

## **Labs 1-5**

Student ID: 24188515

Student Name: Dayu Liu

# Lab 1

## AWS Account and Log in

### [1] Reset and Login into IAM User Account

To start with, I received an email with the initial login credentials for my IAM user account. After navigating to the AWS login portal, I successfully logged in using these credentials and reset my password as instructed.



#### Sign in as IAM user

Account ID (12 digits) or account alias

489389878001

IAM user name

24188516@student.uwa.edu.au

Password

.....

☒ Remember this account

Sign in

[Sign in using root user email](#)

[Forgot password?](#)

## DynamoDB zero-ETL integration with OpenSearch Service

Perform full-text and vector  
search on operational data in  
near real time without building  
and managing data pipelines

[Read the blog ›](#)

This step grants access to the AWS Management Console, which provides UI controls on necessary AWS resources and services for future lab activities.

### [2] Access Identity and Access Management (IAM)

After logging in, locate a clickable with my `IAM user` and `Account ID` information on the top-right corner. Once opened, click at the bottom entry on the user panel to access `Security Credentials`.

24188516@student.uwa.edu.au @ 4893-8987-8001 ▲

Account ID: 4893-8987-8001 

IAM user: 24188516@student.uwa.edu.au



Account

Organization

Service Quotas

Billing and Cost Management

Security credentials

Switch role

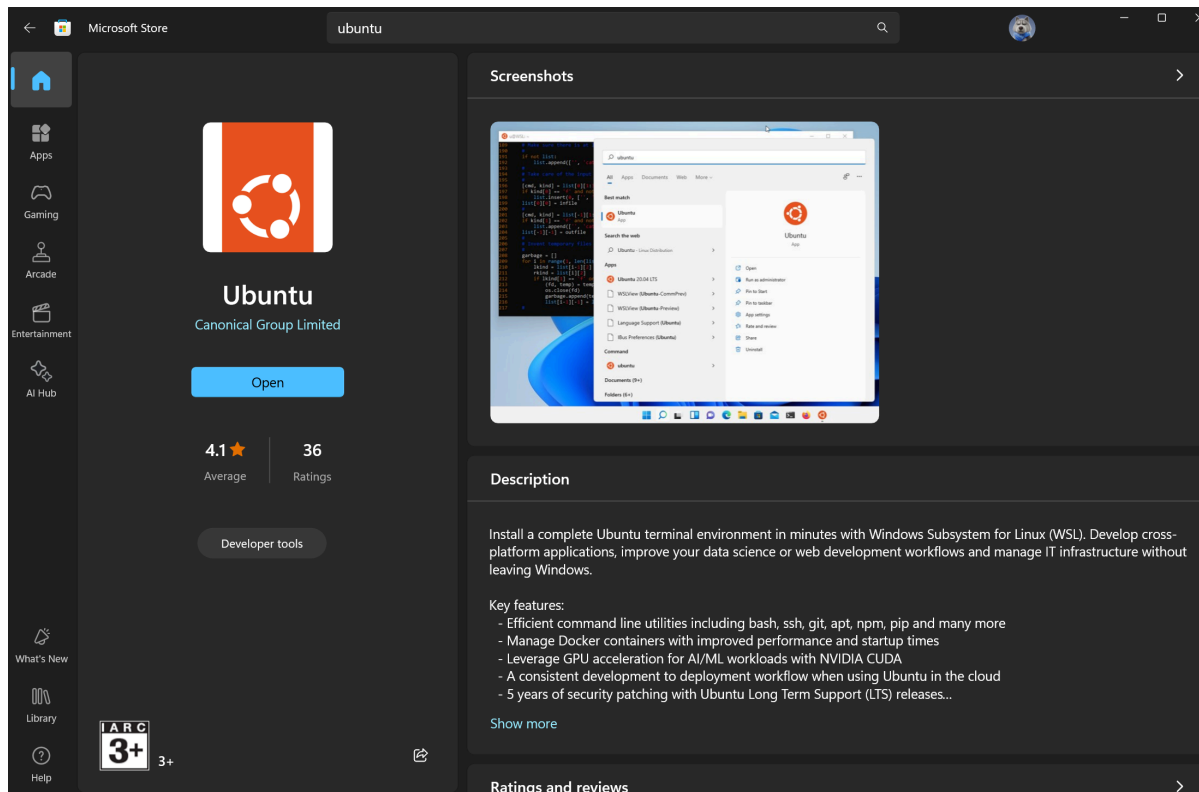
Sign out

Within the `Access keys` tab, I created a new access key and secret. These credentials can be used for programmatic access to AWS services, allowing us to interact with AWS through command-line interfaces (AWS-CLI), SDKs (Boto3 API). Securely store these credentials in a private location, for later AWS configuration setup in this lab.

Access keys (1)		Create access key
Use access keys to send programmatic calls to AWS from the AWS CLI, AWS Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. <a href="#">Learn more</a>		
AKIAXD4PI5LY6DFJVJ6A		
Description	Status	Actions ▼
-	Active	
Last used	Created	
6 days ago	7 days ago	
Last used region	Last used service	
eu-north-1	ec2	

# Set up recent Linux OSes

Since I am running a Windows machine, I chose to use `Ubuntu on windows` via the Windows Subsystem for Linux (WSL). This installation provides an isolated Linux environment with a separate file directory, making file management easier and more organized within the Windows system.



By using WSL, I can run Linux-based commands and utilities without the need for a virtual machine, which saves both time and resources.

## Install Linux Packages

### 1. Install Python 3.10.x

Since my Ubuntu version is `22.04`, my system automatically comes with its corresponding latest Python version, which is `3.10.12`.

### Update Packages

First, we need to ensure our system's packages are up to date. Run the following commands to update and upgrade pre-installed packages:

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

Since upgrading packages involves modifying the system, administrative privileges are required. Hence we need to prefix the command with `sudo`.

## Code Explanation

- `apt update` updates the package lists from **Apt** package management tool. These package lists contain references to the latest versions of packages.
- `apt upgrade` upgrades all installed packages to their latest versions based on the information retrieved from the update command, `-y` automatically answers **YES** to any prompts that might appear during the upgrade process.

```
y only (it is configured to refuse manual start/stop).
See system logs and 'systemctl status snapd.mounts-pre.target' for details.
Could not execute systemctl: at /usr/bin/deb-systemd-invoke line 142.
Setting up libpython3.10-minimal:amd64 (3.10.12-1~22.04.5) ...
Setting up openssl (3.0.2-0ubuntu1.17) ...
Setting up python3.10-minimal (3.10.12-1~22.04.5) ...
Setting up libpython3.10-stdlib:amd64 (3.10.12-1~22.04.5) ...
Setting up libpython3.10:amd64 (3.10.12-1~22.04.5) ...
Setting up python3.10 (3.10.12-1~22.04.5) ...
Setting up libpython3.10-dev:amd64 (3.10.12-1~22.04.5) ...
Setting up python3.10-dev (3.10.12-1~22.04.5) ...
Processing triggers for libc-bin (2.35-0ubuntu3.8) ...
Processing triggers for man-db (2.10.2-1) ...
Processing triggers for dbus (1.12.20-2ubuntu4.1) ...
Processing triggers for mailcap (3.70+nmu1ubuntu1) ...
liudayubob@Dayu:~$
```

## Verify Python Version

To confirm that Python is installed and up to date, use the following command:

```
python3 -V
```

## Code Explanation

- `python3`: Specifies that we are running a version of Python 3.x.
- `-V`: Outputs the installed Python version.

Once executed, the output verifies that Python 3.10.12 is installed.

```
liudayubob@Dayu:~$ python3 -V
Python 3.10.12
```

## Install pip3

To easily install and manage Python libraries, we also need to install **pip**. Install it with:

```
sudo apt install -y python3-pip
```

## Code Explanation

- `python3-pip`: pip package for Python 3 specifically.

Once installed, we can now use `pip3` to install third-party Python packages for future labs.

```
liudayubob@Dayu:~$ sudo apt install -y python3-pip
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
python3-pip is already the newest version (22.0.2+dfsg-1ubuntu0.4).
0 upgraded, 0 newly installed, 0 to remove and 2 not upgraded.
```

## 2. Install AWS CLI

To interact with AWS services from the command line, we use the AWS CLI (Command Line Interface). Install and upgrade it to the latest version using:

```
pip3 install awscli --upgrade
```

### Code Explanation

- `awscli`: This installs the AWS Command Line Interface, enabling us to manage AWS services, directly from the terminal.
- `--upgrade`: Ensures that if an older version of AWS CLI is already installed, it will be replaced with the latest version, which includes new features and updates and guarantees consistency.

Once installed, we can execute AWS CLI commands to interact with various AWS resources such as EC2, S3, etc.

```
Installing collected packages: botocore, awscli
Attempting uninstall: botocore
  Found existing installation: botocore 1.34.149
  Uninstalling botocore-1.34.149:
    Successfully uninstalled botocore-1.34.149
Attempting uninstall: awscli
  Found existing installation: awscli 1.33.31
  Uninstalling awscli-1.33.31:
    Successfully uninstalled awscli-1.33.31
Successfully installed awscli-1.33.35 botocore-1.34.153
```

## 3. Configure AWS CLI

After installation, configure the AWS CLI to connect to our IAM User account. Entering the credentials **Access Key ID**, **Secret Access Key**, **Region** in the prompts after using the following command:

```
aws configure
```

Our credentials can be found from step [3]. These configurations help us to correctly authenticate and set up our AWS environment and to access AWS services securely.

```
liudayubob@Dayu:~$ aws configure
AWS Access Key ID [*****VJ6A]:
AWS Secret Access Key [*****DAfU]:
Default region name [eu-north-1]:
Default output format [None]:
```

## 4. Install boto3

Although `botocore` is included with the AWS CLI package, `boto3` is the Python SDK for AWS, and can be used to send API requests and automate tasks via Python scripts, such as launching EC2 instances in the next step. Install `boto3` using the following command:

```
pip3 install boto3
```

```
liudayubob@Dayu:~$ pip3 install boto3
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: boto3 in ./local/lib/python3.10/site-packages (1.34.149)
Requirement already satisfied: botocore<1.35.0, >=1.34.149 in ./local/lib/python3.10/site-packages (from boto3) (1.34.153)
Requirement already satisfied: s3transfer<0.11.0, >=0.10.0 in ./local/lib/python3.10/site-packages (from boto3) (0.10.2)
Requirement already satisfied: jmespath<2.0.0, >=0.7.1 in /usr/lib/python3/dist-packages (from boto3) (0.10.0)
Requirement already satisfied: urllib3!=2.2.0, <3, >=1.25.4 in /usr/lib/python3/dist-packages (from botocore<1.35.0, >=1.34.149->boto3) (1.26.5)
Requirement already satisfied: python-dateutil<3.0.0, >=2.1 in ./local/lib/python3.10/site-packages (from botocore<1.35.0, >=1.34.149->boto3) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-packages (from python-dateutil<3.0.0, >=2.1->botocore<1.35.0, >=1.34.149->boto3) (1.16.0)
```

## Test the Installed Environment

### 1. Test the AWS Environment

To verify that our AWS CLI is configured correctly and connected to the AWS environment, we run the following command to list the available regions in our AWS account:

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

### Code Explanation

- `aws ec2 describe-regions`: This command queries the AWS EC2 service to list all available regions where AWS services are provided.
- `--output table`: Formats the output in a readable table structure, making it easier to view and interpret the region data.

Once executed, this command allows us to verify that we are connected to AWS, and the output should display a list of regions in a structured table.

```
53.0, >-1.34.149->D0C03) (1.18.0)
liudayubob@Dayu:~$ aws ec2 describe-regions --output table
```

DescribeRegions		
Regions		
Endpoint	OptInStatus	RegionName
ec2.ap-south-1.amazonaws.com	opt-in-not-required	ap-south-1
ec2.eu-north-1.amazonaws.com	opt-in-not-required	eu-north-1
ec2.eu-west-3.amazonaws.com	opt-in-not-required	eu-west-3
ec2.eu-west-2.amazonaws.com	opt-in-not-required	eu-west-2
ec2.eu-west-1.amazonaws.com	opt-in-not-required	eu-west-1
ec2.ap-northeast-3.amazonaws.com	opt-in-not-required	ap-northeast-3
ec2.ap-northeast-2.amazonaws.com	opt-in-not-required	ap-northeast-2
ec2.ap-northeast-1.amazonaws.com	opt-in-not-required	ap-northeast-1
ec2.ca-central-1.amazonaws.com	opt-in-not-required	ca-central-1
ec2.sa-east-1.amazonaws.com	opt-in-not-required	sa-east-1
ec2.ap-southeast-1.amazonaws.com	opt-in-not-required	ap-southeast-1
ec2.ap-southeast-2.amazonaws.com	opt-in-not-required	ap-southeast-2
ec2.eu-central-1.amazonaws.com	opt-in-not-required	eu-central-1
ec2.us-east-1.amazonaws.com	opt-in-not-required	us-east-1
ec2.us-east-2.amazonaws.com	opt-in-not-required	us-east-2
ec2.us-west-1.amazonaws.com	opt-in-not-required	us-west-1
ec2.us-west-2.amazonaws.com	opt-in-not-required	us-west-2

## 2. Test the Python Environment

After confirming that the AWS CLI is working correctly, we now test the Python environment using `boto3`, the AWS SDK for Python language.

The following Python code connects to the AWS EC2 service and retrieves the available regions, similar to the CLI test but now within the Python environment:

```
python3
>>> import boto3
>>> ec2 = boto3.client('ec2')
>>> response = ec2.describe_regions()
>>> print(response)
```

### Code Explanation

- `import boto3`: Imports the `boto3` library, which is used to interact with AWS services via Python.
- `boto3.client('ec2')`: Initializes a client for the EC2 service, allowing us to make requests to EC2, such as querying regions, starting instances, etc.
- `ec2.describe_regions()`: Queries the EC2 service to retrieve a list of available AWS regions. It returns the data in JSON format.
- `print(response)`: Outputs the result, which contains details about the available regions.

Once executed, this code verifies that our Python environment is correctly set up and able to interact with AWS services via `boto3`.



```
liudayubob@Dayu:~$ python3
Python 3.10.12 (main, Jul 29 2024, 16:56:48) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>> ec2 = boto3.client('ec2')
>>> res = ec2.describe_regions()
>>> print(res)
{'Regions': [{'Endpoint': 'ec2.ap-south-1.amazonaws.com', 'RegionName': 'ap-south-1', 'OptInStatus': 'opt-in-not-require
d'}, {'Endpoint': 'ec2.eu-north-1.amazonaws.com', 'RegionName': 'eu-north-1', 'OptInStatus': 'opt-in-not-required'}, {'E
ndpoint': 'ec2.eu-west-3.amazonaws.com', 'RegionName': 'eu-west-3', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint':
'ec2.eu-west-2.amazonaws.com', 'RegionName': 'eu-west-2', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.eu-we
st-1.amazonaws.com', 'RegionName': 'eu-west-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.ap-northeast-3.
amazonaws.com', 'RegionName': 'ap-northeast-3', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.ap-northeast-2.
amazonaws.com', 'RegionName': 'ap-northeast-2', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.ap-northeast-1.
amazonaws.com', 'RegionName': 'ap-northeast-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.ca-central-1.am
azonaws.com', 'RegionName': 'ca-central-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.sa-east-1.amazonaws
.com', 'RegionName': 'sa-east-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.ap-southeast-1.amazonaws.com',
'RegionName': 'ap-southeast-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.ap-southeast-2.amazonaws.com',
'RegionName': 'ap-southeast-2', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.eu-central-1.amazonaws.com',
'RegionName': 'eu-central-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.us-east-1.amazonaws.com', 'Region
Name': 'us-east-1', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.us-east-2.amazonaws.com', 'RegionName': 'us
-east-2', 'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.us-west-1.amazonaws.com', 'RegionName': 'us-west-1',
'OptInStatus': 'opt-in-not-required'}, {'Endpoint': 'ec2.us-west-2.amazonaws.com', 'RegionName': 'us-west-2', 'OptInStat
us': 'opt-in-not-required'}], 'ResponseMetadata': {'RequestId': '40eac83f-7af8-40cf-911d-bc7d3e3f0bdb', 'HTTPStatusCode'
: 200, 'HTTPHeaders': {'x-amzn-requestid': '40eac83f-7af8-40cf-911d-bc7d3e3f0bdb', 'cache-control': 'no-cache, no-store'
, 'strict-transport-security': 'max-age=31536000; includeSubDomains', 'vary': 'accept-encoding', 'content-type': 'text/x
ml; charset=UTF-8', 'content-length': '2890', 'date': 'Mon, 05 Aug 2024 06:44:22 GMT', 'server': 'AmazonEC2'}, 'RetryAtte
mpts': 0}}
```

### 3. Write a Python Script

Now we create a Python script to wrap these commands into a single file. We should also format the response into a structured table. The Python script is located in `~/lab1` folder on our Ubuntu machine.

