learn - VPC Setup Guide

Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
explain these in a very easy manner in brief:

Here's a super-simple, high-level rundown of what you did in each step:

1. **Create a VPC (your own little network)**
   - Think of a VPC as your private address space in AWS (like 10.0.0.0/16).
   - You "create" it so all your resources live inside it.
2. **Make a Public Subnet (a slice of that network you'll expose to the Internet)**
   - You carve out a smaller range (e.g. 10.0.0.0/24) in one AZ.
   - You turn on "auto-assign public IPv4" so any instance there can get a public IP.
3. **Attach an Internet Gateway (the door to the Internet)**
   - An IGW is what lets traffic flow between your VPC and the public Internet.
   - You create it, then "attach" it to your VPC.
4. **Create a Route Table & Hook It Up**
   - Route tables decide where traffic goes.
   - You add a rule that says "all outbound traffic (0.0.0.0/0) → Internet Gateway."
   - Then you associate that table with your public subnet so instances there can talk outside.
5. **Launch an EC2 Instance (your virtual server)**
   - Pick an AMI (e.g. Ubuntu), choose your VPC + public subnet.
   - Pick a small instance type (like t2.micro) and create/download a key-pair (.pem).
   - Assign a security group that allows SSH (port 22) and HTTP (port 80).
6. **SSH into Your New Server**
   - In the EC2 console click "Connect" → "SSH client."
   - Run `chmod 400 MyKey.pem` then

```bash
ssh -i MyKey.pem ubuntu@<your-public-IP>
```

- You're now logged into your Ubuntu box.

7. **Verify Internet Access**
   - From inside the server, do:

```bash
ping google.com
```

   - If you see replies, your instance really is on the Internet.

That's it—your own tiny cloud network with a public VM that can talk to the outside world!

explain all of these step by step in a very easy manner with basics in details with explanation of all the terms:
Launch An Ec2 Instance In A Virtual Private Cloud (vpc)
Overview
This guide will teach you how to set up a Virtual Private Cloud (VPC) from scratch and launch an EC2 instance within it. By the end of this guide, you will have created a VPC, set up a public subnet, configured routing with an Internet Gateway, and launched an EC2 instance.

Learning Objectives
Create a VPC with CIDR 10.0.0.0/16

Create a Public Subnet within the VPC

Create and Attach an Internet Gateway to the VPC

Configure Route Tables for Internet Access

Launch an EC2 Instance in the Subnet

Access the EC2 Instance

Step-by-Step Guide
Step 1: Create Your VPC
Open the AWS Management Console and search for "VPC" in the search bar.

On the left-hand side, click on "Your VPCs".

Click on "Create VPC" at the top right corner.

Name your VPC using a tag.

Set the IPv4 CIDR block to 10.0.0.0/16.

Click "Create VPC" to create your VPC.

Congratulations on creating your first VPC!

Step 2: Create a Public Subnet
After creating your VPC, click on "Subnets" on the left-hand side.

Click on "Create Subnet".

Designate the VPC you just created.

Assign a CIDR block within your VPC's range (e.g., 10.0.0.0/24).

Click "Create Subnet" to create your subnet.

Click on the created subnet and then "Edit subnet settings".

Enable "Auto-assign public IPv4 address" and save.

Step 3: Create and Attach an Internet Gateway
Click on "Internet Gateways" on the left-hand side.

Click "Create internet gateway".

Once created, click "Actions" and then "Attach to VPC".

Select your VPC and attach the Internet Gateway.

Step 4: Create Route Tables
Click on "Route Tables" on the left-hand side.

Click "Create route table".

Associate the new route table with your VPC.

Add a route to allow internet traffic by specifying the destination 0.0.0.0/0 and target as your Internet Gateway.

Click on the "Subnet Associations" tab, then "Edit Subnet Associations".

Select your public subnet and save.

Network Diagram
Below is a visual representation of the VPC setup with subnets in ap-southeast-1a Availability Zone (AZ).

Step 5: Launch an EC2 Instance
In the AWS console, search for "EC2" and click on "Launch Instance".

Give a name to your ec2 instance

Select the Amazon Machine Image (AMI) for Ubuntu image.

Choose the instance type, e.g., t2.micro.

Configure instance details, selecting the VPC and public subnet you created.

Create a key pair (.pem file) and save it securely.

Configure the security group to allow SSH (port 22) and HTTP (port 80) access.

Launch the instance and wait for it to start up.

Step 6: Access the EC2 Instance
Select your running instance in the EC2 console.

Click "Connect".

SSH into the Public Instance:

Open a terminal where you saved the key pair and run: ssh -i MyKeyPair.pem ubuntu@<public_instance_ip>

Replace <public_instance_ip> with the public IP address of the public instance.

We can see that we have connected to our ec2 instance.

Step 7: Verify that your EC2 instance has internet access
Ping a domain name, such as google.com, to test DNS resolution. ping google.com

Congratulations on successfully creating your first VPC and launching an EC2 instance!

Below is a gentle, detailed walk-through of each step—breaking out the jargon, explaining the "why," and showing you exactly what happens at every stage.

🏠 What's a VPC (Virtual Private Cloud)?

- **Imagine**: Your own little corner of AWS's network—like renting a private office suite in a massive building.
- **CIDR 10.0.0.0/16**: This is the address range for your office. 10.0.0.0 through 10.0.255.255—about 65,536 IP addresses you control.

Step 1: Create Your VPC

1. **AWS Console → VPC**

- In the top search box type **VPC**, click the VPC service.

2. **Your VPCs** → **Create VPC**
3. **Name tag**: “my-vpc” (or whatever you like).
4. **IPv4 CIDR block**: enter `` `10.0.0.0/16` ``.
    - **CIDR** (Classless Inter-Domain Routing) tells AWS how big your network is.
5. **Create** → You now have an empty private network.

Step 2: Carve Out a Public Subnet

- **Subnet** = a slice of your VPC’s IP range, isolated in a single Availability Zone (AZ).
- **Public** means any VM in it can get its own public IP and talk to the Internet.

1. **VPC** → **Subnets** → **Create Subnet**
2. **Select VPC**: choose `` `my-vpc` ``.
3. **AZ**: pick one (e.g. ap-southeast-1a).
4. **IPv4 CIDR**: enter `` `10.0.0.0/24` ``.
    - That gives you addresses 10.0.0.0–10.0.0.255 (256 IPs).
5. **Create Subnet**
6. **Edit Subnet Settings** → enable **Auto-assign public IPv4 address**
    - So any instance you launch here automatically gets a public IP.

Step 3: Build & Attach the Internet Gateway (IGW)

- **IGW** = the doorway between your VPC and the Internet.

1. **VPC** → **Internet Gateways** → **Create Internet Gateway**
2. Name it “my-igw.”
3. **Actions** → **Attach to VPC** → select `` `my-vpc` `` → **Attach**

Now your VPC has a door to the outside world—but no “directions” yet.

Step 4: Tell the Subnet How to Use That Door (Route Tables)

- **Route Table**: like a map that says “if you want to go to X, send traffic to Y.”
- Default route table only knows about local VPC traffic. We’ll add a route for “everything else.”

1. **VPC** → **Route Tables** → **Create Route Table**
2. **Name**: “my-public-rt”, **VPC**: `` `my-vpc` `` → **Create**
3. Select `` `my-public-rt` ``, go to the **Routes** tab → **Edit routes** → **Add route**:
    - **Destination**: `` `0.0.0.0/0` `` (means “all IPs on the Internet”)

- **Target**: your IGW (e.g. igw-xxxxxxxx)

4. **Save routes**
5. Switch to **Subnet Associations** → **Edit subnet associations** → tick your public subnet → **Save**

Now anything launched in that subnet sees “To reach anything on the Internet, go out the IGW.”

Step 5: Launch an EC2 Instance (Your VM)

- **EC2** = Elastic Compute Cloud, AWS’s name for virtual servers.

1. **AWS Console** → **EC2** → **Launch Instance**
2. **Name**: “my-web-server”
3. **AMI (Amazon Machine Image)**: pick **Ubuntu Server 24.04 LTS** (free tier-eligible).
4. **Instance type**: choose **t2.micro** (1 vCPU, 1 GB RAM – free tier).
5. **Configure Instance**:
   - **Network**: `` `my-vpc` ``
   - **Subnet**: your public subnet
   - **Auto-assign Public IP**: should be **Enable** (since you turned it on earlier)
6. **Key pair (SSH)**:
   - Create a new one called `` `my-keypair` `` → download the `` `.pem` `` file → keep it safe!
7. **Security Group** (firewall rules):
   - **Add Rule**: SSH, TCP, port 22, Source **0.0.0.0/0** (means “allow SSH from anywhere”)
   - **Add Rule**: HTTP, TCP, port 80, Source **0.0.0.0/0**
8. **Review & Launch** → **Launch Instances** → **View Instances**

AWS will spin up your VM in seconds.

Step 6: SSH into Your New Server

- **SSH** = Secure Shell, a way to remotely log in to Linux boxes.

1. In your terminal, go to where you saved `` `my-keypair.pem` ``.
2. Lock down permissions so SSH is happy:

```bash
chmod 400 my-keypair.pem
```

3. Log in:

```bash
ssh -i my-keypair.pem ubuntu@<PUBLIC_IP>
```

   - Replace `<PUBLIC_IP>` with the address you see in the EC2 console.
4. **Welcome!** You're now at a prompt on your Ubuntu server.

Step 7: Verify Internet Access

- Inside your SSH session, run:

```bash
ping -c 4 google.com
```

- If you see replies, your routing, IGW, and DNS all work!

Key Terms Recap

- **VPC**: Your private network in AWS.
- **CIDR Block**: Defines the size and range of IPs (e.g. 10.0.0.0/16).
- **Subnet**: A subset of your VPC's IP range, in one AZ.
- **Public Subnet**: Has a route out to an Internet Gateway and auto-assigns public IPs.
- **Internet Gateway (IGW)**: Connects your VPC to the Internet.
- **Route Table**: The map for sending packets—0.0.0.0/0 → IGW means "all Internet traffic goes out to the IGW."
- **EC2 Instance**: A virtual server.
- **AMI**: The "image" for that server (OS + pre-installed software).
- **Instance Type**: CPU/RAM sizing (t2.micro is tiny, free-tier).
- **Key Pair (.pem)**: Your SSH key—like a lock & key for secure login.
- **Security Group**: Virtual firewall around your instance; you opened ports 22 (SSH) and 80 (HTTP).
- **SSH**: Securely log in to your Linux box.

🎉 **You now have**:

- A private network (VPC)
- A public subnet inside it
- An Internet Gateway and proper routing
- A live Ubuntu server you can SSH into
- Verified that it can reach the wider Internet

That's the foundation for hosting websites, running apps, or anything else in AWS!

what is VM ; jargon, AZ, instances , SSH, Key pair (SSH):

Create a new one called my-keypair → download the .pem file → keep it safe!, Step 6: SSH into Your New Server
SSH = Secure Shell, a way to remotely log in to Linux boxes.

In your terminal, go to where you saved my-keypair.pem.

Lock down permissions so SSH is happy:

bash
Copy
Edit
chmod 400 my-keypair.pem
Log in:

bash
Copy
Edit
ssh -i my-keypair.pem ubuntu@<PUBLIC_IP>
Replace <PUBLIC_IP> with the address you see in the EC2 console.

Welcome! You're now at a prompt on your Ubuntu server.

Here's a quick glossary of those terms and a bit of context around the SSH steps you quoted:

VM (Virtual Machine)

- **What it is**: A software "computer" that runs inside a physical server.
- **Why we use it**: You get a fully-isolated OS (Ubuntu, Windows, etc.) without needing your own hardware.
- **In AWS**: An EC2 instance is really just a VM that AWS spins up for you.

Jargon

- **Definition**: Specialized words or acronyms used in a particular field.
- **Example**: "AZ" or "key pair" are cloud-computing jargon. If you're new, it can feel like a different language!

AZ (Availability Zone)

- **What it is**: One of multiple, physically separate data-centers within an AWS region.
- **Why it matters**: Spreading your VMs across AZs gives you resilience—if one AZ has a power blip, the others keep running.

Instance

- **In AWS-speak**: A single running VM—that is, one EC2 virtual server.
- **Why it matters**: You pay per instance, choose its size (CPU, RAM), and attach storage, networking, etc.

SSH (Secure Shell)

- **What it is**: A secure, encrypted way to log into a remote machine's command line.
- **Why we use it**: It's the de-facto standard for administering Linux servers from your laptop.

Key Pair (SSH)

- **What it is**: A matching pair of cryptographic keys—a **private key** (the `.pem` file you download) and a **public key** that AWS installs on your VM.
- **How it works**:
    1. You keep the **private key** (`my-keypair.pem`) secret on your computer.
    2. AWS puts the **public key** on the VM.
    3. When you SSH in, your client proves it has the private key—no passwords required.

Why `chmod 400 my-keypair.pem`?

- **SSH requirement**: Your private key file must be readable **only** by you.
- `chmod 400`: Sets permissions so that **only the file's owner** can read it; everyone else has no access.

The SSH Command

```bash
ssh -i my-keypair.pem ubuntu@<PUBLIC_IP>
```

- `-i my-keypair.pem`: "Use this file as my private key."
- `ubuntu@<PUBLIC_IP>`:
    - `ubuntu` is the default username on an Ubuntu AMI.
    - `<PUBLIC_IP>` is the public IP address AWS assigned to your VM.

Once you hit Enter (and pass the key check), you land in your server's shell prompt—congratulations, you're in!

Uploaded image
solve this:

That red box means AWS isn't accepting the block you've typed. Here are the three most common reasons—and how to fix it:

---

1. Check Your Formatting

Make absolutely sure there are **no extra spaces** or hidden characters. The field must be exactly:

```
10.0.0.0/16
```

– No leading/trailing spaces, tabs, or quotes.

---

2. Stay Within AWS's Size Limits

AWS requires your VPC CIDR mask to be between `` `/16` `` (largest) and `` `/28` `` (smallest).

- `` `/16` `` gives you 65,536 addresses (10.0.0.0–10.0.255.255)
- `` `/28` `` gives you 16 addresses

So `` `/16` `` is valid—just make sure you really have `` `/16` ``, not `` `/15` `` or `` `/17` ``.

---

3. Avoid Overlaps with Existing VPCs

You cannot create two VPCs whose CIDRs overlap.

- If you already have a VPC using `` `10.0.0.0/16` ``, AWS will reject a second one.
- **Solution**: pick a different non-overlapping range, for example:
  - `` `10.1.0.0/16` ``
  - or if you only need a small test VPC, `` `10.0.1.0/24` ``

Quick Fix Steps

1. **Delete any stray characters** in the CIDR box so it reads `` `10.0.0.0/16` `` exactly.
2. If it still errors, **check your other VPCs** under "Your VPCs" for an existing `` `10.0.0.0/16` `` block.
3. If you find one, either **delete** that old VPC (if you don't need it) or **choose a different block**, e.g. `` `10.1.0.0/16` ``.

Once you have a properly formatted, non-overlapping block, the red error will go away and you can click **Create VPC**.

Uploaded image
Uploaded image
how to do this:
Step 2: Carve Out a Public Subnet
Subnet = a slice of your VPC's IP range, isolated in a single Availability Zone (AZ).

Public means any VM in it can get its own public IP and talk to the Internet.

VPC → Subnets → Create Subnet

Select VPC: choose my-vpc.

AZ: pick one (e.g. ap-southeast-1a).

IPv4 CIDR: enter 10.0.0.0/24.

That gives you addresses 10.0.0.0–10.0.0.255 (256 IPs).

Create Subnet

Edit Subnet Settings → enable Auto-assign public IPv4 address

So any instance you launch here automatically gets a public IP.

Here's exactly how you carve out and configure your public subnet in the AWS console:

---

1. Open the Subnets page

1. Sign in to the AWS Management Console and make sure you're in the **Asia Pacific (Singapore)** region (or whichever region you created your VPC in).
2. In the left-hand sidebar under **Virtual private cloud**, click **Subnets**.

2. Create the Subnet

1. Click the **Create subnet** button at the top right.
2. In the **Create subnet** form:
    - **Name tag** (optional): enter something like `` `my-public-subnet` `` so you can recognize it later.
    - **VPC**: choose **my-vpc** (the VPC you made in Step 1).
    - **Availability Zone**: pick one AZ, e.g. **ap-southeast-1a**.
    - **IPv4 CIDR block**: type `` `10.0.0.0/24` ``

        This gives you IPs from 10.0.0.0 through 10.0.0.255 (256 addresses).
3. Click **Create subnet** at the bottom.

You'll see your new subnet listed (with Subnet ID like `` `subnet-0abc123def456ghi7` ``).

---

3. Turn It Into a Public Subnet

By default, instances get only private IPs. To auto-assign a public IP:

1. In the Subnets list, click your new subnet's **Subnet ID**.
2. On the subnet details page, click **Actions** → **Edit subnet settings**.
3. Under **Auto-assign IP settings**, check **Enable auto-assign public IPv4 address**.
4. Click **Save changes**.

Now any EC2 you launch into this subnet will automatically get a public IPv4 address—making it "public" (able to talk directly to the Internet, once you've hooked up your IGW and route table in Step 3 and 4).

Quick recap of what you've done:

- **Created a slice** (10.0.0.0/24) of your VPC's IP space.
- **Placed it** in one Availability Zone.
- **Enabled public IP auto-assignment**, so servers here can get Internet-reachable addresses automatically.

Next up: Attach your Internet Gateway and wire up your route table so that traffic actually flows in and out!

Uploaded image
Uploaded image
how to do this:

To see exactly that "Resource map" view—where you get a little diagram showing your VPC, its subnets, route tables, and Internet Gateway—just do the following:

---

1. Go to the VPC Console

1. Sign in to the AWS Management Console.
2. In the top search bar type **VPC** and click on **Virtual Private Cloud**.

2. Select Your VPC

1. In the left-hand menu click **Your VPCs**.
2. In the table you'll see each VPC you own (including the default one, e.g. 172.31.0.0/16, and your custom `` `10.0.0.0/16` ``).
3. Click once on the row for **my-vpc** (so it's highlighted).

3. Open the Resource Map

1. Below the table you'll see a set of tabs: **Details**, **Resource map**, **CIDRs**, etc.
2. Click the **Resource map** tab.

That instantly draws the diagram:

- **VPC** box showing 10.0.0.0/16
- Your **public subnet** (10.0.0.0/24) in ap-southeast-1a
- The **route table** you associated
- The **Internet Gateway** it connects to

If you want to include multiple VPCs in one diagram, you can **check the box** next to each VPC in the Your VPCs list—AWS will overlay them in the same map.

Why Use the Resource Map?

- **Visual confirmation** of which subnets belong to which route table.
- **Quick check** that your IGW is attached and that the 0.0.0.0/0 route exists.
- Great for documentation or just sanity-checking your network layout before you launch any EC2 instances.

That's all! Once you select your VPC and click **Resource map**, AWS handles the rest and draws you the network diagram you saw in the guide.

Uploaded image
Uploaded image
is it ok?

