

VPC Peering & Cross-VPC Connectivity

By Haroon Zaman | November 2025

Introduction

In this project, I learned how to connect two separate VPCs using **VPC Peering**. Each VPC is normally isolated and cannot communicate with other networks unless we explicitly allow it. By setting up VPC peering, I enabled private, secure communication between two VPCs without sending any data over the public internet.

This project helped me understand multi-VPC architectures, private routing, and how cloud resources communicate across isolated networks safely.

In this project, I completed the following tasks:

- **Set up multiple VPCs** – I created two different VPCs, each with its own CIDR range, subnets, route tables, and resources.
- **Created a VPC Peering connection** – I established a private network link between both VPCs so they could communicate.
- **Tested VPC peering with connectivity checks** – I verified communication using private IP addresses, confirming that traffic stayed inside AWS's private backbone network.

VPC peering ensures that data travels **privately**, directly between VPCs.

Without peering, traffic would need to go through the **public internet**, which is less secure and not ideal for internal workloads.

Creating the First VPC

- First, I created a new VPC using the **VPC and More** option.
- I named this VPC **My_Network_VPC1** so it's easy to identify later.

A VPC is an isolated portion of the AWS Cloud populated by AWS objects, such as Amazon EC2 instances. Mouse over a resource to highlight the related resources.

The screenshot displays the AWS Management Console interface for creating a new VPC. On the left, the 'VPC settings' panel is visible, showing options for 'Resources to create' (VPC only or VPC and more), 'Name tag auto-generation' (Auto-generate), 'IPv4 CIDR block' (10.1.0.0/16), 'IPv6 CIDR block' (No IPv6 CIDR block), and 'Tenancy' (Default). On the right, the 'Preview' section shows a network diagram with three components: 'VPC' (My_Network_VPC_1-vpc), 'Subnets (1)' (eu-north-1a, My_Network_VPC_1-subnet-), and 'Route tables (1)' (My_Network_VPC_1-rtb-public). The diagram illustrates the connectivity between these resources.

- I declared the IPv4 CIDR block as **10.1.0.0/16**, giving the VPC plenty of private IP space.
- I chose **No IPv6**, since this project focuses only on IPv4 connectivity.
- I set **Tenancy to Default**, as I did not need dedicated hardware for this setup.

- I selected **1 Availability Zone** with **one public subnet** and **no private subnet**. This is because this VPC will only be used for testing VPC peering, and I only needed a simple network structure with one public subnet.
- I selected **No NAT Gateway**. NAT gateways are only needed when *private subnets* require outbound internet access. Since I did not create a private subnet and didn't need internet access from inside the VPC, a NAT gateway wasn't necessary.
- I also selected **No VPC Endpoints**. VPC endpoints are used when you want to privately connect your VPC to AWS services like S3 or DynamoDB. Since this project focuses only on VPC-to-VPC connectivity, and not on private access to AWS services, VPC endpoints were not required.

Your VPCs (3) [Info](#)

Find VPCs by attribute or tag

Less than a minute ago [Actions](#) [Create VPC](#)

<input type="checkbox"/>	Name	VPC ID	State	Encryption c...	Encryption control ...	Block Public...	IPv4 CIDR	II
<input type="checkbox"/>	My_Network_VPC	vpc-0e69c4b80354b0e63	Available	-	-	Off	10.0.0.0/16	-
<input type="checkbox"/>	My_Network_VPC_1-vpc	vpc-0d207d25eff661a9c	Available	-	-	Off	10.1.0.0/16	-

- After I created both VPCs, I verified their CIDR ranges:
 - **My_Network_VPC1:** 10.1.0.0/16
 - **My_Network_VPC:** 10.0.0.0/16
- Each VPC had its own unique CIDR block, which is required for VPC peering. VPCs **cannot** overlap in IP ranges if they need to communicate, so using different CIDR blocks made them compatible for peering.
- With both VPCs fully created and isolated from each other, it was now time to **connect them using a VPC peering connection** so they could communicate privately.