#### (1) Install Dependencies

We use the `pandas` library to convert un-tabulated data into a structured table. To install this additional dependency, run the following command:

```
pip install pandas
```

#### (2) Create a script

The code in this script adds an extra step. After retrieving the region data from AWS, we pass the response into a `pandas` dataframe to format and print the output in a table structure.

```
import boto3 as bt
import pandas as pd

ec2 = bt.client('ec2')
response = ec2.describe_regions()
regions = response['Regions']
regions_df = pd.DataFrame(regions)
print(regions_df)
```

### Code Explanation

- `boto3 as bt`: Imports `boto3`, aliased as `bt`.
- `pandas as pd`: Imports `pandas`, aliased as `pd`.
- `ec2 = bt.client('ec2')`: Initializes a client for the EC2 service, allowing us to make requests to EC2, such as querying regions, starting instances, etc.

- `response = ec2.describe_regions()`: Queries the EC2 service to retrieve a list of available AWS regions. It returns the data in JSON format.
- `pd.DataFrame(regions)`: Converts the regions data into a pandas DataFrame for structured output.

### (3) Run the Script

To execute the Python script, use the following command:

```
python3 lab1.py
```

## 4. Get the results

After running the Python script, the results are printed in a table format. The table shows the available AWS regions along with the corresponding **Endpoint**, **RegionName**, and **OptInStatus**.

---	Endpoint	RegionName	OptInStatus
0	ec2.ap-south-1.amazonaws.com	ap-south-1	opt-in-not-required
1	ec2.eu-north-1.amazonaws.com	eu-north-1	opt-in-not-required
2	ec2.eu-west-3.amazonaws.com	eu-west-3	opt-in-not-required
3	ec2.eu-west-2.amazonaws.com	eu-west-2	opt-in-not-required
4	ec2.eu-west-1.amazonaws.com	eu-west-1	opt-in-not-required
5	ec2.ap-northeast-3.amazonaws.com	ap-northeast-3	opt-in-not-required
6	ec2.ap-northeast-2.amazonaws.com	ap-northeast-2	opt-in-not-required
7	ec2.ap-northeast-1.amazonaws.com	ap-northeast-1	opt-in-not-required
8	ec2.ca-central-1.amazonaws.com	ca-central-1	opt-in-not-required
9	ec2.sa-east-1.amazonaws.com	sa-east-1	opt-in-not-required
10	ec2.ap-southeast-1.amazonaws.com	ap-southeast-1	opt-in-not-required
11	ec2.ap-southeast-2.amazonaws.com	ap-southeast-2	opt-in-not-required
12	ec2.eu-central-1.amazonaws.com	eu-central-1	opt-in-not-required
13	ec2.us-east-1.amazonaws.com	us-east-1	opt-in-not-required
14	ec2.us-east-2.amazonaws.com	us-east-2	opt-in-not-required
15	ec2.us-west-1.amazonaws.com	us-west-1	opt-in-not-required
16	ec2.us-west-2.amazonaws.com	us-west-2	opt-in-not-required

## Key Parameters:

- **Endpoint:** Indicates a public URL of the AWS service (EC2) for each region. It's the endpoint through which API calls are routed for that specific region.
- **RegionName:** Represents the region code for each AWS region. In the future labs, we will be using `eu-north-1` for the European North region .
- **OptInStatus:** Shows the status of whether a region requires users to opt-in before using it. `opt-in-not-required` means that the region is generally available for all AWS users.

This table helps verify the connection to AWS and confirms that our Python environment is correctly configured to retrieve information from AWS services.

# Lab 2

## Create an EC2 Instance Using AWS CLI

### 1. Create a Security Group

We begin by creating a security group with a unique name based on our student number, `24188516-sg`. A security group acts as an access control mechanism for our EC2 instances, controlling inbound and outbound traffic.

The following command creates the security group:

```
aws ec2 create-security-group --group-name 24188516-sg --description "security group for development environment"
```

#### Key Parameters:

- `--group-name`: Specifies the name of the security group. In this case, we use `24188516-sg` to uniquely identify the group based on our student number.
- `--description`: Provides a human-readable description of the security group's purpose. Here, we describe it as "security group for development environment" to indicate the group will be used for development purposes.

Once executed, this command will create the security group and return the **GroupId**, which is a unique identifier for the newly created security group. Keep this **GroupId** because we will use it for future commands when creating our EC2 instance and etc.

```
liudayubob@Dayu:~$ aws ec2 create-security-group --group-name 24188516-sg --description "security group for development environment"
{
  "GroupId": "sg-0bdec2dfbf5631af5"
}
```

### 2. Authorize Inbound Traffic for SSH

Now that the security group is created, we need to add a rule to allow inbound SSH traffic. This step enables secure access to our EC2 instances using the SSH protocol on port 22.

The following command authorizes inbound traffic for SSH:

```
aws ec2 authorize-security-group-ingress --group-name 24188516-sg --protocol tcp --port 22 --cidr 0.0.0.0/0
```

#### Key Parameters:

- `--group-name`: Specifies the name of the security group to which the rule will be added. In this case, we are adding the rule to the `24188516-sg` security group created in the previous step.
- `--protocol`: Defines the protocol for the rule. Here, we use `tcp` to specify the Transmission Control Protocol, which is the standard protocol used for SSH.
- `--port`: Specifies the port number on which the traffic will be allowed. In this case, we set it to `22`, the default port for SSH connections.

- `--cidr`: Defines the range of IP addresses allowed to access the instance via SSH. `0.0.0.0/0` means that traffic is allowed from any IP address, giving unrestricted access to SSH from anywhere in the world. This is common for testing purposes but should be restricted for production environments.

Once executed, this command creates a rule allowing SSH access on port 22, and the response confirms the rule creation by displaying the details of the newly added rule.

```
liudayubob@Dayu:~$ aws ec2 authorize-security-group-ingress --group-name 24188516-sg --protocol tcp --port 22 --cidr 0.0.0.0/0
{
  "Return": true,
  "SecurityGroupRules": [
    {
      "SecurityGroupRuleId": "sgr-0622084795b10309b",
      "GroupId": "sg-0bdec2dfbf5631af5",
      "GroupOwnerId": "489389878001",
      "IsEgress": false,
      "IpProtocol": "tcp",
      "FromPort": 22,
      "ToPort": 22,
      "CidrIpv4": "0.0.0.0/0"
    }
  ]
}
```

### 3. Create a Key Pair

To securely connect to the EC2 instance, we generate a public and private key pair. The private key will be used to authenticate SSH connections, while the public key is associated with the EC2 instance. This step is crucial for securing the private key and ensuring that it can be used for SSH connections without exposing it to others.

The following command creates a key pair:

```
aws ec2 create-key-pair --key-name 24188516-key --query 'KeyMaterial' --output text > 24188516-key.pem
```

#### Key Parameters:

- `--key-name`: Specifies the name of the key pair being created. In this case, the key pair is named `24188516-key`, which is based on our student number for identification.
- `--query 'KeyMaterial'`: This option extracts the private key (key material) from the response and outputs it as plain text. The key material is the private part of the key pair, which is required to authenticate SSH sessions.
- `--output text`: Specifies that the output format should be plain text (instead of JSON). The output is redirected to a file using the `>` operator, which saves the private key as `24188516-key.pem`.

#### Set Permissions for the Key:

After the key is generated, we ensure it has the correct permissions by using the `chmod` command:

```
chmod 400 24188516-key.pem
```

- `chmod 400`: This changes the file's permissions to **read-only** for the file owner. It ensures that only the owner of the file can read it and prevent unauthorized access.

Once executed, we can check the outputs of successfully created key:

```
liudayubob@Dayu:~$ aws ec2 create-key-pair --key-name 24188516-key --query 'KeyMaterial' --output text > 24188516-key.pem
liudayubob@Dayu:~$ chmod 400 24188516-key.pem
liudayubob@Dayu:~$ vi 24188516-key.pem

[1]+  Stopped                  vi 24188516-key.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAgSuU62oGW97BYTarPR3JSVUiZafPK9h+ToTXzmSD0nkK2JCh
VSQ7wI4dFM2tnOGjRz3I5uNDNi4fRmdr37tVF80I8PUIReiIF7R9aUbxVcibZulw
3YHe0VoX8YU7S1w8i+khSD6nhCM+4VVRRAqK/gp5Ng+WLbmuYLoE55APGkpoi7RB
pUqtJb/UbxQ1HHGw328RvtK+BB3UKyTtWgm/qBKgI+D/yjvP4ValfGxmzpZjMs3rq
BOp+/uPUPKu8Yj1HF84JDSH7q3YABr0tcnABQ/KCBoLGMPZzWmqhLO70dcLbTrgf
A2UCx0b3Au9a0xKvagmLs4Lge05Hf0lhxN550wIDAQABAoIBAFJuKWmARYeqyKcS
xDLdtdk4nVgBvVA9vcSQdk2uDWIdRNedt+ouTuulOdiLMLhTjF2JKi0d4YjT9a
uKKlNoimg+yQE0m2FecSqiMqQbi1mDLwlMAtLDD0nd5qACeLduEaSQ515i8h42Qx
1+iQ2G4ofaGziwtARYTSV3r00H/ZTpArFNYaspFnTjgOGDBgVLqurLNB0Eic4ILJ
hMqb1twlDDG19VmCCYHE3FnZUtTu+JAPNMDDCjMmnfDyDpOEYTKd8vnw4M/Y9R08
Hg3zxpVmnQtQ4xcp6mAvq8yCtXAER5ssIdoSHYNg+0jYunDkXJMHSGU+PcRPzrvh
gYVLv/ECgYEA5ahQtejRj8hIxFLU3Y2yAHWWcQegDwtCsoZHi4VQFn0NHj9Wfy51
v475+q+J1IKPrcn9zu080K7EpqhAY39bjq/anNjVzgLZvQSCXTpe0fVqSVSUM4Ux
bfLpZZDvO2FE9ZkNN2bImk7aE0lyOmML4ySbj2+0IM14FKWx0VKJW0kCgYEAkcxE
FZMVyTmtlw90duLcMgz6RMkFQpmUkFQt1Yc0kvbXzHb2LYaKcCnbzQb5niTg6mG3
+5NLaQq3dWGYThK+YvfpunAfXKqFzBZ1iRIUfclC7AGLZUtDR7xYFXJR3Efem8nQ
IvUQcGUMckizsM6rLL3+8zf1h7mz16DY0+VBh1sCgYEAqk9E83Ihnqgf1n+63kaL
P630EARgnDLz9mKAQiZt+w9noAvGgHca36EPHun4o8Di/jVyZHcoh3Vj3led0GuN
IWamD3T6MV5bzb0G9z5d05BNEa/cuJHrc6jWfAjQWRYnf0uZal8BAebaKII1yCYU
PzFRr3Be04LESMOL97hmpKkCgYASVV5GXD2EFTWnseEyReqsIdH+QMpUcilsYAl
9pq1jkaqlLcBjG6CFjFmgbcH1NLxf7wz7VyXm8DICOkvBppcNZJ1eD3pWCQE+toC
x3w+KC/BmZ9jsmtjZuKop7x1d7AdZO5ARxKvH3W5IVU/KW3K7YVOrj24uHhN6BL7
6js0wKBgCfcq2y00nxRwj8zurDP38VqznBO/B//0KE7Tck9qfcT7jDPIYsmeKEX
zuzUWmYz6Y04uafpqsMig0TT5VV0zEmH8ijk5HXjS3Sa43JyXEJdemuCIzWZ/xfN
SLP5RrzIr/EFZ7zQdw1Bt4DuJMbXSPSqy08U6mw9Wyp7+W5zwZny
-----END RSA PRIVATE KEY-----
```

## 4. Create the Instance

Now, let's create an EC2 instance using the `aws ec2 run-instances` command. Since my student number is `24188516`, create an EC2 instance in the `eu-north-1` region.

```
aws ec2 run-instances --image-id ami-07a0715df72e58928 --security-group-ids
24188516-sg --count 1 --instance-type t3.micro --key-name 24188516-key --query
'Instances[0].InstanceId'
```

### Note:

At the time of running the lab, the `t2.micro` instance type was not available, so I switched to `t3.micro`. The instance was successfully created with the instance ID `i-0553e2ea0492e1c73`.

## Key Parameters:

- `--image-id`: Specifies the Amazon Machine Image (AMI) ID to be used for the instance. In this case, `ami-07a0715df72e58928` is used, refers to a pre-configured image for this class.
- `--security-group-ids`: Links the instance to the previously created security group (`24188516-sg`). This security group defines the allowed inbound and outbound traffic rules, including SSH access on port 22.
- `--count`: Specifies that only one instance will be created. This flag allows you to create multiple instances simultaneously if needed.
- `--instance-type`: Defines the type of EC2 instance to launch. Due to limitations at the time, **t3.micro** was chosen instead of **t2.micro**.
- `--key-name`: Specifies the name of the key pair (`24188516-key`) to associate with the instance. This key will be used to securely access the instance via SSH.
- `--query 'Instances[0].InstanceId'`: This extracts and displays the **InstanceId** of the newly created EC2 instance.

Once the command is executed, the instance is successfully created, and the **InstanceId** is displayed. In this case, the instance ID returned is `i-0553e2ea0492e1c73`.

```
liudayubob@Dayu:~$ aws ec2 run-instances --image-id ami-07a0715df72e58928 --security-group-ids 24188516-sg --count 1 --i
nstance-type t2.micro --key-name 24188516-key --query 'Instances[0].InstanceId' --region eu-north-1

An error occurred (Unsupported) when calling the RunInstances operation: The requested configuration is currently not su
ported. Please check the documentation for supported configurations.
liudayubob@Dayu:~$ aws ec2 run-instances --image-id ami-07a0715df72e58928 --security-group-ids 24188516-sg --count 1 --i
nstance-type t3.micro --key-name 24188516-key --query 'Instances[0].InstanceId' --region eu-north-1
"i-0553e2ea0492e1c73"
```

## 5. Add a Tag to the Instance

Now that we have the instance ID `i-0553e2ea0492e1c73`, we will add a tag to name the instance. The tag key will be `Name`, and the value will be our student number followed by `-vm` to uniquely identify the instance as `24188516-vm`.

```
aws ec2 create-tags --resources i-0553e2ea0492e1c73 --tags
Key=Name,Value=24188516-vm
```

## Key Parameters:

- `--resources`: Specifies the ID of the resource to tag, in this case, the instance ID `i-0553e2ea0492e1c73`.
- `--tags`: Defines the key-value pair for the tag. Here, the key is `Name`, and the value is `24188516-vm`, which labels the instance for identification purposes.

Once the command is executed, the instance will be tagged with `24188516-vm`, making it easier to identify in the AWS console.

## 6. Get the Public IP Address

To retrieve the public IP address of the instance, we use the `describe-instances` command. The query extracts only the `PublicIpAddress` from the instance details:

```
aws ec2 describe-instances --instance-ids i-0553e2ea0492e1c73 --query  
'Reservations[0].Instances[0].PublicIpAddress'
```

### Key Parameters:

- `--instance-ids`: Specifies the instance ID, which is `i-0553e2ea0492e1c73` in this case.
- `--query`: Limits the output to the `PublicIpAddress` of the instance, providing the required IP address for SSH access.

Once executed, the IP address of our instance is queried and printed, save it for SSH connection in the next step.

```
liudayubob@Dayu:~$ aws ec2 describe-instances --instance-ids i-0553e2ea0492e1c73 --query 'Reservations[0].Instances[0].P  
ublicIpAddress'  
"16.171.151.20"
```

## 7. Connect to the Instance via SSH

Now, we connect to the instance using the public IP address `16.171.151.20` via SSH. We need to use the previously generated `.pem` file to authenticate:

```
ssh -i 24188516-key.pem ubuntu@16.171.151.20
```

### Key Parameters:

- `-i`: Specifies the identity file (private key) to use for SSH authentication, which is `24188516-key.pem`.
- `ubuntu@16.171.151.20`: Connects to the instance as the `ubuntu` user, which is the default username.