Yes—that dialog is exactly what you want. A few tips:

1. **Key pair name**
   - Pick something memorable (e.g. `` `my-vpc-keypair` ``), no spaces at the start or end.
2. **Key pair type**
   - **RSA** is the classic choice and works everywhere.
   - **ED25519** is newer, a bit faster, and just as secure—but some older SSH clients don't support it.
   - If you don't have a specific need, RSA is perfectly fine.
3. **Private key file format**
   - `` `.pem` `` is what you use with the standard OpenSSH client on macOS/Linux/WSL/most Linux AMIs.
   - `` `.ppk` `` is only if you plan to use PuTTY on Windows.

   - Since you'll likely run `ssh -i my-keypair.pem ubuntu@<IP>`, choose **.pem**.
4. **After you click "Create key pair"**
   - Your browser will download something like `my-vpc-keypair.pem`.
   - Move it into a safe folder (e.g. `~/.ssh/`) and run:

     ```
     bash

     chmod 400 ~/.ssh/my-vpc-keypair.pem
     ```

   - That permission lock is required for SSH to accept the file.
5. **Never lose it**
   - If you lose the `.pem`, you cannot SSH into instances launched with that key—you'd have to recreate the instance (or add another key via the AWS console/User Data).

So yes—select **RSA** + **.pem**, click **Create key pair**, download the file, secure it with `chmod 400`, and you're all set!

Uploaded image
Uploaded image
Uploaded image
how to do this here :

When you reach the **Firewall (security groups)** section of the **Launch Instance** wizard, you have two choices:

1. **Create a new security group**
2. **Select an existing security group**

Since you're starting fresh, let's **create a new one** and add exactly two inbound rules: SSH (port 22) and HTTP (port 80).

A. Create & Configure a New Security Group

1. **Under "Firewall (security groups)"** make sure **Create security group** is selected.
2. You'll see a default name like `launch-wizard-1`—you can leave it or rename (e.g. `my-web-sg`).
3. **Inbound rules**: these are the "allowed-in" ports. By default it usually adds SSH for you; if not, you'll add it.

Type	Protocol	Port Range	Source	Description (optional)
SSH	TCP	22	Anywhere	e.g. "SSH from my laptop"
HTTP	TCP	80	Anywhere	e.g. "Web traffic"

   - **To add SSH** (if not already present):
     - Click **Add rule**.

        - For **Type**, choose **SSH** (pre-fills TCP/22).
        - For **Source**, choose **Anywhere** (0.0.0.0/0).
    - **To add HTTP**:
        - Click **Add rule**.
        - For **Type**, choose **HTTP** (pre-fills TCP/80).
        - For **Source**, choose **Anywhere** (0.0.0.0/0).
4. **Review the warning**: AWS reminds you that "0.0.0.0/0 allows all IPs"—in production it's safer to lock SSH to your office or home IP. But for a quick demo, Anywhere is fine.
5. Click **Review and launch** (or **Next** depending on your console version) to continue.

B. (Alternative) Attach an Existing Security Group

If you already have a security group with the correct rules:

1. Select **Select existing security group**.
2. In the dropdown, choose your pre-configured SG that has SSH and HTTP inbound rules.
3. Continue to **Review and launch**.

Why These Rules?

- **SSH (TCP/22)** lets you remotely "ssh" into your VM's console.
- **HTTP (TCP/80)** lets web browsers reach any web service you run on port 80.

Once you launch, your instance will automatically be protected by this firewall. Only traffic on those two ports will get through. All other ports (e.g. 3306 for MySQL, or 22/80 on UDP) remain closed by default.

Uploaded image
is this step is ok?

Yep—that's exactly what you should see the **first** time you SSH into a brand-new EC2 host. That message is just OpenSSH warning you haven't seen this server's **host key** before, so it can't yet vouch for its identity. Here's what to do:

1. **Type** `yes` and hit Enter.
    - This tells SSH "okay, I trust this host"; it will then store the server's public host key in your `~/.ssh/known_hosts` file.
2. **You should then be dropped straight into your Ubuntu shell** (no further prompts), assuming your `.pem` file has the correct permissions (`chmod 400 my-key.pem`) and you used the right username/IP.

After that, future SSH connections to the same public IP will skip that fingerprint prompt entirely.

```
PS C:\Users\ASUS\See-SSh> ssh -i "my-key.pem" ubuntu@ec2-13-229-223-173.ap-southeast-1.compute.amazonaws.com
The authenticity of host 'ec2-13-229-223-173.ap-southeast-1.compute.amazonaws.com (13.229.223.173)' can't be established.
ED25519 key fingerprint is SHA256:9BU1HUSkJWbnpSI8FDQjVPmhH9tuM3tPwjnAPhRSS4c.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-13-229-223-173.ap-southeast-1.compute.amazonaws.com' (ED25519) to the list of known hosts.
Connection closed by 13.229.223.173 port 22
PS C:\Users\ASUS\See-SSh> ping -c 4 google.com
>>
Access denied. Option -c requires administrative privileges.
PS C:\Users\ASUS\See-SSh> chmod 400 my-key.pem
PS C:\Users\ASUS\See-SSh> ssh -i "my-key.pem" ubuntu@ec2-13-229-223-173.ap-southeast-1.compute.amazonaws.com
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-1024-aws x86_64)

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

 System information as of Mon May 12 19:36:48 UTC 2025

  System load:  0.0               Processes:             105
  Usage of /:   25.0% of 6.71GB   Users logged in:       0
  Memory usage: 20%               IPv4 address for enX0: 172.31.22.225
  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.

ubuntu@ip-172-31-22-225:~$ ping -c 4 google.com
PING google.com (142.251.12.113) 56(84) bytes of data.
64 bytes from se-in-f113.1e100.net (142.251.12.113): icmp_seq=1 ttl=104 time=1.50 ms
64 bytes from se-in-f113.1e100.net (142.251.12.113): icmp_seq=2 ttl=104 time=1.65 ms
64 bytes from se-in-f113.1e100.net (142.251.12.113): icmp_seq=3 ttl=104 time=1.53 ms
64 bytes from se-in-f113.1e100.net (142.251.12.113): icmp_seq=4 ttl=104 time=1.54 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.499/1.552/1.645/0.055 ms
ubuntu@ip-172-31-22-225:~$
```

Everything here looks good—you've successfully SSH'd into your EC2 and verified outbound Internet access. A couple of small notes:

---

1. Windows vs. Linux `ping`

- **Windows PowerShell / CMD** uses `-n` instead of `-c`.

  ```powershell
  ping google.com -n 4
  ```

- The `-c` (count) flag is a Unix/Linux convention, so on Windows it complains unless you elevate to an admin who's installed compatible tools.

---

2. Your SSH and Ping on the EC2 instance

```bash
ssh -i "my-key.pem" ubuntu@ec2-13-229-223-173.ap-southeast-1.compute.amazonaws.com
```

- You saw the host-key fingerprint prompt, answered `yes`, then got dropped into the Ubuntu shell. ✅
- After `chmod 400 my-key.pem` the key permissions were correct, so SSH logged you in without closing. ✅
- From inside your Ubuntu box you ran:

  ```bash
  ping -c 4 google.com
  ```

  and got replies—this confirms your VPC's subnet, IGW, route table, and security group are all correctly letting outbound ICMP through. ✅

Next Steps

Now that your network and SSH are working, you can:

1. **Install a web server** (e.g. Apache or Nginx):

```bash
sudo apt update
sudo apt install nginx -y
```

2. **Test it** by pointing your browser at your EC2's **public IP**—you should see the Nginx welcome page.
3. **Lock down SSH** (optional) by editing the security group to allow only your home/office IP instead of `0.0.0.0/0`.

But as far as connectivity and SSH go—you're all set!

what does chmod 400 my-key.pem it mean?

why it is needed before ssh -i "my-key.pem" ubuntu@ec2-13-229-223-173.ap-southeast-1.compute.amazonaws.com

The command:

```bash
chmod 400 my-key.pem
```

does two things:

1. `chmod` is the Unix "change mode" command—it sets a file's permission bits.
2. `400` is an octal (base-8) code that means:
   - **4** for the owner's permissions → read only
   - **0** for the group's permissions → none
   - **0** for everyone else's permissions → none

So after `chmod 400 my-key.pem`:

User	Permissions
Owner	Read only
Group	No access
Others	No access

Why does SSH care?

OpenSSH **refuses** to use a private key file that's potentially world-readable—because if other users on the same machine could read your key, they could impersonate you. By locking it down to "readable only by you," you prove to SSH that it's safe to use.

When you run:

```bash
ssh -i my-key.pem ubuntu@your-ec2
```

SSH checks that `my-key.pem` is not accessible by anyone but you. If it is (e.g. `chmod 644` or looser), you'll see an error like:

```vbnet
Permissions 0644 for 'my-key.pem' are too open.
It is required that your private key files are NOT accessible by others.
```

Running `chmod 400 my-key.pem` fixes that by ensuring **only** you can read the file—and then SSH will happily use it to authenticate your login.

how to exit from these terminal

You can end a shell (SSH or local) session a couple of ways:

1. **Type** `exit` and press Enter

   ```bash
   ubuntu@ip-172-31-22-225:~$ exit
   logout
   Connection to 13.229.223.173 closed.
   ```

   That cleanly logs you out of the remote server (or closes your local shell).

2. **Press Ctrl +D**
   - This sends an "EOF" (end-of-file) to your shell, which also makes it exit immediately.

In VS Code's integrated terminal, either method works the same, or you can click the little trash-can ("Kill Terminal") icon in the terminal panel's title bar to force-close it.

Uploaded image
Uploaded image
Uploaded image
explain all of these step by step in a very easy manner with basics in details with explanation of all the terms:
Deploying a Bastion Server in a Public Subnet
VPC allows users to create a private network within AWS, isolating and protecting resources. Within

this VPC, a crucial security component is the bastion server. This server acts as a gateway, enabling secure access to resources in the private VPC environment from the internet. In this lab, we will deploy a bastion server in a public subnet.

What is a Bastion Server?
A bastion server is an EC2 instance that acts as a secure gateway for accessing private instances within an AWS VPC environment. Also known as a jump server, it provides a controlled access point from the internet into the private network.

A bastion server runs on an Ubuntu EC2 instance hosted in a public subnet of an AWS VPC. This instance is configured with a security group that allows SSH access from anywhere over the internet. The bastion server facilitates secure connections to instances within private subnets, ensuring that these instances remain inaccessible directly from the internet, thereby enhancing the overall security of the VPC.

Objectives
Create a VPC.

Create a Public Subnet.

Create Routes and an Internet Gateway.

Launch EC2 instances in the Public Subnets.

Add a Security Group.

SSH into the Public EC2 Instance.

1. Create a VPC
A Virtual Private Cloud (VPC) provides an isolated network environment in AWS, enabling you to launch AWS resources in a virtual network that you define.

Navigate to the VPC Console:

Log into the AWS Management Console.

In the Services menu, select "VPC" or Search VPC.

Create the VPC:

In the left-hand menu, select "Your VPCs".

Click on "Create VPC".

Set the following values:

VPC Name: my-vpc

IPv4 CIDR block: 10.0.0.0/16

IPv6 CIDR block: No IPv6 CIDR block

Tenancy: Default Tenancy

Click "Create VPC".

2. Create Subnets
Subnets allow you to partition your VPC's IP address range into smaller segments, enabling better management of resources within different availability zones.

Navigate to Subnets:

In the left-hand menu, click on "Subnets".

Click on "Create subnet".

Create the Public Subnet:

VPC ID: Select the VPC ID for my-vpc.

Subnet Name: public-subnet

Availability Zone: ap-southeast-1a

IPv4 CIDR block: 10.0.1.0/24

Click "Create Subnet".

3. Create an Internet Gateway
An Internet Gateway allows instances in the VPC to communicate with the internet. It serves as a target for the VPC route tables for internet-routable traffic.

Navigate to Internet Gateways:

In the left-hand menu, select "Internet Gateways".

Click on "Create internet gateway".

Create the Internet Gateway:

Name tag: my-IGW

Click "Create internet gateway".

Attach the Internet Gateway to the VPC:

Select the newly created internet gateway.

Click "Actions" > "Attach to VPC".

Select my-vpc.

Click "Attach internet gateway".

4. Create Route Tables
Route tables control the routing of network traffic within your VPC. Public route tables direct traffic to the Internet Gateway, while private route tables handle internal routing.

Navigate to Route Tables:

In the left-hand menu, select "Route Tables".

Click on "Create route table".

Create the Public Route Table:

Name: public-route-table

VPC: Select my-vpc.

Click "Create route table".

Add Route to the Internet Gateway:

Select the public-route-table.

In the "Routes" tab, click "Edit routes".

Click "Add route".

Destination: 0.0.0.0/0

Target: Select the internet gateway my-IGW.

Click "Save changes".

Associate Public Route Table with Public Subnet:

In the "Subnet associations" tab, click "Edit subnet associations".

Select public-subnet.

Click "Save associations".

5. Create Security Group
Security groups act as a virtual firewall for your EC2 instances to control inbound and outbound traffic. They are essential for defining and enforcing network access rules.

Navigate to Security Groups:

In the left-hand menu, select "Security Groups".

Click on "Create security group".

Create the Security Group:

Security group name: security-group-1

Description: Allow SSH and all outbound traffic

VPC: Select my-vpc

Configure Inbound Rules:

In "Inbound rules" section click on Add rule>

Type: SSH

Source: Anywhere (0.0.0.0/0)

Configure Outbound Rules:

In "Outbound rules" section click on Add rule>

Type: All traffic

Destination: Anywhere (0.0.0.0/0)

Verify Resources:

In the left-hand menu, select "VPC".

View the resources map of our vpc (my-vpc)

6. Launch EC2 Instances
EC2 instances are virtual servers in the cloud, providing scalable computing capacity. Launching instances in public and private subnets allows you to control access and network traffics.

Launch Public EC2 Instance:

Navigate to "EC2" in the Services menu.

Click on "Instances" > "Launch instances".

Name: ec2-instance-1

Application and OS Images (Amazon Machine Image): Select "Ubuntu Server 24.04 LTS".

Instance Type: Select t2.micro.

Key pair (login): Click "Create new key pair".

Key pair name: public-key-pair

Key pair type: RSA

Private key file format: .pem or .ppk

Click "Create key pair".

Note : Download and save the key in your desired directroy,we will need it later

Network settings: Click "Edit".

VPC: Select my-vpc.

Subnet: Select public-subnet.

Auto-assign Public IP: Enable.

Security group: Select security-group-1.

Click "Launch instance".

7. SSH into the Public EC2 Instance
SSH (Secure Shell) allows you to securely access your EC2 instances. We'll first connect to the public instance from your local machine.

Set Permissions on the Key Pair:

Open your terminal.

Navigate to the directory containing your public-key-pair.pem file.

Run the following command to set the correct permissions: chmod 400 "public-key-pair.pem"

Connect to the Public EC2 Instance:

Obtain the public IP address of ec2-instance-1 from the EC2 console.

Run the following command to connect to the instance (replace <ec2-instance-1-public-IP> with your instance's public IP address): ssh -i "public-key-pair.pem" ubuntu@<ec2-instance-1-public-IP>

By following these steps, we created a secure and efficient way to access private instances within an AWS VPC, improving the network's security. This project demonstrates how to use AWS VPC features to implement a bastion server, providing a strong solution for managing access to private resources in the cloud.

Here's a gentle, jargon-free walkthrough of setting up a bastion ("jump") server in AWS—every term explained and every click spelled out. Think of it as building your own locked-down office (the VPC), carving out a front-door lobby (the public subnet), hanging a doorman (the Internet Gateway and route), and placing a secure gatekeeper (the bastion server) so only invited guests can slip deeper inside.

🏗️ 1. Build Your Private Network (VPC)

What is a VPC?

- VPC = Virtual Private Cloud.
- **Imagine**: You've rented a private floor in a huge data-center building. All your servers live on this floor, and only you control who walks in.

How to create it:

1. In the AWS Console search bar, type **VPC** → click **Virtual Private Cloud**.
2. On the left, click **Your VPCs** → **Create VPC**.
3. Name: `my-vpc`
4. IPv4 CIDR block: `10.0.0.0/16`

- This defines your address range: 10.0.0.0–10.0.255.255 (65,536 IPs).

5. Leave IPv6 off, Tenancy default → **Create VPC**.

🌐 2. Carve Out the Front-Door Lobby (Public Subnet)

What is a Subnet?

- A subnet slices your VPC's big address range into smaller pools.
- **Public** means machines here can get Internet-routable addresses.

How to create it:

1. In the VPC console left menu → **Subnets** → **Create subnet**.
2. Choose `` `my-vpc` ``.
3. AZ (Availability Zone): pick one, e.g. **ap-southeast-1a** (a single data-center wing).
4. IPv4 CIDR block: `` `10.0.1.0/24` `` (256 IPs: 10.0.1.0–10.0.1.255).
5. Name tag: `` `public-subnet` `` → **Create subnet**.
6. Select your new subnet → **Actions** → **Edit auto-assign IP settings** → check **Enable auto-assign public IPv4 address** → **Save**.

   Now any VM you launch here automatically gets a public IP.

🚪 3. Hang the Internet Gateway (IGW)

What is an Internet Gateway?

- An IGW is literally the door between your VPC and the rest of the Internet.

How to create & attach:

1. VPC console → **Internet Gateways** → **Create internet gateway** → name `` `my-igw` `` → **Create**.
2. Select `` `my-igw` `` → **Actions** → **Attach to VPC** → choose `` `my-vpc` `` → **Attach internet gateway**.

🗺️ 4. Tell Your Subnet How to Use the Door (Route Table)

What is a Route Table?

- A map saying "to reach IPs in A, go via B."
- We'll add a rule that sends *all* Internet traffic (`` `0.0.0.0/0` ``) to our IGW.

How to set it up:

1. VPC console → **Route Tables** → **Create route table**.
   - Name: `` `public-route-table` ``

- VPC: `my-vpc` → **Create**.

2. Select `public-route-table` → **Routes** tab → **Edit routes** → **Add route**:
   - Destination: `0.0.0.0/0`
   - Target: select **my-igw** → **Save routes**.
3. **Subnet associations** tab → **Edit subnet associations** → check `public-subnet` → **Save**.

   Now anything in `public-subnet` will send outgoing traffic out the IGW.

🔐 5. Create Your Firewall (Security Group)

What is a Security Group?

- A virtual firewall attached to each EC2. You define which ports are open.

How to create one for the bastion:

1. VPC console → **Security Groups** → **Create security group**.
   - Name: `bastion-sg`
   - VPC: `my-vpc`
2. **Inbound rules** (incoming traffic):
   - SSH → TCP port 22 → Source = **Anywhere (0.0.0.0/0)**
3. **Outbound rules** (outgoing traffic):
   - All traffic → Destination = **Anywhere (0.0.0.0/0)**
4. **Create security group**.

   This allows you (from anywhere) to SSH into the bastion, and lets the bastion connect out to any other host.

