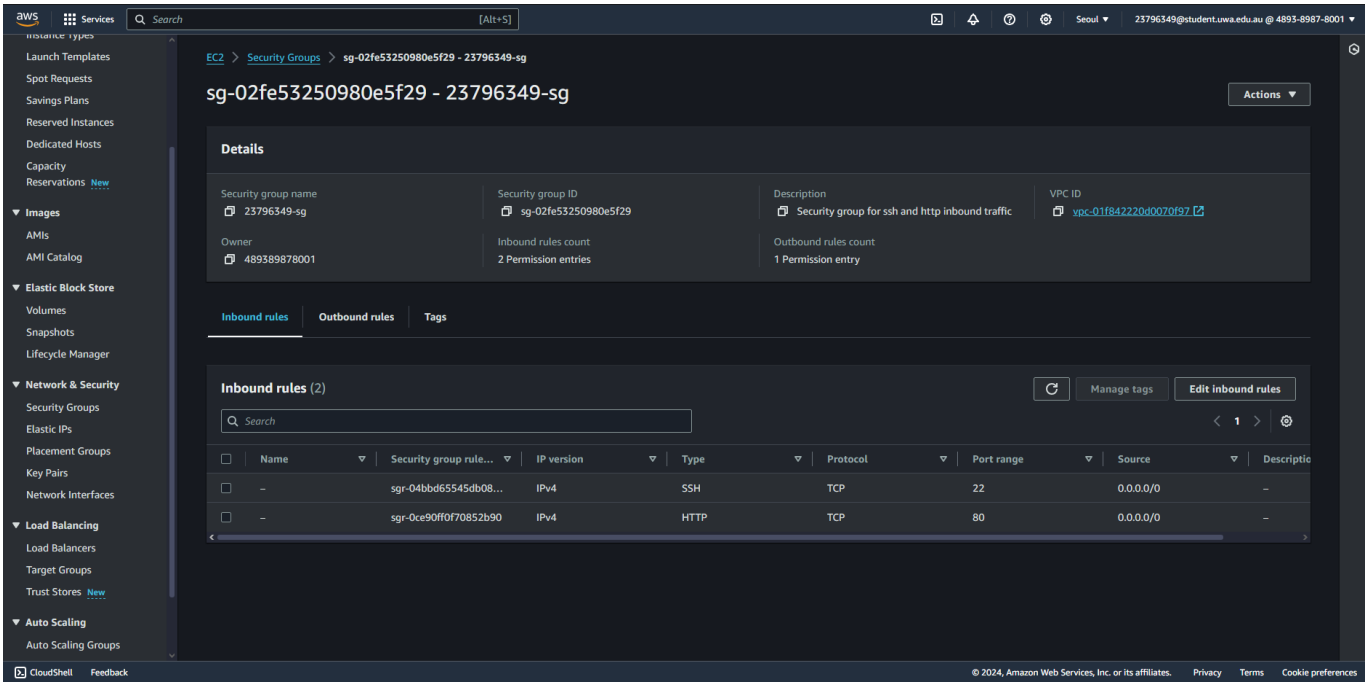
```
    Name='23796349-ALB',
    Subnets=SUBNETS[:2],
    SecurityGroups=[sec_grp_id],
    Scheme='internet-facing',
    Tags=[
        {
            'Key': 'Name',
            'Value': '23796349-ALB'
        },
    ]
)
lb_arn = response['LoadBalancers'][0]['LoadBalancerArn']
print(f"Load Balancer ARN: {lb_arn}")

# Create a target group
response = client.create_target_group(
    Name='23796349-TG',
    Protocol='HTTP',
    Port=80,
    VpcId=VPC_ID,
    Tags=[
        {
            'Key': 'Name',
            'Value': '23796349-TG'
        },
    ]
)
tg_arn = response['TargetGroups'][0]['TargetGroupArn']
print(f"Target Group ARN: {tg_arn}")

# Register targets in the target group
client.register_targets(
    TargetGroupArn=tg_arn,
    Targets=[
        {'Id': instances[0].id},
        {'Id': instances[1].id}
    ]
)

# Create a listener
client.create_listener(
    LoadBalancerArn=lb_arn,
    Protocol='HTTP',
    Port=80,
    DefaultActions=[
        {
            'Type': 'forward',
            'TargetGroupArn': tg_arn
        },
    ]
)

print("2 EC2 instances and an Application Load Balancer successfully created!")
```

- The screenshots below show everything created using the above code:

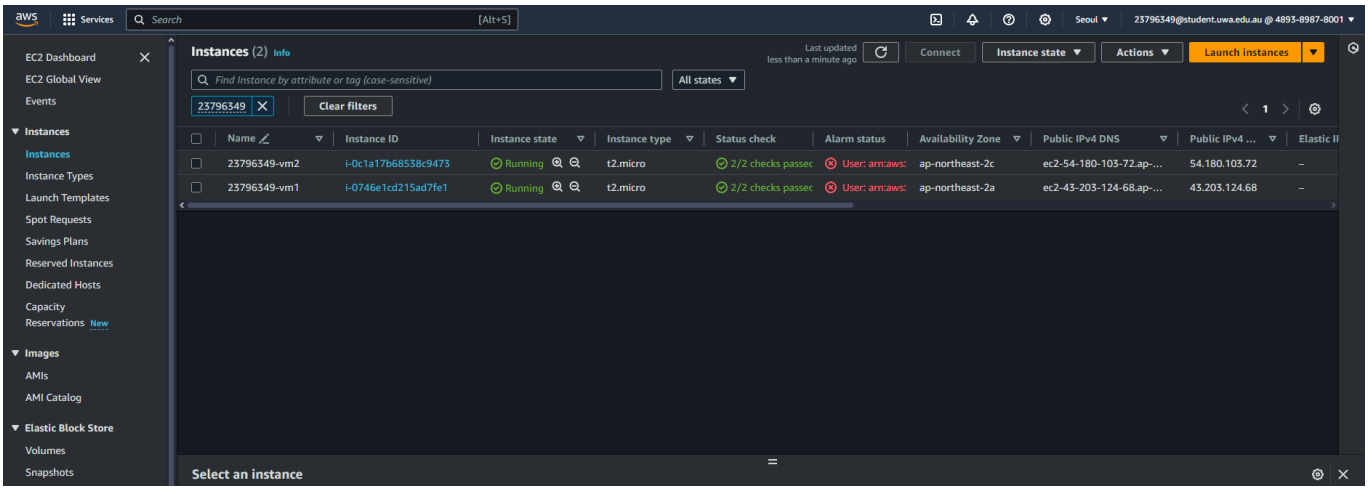
Code output



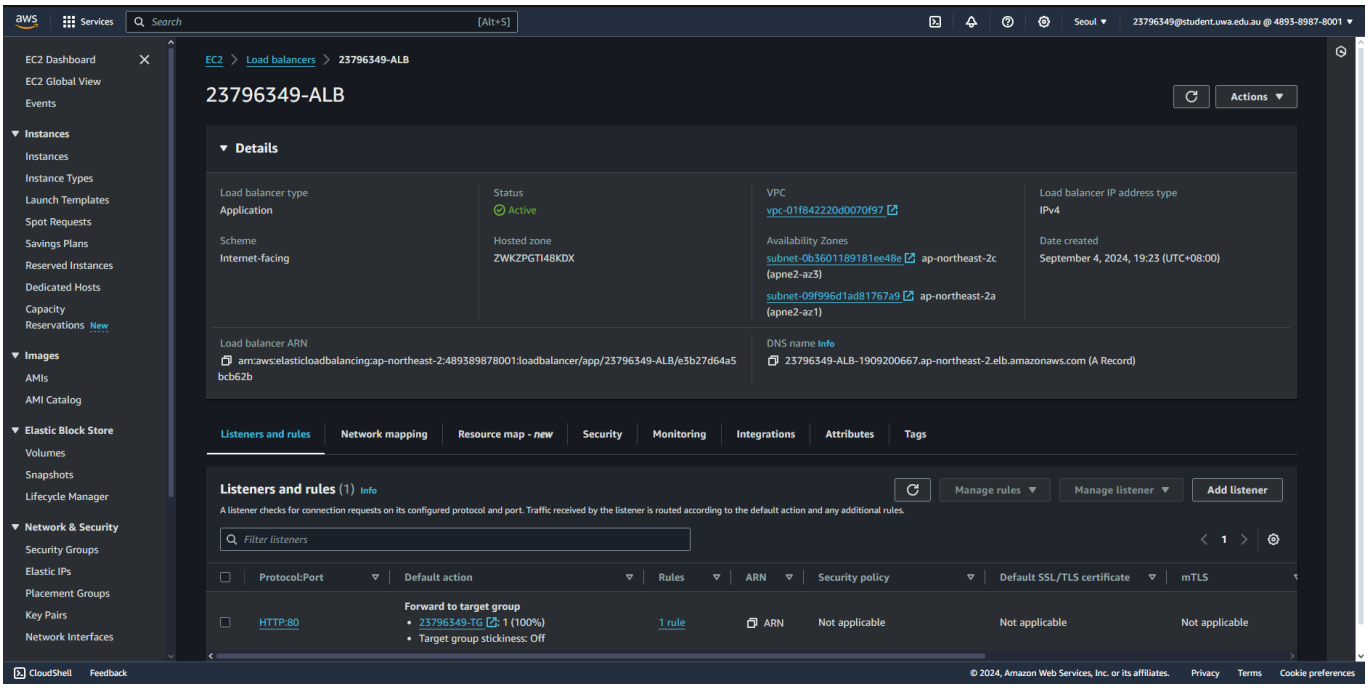
Security group



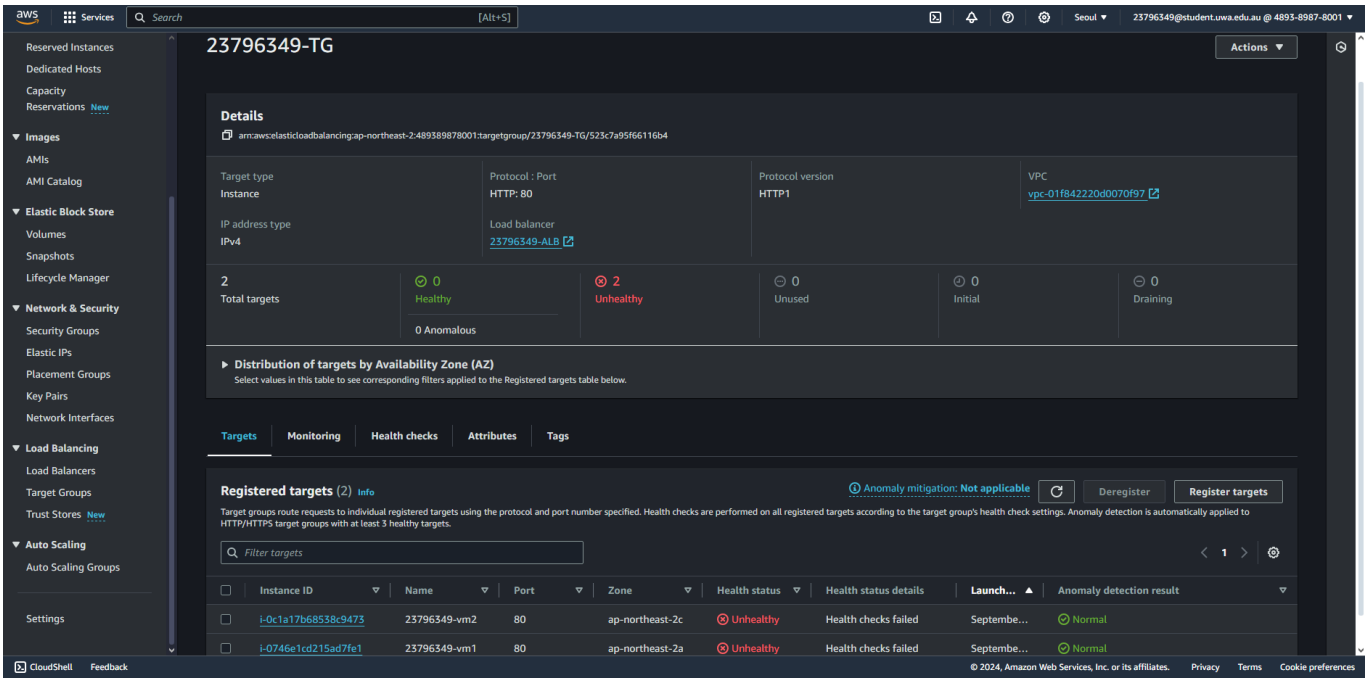
Two EC2 instances



Application Load Balancer



Target Group



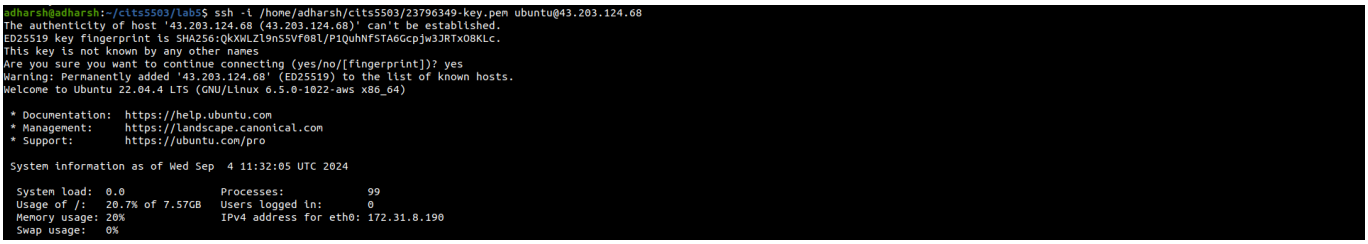
[3] Test the Application Load Balancer

- The load balancer initially wasn't working and the two EC2 instances were showing up as **Unhealthy** targets in the Target Group dashboard.
- The reason being, target group health verification happens over port 80 but the two instances weren't ready to use **port 80 (HTTP)**.
- This problem got sorted a few minutes after **Apache 2** was installed on both instances.
- To connect to the instances using ssh, the following command was used:

```
ssh -i <path to key-pair file> ubuntu@<instance public IP>
```