After connecting, we can see system information on the console, indicating that the connection was successful.



```

ED25519 key fingerprint is SHA256:3k7mfXRpeve2eAlY+qWVKnAuRLN2z0AeJDh9JB3YZ2k.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '16.171.151.20' (ED25519) to the list of known hosts.
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 6.5.0-1022-aws x86_64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/pro

System information as of Mon Aug  5 07:44:03 UTC 2024

System load:  0.0           Processes:            100
Usage of /:   20.7% of 7.57GB Users logged in:      0
Memory usage: 22%          IPv4 address for ens5: 172.31.32.6
Swap usage:   0%

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

```

## 8. List the Created Instance Using the AWS Console

The original instance created in steps 1-7 was destroyed overnight, so I had to run the commands again and the instance ID would differ. Here is a screenshot of the successfully created instance from the AWS console:

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
24188516-vm	i-0c6dd3f1c0c4109dc	Running	t3.micro	2/2 checks passed	User: arn:aws:eu-north-1b	ec2-16-171-28-137.eu-...	16.171.28.137	

## Create an EC2 Instance with Python Boto3

In this step, we create an EC2 instance using the **boto3** Python package instead of AWS CLI commands. While the method names and parameters differ, the outcome is the same as in the previous steps. To differentiate this instance from the previous one, we append `-2` to the **Group name**, **Key name**, and **Instance name**.

The following Python script uses `boto3` to create the EC2 **instance**, **security group**, **key pair**, and **instance tag**:

# Workflow

## 1. Create Security Group:

The script starts by creating a security group ( `24188516-sg-2` ) using `ec2.create_security_group()`.

## 2. Authorize SSH Inbound Rule:

Next, an SSH rule is added using `ec2.authorize_security_group_ingress()`. This allows SSH access on port **22** from all IP addresses ( `0.0.0.0/0` ).

## 3. Create Key Pair:

A key pair ( `24188516-key-2` ) is generated using `ec2.create_key_pair()`, and the private key is saved locally with restricted access permissions using `os.chmod()` to secure it.

## 4. Create EC2 Instance:

The script launches an EC2 instance in the specified security group using `ec2.run_instances()`. The **AMI ID** ( `ami-07a0715df72e58928` ), **instance type** ( `t3.micro` ), and **key name** ( `24188516-key-2` ) are provided as parameters.

## 5. Tag EC2 Instance:

A name tag ( `24188516-vm-2` ) is created for the EC2 instance using `ec2.create_tags()`, which helps in identifying the instance easily.

## 6. Retrieve Public IP Address:

The public IP address of the newly created EC2 instance is retrieved using `ec2.describe_instances()`.

```
import boto3 as bt
import os

# Constants
GroupName = '24188516-sg-2'
KeyName = '24188516-key-2'
InstanceName = '24188516-vm-2'

ec2 = bt.client('ec2')

# 1. Create security group
step1_response = ec2.create_security_group(
    Description="security group for development environment",
    GroupName=GroupName
)

# 2. Authorize SSH inbound rule
step2_response = ec2.authorize_security_group_ingress(
    GroupName=GroupName,
    IpPermissions=[
        {
            'IpProtocol': 'tcp',
            'FromPort': 22,
            'ToPort': 22,
            'IpRanges': [{'CidrIp': '0.0.0.0/0'}]
        }
    ]
)
```

```

# 3. Create key pair
step3_response = ec2.create_key_pair(KeyName=KeyName)
PrivateKey = step3_response['KeyMaterial']

# Save key pair to a file
with open(f'{KeyName}.pem', 'w') as file:
    file.write(PrivateKey)

# Grant file permission to the private key
os.chmod(f'{KeyName}.pem', 0o400)

# 4. Create EC2 instance
step4_response = ec2.run_instances(
    ImageId='ami-07a0715df72e58928',
    SecurityGroupIds=[GroupName],
    MinCount=1,
    MaxCount=1,
    InstanceType='t3.micro',
    KeyName=KeyName
)

# Retrieve the Instance ID
InstanceId = step4_response['Instances'][0]['InstanceId']

# 5. Create a tag for the instance
step5_response = ec2.create_tags(
    Resources=[InstanceId],
    Tags=[{'Key': 'Name', 'Value': InstanceName}]
)

# 6. Get the public IP address of the instance
step6_response = ec2.describe_instances(InstanceIds=[InstanceId])
public_ip_address = step6_response['Reservations'][0]['Instances'][0][
    'PublicIpAddress']

# Print all responses
print(f"
{step1_response}\n{step2_response}\n{PrivateKey}\n{InstanceId}\n{step5_response}\
n{public_ip_address}\n")

```

## Code Explanation

1. `ec2.create_security_group()`:
  - `Description`: Describes the purpose of the security group, here labeled as "security group for development environment".
  - `GroupName`: Defines the name of the security group, in this case, `24188516-sg-2`.
2. `ec2.authorize_security_group_ingress()`:
  - `GroupName`: Specifies the security group where the rule will be added, in this case, `24188516-sg-2`.
  - `IpPermissions`: This parameter contains the rules that specify what type of inbound traffic is allowed.
    - `IpProtocol`: Defines the protocol, here set to `tcp` for SSH access.

- `FromPort` and `ToPort`: Both set to `22`, defining the SSH port.
  - `IpRanges`: Defines the IP range allowed to access the instance. Here, `0.0.0.0/0` allows access from any IP.
3. `ec2.create_key_pair()`:
- `KeyName`: Specifies the name of the key pair, here `24188516-key-2`, generates a new key pair and returns the private key.
4. `file.write()`:
- The private key is saved to a `.pem` file using Python's built-in File library with the `open()` function, and `os.chmod()` is used to set the file's permission to `400` (read-only).
5. `ec2.run_instances()`:
- `ImageId`: Specifies the Amazon Machine Image (AMI) ID, in this case, `ami-07a0715df72e58928`, which contains pre-configured software and settings.
  - `SecurityGroupIds`: Lists the security group IDs that will be associated with the instance. Here, the security group is `24188516-sg-2`.
  - `MinCount` and `MaxCount`: Define how many instances to launch. only one instance will be created in our case.
  - `InstanceType`: Defines the type of instance to launch, in this case, `t3.micro`.
  - `KeyName`: Specifies the name of the key pair, `24188516-key-2`, used for SSH access.
6. `ec2.create_tags()`:
- `Resources`: Specifies the resources to tag, in this case, the instance ID.
  - `Tags`: Defines the key-value pairs for tagging. Here, the tag key is `Name` and the value is `24188516-vm-2`, which labels the instance for easier identification.
7. `ec2.describe_instances()`:
- `InstanceIds`: Specifies the instance ID to describe details on.

Once the script is executed, the responses from each step are printed, showing the security group creation, key pair, instance ID, and public IP address.

```
liudayubob@Dayu: ~/cits5503$ python3 lab2.py
{'GroupId': 'sg-039acf5ea0b94b52b', 'ResponseMetadata': {'RequestId': '77cf0d56-c253-4b1c-9f9f-534f937aaf80', 'HTTPStatus': 200, 'HTTPHeaders': {'x-amzn-requestid': '77cf0d56-c253-4b1c-9f9f-534f937aaf80', 'cache-control': 'no-cache, no-store', 'strict-transport-security': 'max-age=31536000; includeSubDomains', 'content-type': 'text/xml; charset=UTF-8', 'content-length': '283', 'date': 'Mon, 12 Aug 2024 07:27:38 GMT', 'server': 'AmazonEC2'}, 'RetryAttempts': 0}}
{'Return': True, 'SecurityGroupRules': [{'SecurityGroupId': 'sgr-0923b0baaf289af26', 'GroupId': 'sg-039acf5ea0b94b52b', 'GroupOwner': '489389878001', 'IsEgress': False, 'IpProtocol': 'tcp', 'FromPort': 22, 'ToPort': 22, 'CidrIpv4': '0.0.0.0/0'}], 'ResponseMetadata': {'RequestId': 'd2143bd0-fa94-4907-94fc-8fbbc8dc300a', 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': 'd2143bd0-fa94-4907-94fc-8fbbc8dc300a', 'cache-control': 'no-cache, no-store', 'strict-transport-security': 'max-age=31536000; includeSubDomains', 'content-type': 'text/xml; charset=UTF-8', 'content-length': '719', 'date': 'Mon, 12 Aug 2024 07:27:39 GMT', 'server': 'AmazonEC2'}, 'RetryAttempts': 0}}
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAAnYY9+Gq4xLY4JpURQ6bCbJA/28xP5hWLNDoV9CrXETSX8mSN
sxPytGzXGn1pQ0KNT4+ZR77Go4uB6UnBgJ6ocQdwlHMtL3lw55S/KB4osJED
vBN8oESjmeuItNhdaFSzs9L496mcA1CF/ST+11Eg4rWngyJE/s/7cdWqoCEAxY
8hAqbsT6hX4YaNeL4/TvaWm+y6cNIz3x+LUzNVg4XGyikhnPrRHP1ScN3wzdWYXV
g2fQnID9yqp2MOYRp1xzoYJsoVc17xX53WrhQT4rhLrITqCV7mcXV0KYFPAEAsQ
7fBHHzU+gCz0Iogo6hqlLjxzYF0LOtNIs6KL5wIDAQABAoIBAGad3d7nfwLnf3S6
551CBcP+8Gq6NU54uF9Ee6q9Ep+XuGhiUdW30PmmEyBVZL2puF2nWB7Qqpni+u
T4nuz0x/jtFA/F7oJUnBBekGwGRdCTrErT8nHqe1LzKFC1Gj15m2fazdowVLPZ3N
VX7oFTSWRw6FBn8AEAIRwx1uZ4P4c9QjUsGnTIXYb1hNFPVpxidoHaDBnNHDjvXU
jBZRI0V8tIjCTOfPG0L1840xZfrnQnWDay4ZC+tK9pBx88Sv6wRPLo0y1dHAmPO
6miUW7H5SYSGeQeXmz77cYAoy6raRLe7gXf/2T2wFPFq5wasksfLVBuCltNE0u6h
vWfgEXkCYEA8s1a7grBRR04DtJS1sAPRneSrd835v00bMIWZqBiorLC4nXypN7p
hbS9wJLI+0j+6/PvW+LCIG0SaI/T78JGQBAWwGFTzx5mNypR3dZ/WSEC169k8Ee
PcwqzI1ZcZNCvpXH9r0jh32pBaAy9W5IAPZEdHqWv0dxirhEm6HsEw0CgYEAphj2
35rLo4oXeuXp8LaA++hfqM8RfOCrWyr7GasXdkH21LZNpU49ZU55LMAVXUV2kjh
+ZNL1FuZeh1uieFAhBdLD+DjSg0SmpEbVz7IshzN1mrZ0fm/t0G+yo/rvXiR3s8c
Vi5s2/T0b/eisK31rScJ45+cW/h4VfTdyb1j78McGYEA6w0oXJGZL2pk4uybSNAO
ueFF/7fBbnZ5mZBu8ABXpUE+qzIavhpS4phW53fQ/muCB7xV+zaxTRw5jcpZoyE
kvFF+4Lfxp09AcwtKMQrOpik/6H6Q05XT+sPcJRULCDPGkb4Po8yfiWZcxni3ByV
12Cva90Xil6JXY0C105Pu0CgYAPkFVJvJqKV309wnj3KFLlyAL2qviB8EqaP1fK
HvMS3ScA8yl6PuuX20WfQXhrhpuZducBEjNbkvYm7No7hd61Sg6c6s4nCL6+9TB
BxwNPI+5IMgkWemyBVIYbwXxYb1zZILWPfIS1vY0q7Kt9Y8Iy5isoJ8pKw4zmFjQ
x2NquwKBgQDbW9AcyNDh4ydp739/+cWzeCxxLQpMxPdHJLhJx1nFAFhp029pF
mVfQ3zhNuxRZ2hL6LbKHvgWihsa7x4cqrur7b50IoqKVKVP0iFi/iwrLS27bumk
L4B5NocdRji4wFxpTgLSL3848QZydh8Wa+UH1p5Pvzk4oMC1Q0AQAA==
-----END RSA PRIVATE KEY-----
i-05213726d4fa9897
{'ResponseMetadata': {'RequestId': 'b5c5f91a-bf5a-447d-8919-0228a321d8b7', 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': 'b5c5f91a-bf5a-447d-8919-0228a321d8b7', 'cache-control': 'no-cache, no-store', 'strict-transport-security': 'max-age=31536000; includeSubDomains', 'content-type': 'text/xml; charset=UTF-8', 'content-length': '221', 'date': 'Mon, 12 Aug 2024 07:27:41 GMT', 'server': 'AmazonEC2'}, 'RetryAttempts': 0}}
16.171.155.182

liudayubob@Dayu:~/cits5503/lab2$ |
```

Let's verify the created instance in the AWS console as well:

Instances (3) Info

Find Instance by attribute or tag (case-sensitive)

All states

24188516

Clear filters

< 1 >

⊗

<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP	IPv6 IPs	Monitoring
<input type="checkbox"/>	24188516-vm	i-0c6dd3f1c0c4109dc	<span>Running</span>	t3.micro	<span>2/2 checks passed</span>	<span>User: awsaws</span>	eu-north-1b	ec2-16-171-28-137.eu-...	16.171.28.137	-	-	disabled
<input type="checkbox"/>	24188516-vm-1	i-098c5a3899c9fb2e7	<span>Terminated</span>	t3.micro	-	<span>User: awsaws</span>	eu-north-1b	-	-	-	-	disabled
<input type="checkbox"/>	24188516-vm-2	i-05213726d4fa9897	<span>Running</span>	t3.micro	<span>Initializing</span>	<span>User: awsaws</span>	eu-north-1b	ec2-16-171-155-182.eu...	16.171.155.182	-	-	disabled

# Use Docker Inside a Linux OS

## 1. Install Docker

To install Docker, we use the following command to install the necessary packages:

```
sudo apt install docker.io -y
```

### Key Parameters:

- `docker.io`: Specifies the Docker package to install.
- `-y`: Automatically confirms the installation without prompting for user input.

## 2. Start the Docker Service

After installation, we start the Docker service to make it ready for use:

```
sudo systemctl start docker
```

### Key Parameters:

- `start`: Tells the system to start the Docker service.
- `docker`: Specifies the Docker service to start.

## 3. Enable Docker to Start on Boot

To ensure Docker starts automatically whenever the system boots, we enable the Docker service with:

```
sudo systemctl enable docker
```

### Key Parameters:

- `enable`: Enables Docker to start automatically when the system boots.
- `docker`: Specifies the Docker service to enable.

```
Setting up dnsmasq-base (2.90-0ubuntu0.22.04.1) ...
Setting up runc (1.1.12-0ubuntu2~22.04.1) ...
Setting up dns-root-data (2023112702~ubuntu0.22.04.1) ...
Setting up bridge-utils (1.7-1ubuntu3) ...
Setting up pigz (2.6-1) ...
Setting up containerd (1.7.12-0ubuntu2~22.04.1) ...
Created symlink /etc/systemd/system/multi-user.target.wants/containerd.service → /lib/systemd/system/containerd.service.
Setting up ubuntu-fan (0.12.16) ...
Created symlink /etc/systemd/system/multi-user.target.wants/ubuntu-fan.service → /lib/systemd/system/ubuntu-fan.service.
Setting up docker.io (24.0.7-0ubuntu2~22.04.1) ...
Adding group 'docker' (GID 117) ...
Done.
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /lib/systemd/system/docker.service.
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket → /lib/systemd/system/docker.socket.
Processing triggers for dbus (1.12.20-2ubuntu4.1) ...
Processing triggers for man-db (2.10.2-1) ...
liudayubob@Dayu:~$
```

## 4. Check Docker Version

To verify that Docker has been installed and is running properly, check its version using:

```
docker --version
```

### Key Parameters:

- `--version`: Prints the installed Docker version.

Once executed, this command outputs the installed Docker version, ensuring that Docker is ready to use.

```
Processing triggers for man-db (2.10.1-1) ...  
liudayubob@Dayu:~$ sudo systemctl start docker  
liudayubob@Dayu:~$ sudo systemctl enable docker  
liudayubob@Dayu:~$ docker --version  
Docker version 24.0.7, build 24.0.7-0ubuntu2~22.04.1  
liudayubob@Dayu:~$ |
```

## 5. Build and Run an `httpd` Container

In this step, we create an HTML file to be served via an Apache HTTP server running inside a Docker container.

### HTML File Creation

The file `index.html` is located inside the `html` directory and contains the following content:

```
<html>  
  <head></head>  
  <body>  
    <p>Hello, world!</p>  
  </body>  
</html>
```

This file simply displays the message **"Hello, World!"** when accessed via a web browser.