Creating the VPC Peering Connection

- Next, I went to **Peering Connections** from the left-side panel of the VPC console.
- I clicked on **Create Peering Connection** to start setting up the link between my two VPCs.

Name - optional
Create a tag with a key of 'Name' and a value that you specify.
My_VPC <> My_VPC_1

Select a local VPC to peer with

VPC ID (Requester)
vpc-0d207d25eff661a9c (My_Network_VPC_1-vpc)

VPC CIDRs for vpc-0d207d25eff661a9c (My_Network_VPC_1-vpc)

CIDR	Status	Status reason
10.1.0.0/16	Associated	-

Select another VPC to peer with

Account
☒ My account
☐ Another account

Region
☒ This Region (eu-north-1)
☐ Another Region

VPC ID (Acceptor)
vpc-0e69c4b80354b0e63 (My_Network_VPC)

VPC CIDRs for vpc-0e69c4b80354b0e63 (My_Network_VPC)

CIDR	Status	Status reason
10.0.0.0/16	Associated	-

- I named the peering connection **My_VPC <> My_VPC1** so it was easy to recognize.
- First, I selected the **Requestor VPC**, which was **My_Network_VPC1**.
The *requestor* is simply the VPC that **initiates** the peering request.
- Then I selected the **Acceptor VPC**, which was **My_Network_VPC**.
The *acceptor* is the VPC that **receives** the request and must approve it.
- In the acceptor settings, I noticed AWS allows peering with **other accounts** and even **other regions**, making cross-account or cross-region peering possible.
But for this project, I selected my second VPC in the **same region**.
- After selecting both sides, I clicked **Create Peering Connection** to create the request.

pcx-084a2f471c694f98b / My_VPC <> My_VPC_1

Pending acceptance
You can accept or reject this peering connection request using the 'Actions' menu. You have until Monday 15 December 2025 at 10:38:23 GMT+3 to accept or reject the request, otherwise the request will be automatically rejected.

Details **Info**

Requester owner ID
377721963177

Peering connection ID
pcx-084a2f471c694f98b

Status
Pending Acceptance by 377721963177

Expiration time
Monday 15 December 2025 at 10:38:23 GMT+3

Acceptor owner ID
377721963177

Requester VPC
vpc-0d207d25eff661a9c / My_Network_VPC_1-vpc

Requester CIDRs
10.1.0.0/16

Requester Region
Stockholm (eu-north-1)

VPC Peering connection ARN
arn:aws:ec2:eu-north-1:377721963177:pcx-084a2f471c694f98b

Acceptor VPC
vpc-0e69c4b80354b0e63 / My_Network_VPC

Acceptor CIDRs
-

Acceptor Region
Stockholm (eu-north-1)

Actions
Accept request
Reject request
Edit DNS settings
Manage tags
Delete peering connection

Accepting the Peering Request

- After creating the peering connection, it was still in the **Pending Acceptance** state.

- Because both VPCs are in the **same AWS account**, I had full permission to approve the request myself.
- I selected the peering connection, went to **Actions**, and clicked **Accept Request**.
- Once accepted, the peering status changed to **Active**, meaning both VPCs were now allowed to communicate privately.

🟢 Your VPC peering connection (pcx-084a2f471c694f98b | My_VPC <-> My_VPC_1) has been established. To send and receive traffic across this VPC peering connection, you must add a route to the peered VPC in one or more of your VPC route tables. [Info](#) Modify my route tables now ✕

pcx-084a2f471c694f98b / My_VPC <-> My_VPC_1 Actions ▾

Details [Info](#)