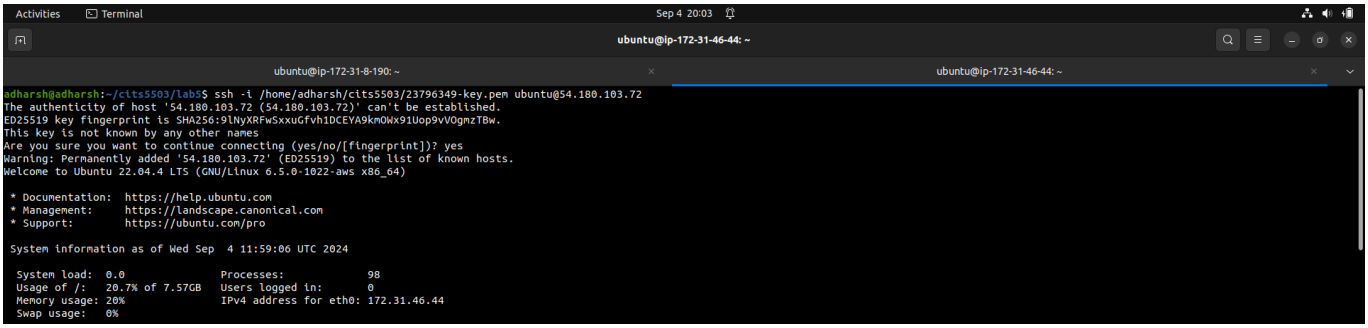
so,

```
#instance 1
ssh -i /home/adharsh/cits5503/23796349-key.pem ubuntu@43.203.124.68
```



and

```
#instance 2
ssh -i /home/adharsh/cits5503/23796349-key.pem ubuntu@54.180.103.72
```

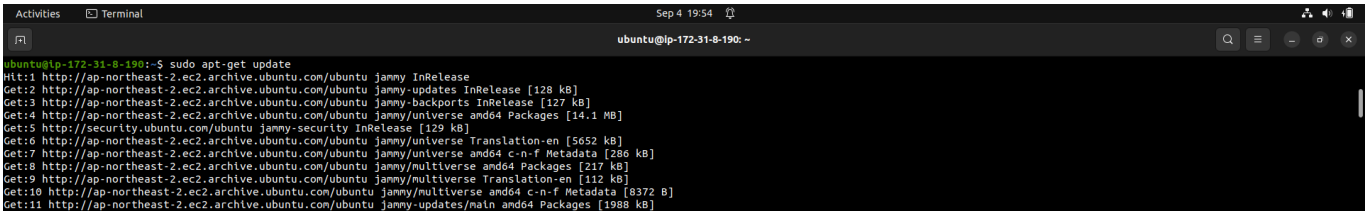


NOTE: This **key-pair** file was created in Lab 2. I used the same file by copying it to this working directory.

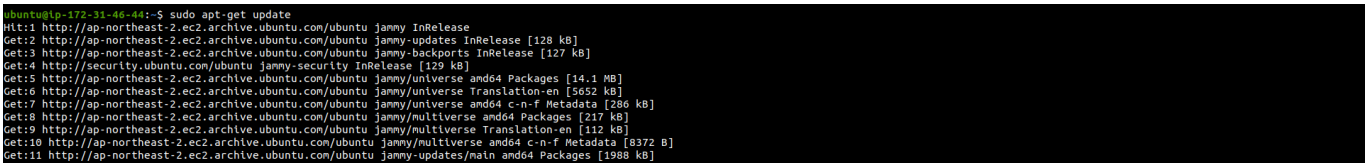
- Each instance was updated using the command:

```
sudo apt-get update
```