🚀 6. Launch the Bastion Server (EC2 Instance)

What is an EC2 Instance?

- A virtual server (VM) in AWS.

How to launch it:

1. AWS Console → search **EC2** → **Instances** → **Launch instances**.
2. Name tag: `bastion-server`
3. AMI: choose **Ubuntu Server 24.04 LTS**
4. Instance type: **t2.micro** (free-tier)
5. **Key pair**: click **Create new key pair** → name `bastion-key` → type RSA → format **.pem** → **Create key pair** → download `bastion-key.pem`.
6. **Network settings**:

  - VPC: `my-vpc`
  - Subnet: `public-subnet`
  - Auto-assign Public IP: **Enable**
  - Security group: select `bastion-sg`
7. **Launch instance**.

   You now have a Linux server sitting publicly in your subnet, reachable only on port 22.

🔑 7. SSH into Your Bastion

What is SSH?

- Secure Shell—a protected way to open a terminal on a remote server.

Why `chmod 400`?

- **chmod 400 bastion-key.pem** makes the file read-only for *you* and unreadable by anyone else. SSH refuses to use a key that could be world-readable.

How to connect:

1. On your local machine open a terminal.
2. Move `bastion-key.pem` into a safe folder (e.g. `~/.ssh/`).
3. Run:

```bash
chmod 400 ~/.ssh/bastion-key.pem
ssh -i ~/.ssh/bastion-key.pem ubuntu@<BASTION_PUBLIC_IP>
```

4. On first connect you'll see a fingerprint warning—type `yes` and press Enter.
5. You arrive at an `ubuntu@ip-...:~$` prompt.

   The bastion is now your secure gateway—any further "private" EC2 you launch in a private subnet can be reached *through* this server, without ever exposing them directly to the Internet.

🔄 What You've Achieved

1. **VPC**—your private network floor.
2. **Public Subnet**—your lobby with a public door.
3. **IGW + Route Table**—the unlocked door that sends Internet traffic in/out.
4. **Security Group**—firewall rules allowing only SSH to the bastion.
5. **Bastion Server**—the doorman who greets your SSH sessions.

With this setup, any backend servers you launch in **private** subnets never get a public IP. You simply SSH to the bastion first, then "hop" from the bastion into those private hosts—keeping them invisible to the outside world and giving you a single, well-secured entry point.

explain in details what is jargon, bastion ("jump") server, and what bastion ("jump") server differs from normal EC2 Launch here

in prev:
Step 5: Launch an EC2 Instance (Your VM)
EC2 = Elastic Compute Cloud, AWS's name for virtual servers.

AWS Console → EC2 → Launch Instance

Name: "my-web-server"

AMI (Amazon Machine Image): pick Ubuntu Server 24.04 LTS (free tier-eligible).

Instance type: choose t2.micro (1 vCPU, 1 GB RAM – free tier).

Configure Instance:

Network: my-vpc

Subnet: your public subnet

Auto-assign Public IP: should be Enable (since you turned it on earlier)

Key pair (SSH):

Create a new one called my-keypair → download the .pem file → keep it safe!

Security Group (firewall rules):

Add Rule: SSH, TCP, port 22, Source 0.0.0.0/0 (means "allow SSH from anywhere")

Add Rule: HTTP, TCP, port 80, Source 0.0.0.0/0

Review & Launch → Launch Instances → View Instances

AWS will spin up your VM in seconds.

and here :

🔐 5. Create Your Firewall (Security Group)
What is a Security Group?

A virtual firewall attached to each EC2. You define which ports are open.

How to create one for the bastion:

VPC console → Security Groups → Create security group.

Name: bastion-sg

VPC: my-vpc

Inbound rules (incoming traffic):

SSH → TCP port 22 → Source = Anywhere (0.0.0.0/0)

Outbound rules (outgoing traffic):

All traffic → Destination = Anywhere (0.0.0.0/0)

Create security group.

This allows you (from anywhere) to SSH into the bastion, and lets the bastion connect out to any other host.

🚀 6. Launch the Bastion Server (EC2 Instance)
What is an EC2 Instance?

A virtual server (VM) in AWS.

How to launch it:

AWS Console → search EC2 → Instances → Launch instances.

Name tag: bastion-server

AMI: choose Ubuntu Server 24.04 LTS

Instance type: t2.micro (free-tier)

Key pair: click Create new key pair → name bastion-key → type RSA → format .pem → Create key pair → download bastion-key.pem.

Network settings:

VPC: my-vpc

Subnet: public-subnet

Auto-assign Public IP: Enable

Security group: select bastion-sg

is there any significant difference

Let's unpack each term and then compare a "plain" EC2 launch to a bastion ("jump") server deployment so you see exactly what changes—and why.

---

1. Jargon

Jargon is any specialized term or acronym used in a field that newcomers might not immediately understand. In AWS networking you'll hear things like:

- **VPC** (Virtual Private Cloud)
- **CIDR** (Classless Inter-Domain Routing)
- **IGW** (Internet Gateway)
- **AZ** (Availability Zone)
- **EC2** (Elastic Compute Cloud)
- **SG** (Security Group)

Knowing the definitions turns "alphabet soup" into a clear map of how the pieces fit together.

---

2. What Is a Bastion (Jump) Server?

- **Definition**:
  A **bastion** (aka **jump**) server is a hardened EC2 instance **solely** exposed to the Internet. It sits in a **public subnet** and acts as the **only** "front door" you—and only you—use to get into your AWS network.
- **Primary purpose**:
  1. **Controlled access**: Instead of opening SSH on every server, you open SSH only on the bastion.
  2. **Audit point**: All admin sessions funnel through this one host, so you can log and monitor exactly who logged in, when, and from where.
  3. **Isolation**: Your true application servers live in **private subnets** with no public IPs—impossible to reach directly from the Internet.
- **Why "jump"?**
  Once you SSH into the bastion, you "jump" (via another SSH command or ProxyJump) from it into private-subnet servers. Without the bastion, those private servers wouldn't have any public address at all.

---

3. A "Normal" EC2 Launch vs. a Bastion Server Launch

Aspect	"Normal" EC2 Server	Bastion (Jump) Server
Subnet	Often a **public** subnet if you want the server to host websites/APIs.	Always a **public** subnet (so you can reach it).
Public IP	Enabled for direct Internet-facing services (web, API).	Enabled so administrators can SSH in.

Aspect	"Normal" EC2 Server	Bastion (Jump) Server
Security Group	Typically opens ports like 22 (SSH), 80/443 (HTTP/HTTPS), maybe 3306 (DB), etc.	**Only** opens port 22 for SSH (lockdown: you shouldn't serve web traffic here).
Outbound access	Often unrestricted (all ports) so your app can call external APIs, databases, etc.	Unrestricted too—so the bastion can reach private-subnet hosts on SSH (port 22).
Role & Hardening	General purpose: web server, application server, database server. You might install your app, web stack, runtime, etc.	Specialized: minimal OS install, only SSH daemon, strict patch management, MFA/2FA, key-based auth only—and nothing else.
Monitoring & Logs	Application-level logs + OS logs.	Extra OS-level auditing (who logged in, from which IP, session recording).

---

4. What Actually Changes in Steps 5 & 6

In a "Web Server" Launch (Step 5)

1. **Security Group**
   - Inbound:
     - SSH (22) → Anywhere
     - HTTP (80) → Anywhere
   - Outbound: All traffic
2. **AMI & Instance Type**
   - Ubuntu + t2.micro (same as bastion)
3. **Network**
   - Public subnet, auto-assign public IP
4. **Role**
   - You'll install Nginx/Apache, deploy code, open port 80 for website visitors.

In a Bastion Server Launch (Step 6)

1. **Security Group**
   - Inbound: **only** SSH (22) → Anywhere
   - Outbound: All traffic
2. **AMI & Instance Type**
   - Still Ubuntu + t2.micro (bastions can be small)
3. **Network**
   - Public subnet, auto-assign public IP
4. **Role**
   - You **do not** install web services. You lock down everything except SSH.
   - You may install monitoring agents, audit tools, and restrict SSH further (e.g. only from your office IP).

5. Why Use a Bastion?

1. **Security Best Practice**
   - Reduce attack surface: private servers have zero open inbound ports.
   - All SSH sessions go through one bastion that you can lock down extra tightly.
2. **Ease of Management**
   - Rotate bastion keys or credentials in one place.
   - Onboard/deprovision admin access centrally—no need to touch every backend server.
3. **Audit and Compliance**
   - Tailor logs on the bastion (session recordings, `last`, `auditd`) for forensic evidence.
   - You can enforce MFA or integrate with AWS SSM Session Manager for an even stronger security posture.

In Short

- **A "normal" EC2** is built for your application (web, database, batch job). It may open multiple ports and run your software stack.
- **A bastion EC2** is a tiny, hardened server whose **only** job is to accept SSH from you, then proxy or jump you into private parts of your network.

That single, tightly-guarded entry point is what makes your entire AWS environment more secure.

Uploaded image
explain the error:

That red warning is AWS telling you that your new subnet's address range overlaps one you already have in the same VPC. Let's break it down:

What "CIDR Address overlaps" Means

- **VPC CIDR**: You gave your VPC the big block `10.0.0.0/16` (which spans `10.0.0.0`–`10.0.255.255`).
- **Existing Subnet**: AWS already has a subnet using `10.0.0.0/20`, which covers `10.0.0.0`–`10.0.15.255`.
- **Your New Subnet**: You tried to carve out `10.0.1.0/24`, i.e. `10.0.1.0`–`10.0.1.255`.
- **Overlap**: Since `10.0.1.0/24` is entirely inside the range `10.0.0.0/20`, the two subnets would share addresses (e.g. `10.0.1.5` belongs to both). AWS forbids that.

Why Subnets Must Not Overlap

Within a single VPC, each subnet must own a distinct, non-overlapping slice of the IP space—otherwise the network can't correctly route traffic or enforce isolation.

How to Fix It

1. **Pick a non-overlapping CIDR**
   - If you already have a `/20` covering `10.0.0.0`–`10.0.15.255`, choose something outside that.
   - For example:
     - `10.0.16.0/24` (covers `10.0.16.0`–`10.0.16.255`)
     - or even `10.0.2.0/24` if your `/20` was narrower.
2. **Or delete the old subnet**
   - If the existing `10.0.0.0/20` is a mistake, you can delete it (from Subnets → select it → Actions → Delete subnet) and then create your intended `/24`.
3. **Then retry**
   - In the Create Subnet form, enter your new, non-overlapping block (e.g. `10.0.16.0/24`), give it a name tag, and hit **Create subnet**.

Once the block you type doesn't intersect any other subnet in that VPC, the red error will disappear and you'll be free to create it.

Uploaded image

The error is telling you that your new subnet `10.0.25.0/24` lives *inside* the already-existing `10.0.16.0/20` subnet, so the two overlap. Here's why that happens and how to fix it:

Why It Overlaps

- **Existing** `/20`:
  - A `10.0.16.0/20` covers addresses from **10.0.16.0** through **10.0.31.255**.
- **Your** `/24`:
  - A `10.0.25.0/24` covers **10.0.25.0** through **10.0.25.255**.
- **Overlap**:
  - Since `10.0.25.x` falls between `10.0.16.0` and `10.0.31.255`, the ranges collide, which AWS disallows.

How to Pick a Non-Overlapping CIDR

You need a `/24` block *outside* **10.0.16.0–10.0.31.255** yet still inside your VPC's **10.0.0.0/16**. Here are two safe examples:

- **10.0.32.0/24** → covers 10.0.32.0–10.0.32.255
- **10.0.2.0/24** → covers 10.0.2.0–10.0.2.255

Either one sits entirely outside the 10.0.16.0/20 range but within your VPC.

Steps to Fix

1. In the **IPv4 subnet CIDR block** field, replace `10.0.25.0/24` with, for example, `10.0.32.0/24`.
2. Make sure the little red error icon goes away.
3. Click **Create subnet**.

Now you'll have a clean, non-overlapping public subnet ready to host your bastion or web servers.

Uploaded image
what to plot here?

That Inbound section is where you "plot" (i.e. define) your firewall rules for the bastion's security group. Here's exactly what to fill in:

1. **Remove** that default "Custom TCP port 0" rule by clicking **Delete**.
2. **Click Add rule** and set:
   - **Type**: **SSH**
   - **Protocol**: **TCP** (auto-filled)
   - **Port range**: **22**
   - **Source**: **Anywhere** (0.0.0.0/0)
   - *(Optional)* Description: "Allow SSH from the Internet"
3. (If you also need HTTP on your bastion for some strange reason, you could add)
   - **Type**: **HTTP**
   - **Port range**: **80**
   - **Source**: **Anywhere** (0.0.0.0/0)
4. Under **Outbound rules**, you can leave the default of **All traffic** → **0.0.0.0/0** so your bastion can reach any private-subnet host.

Why SSH only?

- A bastion server's *only* public-facing service should be SSH.
- You never serve web pages or databases from it; it's purely your "jump box."

Once you've set those two inbound rules (SSH ± HTTP if needed) and left outbound wide open, click **Create security group** at the bottom.

Uploaded image

Security group: select bastion-sg
how??

When you're on the **Launch Instance** page, look under the **Network settings** section for **Firewall (security groups)**. By default it shows **Create security group**. To attach your pre-built `bastion-sg` instead, do this:

1. **Click the radio button** next to **Select existing security group**
2. A multi-select box will appear.
3. **Open that drop-down**, and you'll see a list of every SG in your VPC.
4. **Type "bastion-sg"** (or scroll) to find the one you created.
5. **Check the box** next to **bastion-sg** so it turns blue.
6. You'll see it now listed under **Selected security groups**.

Everything else stays the same—when you click **Launch instance**, AWS will assign that SG (and its SSH-only rule) to your new EC2 "bastion-server" instance.

Uploaded image

That error is simply because PowerShell (and your file's location) doesn't match the Linux-style path you gave to `chmod`. A couple easy fixes:

---

1. Run `chmod` on the actual file path

If your `.pem` is in `C:\Users\ASUS\See-SSH\`, from *that* folder do:

```powershell
# in PowerShell, if you're already in C:\Users\ASUS\See-SSH
chmod 400 .\bastion-key.pem
```

Or give the full path:

```powershell
chmod 400 C:\Users\ASUS\See-SSH\bastion-key.pem
```

---

2. Use WSL or Git Bash if you want "~/" shortcuts

If you've installed WSL (Ubuntu on Windows) or Git Bash, open *that* shell instead of PowerShell, then:

```bash
# in WSL or Git Bash, assuming you've copied the file into your Linux home
mv /mnt/c/Users/ASUS/See-SSH/bastion-key.pem ~/.ssh/
chmod 400 ~/.ssh/bastion-key.pem
```

Why it matters

SSH on Windows still checks your private key's permissions. You just need to run **`chmod 400`** on the real file path—don't use `~/` unless you're in a Linux-compatible shell and the file actually lives under your Linux home directory.

Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
explain all of these step by step in a very easy manner with basics in details with explanation of all the terms:

7. SSH into the Public EC2 Instance
SSH (Secure Shell) allows you to securely access your EC2 instances. We'll first connect to the public instance from your local machine.

Set Permissions on the Key Pair:

Open your terminal.

Navigate to the directory containing your public-key-pair.pem file.

Run the following command to set the correct permissions: chmod 400 "public-key-pair.pem"

Connect to the Public EC2 Instance:

Obtain the public IP address of ec2-instance-1 from the EC2 console.

Run the following command to connect to the instance (replace <ec2-instance-1-public-IP> with your instance's public IP address): ssh -i "public-key-pair.pem" ubuntu@<ec2-instance-1-public-IP>

8. SSH from the Public EC2 Instance to the Private EC2 Instance
To access the private EC2 instance, we'll SSH from the public instance using the private key of the

private instance.

Copy the content of public key from the directory containing private-key-pair.pem file :

Create the Private Key File on the Public Instance:

Once connected to ec2-instance-1, create a file for the private key with nano or vim (text editor) in ec2-instance-1: nano private-key-pair.pem Paste the contents of your private-key-pair.pem file.

Set Permissions on the Private Key File:

Run the following command to set the correct permissions: chmod 400 private-key-pair.pem

Connect to the Private EC2 Instance:

Obtain the private IP address of ec2-instance-2 from the EC2 console.

Run the following command to connect to the instance (replace <ec2-instance-2-public-IP> with your instance's private IP address): ssh -i "private-key-pair.pem" ubuntu@<ec2-instance-2-public-IP>

You have now successfully created a VPC, subnets, internet gateway, route tables, security group, launched EC2 instances, and established SSH connections to both the public and private instances.

Below is a line-by-line, plain-English guide to SSH'ing into your **public** EC2 ("bastion") machine, then **jumping** from there into your **private** EC2—every term and flag unpacked.

🔑 7. SSH into the Public EC2 Instance

What is SSH?

- **SSH** (Secure Shell) is the standard, encrypted way to open a terminal session on a remote Linux box.

Why `chmod 400`?

- Your **private key** file (`public-key-pair.pem`) must be **readable only by you**.
- `chmod 400 public-key-pair.pem` sets permissions to **owner-read only**, blocking group/other access so SSH will accept it.

Step-by-Step

1. **Open your local terminal**
   - On Windows you can use PowerShell, Command Prompt (with OpenSSH installed), Git Bash, or WSL.
   - On macOS/Linux just open Terminal.
2. **Change into the folder** where you saved the key:

```bash
cd /path/to/your/keys
```

3. **Lock down the key** so it's not world-readable:

```bash
chmod 400 public-key-pair.pem
```

   - **400** means "owner can read, nobody else can."

4. **Find the Public IP**
   - In the EC2 console, select **ec2-instance-1** and note its **Public IPv4 address** (e.g. `18.123.45.67`).

5. **SSH in** with your key and the default Ubuntu username:

```bash
ssh -i "public-key-pair.pem" ubuntu@18.123.45.67
```

   - `-i` tells SSH which identity (key) file to use.
   - `ubuntu@…` means "log in as user `ubuntu` on that host."
   - The first time you connect you'll get a "host key" fingerprint prompt—answer **yes**.

You'll end up at a prompt like:

```bash
ubuntu@ip-10-0-1-184:~$
```

—congratulations, you're on the public server!

🔗 8. SSH from the Public EC2 to the Private EC2

Why?

- The **private instance** sits in a subnet with **no public IP**. The only way in is **via** your bastion.

What you need

- The **private key** (`private-key-pair.pem`) that matches the key-pair you chose when launching **ec2-instance-2**.
- The **private IP** of **ec2-instance-2** (e.g. `10.0.2.15`).

Step-by-Step

1. **On your bastion** (public EC2), create a file for the private key:

```bash
nano ~/private-key-pair.pem
```

   - `nano` is a simple command-line text editor. You can also use `vim` or `vi` if you prefer.