<p>Requester owner ID</p> <p>377721963177</p> <p>Peering connection ID</p> <p>pcx-084a2f471c694f98b</p> <p>Status</p> <p>Active</p> <p>Expiration time</p> <p>–</p>	<p>Accepter owner ID</p> <p>377721963177</p> <p>Requester VPC</p> <p>vpc-0d207d25eff661a9c / My_Network_VPC_1-vpc</p> <p>Requester CIDRs</p> <p>10.1.0.0/16</p> <p>Requester Region</p> <p>Stockholm (eu-north-1)</p>	<p>VPC Peering connection ARN</p> <p>arn:aws:ec2:eu-north-1:377721963177:vpc-peering-connection/pcx-084a2f471c694f98b</p> <p>Accepter VPC</p> <p>vpc-0e69c4b80354b0e63 / My_Network_VPC</p> <p>Accepter CIDRs</p> <p>10.0.0.0/16</p> <p>Accepter Region</p> <p>Stockholm (eu-north-1)</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

[DNS](#) | [Route tables](#) | [Tags](#)

Updating Route Tables for VPC Peering

- After activating the peering connection, the two VPCs were *allowed* to communicate — but they still **could not send traffic to each other yet**.
- This is because **VPC peering does NOT automatically update route tables**. Each VPC must be told *where* to send traffic intended for the other VPC.
- In simple words:
Peering creates the link.
Route tables decide the path.
 Without adding routes, the instances inside the VPCs do not know how to reach each other's IP ranges.
- To fix this, I opened **Route Tables** in the VPC console and selected the route table for my first VPC.
- Then I clicked **Edit Routes** to add a new entry pointing to the peering connection.

Route tables (2/5) [Info](#) Last updated less than a minute ago [Actions](#) [Create route table](#)

🔍 Find route tables by attribute or tag

<input type="checkbox"/>	Name	Route table ID	Explicit subnet associ...	Edge associations	Main	VPC	Owner ID
<input type="checkbox"/>	My_Private_RTable	rtb-0650e5a316b4c94bd	subnet-09751e0e45a3b4...	–	No	vpc-0e69c4b80354b0e63 My_...	377721963177
<input type="checkbox"/>	–	rtb-0c36dd84a1f65c88	–	–	Yes	vpc-0d207d25eff661a9c My_...	377721963177
<input checked="" type="checkbox"/>	My_Public_RTable_1	rtb-044ca0465245c2040	subnet-074f0c2844543fe...	–	No	vpc-0d207d25eff661a9c My_...	377721963177
<input checked="" type="checkbox"/>	My_Public_RTable	rtb-05b130f2f43879211	subnet-024df28332ec19...	–	Yes	vpc-0e69c4b80354b0e63 My_...	377721963177
<input type="checkbox"/>	–	rtb-06655d504ed416bba	–	–	Yes	vpc-0b5621f1aac817a13	377721963177

Setting Routes for Cross-VPC Communication

- Each route table belongs to a different VPC, which means they control traffic **inside** their own VPC only.
- To make both VPCs talk to each other, I had to **manually create a path** in each route table that points to the peering connection.
- Without these routes, even though the peering link exists, the instances won't know how to reach the other VPC's IP range.

- So I opened the route table for the first VPC and prepared to add a route that sends traffic to the **other VPC's CIDR block** through the **peering connection**.

Edit routes

Destination	Target	Status	Propagated	Route Origin	
10.1.0.0/16	local	Active	No	CreateRouteTable	
0.0.0.0/0	Internet Gateway	Active	No	CreateRoute	Remove
10.0.0.0/16	Peering Connection	-	No	CreateRoute	Remove

Buttons: Add route, Cancel, Preview, Save changes

Adding Routes to Both VPCs

- My **My_Public_RTable1** belonged to **My_Network_VPC1**, which uses the CIDR block **10.1.0.0/16**.
- My **My_Public_RTable** belonged to **My_Network_VPC**, which uses the CIDR block **10.0.0.0/16**.
- I opened **My_Public_RTable1** first and added a new route.
 - **Destination:** 10.0.0.0/16 (the CIDR block of the second VPC)
 - **Target:** the VPC peering connection
This tells VPC1 that any traffic meant for 10.0.0.0/16 should be sent through the peering link.
- Then I opened **My_Public_RTable** and did the exact same thing in reverse.
 - **Destination:** 10.1.0.0/16 (the CIDR block of the first VPC)
 - **Target:** the same peering connection
This allows VPC2 to send traffic back to VPC1.
- After adding these routes, both VPCs now had a **two-way private path** for communication through the peering connection.