Instance 1



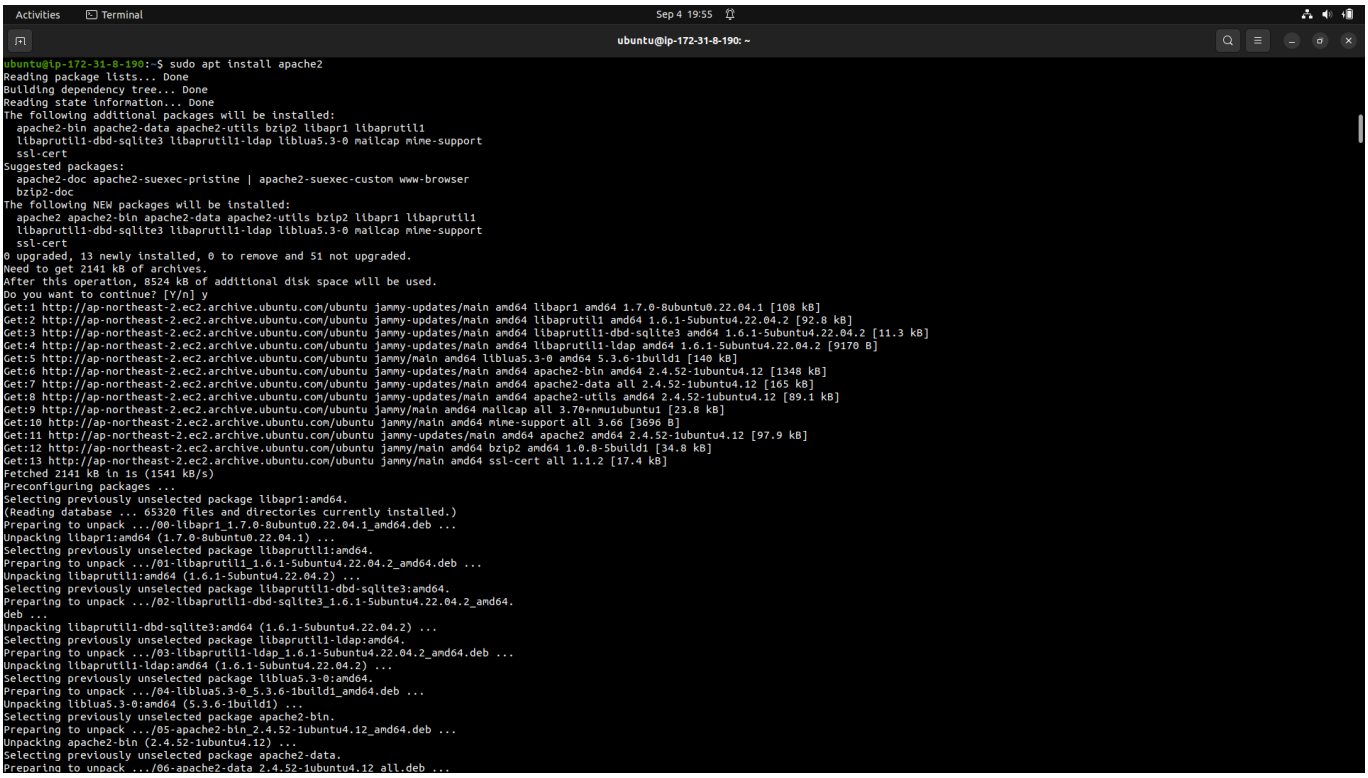
Instance 2



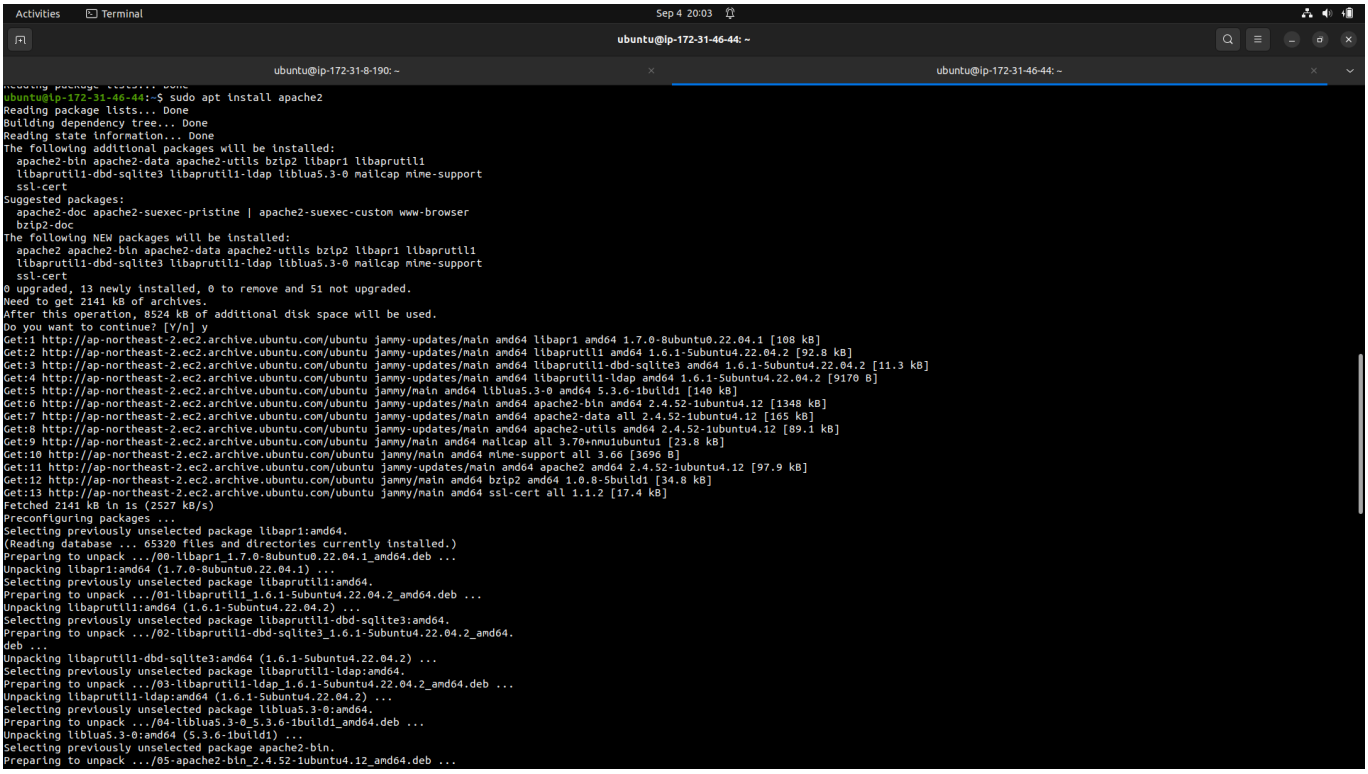
- The following command was used to install Apache 2:

```
sudo apt install apache2
```