### Create a Dockerfile

Outside the `html` directory, we create a `Dockerfile` to define the configuration for our Docker container. The file contains the following:

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

### Key Parameters:

- `FROM`: Specifies the base image for the container. In this case, it uses Apache HTTP Server version 2.4.
- `COPY`: Copies the contents of the `html` directory from the local system into the container's web server directory (`/usr/local/apache2/htdocs/`), making the `index.html` file accessible via the web server.

### Add User to Docker Group

We add our username (`liudayubob`) to the Docker group to grant permission to manage Docker containers, then reboot the system:

```
sudo usermod -a -G docker liudayubob
```

## Key Parameters:

- `usermod -a -G`: Adds the user `liudayubob` to the Docker group (`docker`), allowing us to manage Docker without administrative permissions.

## Build the Docker Image

Once the `Dockerfile` and `html` folder are in place, we build the Docker image using the following command:

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

## Key Parameters:

- `build`: Instructs Docker to build an image based on the `Dockerfile` in the current directory.
- `-t`: Tags the image with the name `my-apache2` for easy reference.
- `.`: Specifies the build context, indicating the current directory (where the `Dockerfile` and `html` folder are located).

Once executed, this command builds the Docker image alias as `my-apache2`, preparing it to run the Apache server that serves the `index.html` file.

```
liudayubob@Dayu:~/cits5503/lab2$ docker build -t my-apache2 .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
             Install the buildx component to build images with BuildKit:
             https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  8.704kB
Step 1/2 : FROM httpd:2.4
--> 19c71fbb7140
Step 2/2 : COPY ./html/ /usr/local/apache2/htdocs/
--> c4423761f4e8
Successfully built c4423761f4e8
Successfully tagged my-apache2:latest
liudayubob@Dayu:~/cits5503/lab2$
```

## Run the Docker Container

After building the image `my-apache2`, we run the Docker container using the following command:

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

## Key Parameters:

- `-p`: Maps the host machine's port to the Docker container's port, enabling access to the container's web server from the host.
- `-dit`: Runs the container in detached mode (`d`), keeps STDIN open (`i`), and allocates a pseudo-TTY (`t`).
- `--name`: Sets the container name to `my-app`.

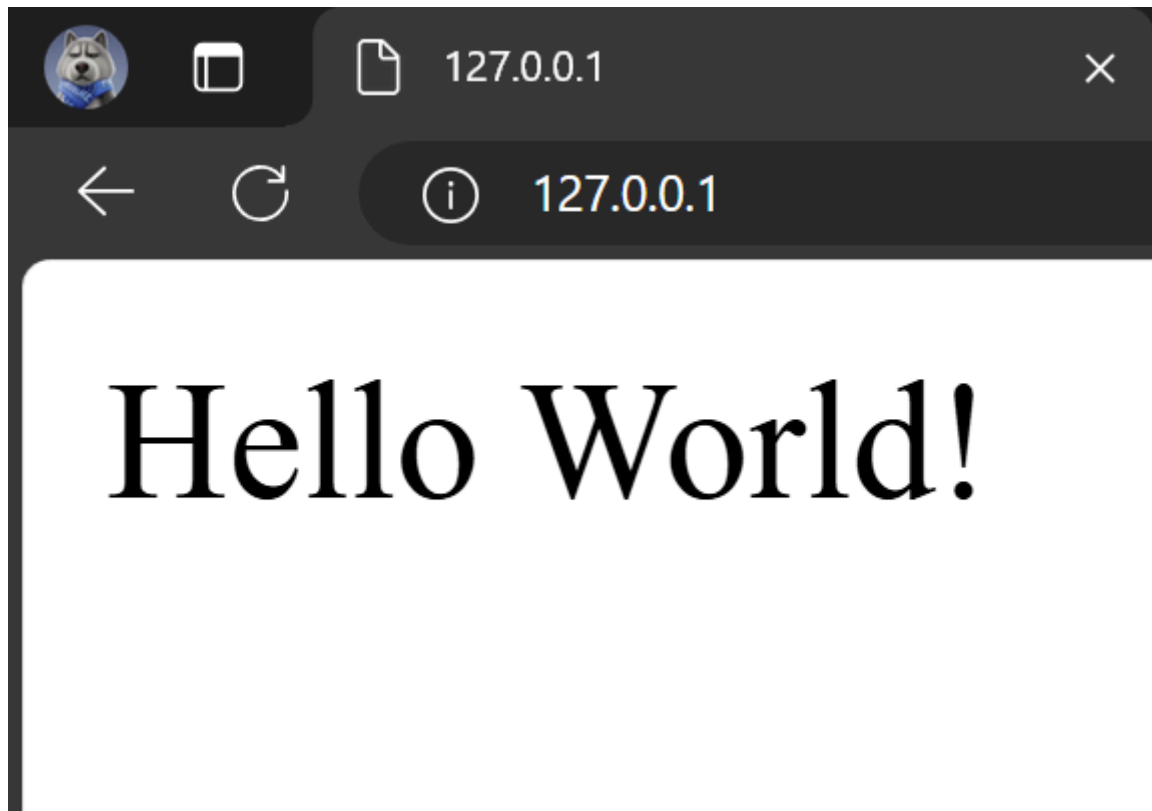
Once executed, this command starts the Apache server inside the container, serving the HTML content at port 80.



```
Successfully tagged my-apache2:latest
liudayubob@Dayu:~/cits5503/lab2$ docker run -p 80:80 -dit --name my-app my-apache2
aab9e3ef7aaf093a900c19bdf860b57cdc3cf70719693d4394abbbd30326eb47
```

## Access the Hosted HTML Page

To view the hosted HTML page, open a browser and navigate to `http://localhost` or `http://127.0.0.1`. The browser will display the **"Hello, World!"** message from the `index.html` file served by the Apache HTTP server inside the Docker container.



## 6. Other Docker Commands

### Check Running Containers

To list all running containers, use the following command:

```
docker ps -a
```

### Key Parameters:

- `ps`: Lists the currently running containers.
- `-a`: Includes all containers, even those that are not running.

Once executed, this command displays the properties of the containers, such as **Container ID**, **STATUS**, **PORTS**, the container name, and the image used.

```
liudayubob@Dayu:~/cits5503/lab2$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAME
aab9e3ef7aaf   my-apache2    "httpd-foreground"      5 minutes ago Up 5 minutes  0.0.0.0:80->80/tcp, :::80->80/tcp  my-app
```

## Stop and Remove the Container

To stop and remove the running container, use the following commands:

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

### Key Parameters:

- `stop`: Stops the running container.
- `rm`: Removes the container from the system.

These commands stop the `my-app` container and then remove it from the system.

# Lab 3

---

## 1. Create Files

We begin by creating the required files and directories. The following file structure contains three files: `cloudstorage.py`, `rootfile.txt`, and `subfile.txt`.

```
liudayubob@Dayu:~/cits5503/lab3$ find .
.
./cloudstorage.py
./rootdir
./rootdir/rootfile.txt
./rootdir/subdir
./rootdir/subdir/subfile.txt
```

## 2. Save to S3 by Updating `cloudstorage.py`

The `cloudstorage.py` script is modified to create an S3 bucket named `24188516-cloudstorage` if it doesn't already exist. The script then traverses all directories and subdirectories in the root directory and uploads any discovered files to the S3 bucket.

## Workflow

### 1. Locate or Create the S3 Bucket:

- The script attempts to create an S3 bucket ( `24188516-cloudstorage` ) using `s3.create_bucket()`.
- It passes the bucket name through the `Bucket` parameter and specifies the region ( `eu-north-1` ) using `CreateBucketConfiguration`.
- If the bucket already exists or the creation fails, the script catches the exception and moves on.

### 2. Recursively Traverse the Local Directory:

- The script uses `os.walk()` to recursively traverse the root directory ( `.` ) and its subdirectories.
- For each file found, it captures the folder name and the file name, constructing the full path.

### 3. Format the File Key and Upload Files:

- For each discovered file, the script generates a file key by concatenating the folder name and the file name using `os.path.join()`, ensuring compatibility across operating systems.
- The function `s3.upload_file()` is called to upload the file to the S3 bucket. It accepts the local file path, the bucket name, and the file key (which determines where the file is stored in the S3 bucket).
- The file is uploaded to the same folder structure in the S3 bucket as it exists on the local machine.

Here is the script down below:

```

import os
import boto3

ROOT_DIR = '.'
ROOT_S3_DIR = '24188516-cloudstorage'
s3 = boto3.client("s3")

bucket_config = {'LocationConstraint': 'eu-north-1'}

def upload_file(folder_name, file, file_name):
    file_key = os.path.join(folder_name, file_name).replace("\\", "/")
    s3.upload_file(file, ROOT_S3_DIR, file_name) # file path, bucket name, key
    print(f"Uploading {file}")

# Main program
try:
    # Create bucket if not there
    response = s3.create_bucket(
        Bucket=ROOT_S3_DIR,
        CreateBucketConfiguration=bucket_config
    )
    print(f"Bucket created: {response}")
except Exception as error:
    print(f"Bucket creation failed: {error}")
    pass

# Traverse 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(f"{dir_name[2:]}/", f"{dir_name}/{fname}", fname)

print("done")

```

## Code Explanation

- `s3.create_bucket()`: Attempts to create an S3 bucket.
  - `Bucket`: Specifies the name of the bucket to be created, which is `24188516-cloudstorage`.
  - `CreateBucketConfiguration`: Defines configuration options for the bucket. In this case, we set the `LocationConstraint` to `eu-north-1`, which places the bucket in the specified AWS region.
- `os.walk()`: Recursively traverses through the root directory (`.`) and subdirectories, finding all files to be uploaded.
- `s3.upload_file()`: Uploads the file to the S3 bucket. It accepts the following parameters:
  - `file`: The local path to the file to upload.
  - `Bucket`: Specifies the destination S3 bucket, `24188516-cloudstorage`.
  - `file_name`: The key under which the file is stored in the S3 bucket, formed by concatenating the folder path and file name.

Now our files are uploaded to the corresponding locations in the S3 bucket, consistent with out local directory structure.

```
liudayubob@Dayu:~/cits5503/lab3$ python3 cloudstorage.py
Bucket creation failed: An error occurred (BucketAlreadyOwnedByYou) when calling the CreateBucket operation: Your previous request to create the named bucket succeeded and you already own it.
Uploading ./rootdir/rootfile.txt
Uploading ./rootdir/subdir/subfile.txt
done
```

### 3. Restore from S3

We create a new program, `restorefromcloud.py`, to restore files from the S3 bucket and write them to the appropriate directories. The program uses `s3.list_objects_v2()` to list all files in the S3 bucket along with their attributes, such as **Key** and **Name**.

We combine the local **ROOT\_TARGET\_DIR** with the **Key** to form the local file path. If the local directory does not exist, we create it using `os.makedirs()`. Finally, we download each file from the S3 bucket using `s3.download_file()`.

## Workflow

#### 1. List Objects in the S3 Bucket:

- The script uses `s3.list_objects_v2()` to list all the files (objects) in the `24188516-cloudstorage` S3 bucket. The response contains details about each file, including its **Key**, which indicates the path of the file within the S3 bucket.

#### 2. Create Local Directory Structure:

- For each file found in the S3 bucket, the script constructs a local file path by combining the **ROOT\_TARGET\_DIR** (the root directory where files will be restored) with the file's S3 **Key**.
- Before downloading the file, the script checks if the directory for the local file path exists using `os.path.exists()`. If the directory does not exist, the script creates it using `os.makedirs()`.

#### 3. Download Files from S3:

- Once the local directory is confirmed or created, the script calls `s3.download_file()` to download the file from the S3 bucket and store it in the corresponding local directory.
- The function accepts the **S3 bucket name**, **S3 key (file path)**, and **local file path** as parameters, ensuring that the files are restored to the correct locations.

Here is the script down below:

```
import os
import boto3

ROOT_TARGET_DIR = '.' # Root directory where files will be restored
ROOT_S3_DIR = '24188516-cloudstorage'
s3 = boto3.client("s3")

def download_file(s3_key, local_file_path):
    local_dir = os.path.dirname(local_file_path)

    # Ensure the local directory exists
```

```

if not os.path.exists(local_dir):
    print(f"Creating directory {local_dir}")
    os.makedirs(local_dir)

# Download the file
s3.download_file(ROOT_S3_DIR, s3_key, local_file_path)
print(f"Downloading {s3_key} to {local_file_path}")

# Main program
# List all objects in the S3 bucket
objects = s3.list_objects_v2(Bucket=ROOT_S3_DIR)

if 'Contents' in objects:
    for obj in objects['Contents']:
        s3_key = obj['Key']
        local_file_path = os.path.join(ROOT_TARGET_DIR, s3_key).replace("/",
os.path.sep)

        # Download the file from S3 to the corresponding local path
        download_file(s3_key, local_file_path)
else:
    print("No objects found in the bucket.")
    pass

print("done")

```

## Code Explanation

- `s3.list_objects_v2()`: Lists all objects stored in the specified S3 bucket.
  - `Bucket`: Specifies the S3 bucket name, which is `24188516-cloudstorage` in our case.
- `s3.download_file()`: Downloads the specified file from S3 to the local directory.
  - `Bucket`: Specifies the S3 bucket name, `24188516-cloudstorage`.
  - `s3_key`: The key (path) of the file in the S3 bucket.
  - `local_file_path`: Specifies the destination file path on the local machine.
- `os.makedirs()`: Creates the specified directory if it doesn't already exist, so we can mirror the local directory structure to our S3 directory structure.

Once executed, this script traverses the S3 bucket, restoring files to the local directory in the same structure they were uploaded.

```

liudayubob@Dayu:~/cits5503/lab3$ python3 restorefromcloud.py
Create directory: ./rootdir
Downloading rootdir/rootfile.txt to ./rootdir/rootfile.txt
Create directory: ./rootdir/subdir
Downloading rootdir/subdir/subfile.txt to ./rootdir/subdir/subfile.txt
done

```

## 4. Write Information About Files to DynamoDB

### 1. Install DynamoDB

First, we create and navigate into the `dynamodb` directory. We then install **JRE** and download the **DynamoDB** package, extracting the necessary files for local use. Once extracted, we have the compiled Java code `DynamoDBLocal.jar` and a folder containing libraries `DynamoDBLocal_lib`, which are required to run a local DynamoDB instance.

```
mkdir dynamodb
cd dynamodb

# Install JRE
sudo apt-get install default-jre

# Download DynamoDB package
wget https://s3-ap-northeast-1.amazonaws.com/dynamodb-1ocal-
tokyo/dynamodb_1ocal_1atest.tar.gz

# Extract DynamoDB
tar -zxvf dynamodb_1ocal_1atest.tar.gz
```

```
liudayubob@Dayu:~/cits5503/lab3/dynamodb$ ls
DynamoDBLocal.jar  DynamoDBLocal_lib  LICENSE.txt  README.txt  THIRD-PARTY-LICENSES.txt  dynamodb_1ocal_1atest.tar.g
z
liudayubob@Dayu:~/cits5503/lab3/dynamodb$
```

Next, we start the DynamoDB instance locally using **JRE**. The port is set to **8001** since **8000** is already in use. The `-sharedDb` flag creates a single database file, `_shared-1ocal-1instance.db`, which is accessed by all programs connecting to this local DynamoDB instance.

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb -
port 8001
```

### Key Parameters:

- `-Djava.library.path`: Specifies the path to the required native libraries for running DynamoDB locally ( `./DynamoDBLocal_lib` ).
- `-jar`: Indicates the JAR file ( `DynamoDBLocal.jar` ) that contains the DynamoDB local service.
- `-sharedDb`: Configures DynamoDB to use a single shared database file ( `_shared-1ocal-1instance.db` ).
- `-port`: Specifies that the service should listen on port 8001.