2. **Paste** the entire contents of your **private-key-pair.pem** (that you downloaded locally) into the editor, then save and exit:
   - In **nano**: press `Ctrl+O` (to write out), Enter (confirm), then `Ctrl+X` (to exit).
3. **Lock down** that key the same way:

```bash
chmod 400 ~/private-key-pair.pem
```

4. **Find the Private IP**
   - In the EC2 console, select **ec2-instance-2** and note its **IPv4 Private IP** (e.g. `10.0.2.15`).
5. **SSH into the private instance**:

```bash
ssh -i ~/private-key-pair.pem ubuntu@10.0.2.15
```

   - No host-key prompt this time if you've already connected before; otherwise answer **yes** again.

Now your prompt changes to something like:

```bash
ubuntu@ip-10-0-2-15:~$
```

—you're inside your private server, reached securely via the bastion.

Quick Recap of Key Terms

- `.pem` **file**: Your **private SSH key**—never share this.
- `chmod 400`: Locks the key so only you can read it.
- `ssh -i key.pem user@IP`: Opens a secure shell using that key.
- **Bastion / jump server**: A publicly-reachable EC2 that you SSH into first.
- **Private instance**: No public IP; only accessible from inside your VPC (via bastion).

With these two hops—local → bastion → private—you've built a layered, secure access model that keeps your sensitive servers hidden from the open Internet.

Uploaded image
Uploaded image

Uploaded image
explain all of these step by step in a very easy manner with basics in details with explanation of all the terms:

Task
Creating and Configuring a Secure AWS VPC

Creating and Configuring a Secure AWS VPC with Public and Private Subnets
This document guides you through the process of creating a Virtual Private Cloud (VPC) and associated resources within AWS. A VPC allows you to launch AWS resources into a virtual network that you have complete control over. By the end of this guide, you will have created a VPC with public and private subnets, route tables, an internet gateway, security groups, and EC2 instances. Additionally, you'll learn how to SSH into the public EC2 instance from your local machine and from the public EC2 instance to the private EC2 instance.

Objectives
Create a VPC.

Create a Public and Private Subnet.

Create Routes and an Internet Gateway.

Launch EC2 instances in the Subnets.

Add a Security Group.

SSH into the Public EC2 Instance.

SSH from the Public EC2 Instance to the Private EC2 Instance.

1. Create a VPC
A Virtual Private Cloud (VPC) provides an isolated network environment in AWS, enabling you to launch AWS resources in a virtual network that you define.

Navigate to the VPC Console:

Log into the AWS Management Console.

In the Services menu, select "VPC" or search for "VPC".

Create the VPC:

Click on "Create VPC".

Set the following values:

VPC Name: my-vpc

IPv4 CIDR block: 10.0.0.0/16

IPv6 CIDR block: No IPv6 CIDR block

Tenancy: Default Tenancy

Click "Create VPC".

2. Create Subnets
Subnets allow you to partition your VPC's IP address range into smaller segments, enabling better management of resources within different availability zones.

Navigate to Subnets:

In the left-hand menu, click on "Subnets".

Click on "Create subnet".

Create the Public Subnet:

VPC ID: Select the VPC ID for my-vpc.

Subnet Name: public-subnet

Availability Zone: ap-southeast-1a

IPv4 CIDR block: 10.0.1.0/24

Click "Create Subnet".

Create the Private Subnet:

Click "Add new subnet".

Subnet Name: private-subnet

Availability Zone: ap-southeast-1b

IPv4 CIDR block: 10.0.2.0/24

Click "Create Subnet".

3. Create an Internet Gateway
An Internet Gateway allows instances in the VPC to communicate with the internet. It serves as a target for the VPC route tables for internet-routable traffic.

Navigate to Internet Gateways:

In the left-hand menu, select "Internet Gateways".

Click on "Create internet gateway".

Create the Internet Gateway:

Name tag: my-IGW

Click "Create internet gateway".

Attach the Internet Gateway to the VPC:

Select the newly created internet gateway.

Click "Actions" > "Attach to VPC".

Select my-vpc.

Click "Attach internet gateway".

4. Create Route Tables
Route tables control the routing of network traffic within your VPC. Public route tables direct traffic to the Internet Gateway, while private route tables handle internal routing.

Navigate to Route Tables:

In the left-hand menu, select "Route Tables".

Click on "Create route table".

Create the Public Route Table:

Name: public-route-table

VPC: Select my-vpc.

Click "Create route table".

Add Route to the Internet Gateway:

Select the public-route-table.

In the "Routes" tab, click "Edit routes".

Click "Add route".

Destination: 0.0.0.0/0

Target: Select the internet gateway my-IGW.

Click "Save changes".

Associate Public Route Table with Public Subnet:

In the "Subnet associations" tab, click "Edit subnet associations".

Select public-subnet.

Click "Save associations".

Create the Private Route Table:

Click "Create route table".

Name: private-route-table

VPC: Select my-vpc.

Click "Create route table".

Associate Private Route Table with Private Subnet:

Select the private-route-table.

In the "Subnet associations" tab, click "Edit subnet associations".

Select private-subnet.

Click "Save associations".

5. Create Security Group
Security groups act as a virtual firewall for your EC2 instances to control inbound and outbound traffic. They are essential for defining and enforcing network access rules.

Navigate to Security Groups:

In the left-hand menu, select "Security Groups".

Click on "Create security group".

Create the Security Group:

Security group name: security-group-1

Description: Allow SSH and all outbound traffic

VPC: Select my-vpc

Configure Inbound Rules:

In "Inbound rules" section click on Add rule>

Type: SSH

Source: Anywhere (0.0.0.0/0)

Configure Outbound Rules:

In "Outbound rules" section click on Add rule>

Type: All traffic

Destination: Anywhere (0.0.0.0/0)

Verify Resources:

In the left-hand menu, select "VPC".

View the resources map of our vpc (my-vpc)

6. Launch EC2 Instances
EC2 instances are virtual servers in the cloud, providing scalable computing capacity. Launching instances in public and private subnets allows you to control access and network traffic.

Launch Public EC2 Instance:

Navigate to "EC2" in the Services menu.

Click on "Instances" > "Launch instances".

Name: ec2-instance-1

Application and OS Images (Amazon Machine Image): Select "Ubuntu Server 24.04 LTS".

Instance Type: Select t2.micro.

Key pair (login): Click "Create new key pair".

Key pair name: public-key-pair

Key pair type: RSA

Private key file format: .pem or .ppk

Click "Create key pair".

Note : Download and save the key in your desired directroy,we will need it later

Network settings: Click "Edit".

VPC: Select my-vpc.

Subnet: Select public-subnet.

Auto-assign Public IP: Enable.

Security group: Select security-group-1.

Click "Launch instance".

Launch Private EC2 Instance:

Navigate back to "Instances" > "Launch instances".

Name: ec2-instance-2

Application and OS Images (Amazon Machine Image): Select "Ubuntu Server 24.04 LTS".

Instance Type: Select t3.micro.

Key pair (login): Click "Create new key pair".

Key pair name: private-key-pair

Key pair type: RSA

Private key file format: .pem or .ppk

Click "Create key pair".

Note : Download and save the key in your desired directroy,we will need it later

Network settings: Click "Edit".

VPC: Select my-vpc.

Subnet: Select private-subnet.

Auto-assign Public IP: Disable.

Security group: Select security-group-1.

Click "Launch instance".

7. SSH into the Public EC2 Instance
SSH (Secure Shell) allows you to securely access your EC2 instances. We'll first connect to the public instance from your local machine.

Set Permissions on the Key Pair:

Open your terminal.

Navigate to the directory containing your public-key-pair.pem file.

Run the following command to set the correct permissions: chmod 400 "public-key-pair.pem"

Connect to the Public EC2 Instance:

Obtain the public IP address of ec2-instance-1 from the EC2 console.

Run the following command to connect to the instance (replace <ec2-instance-1-public-IP> with your instance's public IP address): ssh -i "public-key-pair.pem" ubuntu@<ec2-instance-1-public-IP>

8. SSH from the Public EC2 Instance to the Private EC2 Instance
To access the private EC2 instance, we'll SSH from the public instance using the private key of the private instance.

Copy the content of public key from the directory containing private-key-pair.pem file :

Create the Private Key File on the Public Instance:

Once connected to ec2-instance-1, create a file for the private key with nano or vim (text editor) in ec2-instance-1: nano private-key-pair.pem Paste the contents of your private-key-pair.pem file.

Set Permissions on the Private Key File:

Run the following command to set the correct permissions: chmod 400 private-key-pair.pem

Connect to the Private EC2 Instance:

Obtain the private IP address of ec2-instance-2 from the EC2 console.

Run the following command to connect to the instance (replace <ec2-instance-2-public-IP> with your instance's private IP address): ssh -i "private-key-pair.pem" ubuntu@<ec2-instance-2-public-IP>

You have now successfully created a VPC, subnets, internet gateway, route tables, security group, launched EC2 instances, and established SSH connections to both the public and private instances.

Below is a friendly, jargon-busting walk-through of every step—from building your VPC to hopping through your bastion into a private server—with each term defined and each action explained.

🏗️ 1. Create Your VPC

What is a VPC?
A **Virtual Private Cloud** (VPC) is your own isolated network inside AWS—think of it as renting a private floor in a data-center building.

How to do it:

1. **Open** the AWS console → **Services** → **VPC**.
2. On the left, click **Your VPCs** → **Create VPC**.
3. Give it a name (e.g. `my-vpc`) and set **IPv4 CIDR block** to `10.0.0.0/16`.
   - **CIDR block**: defines your network's IP range (here, 10.0.0.0–10.0.255.255).
4. Leave IPv6 off, Tenancy = Default → **Create VPC**.

🌐 2. Carve Out Subnets

What is a Subnet?
A **subnet** is a slice of your VPC's IP range, placed in one Availability Zone (AZ).

Public Subnet (for your bastion/web server)

1. In VPC console → **Subnets** → **Create subnet**.
2. Select **my-vpc**, AZ = `ap-southeast-1a`, IPv4 CIDR = `10.0.1.0/24` (256 IPs).
3. Name it `public-subnet` → **Create**.

4. Select that subnet → **Actions** → **Edit auto-assign IP settings** → enable **Auto-assign public IPv4 address** → **Save**.
   - Now any EC2 here gets a public IP automatically.

Private Subnet (for your backend server)

1. Still in **Subnets**, click **Add new subnet**.
2. VPC = `` `my-vpc` ``, AZ = `` `ap-southeast-1b` ``, IPv4 CIDR = `` `10.0.2.0/24` ``.
3. Name `` `private-subnet` `` → **Create**.
   - No public IPs here—machines are hidden from the Internet.

🚪 3. Internet Gateway (IGW)

What is an Internet Gateway?
It's the "front door" that lets traffic flow between your VPC and the Internet.

Steps:

1. VPC console → **Internet Gateways** → **Create internet gateway** → name `` `my-IGW` `` → **Create**.
2. Select `` `my-IGW` `` → **Actions** → **Attach to VPC** → choose `` `my-vpc` `` → **Attach**.

🗺️ 4. Route Tables

What is a Route Table?
A set of rules ("routes") that tells network traffic **where** to go next.

Public Route Table

1. VPC console → **Route Tables** → **Create route table** → name `` `public-route-table` ``, VPC = `` `my-vpc` ``.
2. Select it → **Routes** → **Edit routes** → **Add route**:
   - Destination = `` `0.0.0.0/0` `` (all Internet)
   - Target = your IGW (`` `my-IGW` ``) → **Save**.
3. **Subnet associations** → **Edit** → select `` `public-subnet` `` → **Save**.

Machines in `` `public-subnet` `` now send "go to Internet" traffic out the IGW.

Private Route Table

1. **Create route table** → name `` `private-route-table` ``, VPC = `` `my-vpc` ``.
2. **Subnet associations** → select `` `private-subnet` `` → **Save**.
   - No `` `0.0.0.0/0` `` route here, so these machines cannot reach the Internet directly.

🔐 5. Security Group

What is a Security Group?
A virtual firewall you attach to EC2 instances that controls inbound/outbound ports.

For Your Public EC2 (“bastion-sg”)

1. VPC console → **Security Groups** → **Create security group** → name `` `bastion-sg` ``, VPC = `` `my-vpc` ``.
2. **Inbound rules**:
   - **Type** = SSH (TCP/22), **Source** = Anywhere (`` `0.0.0.0/0` ``)
3. **Outbound rules**: leave default (All traffic → Anywhere).
4. **Create**.

For Your Private EC2 (reuse or create similarly)

You can use the same SG if you want both bastion and private to allow outbound SSH—but you won’t attach it to the private subnet instance’s public interface, so it remains hidden.

🚀 6. Launch EC2 Instances

What is an EC2 Instance?
A virtual server (VM) in AWS.

6.1 Public EC2 (“bastion-server”)

1. Services → **EC2** → **Instances** → **Launch instances**.
2. **Name tag**: `` `bastion-server` ``.
3. **AMI**: Ubuntu Server 24.04 LTS.
4. **Instance type**: t2.micro (free-tier).
5. **Key pair**: Create new → name `` `public-key-pair` `` → type RSA + format `` `.pem` `` → **Create key pair** (download `` `.pem` ``).
6. **Network settings**:
   - VPC = `` `my-vpc` ``
   - Subnet = `` `public-subnet` ``
   - Auto-assign Public IP = **Enable**
   - Security group = **bastion-sg**
7. **Launch**.

6.2 Private EC2 (“backend-server”)

1. **Launch instances** again.
2. Name = `` `ec2-instance-2` ``.
3. AMI = Ubuntu Server 24.04 LTS.
4. Type = t3.micro.

5. Key pair: Create `private-key-pair` → download `.pem`.
6. Network:
   - VPC = `my-vpc`
   - Subnet = `private-subnet`
   - Auto-assign Public IP = **Disable**
   - Security group = your SSH-allowing SG (so bastion can SSH in)
7. **Launch**.

🔑 7. SSH into the Public EC2 Instance

1. **On your laptop** open Terminal.
2. `cd` into folder where `public-key-pair.pem` resides.
3. **Lock the key**:

   ```bash
   chmod 400 public-key-pair.pem
   ```

   - This makes it readable only by you (SSH requires strict permissions).
4. **Find the Public IP** of `bastion-server` in the EC2 console.
5. **SSH in**:

   ```bash
   ssh -i "public-key-pair.pem" ubuntu@<BASTION_PUBLIC_IP>
   ```

   - `ssh`: the command
   - `-i`: specifies your key file
   - `ubuntu@…`: logs in as the default "ubuntu" user
6. Type **yes** when prompted about the host key. You'll land at:

   ```bash
   ubuntu@ip-10-0-1-XX:~$
   ```

🔗 8. SSH from Public to Private EC2

1. **On the bastion**, create a key file for the private instance:

   ```bash
   nano ~/private-key-pair.pem
   ```

2. **Paste** the contents of your **local** `` `private-key-pair.pem` `` into this file.
3. **Save & exit** (Ctrl+O → Enter → Ctrl+X).
4. **Lock down** its permissions:

```bash
chmod 400 ~/private-key-pair.pem
```

5. **Get the Private IP** of `` `ec2-instance-2` `` from the EC2 console (e.g. `` `10.0.2.15` ``).
6. **SSH in**:

```bash
ssh -i ~/private-key-pair.pem ubuntu@10.0.2.15
```

7. Again, type **yes** at the host-key prompt. You'll now be at:

```bash
ubuntu@ip-10-0-2-15:~$
```

✅ You're Done!

You've now built:

- A private network (VPC with `` `10.0.0.0/16` ``)
- One public subnet for your bastion (`` `10.0.1.0/24` ``)
- One private subnet for your backend (`` `10.0.2.0/24` ``)
- An Internet Gateway + public route for Internet traffic
- A locked-down security group that only allows SSH
- A "bastion" EC2 you can reach from home
- A private EC2 reachable **only** from that bastion

This design keeps your sensitive servers hidden behind a single hardened jump-host, giving you both strong security and flexible access.

Uploaded image
for 6.2 Private EC2 ("backend-server") what should i select

For your **private EC2** ("backend-server") under **Network settings**, you want:

1. **VPC**: select **my-vpc**
2. **Subnet**: choose **private-subnet** (the one with CIDR 10.0.2.0/24)

3. **Auto-assign public IP**: **Disable** (so it has no Internet-facing address)
4. **Firewall (security groups)**: click **Select existing security group** and pick the one you created earlier (e.g. **security-group-1**)—the same SG that allows SSH from your bastion.

That way:

- The instance lives in the private subnet (no public IP).
- It uses the SG that only permits SSH from the bastion server.
- All other inbound ports remain closed, and outbound is wide open so it can reach out if needed.

Uploaded image
Uploaded image
Uploaded image

You're getting two different errors, so let's tackle them one at a time:

---

1) PowerShell "Cannot find path" & `chmod` on Windows

You tried:

```powershell
chmod 400 ~/.ssh/bastion-key.pem
Set-Location : Cannot find path 'C:\Users\ASUS\Banger\BANGER' because it does not exist.
```

What's happening

- `~/` and Linux-style paths don't automatically map to Windows PowerShell folders.
- You ran `cd BANGER` in a folder that doesn't exist.

How to fix it

1. **Open PowerShell** and confirm where your `.pem` files live.

   ```powershell
   PS C:\Users\ASUS> dir
   ```

2. **Change directory** into the exact folder name you see. If your keys are in `C:\Users\ASUS\See-SSH\` do:

   ```powershell
   cd .\See-SSH\
   ```

3. Now you can run:

```
powershell

chmod 400 .\public-key-pair.pem
```

   - PowerShell's built-in `chmod` will translate that into Windows ACLs so SSH is happy.
4. **Verify** the file is there:

```
powershell

dir *.pem
```

---

2) SSH "Connection timed out" on port 22

```
pgsql

ssh -i "public-key-pair.pem" ubuntu@54.255.2.163
ssh: connect to host 54.255.2.163 port 22: Connection timed out
```

What "timed out" means

Your packet never reaches the VM. The usual culprits are:

1. **No Internet Gateway / Route** on that subnet
2. **No inbound SSH rule** in the instance's security group
3. **Network ACL** blocking port 22
4. **Corporate/ISP firewall** blocking outbound SSH

Checklist to get SSH working

1. **Public Subnet + IGW**
   - In the VPC console, go to **Subnets** → select the subnet of your bastion → **Route Table** → verify there's a route `0.0.0.0/0` → Target = your Internet Gateway (igw-xxxxxx).
2. **Security Group**
   - In the EC2 console, select the **bastion-server** instance → **Networking** tab → scroll to **Security groups** → click into the SG.
   - Under **Inbound rules**, you must see:

```
python

Type    Protocol  Port range   Source
SSH     TCP       22           0.0.0.0/0
```

   - If it's not there, click **Edit inbound rules** and **Add rule** → SSH, port 22, Source = Anywhere → Save.
3. **Network ACLs**

- In the VPC console → **Network ACLs** → find the ACL for your public subnet → ensure there's an ALLOW rule for inbound TCP port 22 and outbound ephemeral ports (1024–65535).

4. **Check from your laptop**
   - Try `telnet 54.255.2.163 22` or `Test-NetConnection -ComputerName 54.255.2.163 -Port 22` in PowerShell to confirm connectivity.
   - If that also times out, see if your corporate/ISP network is blocking SSH. You can test from a different network (home hotspot, phone tethering).