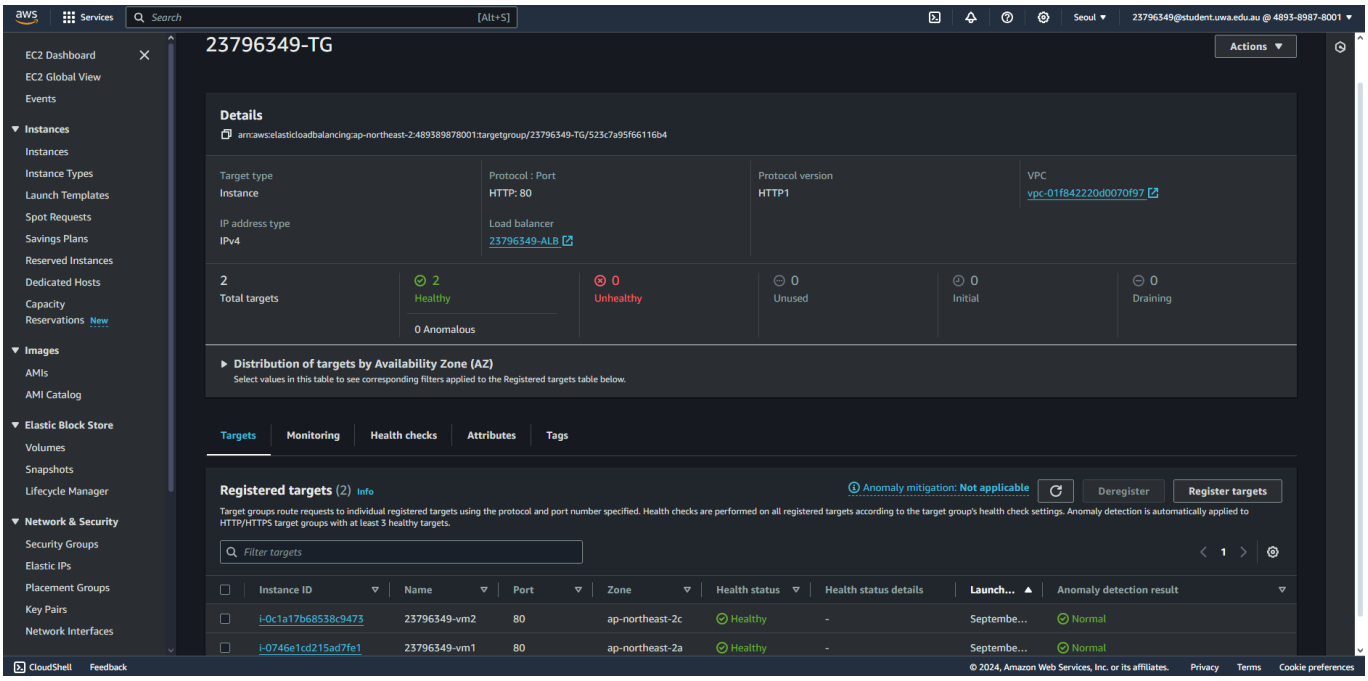
Instance 1



Instance 2



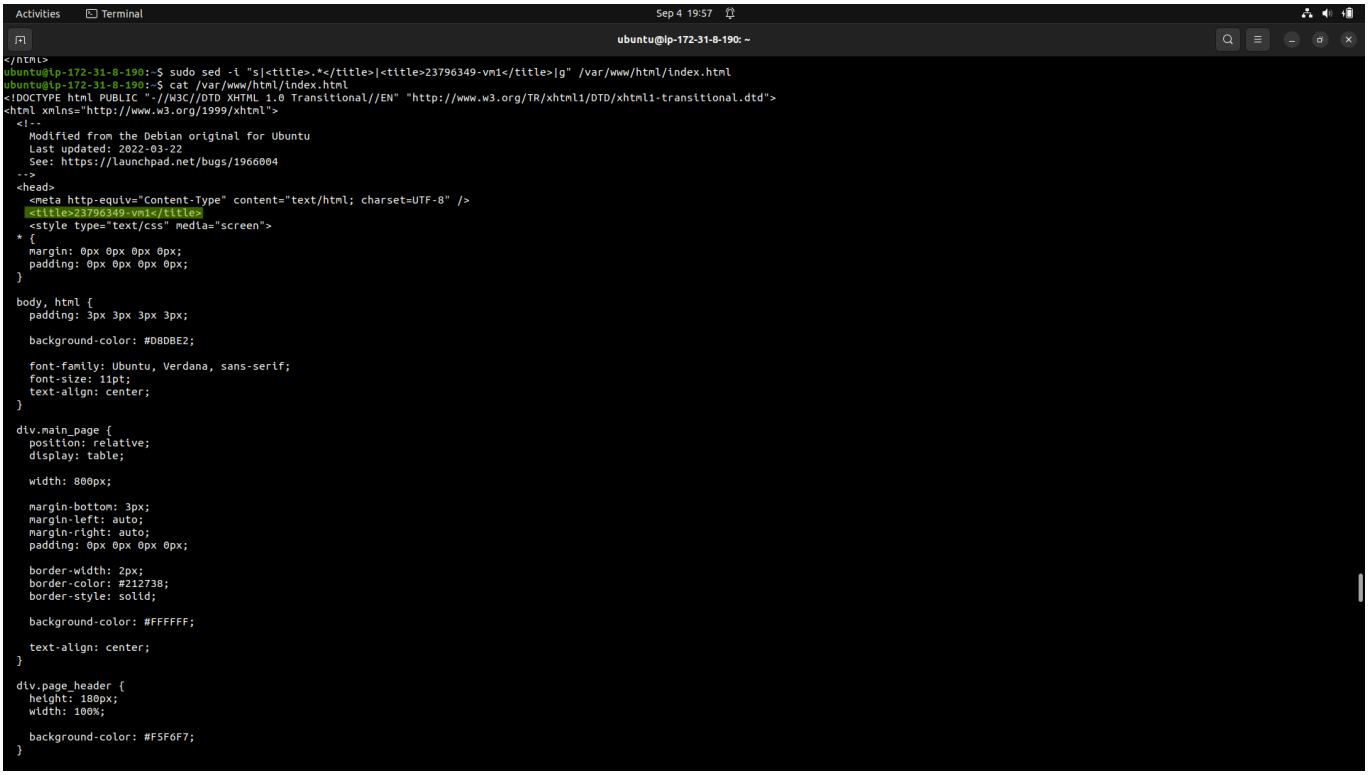
- Now that Apache 2 was installed in both instances, both the instances show up as healthy in the Target Group Dashboard:



- To change the content in the **title** tag, use the following command:

NOTE: I learnt **sed** operations in the unit **CITS4407 - Open Source Tools and Scripting**