rtb-044ca0465245c2040 / My_Public_Rtable_1

Actions

Details Info

Route table ID rtb-044ca0465245c2040	Main No	Explicit subnet associations subnet-074f0c2844543fecc / My_Network_VPC_1-subnet-public1-eu-north-1a	Edge associations -
VPC vpc-0d207d25eff661a9c My_Network_VPC_1~vpc	Owner ID 377721963177		

Routes (3)

Destination	Target	Status	Propagated	Route Origin
0.0.0.0/0	igw-021d9da9d707b28e0	Active	No	Create Route
10.0.0.0/16	pcx-084a2f471c694f98b	Active	No	Create Route
10.1.0.0/16	local	Active	No	Create Route Table

Why We Must Add the Other VPC's CIDR Block (Even Though the Route Table Already Has 0.0.0.0/0)

- Even though the public route table already had the route **0.0.0.0/0 → Internet Gateway**, that only sends traffic **to the internet**, not to another VPC.

- The CIDR block **0.0.0.0/0** means “send all *unknown* traffic to the internet,” but the other VPC’s IP range (10.x.x.x) is *not* on the public internet. It exists **inside AWS’s private network**, not outside.
- Because of this, the route table needs a **specific rule** telling it:
“If the destination is the *other* VPC’s private CIDR block, send it through the peering connection, not the Internet Gateway.”
- Without adding the other VPC’s CIDR block, the instance would try to send that traffic to the **Internet Gateway**, which cannot reach private AWS networks. The traffic would simply get lost.
- By adding the correct CIDR block and pointing it to the **peering connection**, we are telling AWS:
“This traffic is meant for another private network — send it through the private link, not the internet.”
- This is why **every VPC peering setup requires explicit route entries** for both VPCs, even when using public route tables.

▼ Network settings [Info](#)

VPC - required [Info](#)

vpc-0d207d25eff661a9c (My_Network_VPC_1-vpc)
10.1.0.0/16

Subnet [Info](#)

subnet-074f0c2844543fecc My_Network_VPC_1-subnet-public1-eu-north-1a
VPC: vpc-0d207d25eff661a9c Owner: 377721963177 Availability Zone: eu-north-1a (eun1-az1)
Zone type: Availability Zone IP addresses available: 4091 CIDR: 10.1.0.0/20

Auto-assign public IP [Info](#)

Disable

Firewall (security groups) [Info](#)

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

☐ Create security group ☒ Select existing security group

Common security groups [Info](#)

Select security groups

default sg-08c97f68f6676e1d7
VPC: vpc-0d207d25eff661a9c

[Compare security group rules](#)

Security groups that you add or remove here will be added to or removed from all your network interfaces.

► Advanced network configuration

Creating EC2 Instances in Both VPCs for Peering Tests

- To test VPC peering, I needed to run an EC2 instance **inside each VPC**. This allows me to send traffic from one VPC to the other using private IP addresses.
- First, I created an EC2 instance and named it **My_EC2_1**.
 - I selected **My_Network_VPC1** as the VPC.
 - I chose the **public subnet** of that VPC.
 - I **disabled Auto-assign Public IP**, because for peering tests we only want **private-to-private** communication.
 - For the security group, I selected the **default SG** that was automatically created when the VPC was created.
- Then I created another EC2 instance in the **second VPC** and named it **My_EC2**.
 - I placed it inside **My_Network_VPC**.
 - I used the public subnet of this VPC as well.

- I applied the **same settings**: no public IP and the **default Security Group** of its VPC.
- At this point, each VPC had its own EC2 instance, ready to test **cross-VPC communication** through the peering connection.