```
liudayubob@Dayu:~/cits5503/lab3/dynamodb$ java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -share
ddb -port 8001
Initializing DynamoDB Local with the following configuration:
Port:      8001
InMemory:  false
Version:   1.25.0
DbPath:    null
SharedDb:  false
shouldDelayTransientStatuses: false
CorsParams: null
```

## 2. Create a Table in DynamoDB

We create a Python script, `createtable.py`, to define a table named `CloudFiles` in DynamoDB. The table uses `userId` as the partition key and `fileName` as the sort key. We define the keys using `KeyType` (`HASH` for partition key and `RANGE` for sort key), while `AttributeName` and `AttributeType` specify the attributes' names and types.

Although DynamoDB is schema-free, attributes like `path`, `lastUpdated`, `owner`, and `permissions` don't need to be predefined in the table schema, but they can be added later when inserting items into the table.

Here's the table schema:

```
# Database schema
CloudFiles = {
    'userId',
    'fileName',
    'path',
    'lastUpdated',
    'owner',
    'permissions'
}
```

Here's the script to create the table:

```
# createtable.py
import boto3

def create_db_table():
    # Initialize DynamoDB service instance
    dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8001")

    table = dynamodb.create_table(
        TableName='CloudFiles',
        KeySchema=[
            {
                'AttributeName': 'userId',
                'KeyType': 'HASH' # Partition key
            },
            {
                'AttributeName': 'fileName',
                'KeyType': 'RANGE' # Sort key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'userId',
                'AttributeType': 'S' # String type
            },
            {
                'AttributeName': 'fileName',
                'AttributeType': 'S' # String type
            }
        ],
        ProvisionedThroughput={
```



```

        'ReadCapacityUnits': 1,
        'WriteCapacityUnits': 1
    }
)

print("Table status:", table.table_status)

if __name__ == '__main__':
    create_db_table()

```

## Code Explanation

- `boto3.resource("dynamodb")`: Initializes a DynamoDB resource instance, allowing interaction with the DynamoDB service. We specify `endpoint_url="http://localhost:8001"` to connect to the local DynamoDB instance running on port **8001**.
- `dynamodb.create_table()`: Creates a new table in DynamoDB.
  - `TableName`: Specifies the name of the table, here `CloudFiles`.
  - `KeySchema`: Defines the partition key and sort key for the table:
    - `AttributeName`: Specifies the name of the attribute. We use `userId` for the partition key and `fileName` for the sort key.
    - `KeyType`: Specifies whether the attribute is a partition key (`HASH`) or a sort key (`RANGE`).
  - `AttributeDefinitions`: Specifies the types of attributes used in the key schema:
    - `AttributeType`: Defines the type of the attribute. In this case, both `userId` and `fileName` are of type `S` (string).
  - `ProvisionedThroughput`: Defines the read and write capacity for the table. Here, both read and write capacity are set to 1.

Once executed, this script connects to the local DynamoDB instance, creates the `CloudFiles` table, and prints the table status after creation.

```

DynamoDBLocal_CUI README.TXT createtable.py dynamodb_co
liudayubob@Dayu:~/cits5503/lab3/dynamodb$ python3 createtable.py
Table status: ACTIVE

```

## 3. Write Data into the `CloudFiles` Table

In this step, we write data into the `CloudFiles` table. First, we use `s3.list_objects_v2()` to list all files in the `24188516-cloudstorage` bucket. The output contains attributes such as **Key** and **LastModified**. To retrieve additional information like **Owner** and **Permissions**, we make a separate call to `s3.get_object_acl()`, which provides these details under the **Grants** and **Owner** attributes.

After extracting all necessary attributes, we use `dynamodb_table.put_item()` to insert each object into the DynamoDB table. Since our designated region is `eu-north-1`, we populate the `owner` field with the owner's ID.

# Workflow

## 1. Lists all files in the S3 Bucket:

The script uses `s3.list_objects_v2()` to retrieve the list of all objects in the specified S3 bucket (`24188516-cloudstorage`). The response contains metadata for each file, including the `key` (file name) and `LastModified` timestamp.

## 2. Retrieves Owner and Permission Information:

For each file, the script calls `s3.get_object_acl()` to fetch the Access Control List (ACL) associated with the file. The ACL contains details about the file's owner and permission settings, found in the **Grants** and **Owner** attributes.

## 3. Extracts File Attributes:

The script extracts key attributes from each file, including:

- **userId**: The ID of the user or entity who has permissions for the file.
- **fileName**: The name of the file, extracted from the `key` in the S3 response.
- **path**: The full S3 key (file path) of the file.
- **lastUpdated**: The last modification date of the file, extracted from `LastModified`.
- **owner**: The owner's ID, retrieved from the ACL.
- **permissions**: The file's access permissions, extracted from the ACL's `Grants`.

## 4. Inserts File Attributes into the DynamoDB Table:

The extracted file attributes are then inserted into the DynamoDB `CloudFiles` table using `dynamodb_table.put_item()`. This operation stores each file's metadata in the database, associating it with the appropriate `userId` and `fileName`.

Here's the script:

```
# writetable.py
import boto3
import os

BUCKET_NAME = '24188516-cloudstorage'
DB_NAME = 'CloudFiles'

# Set up AWS instances for S3 and DynamoDB
s3 = boto3.client('s3')
dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8001")
dynamodb_table = dynamodb.Table(DB_NAME)

def list_files():
    # List all objects in the S3 bucket
    files = []
    objects = s3.list_objects_v2(Bucket=BUCKET_NAME)

    if 'Contents' in objects:
        for obj in objects['Contents']:
            # Get access control list for owner and permission information
            obj_acl = s3.get_object_acl(Bucket=BUCKET_NAME, Key=obj['key'])
            files.append({**obj, **obj_acl})

    return files
```

```

def extract_file_attributes(file):
    # Extract attributes of a file
    file_attributes = {
        'userId': file['Grants'][0]['Grantee']['ID'],
        'fileName': os.path.basename(file['Key']),
        'path': file['Key'],
        'lastUpdated': file['LastModified'].isoformat(),
        'owner': file['Owner']['ID'],
        'permissions': file['Grants'][0]['Permission']
    }

    return file_attributes

def write_to_table():
    # List all files in the bucket and write them to the DynamoDB table
    try:
        files = list_files()

        # Iterate through each file
        for file in files:
            # Extract attributes for the file
            file_attributes = extract_file_attributes(file)

            # Write the attributes to DynamoDB
            db_res = dynamodb_table.put_item(Item=file_attributes)
            print(f"Inserted {file_attributes['fileName']} into DynamoDB")

    except Exception as error:
        print(f"Database write operation failed: {error}")
        pass

if __name__ == '__main__':
    write_to_table()

```

## Code Explanation

- `s3.list_objects_v2()`: Lists all objects in the specified S3 bucket.
  - `Bucket`: Specifies the name of the bucket to retrieve the object list from, in this case, `24188516-cloudstorage`.
- `s3.get_object_acl()`: Retrieves the access control list (ACL) of the specified object to get details like the owner and permissions.
  - `Bucket`: Specifies the S3 bucket name, `24188516-cloudstorage`.
  - `Key`: Specifies the key (path) of the object for which the ACL is retrieved.
- `dynamodb_table.put_item()`: Inserts an item into the DynamoDB table.
  - `Item`: Specifies the attributes of the item to insert. In this case, it includes attributes like `userId`, `fileName`, `path`, `lastUpdated`, `owner`, and `permissions`.

```

liudayubob@Dayu:~/cits5503/lab3/dynamodb$ python3 writetable.py
Inserted rootdir/rootfile.txt into DynamoDB
Inserted rootdir/subdir/subfile.txt into DynamoDB

```

## 4. Print and Destroy the CloudFiles Table

### Print the Table

We use the AWS CLI to scan and print the contents of the CloudFiles table. The following command retrieves all items in the table and displays them:

```
aws dynamodb scan --table-name CloudFiles --endpoint-url http://localhost:8001
```

### Key Parameters:

- `--table-name`: Specifies the name of the DynamoDB table to scan, in this case, `CloudFiles`.
- `--endpoint-url`: Specifies the endpoint URL for connecting to the local DynamoDB instance running on port **8001**.

Once executed, this command prints the table structure, showing the data we inserted in the previous step.

```
liudayubob@Dayu:~/cits5503/lab3/dynamodb$ aws dynamodb scan --table-name CloudFiles --endpoint-url http://localhost:8001
{
  "Items": [
    {
      "owner": {
        "S": "2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e"
      },
      "path": {
        "S": "rootdir/rootfile.txt"
      },
      "lastUpdated": {
        "S": "2024-08-19T04:12:37+00:00"
      },
      "fileName": {
        "S": "rootfile.txt"
      },
      "userId": {
        "S": "2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e"
      },
      "permissions": {
        "S": "FULL_CONTROL"
      }
    },
    {
      "owner": {
        "S": "2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e"
      },
      "path": {
        "S": "rootdir/subdir/subfile.txt"
      },
      "lastUpdated": {
        "S": "2024-08-19T04:12:38+00:00"
      },
      "fileName": {
        "S": "subfile.txt"
      },
      "userId": {
        "S": "2a5fac7aada1ad2caa48c9ab08cc4e2428d4eb596108daa3b59f1204ae96482e"
      },
      "permissions": {
        "S": "FULL_CONTROL"
      }
    }
  ],
  "Count": 2,
  "ScannedCount": 2,
  "ConsumedCapacity": null
}
```

## Destroy the Table

To delete the `CloudFiles` table, we use the following AWS CLI command:

```
aws dynamodb delete-table --table-name CloudFiles --endpoint-url
http://localhost:8001
```

### Key Parameters:

- `--table-name`: Specifies the name of the DynamoDB table to delete, in this case, `CloudFiles`.
- `--endpoint-url`: Specifies the endpoint URL for connecting to the local DynamoDB instance running on port **8001**.

Once executed, this command deletes the table, removing all data and schema. Also the defined schema (partition key and sort key) will be printed before deletion.

```
liudayubob@Dayu:~/cits5503/lab3/dynamodb$ aws dynamodb delete-table --table-name CloudFiles --endpoint-url http://loca
lhost:8001
{
  "TableDescription": {
    "AttributeDefinitions": [
      {
        "AttributeName": "userId",
        "AttributeType": "S"
      },
      {
        "AttributeName": "fileName",
        "AttributeType": "S"
      }
    ],
    "TableName": "CloudFiles",
    "KeySchema": [
      {
        "AttributeName": "userId",
        "KeyType": "HASH"
      },
      {
        "AttributeName": "fileName",
        "KeyType": "RANGE"
      }
    ],
    "TableStatus": "ACTIVE",
    "CreationDateTime": 1724044579.229,
    "ProvisionedThroughput": {
      "LastIncreaseDateTime": 0.0,
      "LastDecreaseDateTime": 0.0,
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 1,
      "WriteCapacityUnits": 1
    },
    "TableSizeBytes": 489,
    "ItemCount": 2,
    "TableArn": "arn:aws:dynamodb:ddblocal:000000000000:table/CloudFiles",
    "DeletionProtectionEnabled": false
  }
}
```

# Lab 4

## Apply a Policy to Restrict Permissions on Bucket

### 1. Write a Python Script

In this lab, we apply an access permission policy to the S3 bucket `24188516-cloudstorage` created in the previous lab. The policy restricts access to this bucket, allowing only the user with the username `24188516@student.uwa.edu.au` to access the contents.

The policy is defined as a JSON document, where:

- **Sid**: A unique identifier for the policy statement.
- **Effect**: Specifies the result of the policy, set to `"DENY"`, meaning the action is denied if the condition is met.
- **Action**: Specifies the S3 actions being denied, in this case, `"s3:*"` to deny all S3 actions.
- **Resource**: Specifies the resources affected by the policy, here all objects in the `24188516-cloudstorage` bucket.
- **Condition**: Specifies a condition that checks if the `aws:username` is not `24188516@student.uwa.edu.au`. If true, access is denied.

Here's the bucket policy in JSON format:

```
# bucketpolicy.json
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "AllowAllS3ActionsInUserFolderForUserOnly",
    "Effect": "DENY",
    "Principal": "*",
    "Action": "s3:*",
    "Resource": "arn:aws:s3:::24188516-cloudstorage/*",
    "Condition": {
      "StringNotLike": {
        "aws:username": "24188516@student.uwa.edu.au"
      }
    }
  }
}
```

This JSON policy ensures that any user attempting to access the bucket, who is not `24188516@student.uwa.edu.au`, will be denied all actions related to S3. It also applies to all objects within the `24188516-cloudstorage` bucket, as specified by the **Resource**.

## Python Script to Apply the Policy

Since the policy parameter in `s3.put_bucket_policy()` only accepts a JSON string, we load the JSON policy from `bucketpolicy.json`, convert it into a string using `json.dumps()`, and then apply it to the bucket using `s3.put_bucket_policy()`.

## Workflow

### 1. Reads the JSON policy from File system:

The script uses `json.load()` to read the bucket policy from the `bucketpolicy.json` file. This policy outlines the permissions that will be applied to the S3 bucket.

### 2. Converts the policy to a string:

The policy is converted into a string format using `json.dumps()`, as the `s3.put_bucket_policy()` function requires the policy to be passed as a JSON string.

### 3. Applies the policy to the S3 bucket:

The script uses `s3.put_bucket_policy()` to attach the policy to the specified S3 bucket (`24188516-cloudstorage`). This method takes in two parameters:

- **Bucket:** The name of the S3 bucket (`24188516-cloudstorage`).
- **Policy:** The policy in JSON string format that will govern access to the bucket.

Here's the Python script to apply the policy:

```
# addpolicy.py
import boto3
import json

BUCKET_NAME = '24188516-cloudstorage'

# Create an S3 instance
s3 = boto3.client('s3')

def apply_bucket_policy():
    # Import the policy from the JSON file
    with open('bucketpolicy.json', 'r') as policy_file:
        policy = json.load(policy_file)

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

    # Apply the policy to the bucket
    response = s3.put_bucket_policy(Bucket=BUCKET_NAME, Policy=policy_string)
    print("Policy applied!", response)

if __name__ == '__main__':
    apply_bucket_policy()
```

## Code Explanation

- `boto3.client('s3')`: Initializes an S3 client for interacting with the S3 service.
- `json.load()`: Reads and parses the `bucketpolicy.json` file into a Python dictionary.
- `json.dumps()`: Converts the Python dictionary containing the policy into a JSON string format, which is required by the `put_bucket_policy()` method.
- `s3.put_bucket_policy()`: Applies the bucket policy to the specified S3 bucket.
  - `Bucket`: Specifies the name of the S3 bucket, here `24188516-cloudstorage`.
  - `Policy`: Accepts the policy as a JSON string, which defines the access control rules for the bucket.

```
liudayubob@Dayu:~/cits5503/lab4$ python3 addpolicy.py
policy_string: {"Version": "2012-10-17", "Statement": [{"Sid": "AllowAllS3ActionsInUserFolderForUserOnly", "Effect": "Deny", "Principal": "*", "Action": "s3:*", "Resource": "arn:aws:s3:::24188516-cloudstorage/*", "Condition": {"StringNotLike": {"aws:username": "24188516@student.uwa.edu.au"}}}]}
Policy applied! {'ResponseMetadata': {'RequestId': 'MZW1F54J75E4J7N1', 'HostId': '05vj6KfwpagFLNFh2YhLYLMnceu4j04FB8E5czqJz2goS0zLG0CcFsF7uYRYAzyFa7jL59FacS8=', 'HTTPStatusCode': 204, 'HTTPHeaders': {'x-amz-id-2': '05vj6KfwpagFLNFh2YhLYLMnceu4j04FB8E5czqJz2goS0zLG0CcFsF7uYRYAzyFa7jL59FacS8=', 'x-amz-request-id': 'MZW1F54J75E4J7N1', 'date': 'Mon, 19 Aug 2024 06:27:08 GMT', 'server': 'AmazonS3'}, 'RetryAttempts': 0}}
```

## 2. Check Whether the Script Works

After applying the bucket policy, we test to ensure that the policy is working as intended.

### Verify the Policy Using AWS CLI

To check whether the policy has been applied to the `24188516-cloudstorage` bucket, we use the following AWS CLI command:

```
aws s3api get-bucket-policy --bucket 24188516-cloudstorage --query Policy --output text
```

### Key Parameters:

- `--bucket`: Specifies the name of the S3 bucket to check for the applied policy, in this case, `24188516-cloudstorage`.
- `--query Policy`: Filters the output to display only the bucket policy.
- `--output text`: Outputs the policy in plain text format.

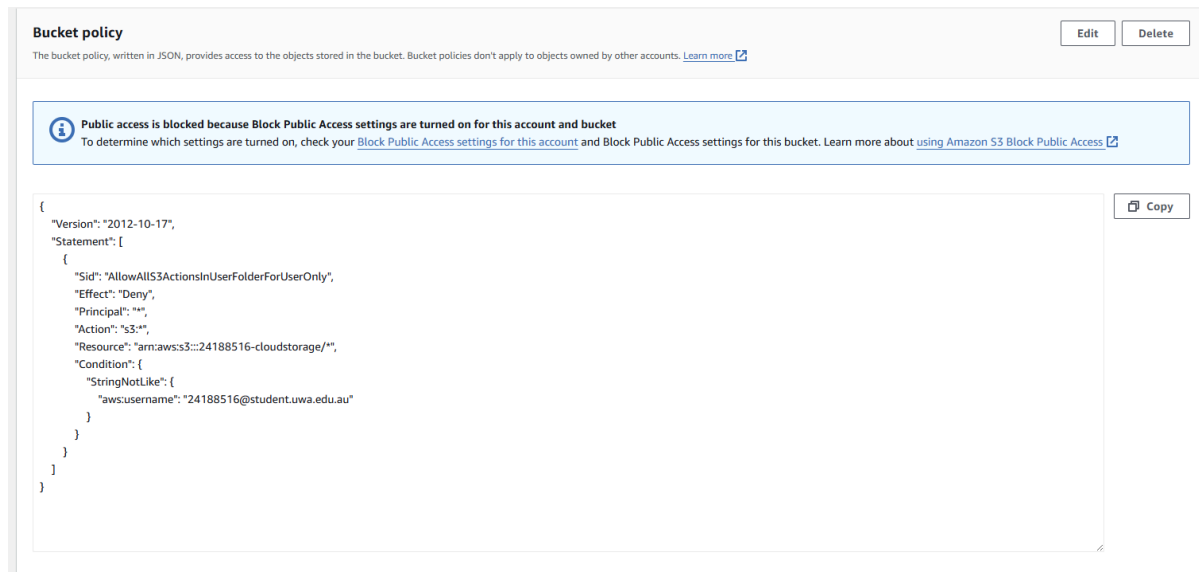
Once executed, this command retrieves the policy attached to the S3 bucket and outputs it in plain text. The expected output is the JSON policy document we applied earlier.