Recap

1. **Fix your local path** so PowerShell `cd` goes to the folder with your `.pem`.
2. **Lock down** your key file with `chmod 400 .\public-key-pair.pem`.
3. **Verify** that your **bastion** instance really:
   - Lives in a public subnet with an IGW route
   - Has an SG allowing SSH inbound
   - (Optionally) check the NACLs
4. **Retry SSH**:

```powershell
ssh -i .\public-key-pair.pem ubuntu@54.255.2.163
```

Once those are all in place, you should get your Ubuntu prompt instead of a timeout. If you still can't connect, try from another network to rule out local firewall issues. Good luck!

PS C:\Users\ASUS\Banger> ssh -i "public-key-pair.pem" ubuntu@54.255.2.163
ssh: connect to host 54.255.2.163 port 22: Connection timed out
PS C:\Users\ASUS\Banger>
PS C:\Users\ASUS\Banger> ssh -i "public-key-pair.pem" ubuntu@54.255.2.163
ssh: connect to host 54.255.2.163 port 22: Connection timed out
PS C:\Users\ASUS\Banger> ssh -i "public-key-pair.pem" ubuntu@54.255.2.163
The authenticity of host '54.255.2.163 (54.255.2.163)' can't be established.
ED25519 key fingerprint is SHA256:dXffKPp1BQ7+ifE4Tx1hTwEfkHpH1lz0AnQPJR+kb0I.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '54.255.2.163' (ED25519) to the list of known hosts.
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-1024-aws x86_64)

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

 System information as of Tue May 20 10:54:40 UTC 2025

```
  System load:  0.0               Processes:             105
  Usage of /:   25.0% of 6.71GB   Users logged in:       0
  Memory usage: 20%               IPv4 address for enX0: 10.0.1.51
  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.

ubuntu@ip-10-0-1-51:~$ ^[[200~nano ~/private-key-pair.pem
nano: command not found
ubuntu@ip-10-0-1-51:~$ nano ~/private-key-pair.pem
ubuntu@ip-10-0-1-51:~$ chmod 400 ~/private-key-pair.pem ~/private-key-pair.pem
ubuntu@ip-10-0-1-51:~$ ^[[200~ssh -i "private-key-pair.pem" ubuntu@10.0.2.64~
ssh: command not found
ubuntu@ip-10-0-1-51:~$ ssh -i "private-key-pair.pem" ubuntu@10.0.2.64
The authenticity of host '10.0.2.64 (10.0.2.64)' can't be established.
ED25519 key fingerprint is SHA256:UqbZe+6EneONaMTJFDvid2TMbAOy16B+Qka14wduEIg.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.0.2.64' (ED25519) to the list of known hosts.
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-1024-aws x86_64)

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

 System information as of Tue May 20 10:59:00 UTC 2025

  System load:  0.08              Processes:             104
  Usage of /:   24.9% of 6.71GB   Users logged in:       0
  Memory usage: 20%               IPv4 address for enX0: 10.0.2.64
  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.

ubuntu@ip-10-0-2-64:~$

You've now gone through every step—creating the network, launching both instances, and finally SSH'ing in—and your logs show exactly what happened. Here's a quick breakdown of that transcript and what each line means:

---

1. Initial "Connection timed out"

```powershell
ssh -i "public-key-pair.pem" ubuntu@54.255.2.163
ssh: connect to host 54.255.2.163 port 22: Connection timed out
```

- **What it means**: Your SSH packets weren't reaching the server.
- **Why it cleared up on retry**: Most likely you added/fixed the Security Group rule to allow SSH (TCP/22) in from 0.0.0.0/0, or refreshed the route/NACL, and then the next attempt worked.

---

2. Successful SSH to the Public ("bastion") Server

```bash
The authenticity of host '54.255.2.163' can't be established…
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.255.2.163' to the list of known hosts.
```

```
Welcome to Ubuntu 24.04.2 LTS …
ubuntu@ip-10-0-1-51:~$
```

- **Host key prompt**: First time you hit that IP, SSH warns you haven't seen its fingerprint before. Typing **yes** lets SSH record it in `~/.ssh/known_hosts`.
- **Login success**: The "Welcome to Ubuntu" banner and shell prompt confirm you're in.

---

3. "nano: command not found"

```bash
nano ~/private-key-pair.pem
nano: command not found
```

- **Why**: AWS's minimal Ubuntu image doesn't include every editor by default.
- **Fix**: Install nano (or use `vi`):

  ```bash
  sudo apt update
  sudo apt install nano   # if you really want nano
  # or simply do:
  vi ~/private-key-pair.pem
  ```

- Alternatively, you can `cat > ~/private-key-pair.pem` and paste, then press `Ctrl+D` to save.

---

4. Copying & Locking the Private Key

```bash
chmod 400 ~/private-key-pair.pem
```

- Locks the file so **only you** can read it—SSH requires this strict permission.

---

5. SSH'ing from Bastion → Private Server

```bash
ssh -i "private-key-pair.pem" ubuntu@10.0.2.64
The authenticity of host '10.0.2.64' can't be established…
Are you sure you want to continue connecting (yes/no)? yes
Welcome to Ubuntu 24.04.2 LTS …
ubuntu@ip-10-0-2-64:~$
```

- **No public IP** on 10.0.2.64, but because you're already inside the VPC (on the bastion), you can reach it directly on its private IP.
- Again you confirm the host key and then land at the private machine's shell.

Summary of What You've Done

1. **Built**: a custom VPC (10.0.0.0/16) with both public (10.0.1.0/24) and private (10.0.2.0/24) subnets.
2. **Hung** an Internet Gateway and wired up routes so the public subnet can talk to the Internet.
3. **Launched** two Ubuntu EC2s:
   - **bastion-server** in the public subnet with a public IP.
   - **private-server** in the private subnet with **no** public IP.
4. **Locked down** a Security Group that only opens SSH (TCP/22).
5. **SSH'd**:
   - Local → bastion (via public IP)
   - Bastion → private (via private IP)

🎉 You now have a fully isolated, best-practice "jump host" setup: only one machine accepts Internet SSH, and from there you tunnel into your hidden backend.

Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Launch an EC2 Instance and Create an S3 Bucket

Launch an EC2 Instance and Create an S3 Bucket
explain all of these step by step along with explain the provided images and given codes and basic terms like what is plumuli,S3 Bucket ;why they are used and how they are used and in a very easy manner with basics in details with explanation of all the terms:

Task
Launch an EC2 Instance and Create an S3 Bucket

Launch an EC2 Instance and Create an S3 Bucket
A Comprehensive Guide for AI Engineering Hackathon 2025 Powered By Brain Station 23

Welcome to the AI Engineering Hackathon 2025, where you'll leverage cloud computing to build innovative AI solutions. This guide is designed for hackathon participants to quickly launch an AWS EC2 instance and create an S3 bucket for your project. We'll use the AWS Management Console for simplicity and speed, perfect for a fast-paced hackathon environment. Let's get your infrastructure up and running!

Prerequisites
AWS Account: Ensure you have an AWS account.

IAM User Permissions: The IAM user should have the appropriate permissions for EC2 and S3.

Instance Profile Permissions: To create and manage instance profiles in AWS, IAM user or role needs specific permissions related to IAM and EC2.

EC2 Instance: An EC2 instance running Ubuntu or a similar Linux distribution.

Step 1: Launch an EC2 instance using pulumi
For this project, we need an instance for mysql and other necessary resouces. We will provision an EC2 instance to host MySQL. While the instance will be created in a public subnet for the purpose of this demonstration, it is important to note that deploying MySQL in a public subnet is not recommended due to security concerns.

1. Configure AWS CLI
Configure AWS CLI with the necessary credentials. Run the following command and follow the prompts to configure it:

aws configure
This command sets up your AWS CLI with the necessary credentials, region (ap-southeast-1), and output format.

You will find the AWS Access key and AWS Seceret Access key on Lab description page,where you generated the credentials

2. Provisioning Compute Resources
In this step, we will provision the necessary AWS resources that will host our DB and pgAdmin server.

1. Create a Directory for Your Infrastructure

Before starting, it’s best to create a dedicated directory for the infrastructure files:

mkdir db-infra
cd db-infra
2. Install Python venv

Set up a Python virtual environment (venv) to manage dependencies for Pulumi or other Python-based tools:

sudo apt update
sudo apt install python3.8-venv -y
This will set up a Python virtual environment which will be useful later when working with Pulumi.

3. Initialize a New Pulumi Project

Pulumi is an Infrastructure-as-Code (IaC) tool used to manage cloud infrastructure. In this tutorial, you'll use Pulumi python to provision the AWS resources required for Database Cluster.

First login to Pulumi by running the command in the terminal. You will require a token to login. You can get the token from the Pulumi website.

```
pulumi login
```

After login, run the following command to initialize a new Pulumi project:

```
pulumi new aws-python
```

You will prompted with serveral questions. Go with the default options. Only change the aws region to ap-southeast-1.

4. Update the __main.py__ file:

Replace the contents of main.py with the following Python code to provision a VPC, subnet, security group, and EC2 instance:

```
import pulumi
import pulumi_aws as aws

# Create a VPC
vpc = aws.ec2.Vpc("my-vpc",
    cidr_block="10.0.0.0/16",
    tags={
        "Name": "my-vpc"
    })

pulumi.export("vpcId", vpc.id)

# Create a public subnet
public_subnet = aws.ec2.Subnet("public-subnet",
    vpc_id=vpc.id,
    cidr_block="10.0.1.0/24",
    availability_zone="ap-southeast-1a",
    map_public_ip_on_launch=True,
    tags={
        "Name": "public-subnet"
    })

pulumi.export("publicSubnetId", public_subnet.id)

# Create an Internet Gateway
internet_gateway = aws.ec2.InternetGateway("internet-gateway",
    vpc_id=vpc.id,
    tags={
        "Name": "igw"
    })

pulumi.export("igwId", internet_gateway.id)

# Create a Route Table
public_route_table = aws.ec2.RouteTable("public-route-table",
    vpc_id=vpc.id,
```

```
    tags={
        "Name": "rt-public"
    })

# Create a route in the route table for the Internet Gateway
route = aws.ec2.Route("igw-route",
    route_table_id=public_route_table.id,
    destination_cidr_block="0.0.0.0/0",
    gateway_id=internet_gateway.id)

# Associate the Route Table with the Public Subnet
route_table_association = aws.ec2.RouteTableAssociation("public-route-table-association",
    subnet_id=public_subnet.id,
    route_table_id=public_route_table.id)

pulumi.export("publicRouteTableId", public_route_table.id)

# Create a Security Group for the Public Instance
public_security_group = aws.ec2.SecurityGroup("public-secgrp",
    vpc_id=vpc.id,
    description="Enable SSH and MySQL access for public instance",
    ingress=[
        aws.ec2.SecurityGroupIngressArgs(
            protocol="tcp",
            from_port=22,
            to_port=22,
            cidr_blocks=["0.0.0.0/0"]),  # SSH
        aws.ec2.SecurityGroupIngressArgs(
            protocol="tcp",
            from_port=3306,
            to_port=3306,
            cidr_blocks=["0.0.0.0/0"]),  # MySQL
    ],
    egress=[
        aws.ec2.SecurityGroupEgressArgs(
            protocol="-1",
            from_port=0,
            to_port=0,
            cidr_blocks=["0.0.0.0/0"])  # Allow all outbound traffic
    ],
    tags={
        "Name": "public-secgrp"
    })

# Use the specified Ubuntu 24.04 LTS AMI
ami_id = "ami-060e277c0d4cce553"

# Create MySQL Instance
mysql_instance = aws.ec2.Instance("mysql-instance",
    instance_type="t2.micro",
    vpc_security_group_ids=[public_security_group.id],
```

```
    ami=ami_id,
    subnet_id=public_subnet.id,
    key_name="MyKeyPair",
    associate_public_ip_address=True,
    tags={
        "Name": "MySQLInstance",
        "Environment": "Development",
        "Project": "MySQLSetup"
    })

pulumi.export("mysqlInstanceId", mysql_instance.id)
pulumi.export("mysqlInstanceIp", mysql_instance.public_ip)
pulumi.export("mysqlInstanceDns", mysql_instance.public_dns)
```

Create Key Pair

Create a new key pair for our instances using the following command:

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

These commands will create key pair for mysql instance.

Set File Permissions of the key files

```
chmod 400 MyKeyPair.pem
```

Deploy the Pulumi Stack

Deploy the stack:

```
pulumi up --yes
```

Review the changes and confirm by selecting yes.

Verify the Deployment

You can verify the created resources such as VPC, Subnet, EC2 instance using AWS console.

Step 2: Create an S3 Bucket and Lifecycle Rule

Create an S3 Bucket

Go to the S3 console.

Click "Create bucket" and follow the prompts.

Make sure versioning is enabled as going forward we have to apply lifecycle policy on our bucket.

Create a Lifecycle Rule

Navigate to the bucket's "Management" tab.

Create a lifecycle rule to expire objects after 3 days.

Give name to your lifecycle rule.

Choose “Apply to all objects in the bucket” rule scope.

Set expiration days to 3 as we will be storing backups of last 3 days.

Click create rule and it's done.

Step 3: Attach an IAM Role to the EC2 Instance
Let's create an IAM role with the necessary permissions for EC2 to write to our S3 bucket.

Create an IAM Role
Go to the IAM console and create a new role.

Select trusted entity type as AWS service and usecase as EC2 as we are creating the role for EC2 instance.

Give a name to the role and click Create role.

Attach Policy for Permissions
On the role summary page, under the "Permissions" tab, click on the "Add permissions" button.

Choose Create inline policy.

Attach the following json file in the policy editor: { "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["s3:PutObject", "s3:GetObject", "s3:ListBucket"], "Resource": ["arn:aws:s3:::your-bucket-name", "arn:aws:s3:::your-bucket-name/*"] }] } Replace your-bucket-name with your bucket name.

Attach the Role to EC2
Go to the EC2 Dashboard.

Select the instance you created to attach the role.

Click on Actions > Security > Modify IAM Role.

In the dropdown list, you should see the role you created. Select it and click Update IAM Role.

Test Connectivity
SSH into your EC2 instance using the following command:

ssh -i path/to/your/MyKeyPair.pem ubuntu@<public-ip>
Verify S3 access with

aws s3 ls
Make sure to replace path/to/your/MyKeyPair.pem with the actual path to your key pair file. And your ubuntu instance should have the awscli installed.

Now you're ready to build your AI solution for the hackathon with a live EC2 instance and S3 storage!

Here's a friendly, step-by-step walkthrough of "Launch an EC2 Instance and Create an S3 Bucket" using Pulumi and the console—complete with the why behind each term and screenshots you provided.

🛠️ Prerequisites & Key Terms

1. **AWS Account & IAM Permissions**
   - You need an AWS account and an IAM user/role that can create VPCs, EC2 instances, S3 buckets, IAM roles, etc.
2. **Pulumi**
   - An **Infrastructure-as-Code** (IaC) tool. You write real code (Python, TypeScript, Go…) to define cloud resources, then `` `pulumi up` `` creates them.
   - Why? It gives you versionable, repeatable infra instead of clicking manually.
3. **EC2 (Elastic Compute Cloud)**
   - AWS's virtual servers ("VMs in the cloud"). You pick an AMI (OS image), instance type (CPU/RAM), network, security group, and SSH key.
4. **S3 Bucket**
   - A massively scalable object store ("buckets of files"). Perfect for storing database backups, logs, models, etc.
5. **Lifecycle Rule**
   - A policy on your S3 bucket that automatically expires or archives objects after a certain time—great for keeping only the last 3 days of backups, for instance.
6. **IAM Role & Instance Profile**
   - A bucket-write job running on EC2 needs permission to call S3 APIs. Instead of baking credentials into your server, you attach an **IAM Role** to the instance ("instance profile") granting it exactly the S3 rights it needs.

🔧 Step 1: Provision Your EC2 & Networking with Pulumi

1.1 Configure AWS CLI

```bash
aws configure
```

Enter your **Access Key**, **Secret Key**, **Default region** (`` `ap-southeast-1` ``), and **output** (`` `json` ``).

1.2 Create a project directory & venv

```bash
mkdir db-infra && cd db-infra
sudo apt update
sudo apt install python3.8-venv -y
python3.8 -m venv venv && source venv/bin/activate
```

1.3 Install & login to Pulumi

```bash
bash

pip install pulumi pulumi-aws
pulumi login             # follow prompts to authenticate
pulumi new aws-python    # choose defaults, set region = ap-southeast-1
```

1.4 Edit `__main__.py`

Replace it with:

```python
python

import pulumi
import pulumi_aws as aws

# 1) VPC
vpc = aws.ec2.Vpc("my-vpc",
    cidr_block="10.0.0.0/16",
    tags={"Name":"my-vpc"})
pulumi.export("vpcId", vpc.id)

# 2) Public Subnet
public_subnet = aws.ec2.Subnet("public-subnet",
    vpc_id=vpc.id,
    cidr_block="10.0.1.0/24",
    availability_zone="ap-southeast-1a",
    map_public_ip_on_launch=True,
    tags={"Name":"public-subnet"})
pulumi.export("publicSubnetId", public_subnet.id)

# 3) Internet Gateway
igw = aws.ec2.InternetGateway("internet-gateway",
    vpc_id=vpc.id, tags={"Name":"igw"})
pulumi.export("igwId", igw.id)

# 4) Route Table + Route
rt = aws.ec2.RouteTable("public-route-table", vpc_id=vpc.id, tags={"Name":"rt-public"})
aws.ec2.Route("igw-route",
    route_table_id=rt.id,
    destination_cidr_block="0.0.0.0/0",
    gateway_id=igw.id)
aws.ec2.RouteTableAssociation("public-rta",
    subnet_id=public_subnet.id,
    route_table_id=rt.id)
pulumi.export("rtId", rt.id)

# 5) Security Group (SSH + MySQL)
sg = aws.ec2.SecurityGroup("public-secgrp",
    vpc_id=vpc.id,
    description="SSH & MySQL",
    ingress=[
      aws.ec2.SecurityGroupIngressArgs( protocol="tcp", from_port=22, to_port=22, cidr_blocks=["0.0
      aws.ec2.SecurityGroupIngressArgs( protocol="tcp", from_port=3306, to_port=3306, cidr_blocks=[
    ],
    egress=[ aws.ec2.SecurityGroupEgressArgs(protocol="-1", from_port=0, to_port=0, cidr_blocks=["0
    tags={"Name":"public-secgrp"})

# 6) EC2 Instance for MySQL
ami_id = "ami-060e277c0d4cce553"  # Ubuntu 24.04 LTS in ap-southeast-1
mysql = aws.ec2.Instance("mysql-instance",
    instance_type="t2.micro",
    vpc_security_group_ids=[sg.id],
    subnet_id=public_subnet.id,
    ami=ami_id,
    associate_public_ip_address=True,
    key_name="MyKeyPair",
    tags={"Name":"MySQLInstance","Project":"Hackathon"})
pulumi.export("mysqlIp", mysql.public_ip)
pulumi.export("mysqlDns", mysql.public_dns)
```

1.5 Create & Protect Key Pair

```bash
aws ec2 create-key-pair --key-name MyKeyPair \
  --query 'KeyMaterial' --output text > MyKeyPair.pem
chmod 400 MyKeyPair.pem
```

1.6 Deploy with Pulumi

```bash
pulumi up --yes
```

Pulumi will show a "plan" of what it will create. Confirm, and watch it spin up VPC, subnet, SG, IGW, route, and EC2.

> **What you see**
>
> - In the Pulumi CLI: exported outputs `` `mysqlIp` ``, `` `publicSubnetId` ``, etc.
> - In AWS Console → EC2 → Instances: a running instance named "mysql-instance" with a public IP.

📦 Step 2: Create an S3 Bucket & Lifecycle Rule

2.1 Create the Bucket

1. Console → S3 → **Create bucket**
2. **Bucket name**: `` `mysql-data-backup-bucket` `` (globally unique!)
3. **Region**: ap-southeast-1
4. **Block all public access**: **On**
5. **Bucket Versioning**: **Enable** (to keep versions of your backups)
6. Click **Create bucket**.

> **Screenshot Highlights**
>
> - The "Block all public access" section (all four boxes checked) ensures no one on the Internet can GET/PUT objects unless explicitly permitted.
> - Enabling Versioning gives you the ability to recover previous copies if something is overwritten.

2.2 Set a Lifecycle Rule (Expire after 3 days)

1. Open your new bucket → **Management** tab → **Create lifecycle rule**.
2. **Rule name**: `` `mysql-backup-expire` ``
3. **Scope**: **Apply to all objects in the bucket** ✓

4. **Lifecycle rule actions** → **Expiration** → **Set current version expiration** to **3 days**.
5. Save.

   Why? This auto-deletes any backup older than three days, keeping storage costs down and retaining just your most recent snapshots.

🔐 Step 3: Give Your EC2 Permission to Write to S3

3.1 Create an IAM Role for EC2

1. IAM Console → **Roles** → **Create role**
2. **Trusted entity** → **AWS service** → **EC2** → Next
3. Skip attaching any managed policy for now → **Next**
4. **Name** it `mysql-data-backup-bucket-role` → **Create role**

3.2 Attach an Inline Policy Allowing S3 Put/Get/List

1. In IAM → open your new role → **Permissions** tab → **Add permissions** → **Create inline policy**
2. Choose the **JSON** editor and paste:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::mysql-data-backup-bucket",
        "arn:aws:s3:::mysql-data-backup-bucket/*"
      ]
    }
  ]
}
```

3. **Review & create** the policy.