Instances (2) [Info](#)

Find Instance by attribute or tag (case-sensitive) All states Last updated less than a minute ago Connect Instance state Actions Launch instances

<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv6 DNS
<input type="checkbox"/>	My_EC2_1	i-0159eb3f5df6b86db	Running	t3.micro	Initializing	View alarms +	eu-north-1a	-	-
<input type="checkbox"/>	My_EC2	i-0f8c774560fae2c49	Running	t3.micro	3/3 checks passed	View alarms +	eu-north-1a	-	-

Connection Error – No Public IP

- After both instances were created, I selected one of them and tried to connect using the AWS **Connect** button.
- However, I received an error saying that the instance has **no public IP address**.

[EC2](#) > [Instances](#) > [i-0159eb3f5df6b86db](#) > [Connect to instance](#)

Connect [Info](#)

Connect to an instance using the browser-based client.

[EC2 Instance Connect](#) | [Session Manager](#) | [SSH client](#) | [EC2 serial console](#)

No public IPv4 or IPv6 address assigned

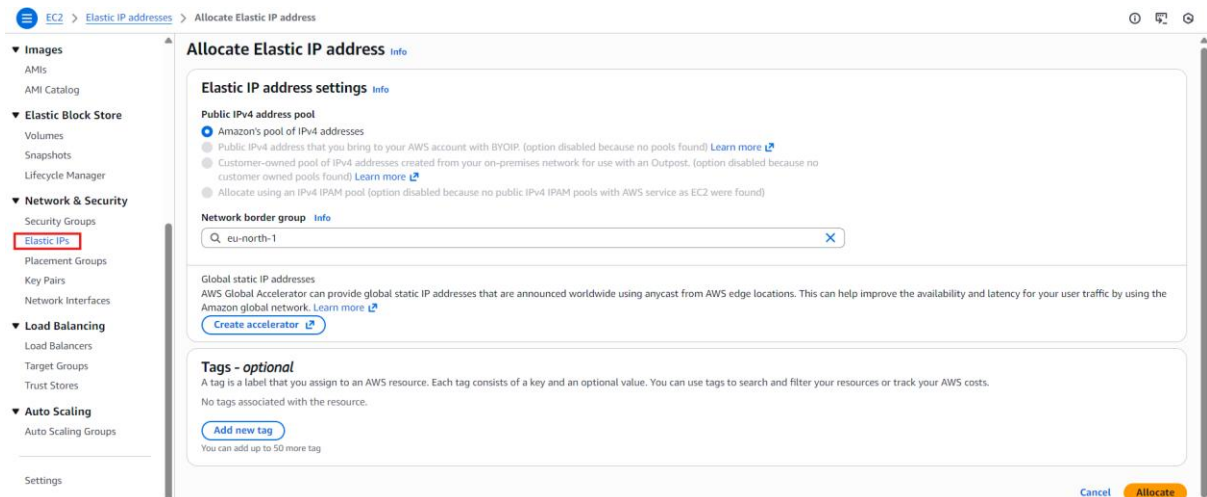
With no public IPv4 or IPv6 address, you can't use EC2 Instance Connect. Alternatively, you can try connecting using [EC2 Instance Connect Endpoint](#).

Instance ID
[i-0159eb3f5df6b86db](#) (My_EC2_1)

- The reason is simple:
The AWS browser-based **EC2 Connect** method requires a **public IP** so AWS can reach the instance over the internet.
Without a public IP, the instance cannot be accessed directly from the AWS console or the outside world.
- Since I purposely **disabled Auto-assign Public IP** on both instances (because peering tests use only private IPs), the console connection method could not work.