```
liudayubob@Dayu:~/cits5503/lab4$ aws s3api get-bucket-policy --bucket 24188516-cloudstorage --query Policy --output text
{"Version": "2012-10-17", "Statement": [{"Sid": "AllowAllS3ActionsInUserFolderForUserOnly", "Effect": "Deny", "Principal": "*", "Action": "s3:*", "Resource": "arn:aws:s3:::24188516-cloudstorage/*", "Condition": {"StringNotLike": {"aws:username": "24188516@student.uwa.edu.au"}}}]}]
```



## Visual Confirmation via AWS Console

Next, we navigate to the AWS console to visually confirm that the policy is in place for the `24188516-cloudstorage` bucket. The console should display the same policy, with the conditions we set for restricting access based on the username.



## Test Denied Access with Incorrect Username

To test whether the policy is correctly restricting access, we deliberately alter the username in the policy. For example, we change the username condition to only allow access to `12345678@student.uwa.edu.au`, effectively denying access to the current user `24188516@student.uwa.edu.au`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAllS3ActionsInUserFolderForUserOnly",
      "Effect": "Deny",
      "Principal": "*",
      "Action": "s3:*",
      "Resource": "arn:aws:s3:::24188516-cloudstorage/*",
      "Condition": {
        "StringNotLike": {
          "aws:username": "12345678@student.uwa.edu.au"
        }
      }
    }
  ]
}
```

Grant to the wrong one, should now deny access to 24188516 (my account)


Amazon S3 > Buckets > 24188516-cloudstorage > rootdir/ > rootfile.txt

## rootfile.txt [Info](#)

Properties | **Permissions** | Versions

### Access control list (ACL)

Grant basic read/write permissions to AWS accounts. [Learn more](#)

 **You don't have permission to get object ACL**  
After you or your AWS administrator has updated your permissions to allow the `s3:GetObjectAcl` action, refresh the page. [Learn more](#)

▶ API response

As expected, when trying to access the bucket resources under the user `24188516@student.uwa.edu.au`, the access is denied.

## AES Encryption Using KMS

### 1. Policy to be Attached to the KMS Key

The following JSON file, `kmspolicy.json`, defines the access control policy to be attached to the KMS (Key Management Service) key. This policy grants permissions to both the root account and the IAM user (`24188516@student.uwa.edu.au`), ensuring appropriate access levels for key management and cryptographic operations.

#### Four Policy Statements:

The policy contains four main statements:

1. **Enable IAM User Permissions:** Grants the root account full access to KMS operations.
2. **Allow access for Key Administrators:** Grants the IAM user permissions for key management tasks.
3. **Allow use of the key:** Grants the IAM user permissions for encryption, decryption, and other cryptographic operations.
4. **Allow attachment of persistent resources:** Allows the IAM user to manage grants, ensuring the grants are for AWS resources.

Here's the full JSON policy:

```
# kmspolicy.json
{
  "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/24188516@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/24188516@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/24188516@student.uwa.edu.au"
    },
    "Action": [
        "kms:CreateGrant",
        "kms:ListGrants",

```

```

        "kms:RevokeGrant"
    ],
    "Resource": "*",
    "Condition": {
        "Bool": {
            "kms:GrantIsForAWSResource": "true"
        }
    }
}
]
}

```

## Code Explanation

- **Statement 1:** Grants full access (`kms:*`) to the root account (`arn:aws:iam::489389878001:root`) for all KMS operations on all resources.
- **Statement 2:** The IAM user (`24188516@student.uwa.edu.au`) is granted permissions to perform key management tasks such as **creating, describing, enabling, disabling, tagging, and deleting** keys (`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`).
- **Statement 3:** The IAM user can use the key for cryptographic functions like **encrypting, decrypting, re-encrypting, and generating** keys (`kms:Encrypt`, `kms:Decrypt`, `kms:ReEncrypt`, `kms:GenerateDataKey`, `kms:DescribeKey`).
- **Statement 4:** Only when the grant is for an AWS resource (`kms:GrantIsForAWSResource`), allows the IAM user to manage grants like **creating, listing, and revoking** keys (`kms:CreateGrant`, `kms:ListGrants`, `kms:RevokeGrant`).

This policy ensures secure management of the KMS key, allowing only authorized users to perform key management and cryptographic operations.

## 2. Attach a Policy to the Created KMS Key

In this step, we create a symmetric encryption KMS key and apply the policy from the `kmspolicy.json` file that was defined earlier. The KMS key is specified for encryption and decryption purposes. After the key is created, we assign an alias using the student's ID, following the format `alias/*`, which results in `alias/24188516`.

Here's the Python script that performs these operations:

```

import boto3
import json

STUDENT_NUMBER = '24188516'

def create_kms_key():
    # Import the policy from the JSON file
    with open('kmspolicy.json', 'r') as policy_file:

```

```

policy = json.load(policy_file)

# Create a new KMS key with the imported policy
kms = boto3.client('kms')
key_response = kms.create_key(
    Policy=json.dumps(policy),
    KeyUsage='ENCRYPT_DECRYPT',
    Origin='AWS_KMS'
)

# Extract the KeyId from the response
key_id = key_response['KeyMetadata']['KeyId']

# Create an alias for the KMS key using the student number
alias_name = f'alias/{STUDENT_NUMBER}'
alias_response = kms.create_alias(
    AliasName=alias_name,
    TargetKeyId=key_id
)

print(f"Key and alias generated successfully!")

if __name__ == "__main__":
    create_kms_key()

```

## Code Explanation

- `boto3.client('kms')`: Initializes a KMS client for interacting with the AWS Key Management Service.
- `kms.create_key()`: Creates a new KMS key.
  - `Policy`: Specifies the access control policy (loaded from `kmspolicy.json`) that defines who can manage and use the key.
  - `KeyUsage`: Defines the purpose of the key, here set to `ENCRYPT_DECRYPT` for symmetric encryption and decryption.
  - `Origin`: Specifies the key material source, set to `AWS_KMS` to have AWS manage the key material.
- `key_response['KeyMetadata']['KeyId']`: Extracts the key ID from the response returned by `kms.create_key()`. The key ID uniquely identifies the key for future operations.
- `kms.create_alias()`: Assigns a human-readable alias to the KMS key.
  - `AliasName`: Defines the alias for the key, here set to `alias/24188516`.
  - `TargetKeyId`: Specifies the key ID to which the alias is assigned.

Once the script is executed, a symmetric KMS key is created with the policy applied, and an alias (`alias/24188516`) is assigned to the key.

```

liudayubob@Dayu:~/cits5503/lab4$ python3 createkmskey.py
Key and alias generated successfully!

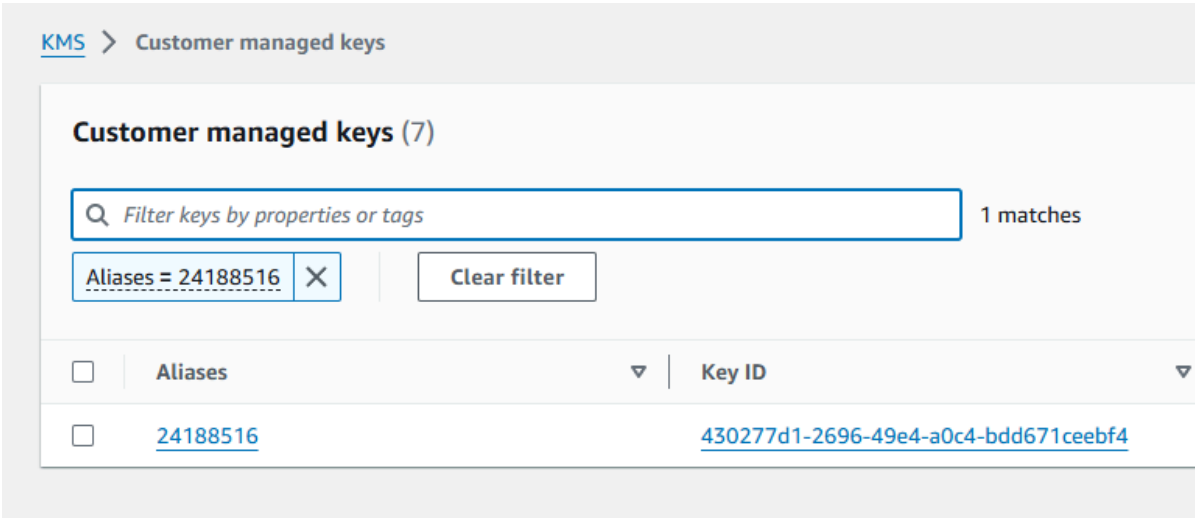
```

### 3. Check Whether the Script Works

To verify that the script has successfully created the KMS key and applied the policy, follow these steps:

#### 1. Check the KMS Key in the AWS Console

Navigate to the **KMS service** in the AWS console. In the list of keys, you should see the newly created key with the alias `alias/24188516`. This confirms that the KMS key and alias have been successfully generated.




#### 2. Verify the Policy

In the **Policy** section of the KMS key, you should see that the user `24188516@student.uwa.edu.au` has been assigned the roles of **Key Administrator** and **Key User**. This confirms that the policy from the `kmspolicy.json` file has been correctly applied to the key, granting the appropriate permissions to the IAM user.

430277d1-2696-49e4-a0c4-bdd671ceebf4

**General configuration**

Alias	Status
24188516	Enabled
ARN	Description
 arn:aws:kms:eu-north-1:489389878001:key/430277d1-2696-49e4-a0c4-bdd671ceebf4	-

[Key policy](#) | [Cryptographic configuration](#) | [Tags](#) | [Key rotation](#) | [Aliases](#)**Key policy**

```
"AWS": "arn:aws:iam::489389878001:root",
},
"Action": "kms:*",
"Resource": "*"
},
{
  "Sid": "Allow access for Key Administrators",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::489389878001:user/24188516@student.uwa.edu.au"
  },
  "Action": [
    "kms:Create*",
    "kms:Describe*",
    "kms:Enable*",
    "kms:List*",
    "kms:Put*",
```

## 4. Use the Created KMS Key for Encryption/Decryption

The following script, `cryptwithkms.py`, encrypts and decrypts files in the S3 bucket `24188516-cloudstorage` using the KMS key we created earlier (`alias/24188516`).

### Workflow

#### 1. Traverse all Files in the S3 Bucket:

The script first calls `process_files()` to list all files in the specified S3 bucket:

- Lists all files in the specified S3 bucket.
- Iterates through each file, calling `encrypt_file()` for encryption and subsequent decryption.

#### 2. Encrypt original Files:

For each file, `encrypt_file()` function retrieves the file content from S3, encrypts it using the specified KMS key, and uploads the encrypted file back to the bucket with a new key that appends `.encrypted` to the original file name:

- Retrieves the file from the S3 bucket using `s3.get_object()`.
- Encrypts the file content using the KMS key with `kms.encrypt()`.
- Uploads the encrypted content back to the bucket with a new key that appends `.encrypted` to the original file name.

- Calls `decrypt_file()` to decrypt the encrypted file.

### 3. Decrypt encrypted Files:

`decrypt_file()` function decrypts the file content and uploads the decrypted file back to the bucket with a new key that appends `.decrypted` to the encrypted file name:

- Retrieves the encrypted file from the bucket using `s3.get_object()`.
- Decrypts the file content using the KMS key with `kms.decrypt()`.
- Converts the decrypted content from bytes to a regular string using `.decode('utf-8')`.
- Uploads the decrypted content back to the bucket with a new key that appends `.decrypted` to the encrypted file name.

Here's the Python script:

```
# cryptwithkms.py
import boto3

s3 = boto3.client('s3')
kms = boto3.client('kms')

BUCKET_NAME = "24188516-cloudstorage"
KMS_KEY = "alias/24188516"

def encrypt_file(file_key):
    # Get the file from bucket and read its content
    s3_object = s3.get_object(Bucket=BUCKET_NAME, Key=file_key)
    file_content = s3_object['Body'].read()

    # Encrypt the file content using KMS
    encrypt_res = kms.encrypt(
        KeyId=KMS_KEY,
        Plaintext=file_content
    )
    file_body = encrypt_res['CiphertextBlob']
    encrypt_file_key = f"{file_key}.encrypted"

    # Upload the encrypted file back to the bucket
    s3.put_object(Bucket=BUCKET_NAME, Key=encrypt_file_key, Body=file_body)
    print(f"File encrypted as: {encrypt_file_key} with content: \n{file_body}\n")

    # After encrypting, decrypt the file
    decrypt_file(encrypt_file_key)

def decrypt_file(file_key):
    # Get the encrypted file from the bucket and read its content
    s3_object = s3.get_object(Bucket=BUCKET_NAME, Key=file_key)
    file_content = s3_object['Body'].read()

    # Decrypt the file content using KMS
    decrypt_res = kms.decrypt(
        KeyId=KMS_KEY,
        CiphertextBlob=file_content
    )
    plain_text = decrypt_res['Plaintext']
```



```

file_body = plain_text.decode('utf-8') # Convert plain text bytes to a
regular string
decrypted_file_key = f"{file_key}.decrypted"

# Upload the decrypted content back to the bucket
s3.put_object(Bucket=BUCKET_NAME, Key=decrypted_file_key, Body=file_body)
print(f"File decrypted as: {decrypted_file_key} with content:
\n{file_body}\n")

def process_files(BUCKET_NAME, KMS_KEY):
    # List all files in the bucket
    response = s3.list_objects_v2(Bucket=BUCKET_NAME)

    if 'Contents' in response:
        for obj in response['Contents']:
            key = obj['Key']
            encrypt_file(key)

if __name__ == "__main__":
    process_files(BUCKET_NAME, KMS_KEY)

```

## Code Explanation

- `s3.get_object()`: Retrieves the specified file from the S3 bucket.
  - `Bucket`: The name of the S3 bucket ( `24188516-cloudstorage` ).
  - `Key`: The key (file name) of the file to retrieve.
- `kms.encrypt()`: Encrypts the file content using the KMS key.
  - `KeyId`: Specifies the KMS key to use for encryption, here `alias/24188516`.
  - `Plaintext`: The file content to be encrypted.
- `s3.put_object()`: Uploads the encrypted or decrypted file back to the S3 bucket.
  - `Bucket`: The name of the S3 bucket ( `24188516-cloudstorage` ).
  - `Key`: The key (file name) for the uploaded file.
  - `Body`: The content of the file being uploaded.
- `kms.decrypt()`: Decrypts the encrypted file content using the KMS key.
  - `KeyId`: The KMS key to use for decryption, here `alias/24188516`.
  - `CiphertextBlob`: The encrypted content to be decrypted.

```

liudayubob@Dayu:~/cits5503/lab4$ python3 encryptanddecrypt.py
File encrypted as: rootdir/rootfile.txt.encrypted with content:
b"\x01\x02\x02\x00x\xf3A\x9d\x83\x83 \xc6#\xeaKL\xda\xb2\x19\xba\xf6J!\x97\xe4^S\x83\xfa\xfb\x8a\x86\xfe\xcd)\x81;\x01\xbb\xedN\x83A\x0c\x7f\xa8\x8dB\xfd\xb9R\x84\xf8s\x00\x00m0k\x06\t*\x86H\x86\xf7r\x01\x07\x06\xa0^0\\\x02\x01\x000W\x06\t*\x86H\x86\xf7r\x01\x07\x010\x1e\x06\t'\x86H\x01e\x03\x04\x01.0\x11\x04\x0c\x89$N5/\xe7a#\n\xe307\x02\x01\x10\x80*\xcb\x03\x05\xe8\xb6\x97\x8a\xc7\x8e~|\x94\xb1\xfa!\xc0\x87\xc0\xc1\xdc\xf5\x86\r\x15\xfe2\x81\xcc'\xc3M\xc7\x97\xdb?>\xc6\xedX\xe3\x99\xc8"
File encrypted as: rootdir/rootfile.txt.encrypted.decrypted with content:
1\n2\n3\n4\n5\n
File encrypted as: rootdir/subdir/subfile.txt.encrypted with content:
b"\x01\x02\x02\x00x\xf3A\x9d\x83\x83 \xc6#\xeaKL\xda\xb2\x19\xba\xf6J!\x97\xe4^S\x83\xfa\xfb\x8a\x86\xfe\xcd)\x81;\x01\x1d\xf9FCkP\t\xe28S\x8fo1\xb1\x7f\xb1\x00\x00m0k\x06\t*\x86H\x86\xf7r\x01\x07\x06\xa0^0\\\x02\x01\x000W\x06\t*\x86H\x86\xf7r\x01\x07\x010\x1e\x06\t'\x86H\x01e\x03\x04\x01.0\x11\x04\x0c\x14o\xc7bFJ\t\x84\x97\x11h\x02\x01\x10\x80*\xcb\x03\x05\xe8\xb6\x97\x8a\xc7\x8e~|\x94\xb1\xfa!\xc0\x87\xc0\xc1\xdc\xf5\x86\r\x15\xfe2\x81\xcc'\xc3M\xc7\x97\xdb?>\xc6\xedX\xe3\x99\xc8"
File encrypted as: rootdir/subdir/subfile.txt.encrypted.decrypted with content:
1\n2\n3\n4\n5\n

```

## Verify results in the AWS S3 Console

After running the script, you can verify the encrypted and decrypted files in the AWS S3 console. The original files will have additional encrypted and decrypted versions as shown below.