3.3 Attach the Role to Your EC2 Instance

1. EC2 Console → **Instances** → select `MySQLInstance` → **Actions** → **Security** → **Modify IAM role**
2. Select `mysql-data-backup-bucket-role` → **Update IAM role**

   What this does:

   - Now, when you log into your EC2 and run `aws s3 cp` or backup scripts, the instance will automatically have the AWS credentials to write into your bucket—no hardcoded keys needed.

🚪 Step 4: Test Everything

1. **SSH into your MySQL EC2**:

```bash
ssh -i MyKeyPair.pem ubuntu@<mysqlIp>
```

2. **Install AWS CLI** (if not already):

```bash
sudo apt update && sudo apt install awscli -y
```

3. **Verify S3 access**:

```bash
aws s3 ls s3://mysql-data-backup-bucket
```

   You should see an (initially empty) list—no errors.

4. **Simulate a backup**:

```bash
echo "fake-backup-$(date +%F).sql" > backup.sql
aws s3 cp backup.sql s3://mysql-data-backup-bucket/
```

5. In the S3 console, refresh the **Objects** tab to see `backup.sql`.
6. Wait 3 days (or shorten the rule to 1 day for testing), then confirm the object auto-expires under **Management** → **Lifecycle**.

🎉 You're Done!

You've now:

- Written code in **Pulumi** to spin up VPC, subnet, security groups, IGW, routes, and an EC2 instance.
- Launched an **EC2** VM running Ubuntu for MySQL.
- Created a versioned **S3 bucket** with a **lifecycle rule** to auto-expire old backups.
- Built an **IAM role** that lets your EC2 securely `PutObject` into S3 without any embedded credentials.

This setup gives you a sound, hackathon-ready playground—spin up your MySQL, dump to S3, and let your AI code iterate on fresh data! 🚀

Uploaded image

in my vscode PowerShell

That error simply means PowerShell can't find the AWS CLI executable on your PATH. You haven't yet installed (or configured) the AWS CLI on your Windows machine, so typing `aws` does nothing. Here's how to fix it:

---

1. Install the AWS CLI on Windows

1. **Download the MSI installer**
   - Go to https://awscli.amazonaws.com/AWSCLIV2.msi and download the MSI.
   - Or visit the AWS docs: "Install the AWS CLI version 2 on Windows (msi)".
2. **Run the installer**
   - Double-click the downloaded `AWSCLIV2.msi` and follow the wizard.
   - It will install the `aws` binary into `C:\Program Files\Amazon\AWSCLIV2\` and add it to your system PATH.
3. **Restart your terminal**
   - Close all PowerShell (and VS Code) terminals and open a new one so the updated PATH takes effect.
4. **Verify the install**

   ```powershell
   aws --version
   # e.g. aws-cli/2.11.10 Python/3.9.11 Windows/10 exe/AMD64 prompt/off
   ```

---

2. Configure your Credentials

Once you see `aws --version` succeed, run:

```powershell
aws configure
```

You'll be prompted for:

- **AWS Access Key ID** → paste the key from your Lab page
- **AWS Secret Access Key** → paste the secret from your Lab page
- **Default region name** → `ap-southeast-1`
- **Default output format** → (e.g.) `json`

That creates a `~/.aws/credentials` file and lets the CLI make authenticated calls.

---

3. Re-run Your `aws` Commands

Now in that same VS Code PowerShell (or a plain PowerShell) you can do:

```powershell
aws s3 ls
aws ec2 describe-instances
# or aws configure again
```

and it will work.

Alternative: Use the VS Code "AWS Toolkit" Extension

If you prefer a graphical side-panel, you can also install the **AWS Toolkit for VS Code**. It bundles its own CLI and will prompt you to add credentials. But the straight CLI install above is the fastest way to get `aws configure` working on PowerShell.

Feel free to let me know if you hit any errors during the install!

```
in vs code:
PS C:\Users\ASUS\walter> aws --version
aws-cli/2.27.18 Python/3.13.3 Windows/11 exe/AMD64
PS C:\Users\ASUS\walter> aws configure
>>
PS C:\Users\ASUS\walter> aws configure
>>
AWS Access Key ID [None]: AKIA6F4G6SC4ET4YTGBP
AWS Secret Access Key [None]: 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
Default region name [None]: ap-southeast-1
Default output format [None]: json
PS C:\Users\ASUS\walter> mkdir db-infra

    Directory: C:\Users\ASUS\walter

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          20/05/2025    18:10                db-infra

PS C:\Users\ASUS\walter> cd db-infra
PS C:\Users\ASUS\walter\db-infra> sudo apt update
Sudo is disabled on this machine. To enable it, go to the Developer Settings page in the Settings app
PS C:\Users\ASUS\walter\db-infra> sudo apt install python3.8-venv -y
Sudo is disabled on this machine. To enable it, go to the Developer Settings page in the Settings app
PS C:\Users\ASUS\walter\db-infra> python3.8 -m venv venv && source venv/bin/activate
```

On Windows, you won't be able to use the `apt` commands – those only work on Linux. You have two main options:

Option A: Use Windows-native Python & venv

1. **Make sure Python 3 is installed on Windows**
    - Download the latest Python 3.x installer from python.org and run it.
    - **Important**: check "Add Python to PATH" on the installer.
2. **Create your virtual environment**
   In PowerShell, from `C:\Users\ASUS\walter\db-infra` run:

   ```powershell
   python -m venv venv
   ```

   This will drop a folder called `venv` containing a Windows-style venv.
3. **Activate it**

   ```powershell
   .\venv\Scripts\Activate.ps1
   ```

   You should see your prompt change to `(venv) PS C:\Users\ASUS\walter\db-infra>`.
4. **Now install any Python packages you need**

   ```powershell
   pip install pulumi pulumi-aws
   ```

   and continue with your Pulumi steps.

Option B: Use Windows Subsystem for Linux (WSL)

If you really need an Ubuntu-style environment (so you can do `sudo apt …` exactly as in the guide), you can:

1. **Enable WSL**
   Open PowerShell as Administrator and run:

   ```powershell
   wsl --install
   ```

   This will install Ubuntu and drop you into a Linux shell on first reboot.
2. **Switch into your Ubuntu distro**
   In any PowerShell window:

   ```powershell
   wsl
   ```

Now you're in a `/home/your-user` Linux shell.

3. **Recreate your project folder inside WSL**

```bash
mkdir ~/db-infra && cd ~/db-infra
sudo apt update
sudo apt install python3.8-venv -y
python3.8 -m venv venv
source venv/bin/activate
pip install pulumi pulumi-aws
```

Then proceed with the Pulumi commands exactly as in the Linux-based guide.

Which to choose?

- **If you only need Python/venv** and Pulumi, then **Option A** is simplest.
- **If you'd prefer the Ubuntu toolchain (`apt`, `sudo`, etc.)**, go with **Option B** (WSL).

Let me know if you'd like more detail on either setup!

Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
give all these step by step execution work for vs code PowerShell along with the explanation:

Task
Launch an EC2 Instance and Create an S3 Bucket

Launch an EC2 Instance and Create an S3 Bucket
A Comprehensive Guide for AI Engineering Hackathon 2025 Powered By Brain Station 23

Welcome to the AI Engineering Hackathon 2025, where you'll leverage cloud computing to build innovative AI solutions. This guide is designed for hackathon participants to quickly launch an AWS EC2 instance and create an S3 bucket for your project. We'll use the AWS Management Console for simplicity and speed, perfect for a fast-paced hackathon environment. Let's get your infrastructure up and running!

Prerequisites
AWS Account: Ensure you have an AWS account.

IAM User Permissions: The IAM user should have the appropriate permissions for EC2 and S3.

Instance Profile Permissions: To create and manage instance profiles in AWS, IAM user or role needs specific permissions related to IAM and EC2.

EC2 Instance: An EC2 instance running Ubuntu or a similar Linux distribution.

Step 1: Launch an EC2 instance using pulumi
For this project, we need an instance for mysql and other necessary resouces. We will provision an EC2 instance to host MySQL. While the instance will be created in a public subnet for the purpose of this demonstration, it is important to note that deploying MySQL in a public subnet is not recommended due to security concerns.

1. Configure AWS CLI
Configure AWS CLI with the necessary credentials. Run the following command and follow the prompts to configure it:

aws configure
This command sets up your AWS CLI with the necessary credentials, region (ap-southeast-1), and output format.

You will find the AWS Access key and AWS Seceret Access key on Lab description page,where you generated the credentials

2. Provisioning Compute Resources
In this step, we will provision the necessary AWS resources that will host our DB and pgAdmin server.

1. Create a Directory for Your Infrastructure

Before starting, it’s best to create a dedicated directory for the infrastructure files:

mkdir db-infra
cd db-infra
2. Install Python venv

Set up a Python virtual environment (venv) to manage dependencies for Pulumi or other Python-based tools:

sudo apt update
sudo apt install python3.8-venv -y
This will set up a Python virtual environment which will be useful later when working with Pulumi.

3. Initialize a New Pulumi Project

Pulumi is an Infrastructure-as-Code (IaC) tool used to manage cloud infrastructure. In this tutorial, you'll use Pulumi python to provision the AWS resources required for Database Cluster.

First login to Pulumi by running the command in the terminal. You will require a token to login. You can get the token from the Pulumi website.

pulumi login

After login, run the following command to initialize a new Pulumi project:

```
pulumi new aws-python
```

You will prompted with serveral questions. Go with the default options. Only change the aws region to ap-southeast-1.

4. Update the __main.py__ file:

Replace the contents of main.py with the following Python code to provision a VPC, subnet, security group, and EC2 instance:

```python
import pulumi
import pulumi_aws as aws

# Create a VPC
vpc = aws.ec2.Vpc("my-vpc",
    cidr_block="10.0.0.0/16",
    tags={
        "Name": "my-vpc"
    })

pulumi.export("vpcId", vpc.id)

# Create a public subnet
public_subnet = aws.ec2.Subnet("public-subnet",
    vpc_id=vpc.id,
    cidr_block="10.0.1.0/24",
    availability_zone="ap-southeast-1a",
    map_public_ip_on_launch=True,
    tags={
        "Name": "public-subnet"
    })

pulumi.export("publicSubnetId", public_subnet.id)

# Create an Internet Gateway
internet_gateway = aws.ec2.InternetGateway("internet-gateway",
    vpc_id=vpc.id,
    tags={
        "Name": "igw"
    })

pulumi.export("igwId", internet_gateway.id)

# Create a Route Table
public_route_table = aws.ec2.RouteTable("public-route-table",
    vpc_id=vpc.id,
    tags={
        "Name": "rt-public"
    })
```

```
# Create a route in the route table for the Internet Gateway
route = aws.ec2.Route("igw-route",
    route_table_id=public_route_table.id,
    destination_cidr_block="0.0.0.0/0",
    gateway_id=internet_gateway.id)

# Associate the Route Table with the Public Subnet
route_table_association = aws.ec2.RouteTableAssociation("public-route-table-association",
    subnet_id=public_subnet.id,
    route_table_id=public_route_table.id)

pulumi.export("publicRouteTableId", public_route_table.id)

# Create a Security Group for the Public Instance
public_security_group = aws.ec2.SecurityGroup("public-secgrp",
    vpc_id=vpc.id,
    description="Enable SSH and MySQL access for public instance",
    ingress=[
        aws.ec2.SecurityGroupIngressArgs(
            protocol="tcp",
            from_port=22,
            to_port=22,
            cidr_blocks=["0.0.0.0/0"]),  # SSH
        aws.ec2.SecurityGroupIngressArgs(
            protocol="tcp",
            from_port=3306,
            to_port=3306,
            cidr_blocks=["0.0.0.0/0"]),  # MySQL
    ],
    egress=[
        aws.ec2.SecurityGroupEgressArgs(
            protocol="-1",
            from_port=0,
            to_port=0,
            cidr_blocks=["0.0.0.0/0"])  # Allow all outbound traffic
    ],
    tags={
        "Name": "public-secgrp"
    })

# Use the specified Ubuntu 24.04 LTS AMI
ami_id = "ami-060e277c0d4cce553"

# Create MySQL Instance
mysql_instance = aws.ec2.Instance("mysql-instance",
    instance_type="t2.micro",
    vpc_security_group_ids=[public_security_group.id],
    ami=ami_id,
    subnet_id=public_subnet.id,
    key_name="MyKeyPair",
    associate_public_ip_address=True,
```

```
    tags={
        "Name": "MySQLInstance",
        "Environment": "Development",
        "Project": "MySQLSetup"
    })

pulumi.export("mysqlInstanceId", mysql_instance.id)
pulumi.export("mysqlInstanceIp", mysql_instance.public_ip)
pulumi.export("mysqlInstanceDns", mysql_instance.public_dns)
```

Create Key Pair

Create a new key pair for our instances using the following command:

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

These commands will create key pair for mysql instance.

Set File Permissions of the key files

```
chmod 400 MyKeyPair.pem
```

Deploy the Pulumi Stack
Deploy the stack:

```
pulumi up --yes
```

Review the changes and confirm by selecting yes.

Verify the Deployment
You can verify the created resources such as VPC, Subnet, EC2 instance using AWS console.

Step 2: Create an S3 Bucket and Lifecycle Rule
Create an S3 Bucket
Go to the S3 console.

Click "Create bucket" and follow the prompts.

Make sure versioning is enabled as going forward we have to apply lifecycle policy on our bucket.

Create a Lifecycle Rule
Navigate to the bucket's "Management" tab.

Create a lifecycle rule to expire objects after 3 days.

Give name to your lifecycle rule.

Choose “Apply to all objects in the bucket” rule scope.

Set expiration days to 3 as we will be storing backups of last 3 days.

Click create rule and it’s done.

Step 3: Attach an IAM Role to the EC2 Instance
Let’s create an IAM role with the necessary permissions for EC2 to write to our S3 bucket.

Create an IAM Role
Go to the IAM console and create a new role.

Select trusted entity type as AWS service and usecase as EC2 as we are creating the role for EC2 instance.

Give a name to the role and click Create role.

Attach Policy for Permissions
On the role summary page, under the "Permissions" tab, click on the "Add permissions" button.

Choose Create inline policy.

Attach the following json file in the policy editor: { "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["s3:PutObject", "s3:GetObject", "s3:ListBucket"], "Resource": ["arn:aws:s3:::your-bucket-name", "arn:aws:s3:::your-bucket-name/*"] }] } Replace your-bucket-name with your bucket name.

Attach the Role to EC2
Go to the EC2 Dashboard.

Select the instance you created to attach the role.

Click on Actions > Security > Modify IAM Role.

In the dropdown list, you should see the role you created. Select it and click Update IAM Role.

Test Connectivity
SSH into your EC2 instance using the following command:

ssh -i path/to/your/MyKeyPair.pem ubuntu@<public-ip>
Verify S3 access with

aws s3 ls
Make sure to replace path/to/your/MyKeyPair.pem with the actual path to your key pair file. And your ubuntu instance should have the awscli installed.

Now you're ready to build your AI solution for the hackathon with a live EC2 instance and S3 storage!

Re

Below is a step-by-step walkthrough, written for PowerShell in VS Code on Windows, showing you how to:

1. Provision an EC2 instance (to host MySQL) via Pulumi
2. Create and configure an S3 bucket (with versioning and a 3-day lifecycle rule)
3. Attach an IAM role to your EC2 instance so it can put objects into that bucket

I've broken each piece down with the exact commands you'd type in your VS Code integrated terminal, plus a plain-English explanation of what each step and term means.

🔧 Prerequisites

1. **AWS CLI installed & configured**
   - You've already run `` `aws --version` `` and `` `aws configure` ``.
   - That stored your Access Key, Secret Key, default region (`` `ap-southeast-1` ``), and output format (`` `json` ``) in `` `~/.aws/credentials` `` and `` `~/.aws/config` ``.
2. **Pulumi CLI installed**
   - Download from https://pulumi.com and install.
   - You'll use it to turn Python code into AWS resources.
3. **Python 3 installed**
   - On Windows: install from https://python.org and check "Add Python to PATH."
4. **A VS Code workspace**
   - Open a folder (like `` `C:\Users\ASUS\walter\db-infra` ``) in VS Code.
   - Use its integrated PowerShell terminal.

1️⃣ Launch an EC2 Instance Using Pulumi

1.1 Create & enter your project folder

```powershell
PS C:\Users\ASUS\walter> mkdir db-infra
PS C:\Users\ASUS\walter> cd db-infra
```

> **mkdir** makes a new directory; **cd** moves into it.