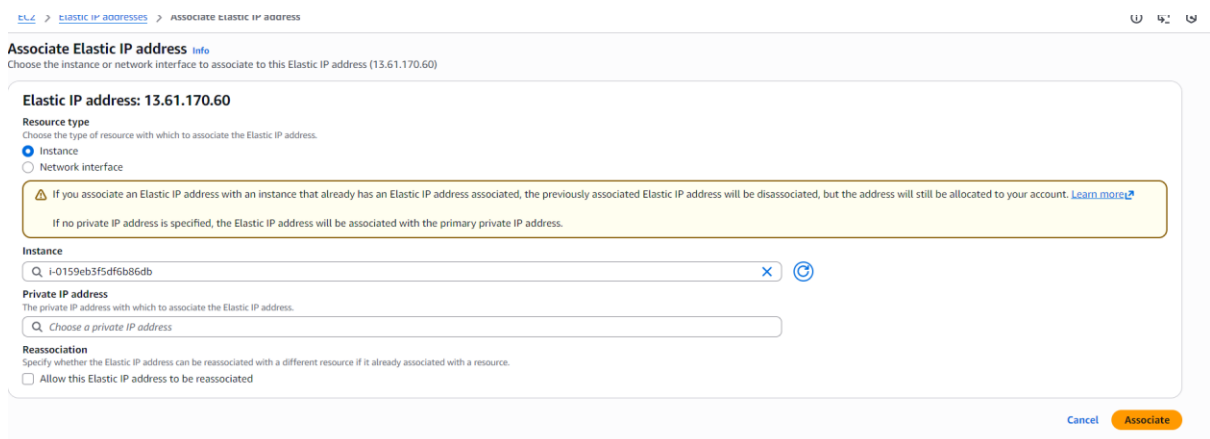
What Are Elastic IPs?

- **Elastic IPs are static IPv4 addresses** that AWS allocates to your account.
 - “Static” means the IP stays the **same**, unlike the **dynamic public IPs** that EC2 instances normally get.
 - When an EC2 instance restarts, its normal public IP changes — but an Elastic IP **does not**.
 - Having an Elastic IP is like having a **permanent address**, instead of moving to a new house every time the instance restarts.
-



Allocating an Elastic IP

- In the EC2 console, under **Network & Security**, I selected **Elastic IPs**.
- Then I clicked **Allocate Elastic IP address**.
- I confirmed the **correct border group** (this ensures the IP is allocated in the right AWS network location).
- Finally, I selected **Amazon's IPv4 address pool** and allocated the IP.



Associating the Elastic IP

- After allocating the Elastic IP, the next step was to **associate** it with the EC2 instance I wanted to connect to.
- I selected the Elastic IP from the list and clicked **Actions** → **Associate Elastic IP address**.
- In the association window, I chose the instance **My_Instance_VPC1**, which was the instance that needed a public, static IP for connection.
- I confirmed the settings and completed the association.
- Once associated, **My_Instance_VPC1** instantly received the Elastic IP, making it reachable from the internet.

Instance summary for i-0159eb3f5df6b86db (My_EC2_1) [Info](#)

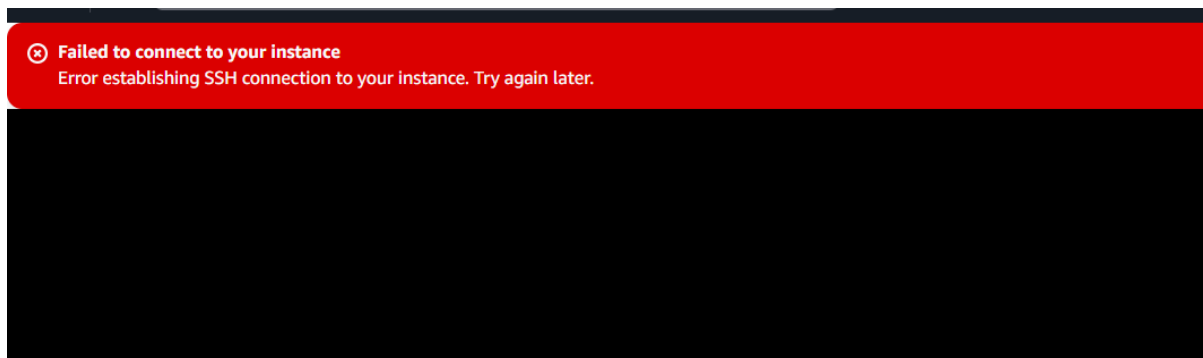
Updated less than a minute ago

[Refresh](#)
[Connect](#)
[Instance state ▼](#)
[Actions ▼](#)