Objects (3) <a href="#">Info</a>		
Objects are the fundamental entities stored in Amazon S3. You can use <a href="#">Amazon S3 inventory</a> to get a list of all		
<input type="text" value="Find objects by prefix"/>		
<input type="checkbox"/>	Name	Type
<input type="checkbox"/>	<a href="#">subfile.txt</a>	txt
<input type="checkbox"/>	<a href="#">subfile.txt.encrypted</a>	encrypted
<input type="checkbox"/>	<a href="#">subfile.txt.encrypted.decrypted</a>	decrypted

<input type="checkbox"/>	Name	Type
<input type="checkbox"/>	<a href="#">rootfile.txt</a>	txt
<input type="checkbox"/>	<a href="#">rootfile.txt.encrypted</a>	encrypted
<input type="checkbox"/>	<a href="#">rootfile.txt.encrypted.decrypted</a>	decrypted
<input type="checkbox"/>	<a href="#">subdir/</a>	Folder

## 5. Apply pycryptodome for Encryption/Decryption

Since AWS KMS uses AES with 256-bit encryption, we can apply the same encryption standard using the `pycryptodome` package for consistency. Here's how we implement AES encryption and decryption with `pycryptodome`.

### 1. Install pycryptodome

First, install the `pycryptodome` package by running the following command:

```
pip install pycryptodome
```

This package provides AES encryption functionality similar to what AWS KMS offers.

```
liudayubob@Dayu:~/cits5503/lab4$ pip install pycryptodome
Defaulting to user installation because normal site-packages is not writeable
Collecting pycryptodome
  Downloading pycryptodome-3.20.0-cp35-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)
    2.1/2.1 MB 11.8 MB/s eta 0:00:00
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.20.0
```

## 2. Modify the Code in `cryptwithpycryptodome.py`

The code is similar to the `cryptwithkms.py` script from the previous step, but now we use `pycryptodome` for encryption and decryption.

### Workflow

#### 1. Import AES and Random Byte Generation:

We import `AES` from `pycryptodome` for encryption/decryption and `get_random_bytes` for random key generation. The **AES\_KEY** is **32 bytes** (256 bits) long, aligning with the AWS KMS approach.

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

AES_KEY = get_random_bytes(32) # 32 bytes = 256 bits-long key
```

#### 2. Encryption Process:

- We initialize an AES cipher object in EAX mode with the generated `AES_KEY`:  
`AES.new(AES_KEY, AES.MODE_EAX)`.
- The file content is encrypted using `cipher.encrypt_and_digest()`, which generates the ciphertext and an authentication tag for integrity verification.
- We concatenate the **nonce**, **tag**, and **ciphertext** in that order to create the encrypted file content. The nonce is used to ensure unique ciphertexts for the same plaintext, preventing issues like hash collisions.

```
# Encrypt the file content using AES with PyCryptodome in EAX mode
cipher = AES.new(AES_KEY, AES.MODE_EAX)
cipher_text, tag = cipher.encrypt_and_digest(file_content) # Encrypt and
generate tag
encrypt_file_key = f"{file_key}.encrypted"

# Concatenate the nonce, tag, and the ciphertext
file_body = cipher.nonce + tag + cipher_text
```

#### 3. Decryption Process:

- We extract the **nonce**, **tag**, and **ciphertext** from the concatenated file content (`file_body`). The nonce is the first 16 bytes, the tag is the next 16 bytes, and the remaining content is the ciphertext.
- Using the extracted nonce, we create a new AES cipher object to decrypt the file and verify its integrity with the tag.

```

# Parse the nonce, tag, and the ciphertext from the file content
nonce = file_body[:16] # First 16 bytes for the nonce
tag = file_body[16:32] # Next 16 bytes for the tag
cipher_text = file_body[32:] # The remaining bytes are the ciphertext

# Decrypt the file content using AES with PyCryptodome in EAX mode
cipher = AES.new(AES_KEY, AES.MODE_EAX, nonce=nonce)
plain_text = cipher.decrypt_and_verify(cipher_text, tag)
file_body = plain_text.decode('utf-8') # Convert decrypted content to a string

```

Here's the full script:

```

# cryptwithpycryptodome.py
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import boto3

s3 = boto3.client('s3')

BUCKET_NAME = "24188516-cloudstorage"
AES_KEY = get_random_bytes(32) # 256-bit key

def encrypt_file(file_key):
    # Get the file from the bucket and read content
    s3_object = s3.get_object(Bucket=BUCKET_NAME, Key=file_key)
    file_content = s3_object['Body'].read()

    # Encrypt the file content using AES with PyCryptodome in EAX mode
    cipher = AES.new(AES_KEY, AES.MODE_EAX)
    cipher_text, tag = cipher.encrypt_and_digest(file_content)
    encrypt_file_key = f"{file_key}.encrypted"

    # Concatenate the nonce, tag, and ciphertext
    file_body = cipher.nonce + tag + cipher_text

    # Upload the encrypted file back to the bucket
    s3.put_object(Bucket=BUCKET_NAME, Key=encrypt_file_key, Body=file_body)
    print(f"File encrypted as: {encrypt_file_key} with content: \n{file_body}\n")

    # Decrypt the file after encryption
    decrypt_file(encrypt_file_key)

def decrypt_file(file_key):
    # Get the encrypted file from the bucket and read content
    s3_object = s3.get_object(Bucket=BUCKET_NAME, Key=file_key)
    file_body = s3_object['Body'].read()

    # Parse the nonce, tag, and ciphertext from the file content
    nonce = file_body[:16] # First 16 bytes for the nonce
    tag = file_body[16:32] # Next 16 bytes for the tag
    cipher_text = file_body[32:] # The rest of the file content is the
    ciphertext

    # Decrypt the file content using AES with PyCryptodome in EAX mode
    cipher = AES.new(AES_KEY, AES.MODE_EAX, nonce=nonce)

```

```

plain_text = cipher.decrypt_and_verify(cipher_text, tag)
file_body = plain_text.decode('utf-8') # Convert plain text bytes to a
regular string
decrypted_file_key = f"{file_key}.decrypted"

# Upload the decrypted content back to the bucket
s3.put_object(Bucket=BUCKET_NAME, Key=decrypted_file_key, Body=file_body)
print(f"File decrypted as: {decrypted_file_key} with content:
\n{file_body}\n")

def process_files(BUCKET_NAME):
    # List all files in the bucket
    response = s3.list_objects_v2(Bucket=BUCKET_NAME)

    if 'Contents' in response:
        for obj in response['Contents']:
            key = obj['key']
            encrypt_file(key)

if __name__ == "__main__":
    process_files(BUCKET_NAME)

```

## Code Explanation

1. `get_random_bytes()`: This function generates a secure random byte sequence to use as the AES encryption key. In this case, we generate 32 bytes (256 bits) to match the AWS KMS key length.
2. `AES.new(AES_KEY, AES.MODE_EAX)`: Initializes a new AES cipher object in EAX mode using the generated AES key. EAX mode provides both encryption and authentication, ensuring data integrity during decryption.
3. `cipher.encrypt_and_digest(file_content)`: Encrypts the provided file content and generates a cryptographic tag to verify the integrity of the encrypted data during decryption.
4. `cipher.decrypt_and_verify(cipher_text, tag)`: Decrypts the ciphertext using the AES key and verifies the integrity of the decrypted data with the provided tag.

## 3. See It in Action

Now, let's run the script using:

```
python3 cryptwithpycryptodome.py
```

The encrypted content will differ from the previous method since a different encryption key is used.

```

liudayubob@Dayu:~/cits5503/lab4$ python3 cryptwithpycryptodome.py
Generated AES_KEY: b'I\x14\xcfL5\xd20(00\xa0\xe1'lf3\n\x89H\x93\x9f\x16\xb9\x92\xe7\xcfW\x84\x05\xa7\x0bA"

File encrypted as: rootdir/rootfile.txt.encrypted with content:
b'\xe1m\x0f\xc3\xde\xd1\xd1\x0bJ\r\xdf\xac\x1ds\xfb\x81\xfe\x076\xe6\xd1\x070\xd5U\xb0\x9e\xbf\xb0z\xeezy\x1ff\x45t\xb8\xde\xedE\xb5\xa1L3\xb0'

File decrypted as: rootdir/rootfile.txt.encrypted.decrypted with content:
1\n2\n3\n4\n5\n

File encrypted as: rootdir/subdir/subfile.txt.encrypted with content:
b"\xbeg\xe9\xa1\x9cr\xa6\xe7\x0c\xfb\xfb\x00h\xd6\xa3\xf6\xb9\x9fk6,{\xe1\x9f\x85?\x87\x1f\xfc\xdc\xb7V\x17E'a\xec;j\x81\x06\x871\x88\xad\x97\t"

File decrypted as: rootdir/subdir/subfile.txt.encrypted.decrypted with content:
1\n2\n3\n4\n5\n

```

You can verify the encrypted and decrypted files in the AWS S3 console:

**Objects (3)** [Info](#)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all

<input type="checkbox"/>	Name	Type
<input type="checkbox"/>	<a href="#">subfile.txt</a>	txt
<input type="checkbox"/>	<a href="#">subfile.txt.encrypted</a>	encrypted
<input type="checkbox"/>	<a href="#">subfile.txt.encrypted.decrypted</a>	decrypted

<input type="checkbox"/>	Name	Type
<input type="checkbox"/>	<a href="#">rootfile.txt</a>	txt
<input type="checkbox"/>	<a href="#">rootfile.txt.encrypted</a>	encrypted
<input type="checkbox"/>	<a href="#">rootfile.txt.encrypted.decrypted</a>	decrypted
<input type="checkbox"/>	<a href="#">subdir/</a>	Folder

## Answer the following question (Marked)

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

Answer:

**KMS** outperforms in its ease of maintenance and high scalability. It offers automated key management so we don't need to manually save our keys. KMS is also highly scalable because they are based on cloud infrastructure, which is critical under significant workload.

**PyCryptodome** is better for its extensibility and low internet overhead. PyCryptodome offers more room of customization with more cryptography algorithms and combinations with different configurations. It doesn't rely on API calls which are subject to connectivity and rate limits. However since encryption/decryption are done on local machine, it doesn't scale well with high workload.

# Lab 5

## Application Load Balancer

### 1. Create 2 EC2 Instances & Add Application Load Balancer

In this section, we will replicate some of the steps from **Lab 2** to create two EC2 instances, with a few changes to add an additional application load balancer. We will append the suffix `-lab5` to resource names like **security group** and **key pair** to differentiate them from the resources in **Lab 2**.

#### Workflow (on top of Lab 2):

- **Subnets and Availability Zones:** We will create the two EC2 instances in different **availability zones** by using `ec2.describe_subnets()` to fetch the subnets, and specifying the **SubnetId** parameter when launching the EC2 instances.
- **Load Balancer and Target Group:**
  - **Create Load Balancer:** Using `elbv2.create_load_balancer()` with the required subnets, security groups, and settings.
  - **Create Target Group:** Using `elbv2.create_target_group()` with the VPC ID, protocol, and port.
  - **Register Targets:** Register the EC2 instances to the load balancer target group.
  - **Create Listener:** Set up a listener to forward HTTP traffic from **port 80** to the **target group**.

This is the python script applied these changes:

```
import boto3 as bt
import os

GroupName = '24188516-sg-lab5'
KeyName = '24188516-key-lab5'
InstanceName1 = '24188516-vm1'
InstanceName2 = '24188516-vm2'
LoadBalancerName = '24188516-elb'
TargetGroupName = '24188516-tg'

# Initialize EC2 and ELBv2 clients
ec2 = bt.client('ec2', region_name='eu-north-1')
elbv2 = bt.client('elbv2')

# 1. Create security group
step1_response = ec2.create_security_group(
    Description="Security group for lab5 environment",
    GroupName=GroupName
)

# 2. Authorize SSH (port 22) and HTTP (port 80) inbound rules
step2_response = ec2.authorize_security_group_ingress(
    GroupName=GroupName,
```

```

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'}]
    }
]
)

# 3. Create key-pair
step3_response = ec2.create_key_pair(KeyName=KeyName)
PrivateKey = step3_response['KeyMaterial']
# Save key-pair
with open(f'{KeyName}.pem', 'w') as file:
    file.write(PrivateKey)
# Grant file permission
os.chmod(f'{KeyName}.pem', 0o400)

# 4. Get two subnets in different availability zones
step4_response = ec2.describe_subnets()['Subnets']
Subnets = [subnet['SubnetId'] for subnet in step4_response[:2]]

# 5. Create instances in two availability zones
Instances = []
for idx, SubnetId in enumerate(Subnets):
    InstanceName = f"24188516-vm{idx + 1}"
    step5_response = ec2.run_instances(
        ImageId='ami-07a0715df72e58928',
        SecurityGroupIds=[step1_response['GroupId']],
        MinCount=1,
        MaxCount=1,
        InstanceType='t3.micro',
        KeyName=KeyName,
        SubnetId=SubnetId
    )
    InstanceId = step5_response['Instances'][0]['InstanceId']
    Instances.append(InstanceId)

# Tag instance with name
ec2.create_tags(
    Resources=[InstanceId],
    Tags=[{'Key': 'Name', 'Value': InstanceName}]
)

# 6. Create application load balancer
step6_response = elbv2.create_load_balancer(
    Name=LoadBalancerName,
    Subnets=Subnets,
    SecurityGroups=[step1_response['GroupId']],

```



```

        Scheme='internet-facing',
        Type='application'
    )
    LoadBalancerArn = step6_response['LoadBalancers'][0]['LoadBalancerArn']

# 7. Create target group
VpcId = ec2.describe_vpcs()['Vpcs'][0]['VpcId']
step7_response = elbv2.create_target_group(
    Name=TargetGroupName,
    Protocol='HTTP',
    Port=80,
    VpcId=VpcId,
    TargetType='instance'
)
TargetGroupArn = step7_response['TargetGroups'][0]['TargetGroupArn']

# 8. Register instances as targets
elbv2.register_targets(
    TargetGroupArn=TargetGroupArn,
    Targets=[{'Id': InstanceId} for InstanceId in Instances]
)

# 9. Create a listener for the load balancer
elbv2.create_listener(
    LoadBalancerArn=LoadBalancerArn,
    Protocol='HTTP',
    Port=80,
    DefaultActions=[{
        'Type': 'forward',
        'TargetGroupArn': TargetGroupArn
    }]
)