1.2 Create a Python virtual environment

```powershell
PS C:\Users\ASUS\walter\db-infra> python -m venv venv
PS C:\Users\ASUS\walter\db-infra> .\venv\Scripts\Activate.ps1
(venv) PS C:\Users\ASUS\walter\db-infra>
```

- **venv**: isolates your project's Python packages.
- The prompt changes to `` `(venv)` `` to show it's active.

1.3 Install Pulumi Python packages

```powershell
(venv) PS> pip install pulumi pulumi-aws
```

- **pulumi**: the core SDK.
- **pulumi-aws**: AWS provider plugin.

1.4 Log in to Pulumi

```powershell
(venv) PS> pulumi login
```

Follow the link it prints to authenticate. This lets Pulumi track your stacks in the Pulumi Service.

1.5 Initialize a new Pulumi project

```powershell
(venv) PS> pulumi new aws-python --name mysql-infra-backup --yes
```

- **pulumi new aws-python**: scaffolds a Python project configured for AWS.
- `` `--yes` `` accepts defaults; you'll be prompted only for region—enter `` `ap-southeast-1` ``.

1.6 Replace `` `__main__.py` `` with your infrastructure code

Open `` `__main__.py` `` and paste:

```python
import pulumi
import pulumi_aws as aws

# 1) VPC
vpc = aws.ec2.Vpc("my-vpc",
    cidr_block="10.0.0.0/16",
    tags={"Name":"my-vpc"})
pulumi.export("vpcId", vpc.id)

# 2) Public Subnet
public_subnet = aws.ec2.Subnet("public-subnet",
    vpc_id=vpc.id,
    cidr_block="10.0.1.0/24",
    availability_zone="ap-southeast-1a",
    map_public_ip_on_launch=True,
    tags={"Name":"public-subnet"})
pulumi.export("publicSubnetId", public_subnet.id)

# 3) Internet Gateway
igw = aws.ec2.InternetGateway("internet-gateway",
    vpc_id=vpc.id, tags={"Name":"igw"})
pulumi.export("igwId", igw.id)

# 4) Route Table + route + association
rt = aws.ec2.RouteTable("public-route-table",
    vpc_id=vpc.id, tags={"Name":"rt-public"})
```

```
route = aws.ec2.Route("igw-route",
    route_table_id=rt.id,
    destination_cidr_block="0.0.0.0/0",
    gateway_id=igw.id)
rta = aws.ec2.RouteTableAssociation("public-route-table-association",
    subnet_id=public_subnet.id,
    route_table_id=rt.id)
pulumi.export("publicRouteTableId", rt.id)

# 5) Security Group: allow SSH (22) & MySQL (3306) in, any out
public_sg = aws.ec2.SecurityGroup("public-secgrp",
    vpc_id=vpc.id,
    description="SSH & MySQL inbound",
    ingress=[
        aws.ec2.SecurityGroupIngressArgs("tcp",22,22,["0.0.0.0/0"]),
        aws.ec2.SecurityGroupIngressArgs("tcp",3306,3306,["0.0.0.0/0"]),
    ],
    egress=[aws.ec2.SecurityGroupEgressArgs("-1",0,0,["0.0.0.0/0"])],
    tags={"Name":"public-secgrp"})

# 6) EC2 Instance
ami_id = "ami-060e277c0d4cce553"  # Ubuntu 24.04 LTS
mysql_instance = aws.ec2.Instance("mysql-instance",
    instance_type="t2.micro",
    vpc_security_group_ids=[public_sg.id],
    ami=ami_id,
    subnet_id=public_subnet.id,
    key_name="MyKeyPair",
    associate_public_ip_address=True,
    tags={"Name":"MySQLInstance"})
pulumi.export("mysqlInstanceId", mysql_instance.id)
pulumi.export("mysqlInstanceIp", mysql_instance.public_ip)
```

Key terms:

- **VPC**: your private network inside AWS.
- **Subnet**: a slice of the VPC's IP range.
- **IGW**: lets traffic flow between your VPC and the internet.
- **Route Table**: maps destinations (like `0.0.0.0/0`) to targets (your IGW).
- **Security Group**: a firewall "attached" to your EC2.
- **EC2 Instance**: a virtual server; here it'll run MySQL.

1.7 Create an EC2 key pair

In **PowerShell** (still in `db-infra`):

```
powershell

(venv) PS> aws ec2 create-key-pair `
    --key-name MyKeyPair `
    --query 'KeyMaterial' `
    --output text > MyKeyPair.pem
```

- This calls AWS to generate a new SSH keypair named **MyKeyPair**, then writes the **private key** into `MyKeyPair.pem`.

1.8 Secure your private key

```powershell
(venv) PS> icacls .\MyKeyPair.pem /inheritance:r /grant:r "${env:USERNAME}:(R)"
```

- On Windows we use **icacls** instead of `chmod 400` to give only your user read (`R`) permission.

1.9 Deploy with Pulumi

```powershell
(venv) PS> pulumi up --yes
```

- Pulumi shows a preview of the 9 resources it will create; `--yes` skips your confirmation and applies them immediately.
- You'll see the outputs printed at the end: your VPC ID, Subnet ID, IGW ID, Route Table ID, **and** the new EC2's Public IP.

2️⃣ Create an S3 Bucket & Lifecycle Rule

2.1 Create the bucket

In **PowerShell**:

```powershell
PS> aws s3api create-bucket `
    --bucket mysql-data-backup-bucket `
    --region ap-southeast-1 `
    --create-bucket-configuration LocationConstraint=ap-southeast-1
```

> **S3 Bucket**: a globally unique "container" for objects (files). We'll store MySQL backups here.

2.2 Enable versioning

```powershell
PS> aws s3api put-bucket-versioning `
    --bucket mysql-data-backup-bucket `
    --versioning-configuration Status=Enabled
```

- **Versioning**: keeps multiple revisions of each object so you can recover old backups if you need to.

2.3 Add a 3-day lifecycle rule

Create a file `lifecycle.json`:

```json
{
  "Rules": [
    {
      "ID": "expire-backups-after-3-days",
```

```
        "Status": "Enabled",
        "Filter": { "Prefix": "" },
        "Expiration": { "Days": 3 },
        "NoncurrentVersionExpiration": { "NoncurrentDays": 3 }
      }
    ]
  }
```

Apply it:

```powershell
PS> aws s3api put-bucket-lifecycle-configuration `
    --bucket mysql-data-backup-bucket `
    --lifecycle-configuration file://lifecycle.json
```

- **Lifecycle rule**: automatically deletes (or transitions) objects. Here we delete any version older than 3 days.

3️⃣ Attach an IAM Role to Your EC2

So your instance can write backups into the S3 bucket without embedding credentials in code, we give it an IAM role.

3.1 Create an IAM role for EC2

1. In the AWS Console → IAM → Roles → **Create role**.
2. **Trusted entity**: AWS service → **EC2** → **Next**.
3. Name it `` `mysql-data-backup-bucket-role` `` → Create role.

3.2 Add an inline policy granting S3 permissions

In the role's **Permissions** tab → **Add permissions** → **Create inline policy** → chose JSON editor and paste:

```json
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:PutObject","s3:GetObject","s3:ListBucket"],
    "Resource": [
      "arn:aws:s3:::mysql-data-backup-bucket",
      "arn:aws:s3:::mysql-data-backup-bucket/*"
    ]
  }]
}
```

→ Review → Create policy.

> **IAM Role**: an identity your EC2 can assume to get temporary credentials.
> **Inline policy**: directly attached here, lets it list the bucket and put/get objects.

3.3 Attach the IAM role to your EC2 instance

In the AWS Console → EC2 → Instances → select your **MySQLInstance** → **Actions** → **Security** → **Modify IAM role** → pick `mysql-data-backup-bucket-role` → **Update IAM role**.

4️⃣ Verify Everything

1. **SSH into your new EC2** (replace `<IP>` with the output from Pulumi—e.g. `54.251.209.150`):

```powershell
PS> ssh -i .\MyKeyPair.pem ubuntu@<IP>
```

2. **On the EC2**, install the AWS CLI if needed and test S3 access:

```bash
$ sudo apt update && sudo apt install awscli -y
$ aws s3 ls s3://mysql-data-backup-bucket
```

   You should see an empty list (no errors), proving your role is working.

3. **(Optional)** run a dummy upload:

```bash
$ echo "hello" > test.txt
$ aws s3 cp test.txt s3://mysql-data-backup-bucket/backups/test.txt
```

   Back in the Console → S3 → your bucket → you'll see `backups/test.txt`.

🎉 You're All Set!

- **Infrastructure as Code**: Pulumi spins up VPC, subnet, route table, IGW, security group, and EC2 in one go.
- **Secure Backups**: your EC2 instance can put files into S3 without hard-coding credentials.
- **Lifecycle Management**: S3 auto-cleans any backups older than 3 days.

Now build your AI application on top of this reliable MySQL + S3 foundation!

Uploaded image
download which one?

On Windows you have two easy ways to install Pulumi:

1. **Installer (MSI)** ("amd64" button under **Installer (MSI)**)
   - This is a standard Windows MSI package.
   - Double-click it, follow the wizard, and Pulumi will be added to your PATH automatically.

- It also registers the "Pulumi" entry in your Start menu.

2. **Windows Binary Download** ("amd64" button under **Windows Binary Download")
   - This is just the raw `pulumi.exe` file.
   - You'd download it, unzip it somewhere (e.g. `C:\Tools\pulumi`), and then add that folder to your PATH manually.

Which should you pick?

- **Go with the MSI** if you want the simplest, most "Windows-native" install.
- Use the **Binary** only if you prefer no-installer, portable tools (for example, on a locked-down machine).

So—click the **amd64** button under **Installer (MSI)**, run the `.msi`, and you'll be ready to go.

```
PS C:\Users\ASUS\walter\db-infra> python -m venv venv
PS C:\Users\ASUS\walter\db-infra> .\venv\Scripts\Activate.ps1
(venv) PS C:\Users\ASUS\walter\db-infra>  pip install pulumi pulumi-aws
Collecting pulumi
  Downloading pulumi-3.170.0-py3-none-any.whl (338 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 338.4/338.4 kB 1.5 MB/s eta 0:00:00
Collecting pulumi-aws
  Downloading pulumi_aws-6.80.0-py3-none-any.whl (10.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.4/10.4 MB 1.2 MB/s eta 0:00:00
Collecting dill~=0.3
  Downloading dill-0.4.0-py3-none-any.whl (119 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 119.7/119.7 kB ? eta 0:00:00
Collecting semver~=3.0
  Downloading semver-3.0.4-py3-none-any.whl (17 kB)
Collecting protobuf~=4.21
  Using cached protobuf-4.25.7-cp310-abi3-win_amd64.whl (413 kB)
Collecting pyyaml~=6.0
  Using cached PyYAML-6.0.2-cp310-cp310-win_amd64.whl (161 kB)
Collecting debugpy~=1.8.7
  Downloading debugpy-1.8.14-cp310-cp310-win_amd64.whl (5.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.2/5.2 MB 1.3 MB/s eta 0:00:00
Collecting pip<26,>=24.3.1
  Using cached pip-25.1.1-py3-none-any.whl (1.8 MB)
Collecting grpcio~=1.66.2
  Downloading grpcio-1.66.2-cp310-cp310-win_amd64.whl (4.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.3/4.3 MB 1.3 MB/s eta 0:00:00
Collecting parver>=0.2.1
  Downloading parver-0.5-py3-none-any.whl (15 kB)
Collecting typing-extensions>=4.11
  Using cached typing_extensions-4.13.2-py3-none-any.whl (45 kB)
Collecting arpeggio>=1.7
  Downloading Arpeggio-2.0.2-py2.py3-none-any.whl (55 kB)
```

━━ 55.3/55.3 kB ? eta 0:00:00
Collecting attrs>=19.2
Downloading attrs-25.3.0-py3-none-any.whl (63 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 63.8/63.8 kB 3.6 MB/s eta 0:00:00
ERROR: To modify pip, please run the following command:
C:\Users\ASUS\walter\db-infra\venv\Scripts\python.exe -m pip install pulumi pulumi-aws

[notice] A new release of pip is available: 23.0.1 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Users\ASUS\walter\db-infra> pulumi login
pulumi: The term 'pulumi' is not recognized as a name of a cmdlet, function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
(venv) PS C:\Users\ASUS\walter\db-infra> pulumi login
pulumi: The term 'pulumi' is not recognized as a name of a cmdlet, function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
(venv) PS C:\Users\ASUS\walter\db-infra>

When you ran:

```powershell
(venv) PS C:\Users\ASUS\walter\db-infra> pip install pulumi pulumi-aws
```

you only installed the **Python SDK** libraries that let your Pulumi **program** talk to AWS. You still need the actual **Pulumi CLI** executable in your PATH in order to run `pulumi login`, `pulumi new`, `pulumi up`, etc.

On Windows the easiest way is to install Pulumi with the **MSI installer** once (not inside the venv). That will drop a `pulumi.exe` into your `C:\Program Files\Pulumi\bin` (or similar) and add it to your system PATH automatically.

---

1. Install Pulumi CLI via MSI

1. Go to https://www.pulumi.com/docs/get-started/install/
2. Under **Windows** → **Installer (MSI)** click the “amd64” button.
3. Run the downloaded `.msi`.
4. Accept all defaults.

Once the MSI finishes, open a **new** PowerShell window (so it picks up the updated PATH) and type:

```powershell
pulumi version
```

You should see something like:

```text
Pulumi v3.170.0
```

---

2. Re-activate your virtual environment

If you still have your venv open, close that shell and then:

```powershell
cd C:\Users\ASUS\walter\db-infra
.\venv\Scripts\Activate.ps1
```

---

3. Verify both pieces are installed

Now you should have:

- **pulumi.exe** on your PATH (the CLI)
- the **pulumi** and **pulumi-aws** Python packages in your venv

Try:

```powershell
(venv) PS> pulumi login
(venv) PS> pulumi version
(venv) PS> python -c "import pulumi; print(pulumi.__version__)"
```

All three commands should succeed.

Why this split exists

- **Pulumi CLI** (`` `pulumi.exe` ``) is the command-line tool that orchestrates deployments: logging in, keeping state, diffing your desired state vs. real AWS, etc.
- **Pulumi Python SDK** (`` `pip install pulumi pulumi-aws` ``) is the library your `` `__main__.py` `` uses to *declare* your infrastructure.

Both are required—but they install separately:

1. CLI via MSI (or Homebrew, or `` `scoop install pulumi` ``, whatever you prefer)
2. Python SDK inside whatever virtual environment you're authoring in

Once you've done that, you can:

```powershell
(venv) PS> pulumi new aws-python
(venv) PS> pulumi up
```

—and your Pulumi-driven VPC/EC2/S3 will spring to life.

Uploaded image
Uploaded image
now to do what?

At this point the Pulumi CLI is waiting for your **Personal Access Token** so it can talk to the Pulumi Service (where it keeps your stack's state, outputs, etc.). Here's exactly what to do next:

---

1. Create a Personal Access Token in Pulumi

1. In your browser, navigate to https://app.pulumi.com/account/tokens
2. Click **Create token**
3. Give it a name like `db-infra-dev-token` and click **Create**
4. **Copy** the long token string that appears (it will look something like `pul-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`) and store it somewhere safe

---

2. Paste the token back into your terminal

Back in your activated venv PowerShell, you should see:

```text
Run `pulumi login --help` for alternative login options.
Enter your access token from https://app.pulumi.com/account/tokens
or hit <ENTER> to log in using your browser:
:
```

1. **Paste** the token you just copied and press **Enter**
2. You should see a success message:

   ```text
   Success! Logged into Pulumi service at https://api.pulumi.com
   ```

---

3. Initialize a new Pulumi project

Now that you're logged in, you can create your stack:

```powershell
# In your db-infra folder, still inside your venv
pulumi new aws-python --name mysql-infra-backup
```

- Accept the defaults, except change the **region** to `ap-southeast-1` when prompted.

- This will scaffold a `Pulumi.yaml` and a `__main__.py` for you.

---

4. Copy in your main.py

Open the newly created `__main__.py` and replace its contents with the code you prepared (the VPC, subnet, IGW, route table, security group, and MySQL EC2 instance). Save the file.

---

5. Create the AWS Key Pair

Run:

```powershell
aws ec2 create-key-pair --key-name MyKeyPair --query 'KeyMaterial' --output text > MyKeyPair.pem
icacls MyKeyPair.pem /inheritance:r /grant:r "%USERNAME%:R"
```

(This sets proper permissions on Windows.)

---

6. Deploy your stack

Finally, deploy everything in one go:

```powershell
pulumi up --yes
```

You'll see Pulumi talk to AWS, create your VPC, EC2, etc., and then print out the new instance's public IP and other exports.

What just happened?

- **pulumi login**: authenticated your CLI against the Pulumi Service
- **pulumi new**: bootstrapped a Python-based IaC project
- **pulumi up**: read `__main__.py`, diff'd against AWS, and applied the changes

From here you can move on to Step 2 (creating your S3 bucket and lifecycle rules) or continue customizing your infrastructure in code.

Uploaded image

Great—you're now inside a *fresh* folder and `pulumi new aws-python` is kicking off the interactive project scaffolding. Here's what to do next:

1. **Answer the prompts**
   You'll see something like:

   ```css
   Project description (A minimal AWS Python Pulumi program):
   ```

   - You can simply press **Enter** to accept the default (or type something like "MySQL infra for hackathon").

   Next you'll get:

   ```css
   Project name [mysql-infra-backup]:
   ```

   - Press **Enter** again unless you want to change it.

   Then:

   ```css
   Stack name [dev]:
   ```

   - Press **Enter** to use the built-in `` `dev` `` stack.

   Finally:

   ```css
   AWS Region [us-west-2]:
   ```

   - **Type** `` `ap-southeast-1` `` and hit **Enter**.

2. **Review the generated files**
   Pulumi will create:
   - `` `Pulumi.yaml` `` (your project definition)
   - `` `__main__.py` `` (the stub Python program)
   - A `` `requirements.txt` `` that lists `` `pulumi` `` and `` `pulumi_aws` ``

3. **Replace `` `__main__.py` ``**
   Open `` `__main__.py` ``, delete its stub contents, and paste in your VPC+Subnet+IGW+RouteTable+SecurityGroup+EC2 code.

4. **Install any missing dependencies**
   Still in your venv:

   ```powershell
   pip install -r requirements.txt
   ```

   (You already have `` `pulumi` `` and `` `pulumi-aws` `` installed.)

5. **Create your EC2 keypair**
   Run (from the **same** PowerShell prompt):

   ```powershell
   aws ec2 create-key-pair --key-name MyKeyPair --query 'KeyMaterial' --output text > MyKeyPair.pe
   icacls MyKeyPair.pem /inheritance:r /grant:r "%USERNAME%:R"
   ```

   That gives your instance a key you'll use for SSH.

6. **Deploy**
   Finally:

   ```powershell
   pulumi up --yes
   ```

   Pulumi will show you the plan, create your VPC, subnet, IGW, route table, security group, and EC2 instance, then spit out the new instance's public IP.