Instance ID i-0159eb3f5df6b86db	Public IPv4 address 13.61.170.60 open address ↗	Private IPv4 addresses 10.1.14.101
IPv6 address —	Instance state Running	Public DNS ec2-13-61-170-60.eu-north-1.compute.amazonaws.com open address ↗
Hostname type IP name: in-10-1-14-101.eu-north-1.comoute.internal	Private IP DNS name (IPv4 only) in-10-1-14-101.eu-north-1.comoute.internal	

Elastic IP Successfully Assigned

- After associating the Elastic IP with **My_Instance_VPC1**, I checked the instance details.
- I could clearly see that the instance had now received a **public IP address**.
- This public IP is the **Elastic IP** that was assigned to me from the Amazon IP pool.
- Because it is an Elastic IP, it will **remain the same** even if the instance is stopped or restarted.
- With this public IP in place, the instance became fully reachable from the internet.



SSH Connection Issue

- After assigning the Elastic IP, I tried connecting to the instance again.
- This time, instead of the “no public IP” error, I ran into a **new SSH error**.
- The browser-based terminal couldn’t establish an SSH connection with the instance.
- This meant the issue was no longer about the public IP — it was now related to **security group settings, firewall rules, or SSH ports** not being open.
- So I had to troubleshoot the SSH configuration to understand what was blocking the connection.

Edit inbound rules [Info](#)

Inbound rules control the incoming traffic that's allowed to reach the instance.

Inbound rules [Info](#)

Security group rule ID: sgr-03f4a46435d97e6a6

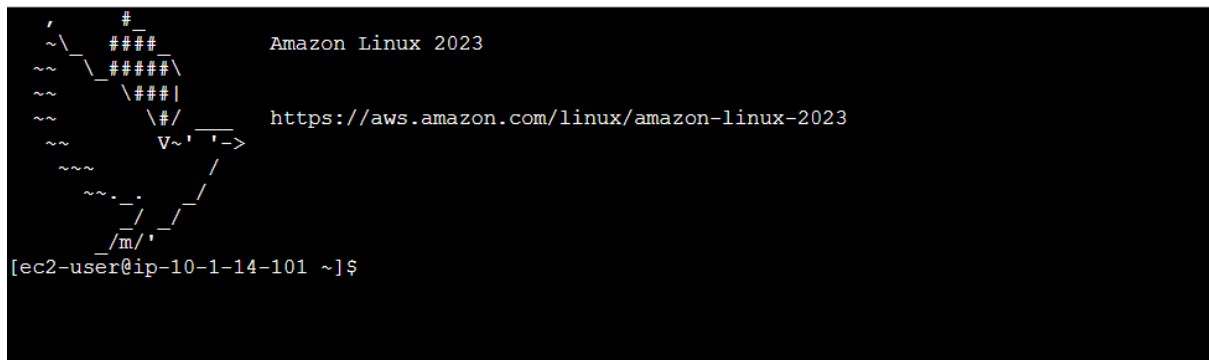
Type	Protocol	Port range	Source	Description - optional	
All traffic	All	All	Custom	Q	Delete
SSH	TCP	22	Anywhere...	Q sg-08c97f68f6676e1d7	Delete
				Q 0.0.0.0/0	Delete

[Add rule](#)

[Cancel](#)
[Preview changes](#)
[Save rules](#)