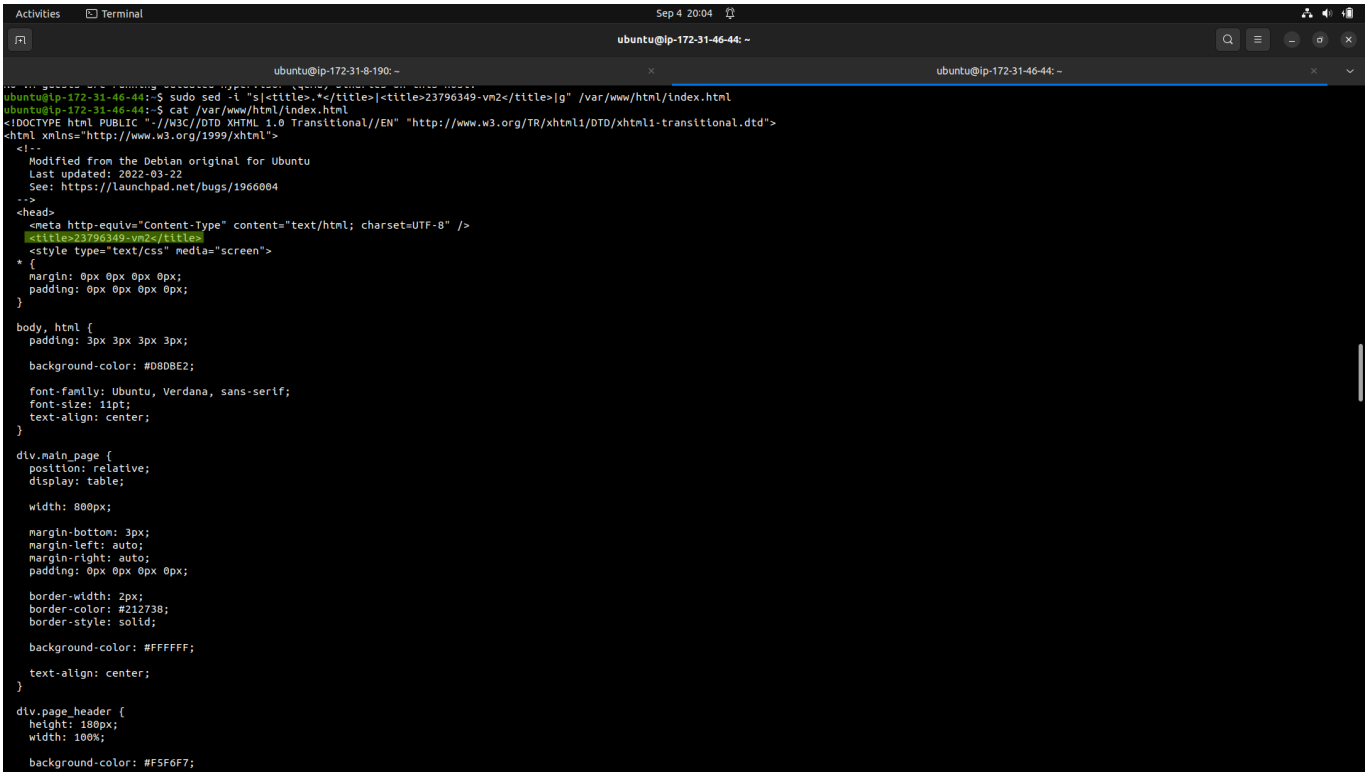
```
#instance 1
sudo sed -i "s|<title>.*</title>|<title>23796349-vm1</title>|g"
/var/www/html/index.html
```



and

```
#instance 2
sudo sed -i "s|<title>.*</title>|<title>23796349-vm2</title>|g"
```

```
/var/www/html/index.html
```



NOTE: To see the changes, the following command can be used:

```
cat /var/www/html/index.html
```

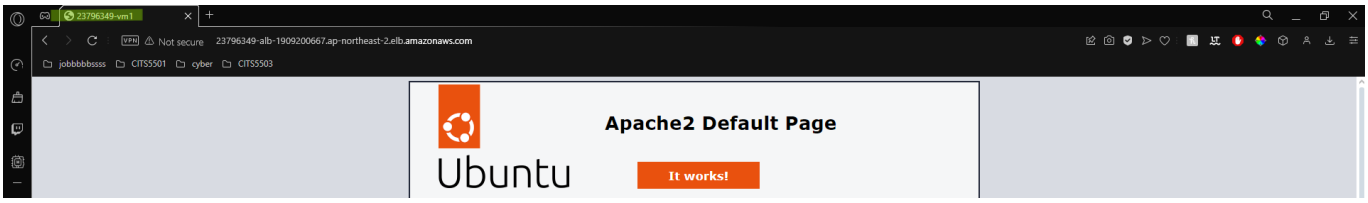
- Alternatively, visiting the public IP of each instance through browser can also be used to verify the changes (screenshots below):

Instance 1

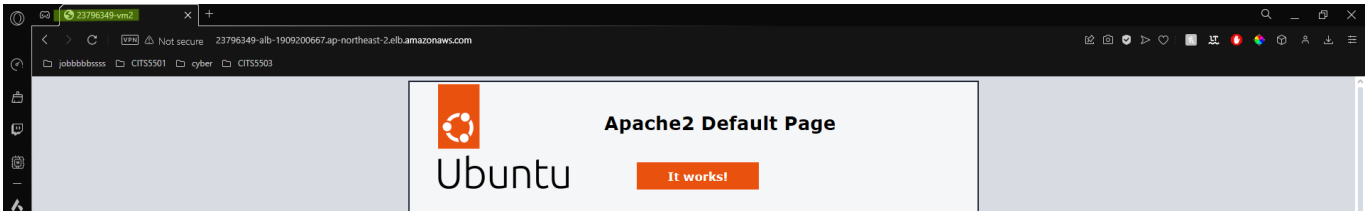
Instance 2

- To check if the load balancer works, copy the **DNS Name** of your load balancer and use it in the browser.
- Each time the page is reloaded, the index.html file of instance 1 and instance 2 are loaded equally, one for each reload (screenshots below):

When the page is loaded



After page reload



- All resources created for this lab were deleted using AWS Console after the completion.