# Print results
print(f"Instance IDs: {Instances}")
print(f"Load Balancer ARN: {LoadBalancerArn}")
print(f"Target Group ARN: {TargetGroupArn}")

```

## Code Explanation

1. `ec2.create_security_group()`: Creates a new security group.
  - **GroupName**: Specifies the name of the security group, which is `24188516-sg-1ab5`.
  - **Description**: A description for the security group.
2. `ec2.authorize_security_group_ingress()`: Authorizes inbound traffic.
  - **IpProtocol**: Specifies `tcp` for the protocol.
  - **FromPort/ToPort**: Specifies ports `22` and `80` for SSH and HTTP access.
  - **IpRanges**: Allows access from `0.0.0.0/0`, meaning any IP address.
3. `ec2.create_key_pair()`: Creates an SSH key pair for secure access.
  - **KeyName**: `24188516-key-1ab5`, specifies the name of the key pair.

4. `ec2.describe_subnets()`: Retrieves available subnets and selects two for launching instances in different availability zones.
5. `ec2.run_instances()`: Launches EC2 instances in separate subnets.
  - **ImageId**: Specifies `ami-07a0715df72e58928` as the AMI ID for the instances.
  - **SecurityGroupIds**: Associates the instances with the previously created security group (`24188516-sg-1ab5`).
  - **SubnetId**: Assigns instances to specific subnets to ensure they are in different availability zones.
  - **InstanceType**: Defines `t3.micro` as the instance type.
6. `elbv2.create_load_balancer()`: Creates an application load balancer.
  - **Name**: `24188516-elb`, defines the name of the load balancer.
  - **Subnets**: Assigns the load balancer to the subnets created earlier.
  - **SecurityGroups**: Associates the load balancer with the security group.
7. `elbv2.create_target_group()`: Creates a target group for the instances.
  - **Name**: `24188516-tg`, specifies the target group name.
  - **Protocol**: HTTP is specified as the protocol for communication.
  - **Port**: Uses port `80` for HTTP traffic.
  - **VpcId**: Associates the target group with the VPC where the instances are launched.
8. `elbv2.register_targets()`: Registers the EC2 instances as targets for the load balancer.
  - **TargetGroupArn**: Registers instances to the specified target group.
9. `elbv2.create_listener()`: Adds a listener to the load balancer.
  - **LoadBalancerArn**: Assigns the listener to the specified load balancer.
  - **Protocol**: HTTP is set as the protocol.
  - **Port**: Listens on port `80`.
  - **DefaultActions**: Forwards traffic to the target group.

## Verify in the AWS Console:

After the script is executed, you can verify the creation of the **load balancer** and **target group** in the AWS console.

EC2 > Load balancers

Load balancers (1)

Elastic Load Balancing scales your load balancer capacity automatically in response to changes in incoming traffic.

Filter load balancers

	Name	DNS name	State	VPC ID	Availability Zones	Type	Date created
<input type="checkbox"/>	24188516-elb	24188516-elb-82154599....	Active	vpc-078a6052bab379...	2 Availability Zones	application	September 9, 2024, 14:35 (UTC+08:00)

EC2 > Target groups

Target groups (1)

Filter target groups

	Name	ARN	Port	Protocol	Target type	Load balancer	VPC ID
<input type="checkbox"/>	24188516-tg	arn:aws:elasticloadbalanci...	80	HTTP	Instance	24188516-elb	vpc-078a6052bab379acd

## Record Public IP Addresses:

The public IPv4 addresses for both EC2 instances are recorded for verification.

Instances (2)

Find Instance by attribute or tag (case-sensitive)

All states

Last updated less than a minute ago

Connect

Instance state

Actions

Launch instances

	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP	IPv6 IPs	Monitoring	Security group name	Key name	Launch time
passsec	User: am:aws: eu-north-1c	eu-north-1c	ec2-16-16-172-60.eu-n...	16.16.172.60	-	-	disabled	24188516-sg-lab5	24188516-key...	2024/09/09 14:35 G
	User: am:aws: eu-north-1a	eu-north-1a	ec2-13-60-244-2.eu-no...	13.60.244.2	-	-	disabled	24188516-sg-lab5	24188516-key...	2024/09/09 14:35 G

## 2. SSH to Our Instances

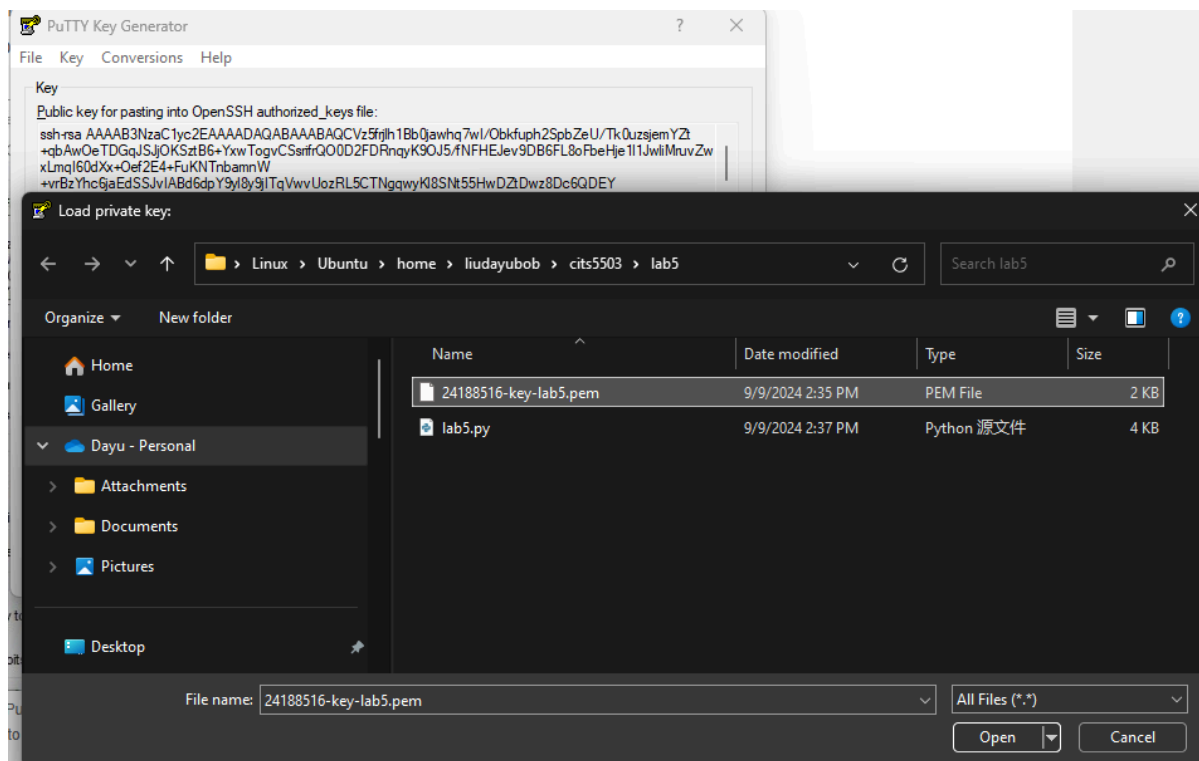
In this step, we will SSH into the EC2 instances to install Apache and start the web server, allowing us to see the load balancer in action.

Use Putty to Connect to EC2 Instances

Since we are using Windows and Putty as our SSH client, we need to convert the private key (`24188516-key-1ab5.pem`) to **PPK format** for Putty to use.

### 1. Convert PEM Key to PPK Format

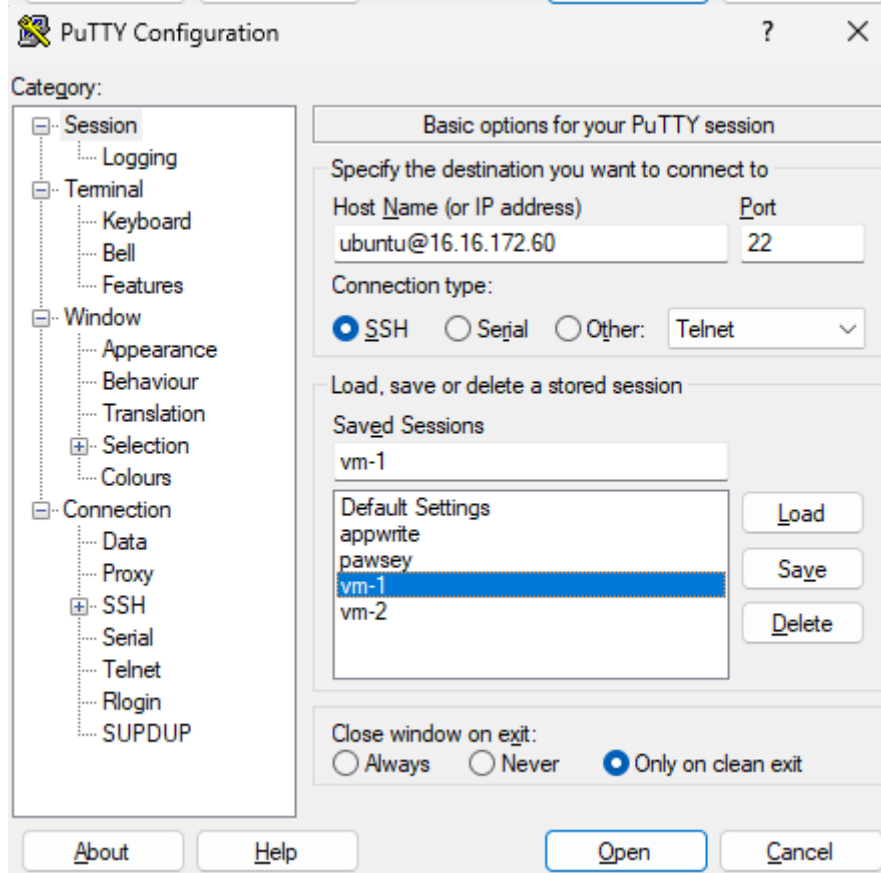
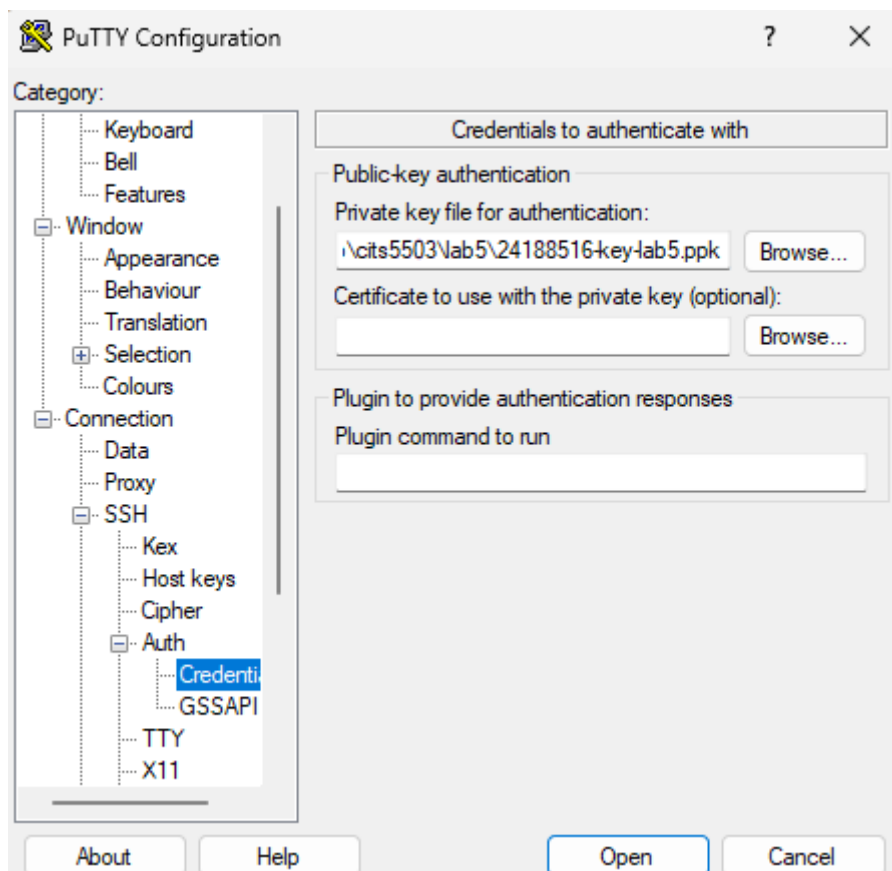
1. Open **PuttyGen** and load the `.pem` key file that was generated in step [3].
2. Convert the file into `.ppk` format by saving it after loading.



## 2. Configure Putty for SSH Access

Once the key is converted, we can configure Putty to use the correct authentication credentials and the public IP addresses of the two EC2 instances we recorded in the last step.

1. **Host:** Enters the public IP address of the EC2 instance we would like to connect to.
2. **Authentication:** Under "Connection -> SSH -> Auth", loads the `.ppk` file for the private key.



### 3. SSH into the EC2 Instance

Now, click "Open" to open a SSH connection. We are logged into the EC2 instance.

```
ubuntu@ip-172-31-8-174: ~
* Using username "ubuntu".
* Authenticating with public key "imported-openssh-key"
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 6.5.0-1022-aws x86_64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/pro

System information as of Mon Sep  9 06:48:00 UTC 2024

System load:  0.0           Processes:            100
Usage of /:   20.7% of 7.57GB Users logged in:          0
Memory usage: 22%          IPv4 address for ens5: 172.31.8.174
Swap usage:   0%

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Mon Sep  9 06:48:01 2024 from 130.95.40.98
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-8-174:~$
```

### 3. Install Apache & Access Results Using IP Addresses

In this step, we will install **Apache** on each EC2 instance, modify the HTML content, and verify the setup by accessing the instances via their public IP addresses.

#### 1. Update and Install Apache

On each EC2 instance, first update the package list and then install **Apache2** using the following commands:

```
sudo apt-get update
sudo apt install apache2
```

Once the installation is complete, Apache will start serving contents from the default directory `/var/www/html/`.

```
ubuntu@ip-172-31-8-174: ~  
Enabling module mime.  
Enabling module negotiation.  
Enabling module setenvif.  
Enabling module filter.  
Enabling module deflate.  
Enabling module status.  
Enabling module reqtimeout.  
Enabling conf charset.  
Enabling conf localized-error-pages.  
Enabling conf other-vhosts-access-log.  
Enabling conf security.  
Enabling conf serve-cgi-bin.  
Enabling site 000-default.  
Created symlink /etc/systemd/system/multi-user.target.wants/apache2.service → /lib/systemd/system/apache2.service.  
Created symlink /etc/systemd/system/multi-user.target.wants/apache-htcacheclean.service → /lib/systemd/system/apache-htcacheclean.service.  
Processing triggers for ufw (0.36.1-4ubuntu0.1) ...  
Processing triggers for man-db (2.10.2-1) ...  
Processing triggers for libc-bin (2.35-0ubuntu3.8) ...  
Scanning processes...  
Scanning linux images...  
  
Running kernel seems to be up-to-date.  
  
No services need to be restarted.  
  
No containers need to be restarted.  
  
No user sessions are running outdated binaries.  
  
No VM guests are running outdated hypervisor (qemu) binaries on this host.  
ubuntu@ip-172-31-8-174:~$
```

## 2. Modify the HTML File to Display Instance Name

To help us identify which EC2 instance is serving the content, we will edit the `<title>` tag of the default `index.html` file to include the instance name. Use the following command to edit the file:

```
sudo vi /var/www/html/index.html
```

Here's an example of the modified HTML file for **VM1**:

```
# index.html  
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />  
<title>Hello, this is VM1!</title>  
<style type="text/css" media="screen">
```

We now repeat this step for the second instance (VM2) and modify the `<title>` tag to **VM2** instead.

```
ubuntu@ip-172-31-27-118: ~  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
  <!--  
    Modified from the Debian original for Ubuntu  
    Last updated: 2022-03-22  
    See: https://launchpad.net/bugs/1966004  
  -->  
  <head>  
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />  
    <title>Hello, this is VM2!</title>  
    <style type="text/css" media="screen">  
    * {  
      margin: 0px 0px 0px 0px;  
      padding: 0px 0px 0px 0px;  
    }  
  
    body, html {  
      padding: 3px 3px 3px 3px;  
  
      background-color: #D8DBE2;  
  
      font-family: Ubuntu, Verdana, sans-serif;
```

## 4. Access the EC2 Instances via Public IP Addresses

Now that Apache is running and the HTML content has been updated, we can access each instance using its public IP address. Open your browser and visit the public IP addresses assigned to each instance.

- **VM1** will display the title: "Hello, this is VM1!"



- **VM2** will display the title: "Hello, this is VM2!"