Once that finishes, you've got a live MySQL EC2 in your VPC. From there you can move on to:

- **Step 2**: Creating your S3 bucket and lifecycle rule
- **Step 3**: Attaching an IAM role so your EC2 can snapshot and upload to S3

But right now, just complete those `pulumi new` prompts and run `pulumi up`. That'll get your infra standing up in AWS.

Uploaded image
is it ok or do i have to step back:

No need to roll back—what you've done is perfectly fine. When Pulumi asks:

```pgsql
The toolchain to use for installing dependencies and running the program [Use arrows to move, type t
  > pip
    poetry [not found]
    uv [not found]
```

all you have to do is **press Enter** on "`pip`" to accept it as your toolchain. Pulumi will then:

1. Generate your `Pulumi.yaml`, `__main__.py`, etc.
2. Leave the default `requirements.txt` in place.
3. Let you immediately run `pulumi up`.

So just hit Enter, answer the remaining prompts (stack name, region), and then run:

```powershell
pulumi up --yes
```

to deploy your VPC, subnet, Internet Gateway, route table, security group, and EC2 instance.

Project description (A minimal AWS Python Pulumi program):
Created project 'mysql-infra-backup'

Please enter your desired stack name.
To create a stack in an organization, use the format <org-name>/<stack-name> (e.g. `acmecorp/dev`).
Stack name (dev):
Sorry, could not create stack 'dev': stack 'junayed-org/mysql-infra-backup/dev' already exists
Stack name (dev):
Sorry, could not create stack 'dev': stack 'junayed-org/mysql-infra-backup/dev' already exists
error: EOF
(venv) PS C:\Users\ASUS\walter\db-infra\tial> cd ..
(venv) PS C:\Users\ASUS\walter\db-infra> cd mizz
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> pulumi new aws-python --name mysql-infra-backup

A project with the name `mysql-infra-backup` already exists. Use 'mysql-infra-backup' anyway
This command will walk you through creating a new Pulumi project.

Enter a value or leave blank to accept the (default), and press <ENTER>.
Press ^C at any time to quit.

Project description (A minimal AWS Python Pulumi program):
Created project 'mysql-infra-backup'

Please enter your desired stack name.
To create a stack in an organization, use the format <org-name>/<stack-name> (e.g. `acmecorp/dev`).
Stack name (dev):
Sorry, could not create stack 'dev': stack 'junayed-org/mysql-infra-backup/dev' already exists
Stack name (dev):

how to utilize this??

You're seeing that error because you already spun up a **dev** stack for the **mysql-infra-backup** project, so Pulumi will not let you create "junayed-org/mysql-infra-backup/dev" twice.

You have two options:

---

1) Re-use the existing dev stack

1. **Select** the existing stack instead of trying to re-create it:

```powershell
pulumi stack select dev
```

2. (Optional) **Inspect** your current config:

```powershell
pulumi config
```

3. **Deploy** what's in your `__main__.py` (or preview first):

```powershell
pulumi up
# or preview only:
pulumi preview
```

That will pick up your existing project & stack and provision any new resources.

---

2) Create a brand-new stack

If you really want a fresh stack (say for a "test" or "prod" environment), just give it a different name:

```powershell
pulumi new aws-python --name mysql-infra-backup \
    --force        # override the "project exists" check
```

When it asks for a stack name:

```text
Stack name (dev): test
```

Now you'll have two stacks:

- **dev** → `junayed-org/mysql-infra-backup/dev`
- **test** → `junayed-org/mysql-infra-backup/test`

You can switch between them with:

```powershell
pulumi stack select dev
# …or…
pulumi stack select test
```

Quick Recap

- **To re-use**: `pulumi stack select dev` + `pulumi up`

- **To make a new one**: `pulumi new aws\-python --force` + answer “test” (or any new name) at the stack prompt

Either approach will get you out of the “stack already exists” error and back to deploying your VPC, subnet, IGW, route table and EC2.

```
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> aws ec2 create-key-pair `
`>>     --key-name MyKeyPair `
`>>     --query 'KeyMaterial' `
`>>     --output text > MyKeyPair.pem
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> icacls .\MyKeyPair.pem /inheritance:r /grant:r
"${env:USERNAME}:(R)"
processed file: .\MyKeyPair.pem
Successfully processed 1 files; Failed processing 0 files
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> pulumi up --yes
Please choose a stack, or create a new one: dev
Previewing update (dev)

View in Browser (Ctrl+O): https://app.pulumi.com/junayed-org/mysql-infra-
backup/dev/previews/0d6946be-c557-4a7a-91cf-8aa98edcf998

     Type                 Name                    Plan     Info
     pulumi:pulumi:Stack  mysql-infra-backup-dev           1 error; 6
Diagnostics:
  pulumi:pulumi:Stack (mysql-infra-backup-dev):
    error: an unhandled error occurred: program exited with non-zero exit code: 1

    Traceback (most recent call last):
      File "C:\Program Files (x86)\Pulumi\pulumi-language-python-exec", line 17, in <module>
        import pulumi
    ModuleNotFoundError: No module named 'pulumi'
    It looks like the Pulumi SDK has not been installed. Have you run pip install?
    If you are running in a virtualenv, you must run pip install -r requirements.txt from inside the
virtualenv.

    [Pulumi Copilot] Would you like help with these diagnostics?
    https://app.pulumi.com/junayed-org/mysql-infra-backup/dev/previews/0d6946be-c557-4a7a-91cf-
8aa98edcf998?explainFailure

Resources:

(venv) PS C:\Users\ASUS\walter\db-infra\mizz>
```

You’re hitting that Python `ModuleNotFoundError: No module named 'pulumi'` because your Pulumi program is being executed with a Python interpreter that doesn’t have the Pulumi SDK installed. When you did:

```powershell
PS> pip install pulumi pulumi-aws
```

you installed those packages into the venv sitting in `` `db-infra\venv` ``, but your actual Pulumi project lives under `` `db-infra\mizz` `` and Pulumi is spawning a *different* Python process there (it uses whatever `` `python` `` is on PATH, not necessarily your venv).

Fix: install your deps into the *exact* interpreter Pulumi will use

1. **Activate** your venv from inside the project folder:

   ```powershell
   PS> cd C:\Users\ASUS\walter\db-infra\mizz
   PS> ..\venv\Scripts\Activate.ps1       # now your prompt shows (venv)
   ```

2. **Install** the requirements that Pulumi generated:

   ```powershell
   (venv) PS> pip install -r requirements.txt
   ```

   That `` `requirements.txt` `` was scaffolded by `` `pulumi new aws-python` `` and should contain at least:

   ```shell
   pulumi>=3.0.0
   pulumi-aws>=6.0.0
   ```

3. **Verify** that you can `` `import pulumi` `` from that same interpreter:

   ```powershell
   (venv) PS> python -c "import pulumi; print(pulumi.__version__)"
   ```

   You should see the version number, not an import error.

4. **Re-run** your deployment:

   ```powershell
   (venv) PS> pulumi up --yes
   ```

   Now Pulumi will invoke the *venv's* `` `python` ``, find the Pulumi SDK, and successfully synth & deploy your resources.

Why this happens

- **Pulumi CLI** (the `` `pulumi` `` command you installed via `` `choco` `` or MSI) is just a thin wrapper.

- When you do `pulumi up`, it shells out to run **your** `__main__.py` with **whatever** `python` it finds in PATH.
- If that Python isn't your virtual-env interpreter (or doesn't have pulumi libs installed), you get a `ModuleNotFoundError`.

By explicitly activating your venv in the same shell where you call `pulumi up`, you guarantee that "`python`" is the one with Pulumi installed.

Let me know if after installing requirements inside `mizz` you still see errors!

Uploaded image
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> pip install -r requirements.txt
Collecting pulumi<4.0.0,>=3.0.0
 Using cached pulumi-3.170.0-py3-none-any.whl (338 kB)
Collecting pulumi-aws<7.0.0,>=6.0.2
 Using cached pulumi_aws-6.80.0-py3-none-any.whl (10.4 MB)
Collecting grpcio~=1.66.2
 Using cached grpcio-1.66.2-cp310-cp310-win_amd64.whl (4.3 MB)
Collecting dill~=0.3
 Using cached dill-0.4.0-py3-none-any.whl (119 kB)
Collecting protobuf~=4.21
 Using cached protobuf-4.25.7-cp310-abi3-win_amd64.whl (413 kB)
Collecting semver~=3.0
 Using cached semver-3.0.4-py3-none-any.whl (17 kB)
Collecting pip<26,>=24.3.1
 Using cached pip-25.1.1-py3-none-any.whl (1.8 MB)
Collecting pyyaml~=6.0
 Using cached PyYAML-6.0.2-cp310-cp310-win_amd64.whl (161 kB)
Collecting debugpy~=1.8.7
 Using cached debugpy-1.8.14-cp310-cp310-win_amd64.whl (5.2 MB)
Collecting typing-extensions>=4.11
 Using cached typing_extensions-4.13.2-py3-none-any.whl (45 kB)
Collecting parver>=0.2.1
 Using cached parver-0.5-py3-none-any.whl (15 kB)
Collecting attrs>=19.2
 Using cached attrs-25.3.0-py3-none-any.whl (63 kB)
Collecting arpeggio>=1.7
 Using cached Arpeggio-2.0.2-py2.py3-none-any.whl (55 kB)
ERROR: To modify pip, please run the following command:
C:\Users\ASUS\walter\db-infra\venv\Scripts\python.exe -m pip install -r requirements.txt

[notice] A new release of pip is available: 23.0.1 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Users\ASUS\walter\db-infra\mizz>
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> V
V: The term 'V' is not recognized as a name of a cmdlet, function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

```
(venv) PS C:\Users\ASUS\walter\db-infra\mizz> pulumi up --yes
Previewing update (dev)

     pulumi:pulumi:Stack  mysql-infra-backup-dev          1 error; 6
Diagnostics:
  pulumi:pulumi:Stack (mysql-infra-backup-dev):
    Traceback (most recent call last):
      File "C:\Program Files (x86)\Pulumi\pulumi-language-python-exec", line 17, in <module>
        import pulumi
    ModuleNotFoundError: No module named 'pulumi'
    It looks like the Pulumi SDK has not been installed. Have you run pip install?
    If you are running in a virtualenv, you must run pip install -r requirements.txt from inside the
virtualenv.

    error: an unhandled error occurred: program exited with non-zero exit code: 1

    [Pulumi Copilot] Would you like help with these diagnostics?
    https://app.pulumi.com/junayed-org/mysql-infra-backup/dev/previews/15a84a28-e038-404f-9e3f-
52f806ad1a3b?explainFailure

Resources:

(venv) PS C:\Users\ASUS\walter\db-infra\mizz>
(venv) PS C:\Users\ASUS\walter\db-infra\mizz>
```

It's still the same root cause: when Pulumi runs your Python program, it isn't finding the SDK in the interpreter it's using. On Windows, even though you've activated the venv and run `pip install …`, sometimes VS Code's Python path or the way you invoked pip didn't actually install into that venv's site-packages.

Let's fix it step by step:

---

1. Install your dependencies with the venv's own python

Whenever Windows tells you:

> To modify pip, please run the following command:
> `C:\Users\ASUS\walter\db-infra\venv\Scripts\python.exe -m pip install -r requirements.txt`

that means **use that python.exe** — not a global pip — to install into the venv. So from your `mizz` folder, run:

```powershell
(venv) PS> .\venv\Scripts\python.exe -m pip install -r requirements.txt
```

You should see it unpack and install `pulumi`, `pulumi-aws`, etc., without that "To modify pip…" error.

---

2. Double-check the install

Still in the same activated venv, test:

```powershell
(venv) PS> python -c "import pulumi; print(pulumi.__version__)"
```

You should get a version like `3.170.0`. If you get an import error here, the install didn't go into this venv.

---

3. Re-run your deployment

Still in the **same PowerShell** where `(venv)` is shown and after a successful import check:

```powershell
(venv) PS> pulumi up --yes
```

Because your venv's `Scripts` folder (with its `python.exe` and the newly installed `pulumi` library) is at the front of `PATH`, Pulumi will now invoke that interpreter and succeed.

Why this matters

- The Pulumi CLI (`pulumi`) is a *wrapper* that shells out to `python __main__.py` under the hood.
- If you haven't installed the SDK into the exact interpreter that ends up running your code — you'll see `ModuleNotFoundError: No module named 'pulumi'`.

By calling the venv's `python.exe -m pip install …`, and staying in the same shell session where the venv is active, you guarantee the correct interpreter and site-packages get used.

Let me know if that clears it up!

explain what is pulumi and it's functionality in these case along with it's importance

Pulumi is an **Infrastructure as Code (IaC)** tool that lets you define, provision, and manage cloud resources (like VPCs, subnets, EC2 instances, S3 buckets, IAM roles, and more) using real programming languages—in our case, **Python**. Instead of clicking around in the AWS console, you write code that **declaratively** specifies the exact cloud architecture you want, and Pulumi takes care of turning that into API calls.

Key Pulumi Concepts

1. **Programs & Stacks**
   - A **Pulumi program** is just a Python script (e.g. `__main__.py`) that uses the `pulumi_aws` library to declare resources.

- A **stack** (e.g. `` `dev` ``) is an isolated instance of that program's infrastructure. You can have multiple stacks (dev, prod, staging) from the same code.

2. **Declarative Resources**

```python
vpc = aws.ec2.Vpc("my-vpc", cidr_block="10.0.0.0/16")
```

You declare "I want a VPC named `` `my-vpc` `` with this CIDR block." Pulumi figures out whether to create, update, or delete based on what's already running.

3. **State Management**
Pulumi keeps track of what's been created (in either its managed cloud backend or a local/state file). When you run `` `pulumi up` ``, it shows you a **preview** of the exact changes—no surprises.

4. **Language Power**
Since you're writing Python, you can use loops, conditionals, abstractions, modules, unit tests, and package managers. This is far richer than static YAML or JSON.

How Pulumi Works in Our Walk-Through

1. `` `pulumi new aws-python` ``
Bootstraps a new Python project with the right files (`` `Pulumi.yaml` ``, `` `requirements.txt` ``, a `` `__main__.py` ``).

2. **Writing Code**
We import the AWS SDK wrapper:

```python
import pulumi, pulumi_aws as aws
```

Then we declare:

   - A **VPC**
   - A **public subnet**, **Internet Gateway**, **route table**,
   - A **security group** with SSH & MySQL ports open
   - An **EC2 instance** running on Ubuntu

3. `` `pulumi up` ``
   - Pulumi computes a diff ("I need to create these 9 resources").
   - It prompts you, "OK to go?"
   - Once confirmed, it calls AWS APIs to create each resource in the right order.

4. **Outputs & Exports**
You can export values (like the instance's public IP) and Pulumi prints them at the end:

```python
pulumi.export("mysqlInstanceIp", mysql_instance.public_ip)
```

Why Pulumi Matters

1. **Reproducibility & Consistency**
   Every team member runs the same code, so staging matches production exactly.
2. **Speed & Automation**
   Spinning up a full VPC + DB server + IAM roles + bucket lifecycle rules goes from dozens of clicks to a single `` `pulumi up` ``.
3. **Version Control & Review**
   Infrastructure definitions live alongside your app code in Git. Pull requests can review infra changes just like code changes.
4. **Multi-Cloud & Multi-Language**
   If you ever need GCP, Azure, Kubernetes, or serverless functions, you reuse the same Pulumi shape and language—even the same Python program.
5. **Preview & Safety**
   Seeing the planned changes before anything executes avoids accidental deletes or misconfigurations.

In this hackathon scenario...

- **Speed**: You can get your MySQL instance and S3 bucket up in minutes.
- **Flexibility**: Tweak security rules, IP ranges, instance sizes—all in code.
- **Cleanup**: Running `` `pulumi destroy` `` tears everything down safely when the hack ends.

Pulumi turns the cloud console into code—giving you full control, reliability, and the power of your favorite programming language to manage your AI hackathon infrastructure.

explain what is S3 and it's functionality in these case along with it's importance

Amazon S3 (Simple Storage Service) is AWS's highly-scalable, durable, and secure **object storage** service. Instead of files on a disk or blocks on a drive, S3 stores *objects*—each object is made up of your data plus metadata you define, and is identified by a unique key (its name) inside a **bucket** (a container for objects).

Core Concepts

Term	Description
Bucket	A top-level container you create (e.g. `` `mysql-data-backup-bucket` ``). Buckets live in a single AWS region.
Object	The individual file you store—like `` `backup-2025-05-21.sql.gz` ``.
Key	The unique identifier for an object in a bucket (essentially the object's "path" or filename).
Versioning	An optional feature that keeps multiple versions of an object when it's overwritten or deleted.
Lifecycle Rule	A policy you attach to a bucket that automatically transitions or expires objects after set days.

Why S3 for MySQL Backups?

1. **Durability & Availability**
   - S3 replicates your data across multiple facilities within a region, giving you **99.999999999% (11-9's) of durability**.
   - Your backups are safe even if hardware fails.
2. **Scalability**
   - Store gigabytes or petabytes of data without worrying about provisioning storage servers.
   - Pay only for what you use.
3. **Cost-Effectiveness**
   - You can choose storage classes (Standard, Infrequent Access, Glacier) to balance cost vs. retrieval speed.
   - Lifecycle rules can automatically move older backups to cheaper tiers or delete them.
4. **Security & Access Control**
   - Buckets and objects are private by default.
   - Use IAM roles (as we did) to give only your EC2 instance permission to **PutObject**, **GetObject**, and **ListBucket**.
   - Block public access so nobody on the internet can read your backups.
5. **Integration with AWS Services**
   - Easily plug into Lambda, Athena, Glacier, and more.
   - You could later trigger a Lambda function when a new backup lands in S3.

How We Used S3 in This Guide

1. **Bucket Creation**
   - We created `` `mysql-data-backup-bucket` `` in the same region (`` `ap-southeast-1` ``) as our EC2 instance.
2. **Enabling Versioning**
   - Turned on **versioning** so if our backup script accidentally overwrote a file, we'd still have the previous version.

3. **Lifecycle Rule**
    - Configured a rule to **expire** (delete) objects older than 3 days.
    - Keeps only the last 3 days of backups, saving costs and avoiding manual cleanup.
4. **IAM Role Attachment**
    - Created an **IAM role** and attached it to our EC2/MySQL instance.
    - The inline policy allowed `` `s3:PutObject` ``, `` `s3:GetObject` ``, and `` `s3:ListBucket` `` on our bucket only.
    - Ensured our backup script can upload and verify backups without exposing credentials.
5. **Backup Workflow**
    - Your EC2 instance runs a cron job or script that:
        1. Dumps the MySQL database to a file.
        2. Uses the AWS CLI to `` `aws s3 cp backup.sql.gz s3://mysql-data-backup-bucket/` ``.

Because S3 is so reliable and cost-effective, it's the perfect place to land automated database backups. You get peace of mind—your critical data is durably stored, old backups are cleaned up automatically, and access is safely locked down through IAM.