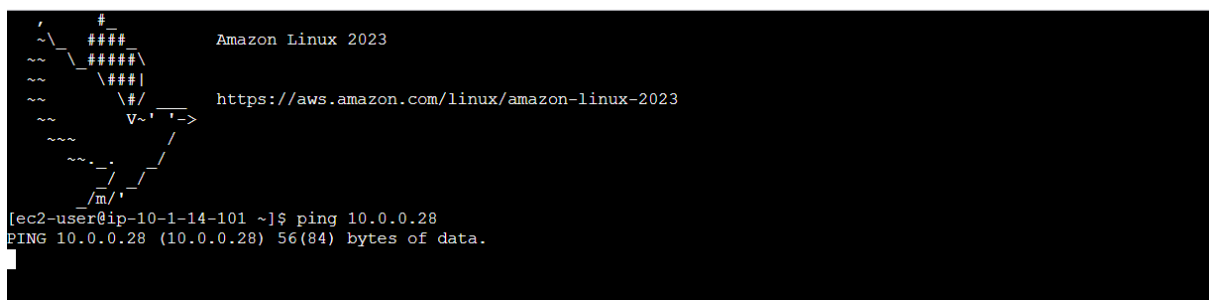
Fixing the SSH Connection Issue

- First, I checked the **Network ACL**, and everything looked correct. Both inbound and outbound rules allowed the traffic that SSH needs.
- Then I checked the **Security Group** attached to the instance. By default, the SG allowed all traffic **only from a specific security group**, not from the internet.
- Because of this, SSH (port 22) was **not accessible from my public IP** or from anywhere outside that SG.
- To fix it, I added a new **Inbound Rule**:
 - **Type:** SSH
 - **Port:** 22
 - **Source:** 0.0.0.0/0 (allows connection from anywhere for testing)
- After saving the rule, the instance became reachable again through SSH using the Elastic IP.



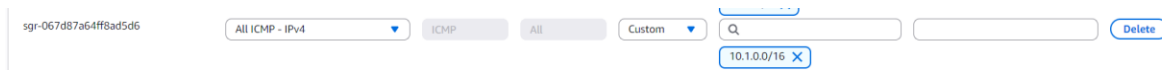
Connection Successfully Established

- After updating the security group and allowing SSH from anywhere, I retried the connection.
- This time, the SSH session opened successfully on **My_EC2_1**.
- The Elastic IP, route settings, NACL, and security group were now correctly configured, allowing the instance to be accessed without any issues.



Initial VPC Peering Ping Test

- After successfully connecting to **My_EC2_1**, I tested cross-VPC connectivity.
- I tried to **ping the private IP** of the second instance (**My_EC2**) located in the other VPC.
- Only **one ping was delivered**, and then the replies stopped completely.
- This meant that although the VPC peering connection existed, something was still **blocking the return traffic** between the two instances.
- I had to troubleshoot the security layers (SG + NACL) to find out what was preventing full communication across the VPCs.



Fixing the ICMP Issue in the Second VPC

- After checking the setup, I discovered that the **Security Group** of the EC2 instance in the second VPC had **no ICMP rules** at all.
- This meant the instance was **not allowing ping (ICMP) traffic in or out**, which explained why only one ping was delivered and no replies were sent back.
- To fix this, I added a new **Inbound Rule**:
 - **Type:** ICMP – IPv4
 - **Source:** **10.1.0.0/16** (the CIDR block of VPC1)
- This ensured that only traffic **coming from the first VPC** was allowed, keeping the setup secure while still enabling cross-VPC communication.
- After saving the changes, the instance in the second VPC was now able to receive and respond to ICMP traffic from the first VPC.

```
64 bytes from 10.0.0.28: icmp_seq=326 ttl=127 time=0.184 ms
64 bytes from 10.0.0.28: icmp_seq=327 ttl=127 time=0.182 ms
64 bytes from 10.0.0.28: icmp_seq=328 ttl=127 time=0.174 ms
64 bytes from 10.0.0.28: icmp_seq=329 ttl=127 time=0.155 ms
64 bytes from 10.0.0.28: icmp_seq=330 ttl=127 time=0.181 ms
64 bytes from 10.0.0.28: icmp_seq=331 ttl=127 time=0.190 ms
64 bytes from 10.0.0.28: icmp_seq=332 ttl=127 time=0.174 ms
64 bytes from 10.0.0.28: icmp_seq=333 ttl=127 time=0.180 ms
64 bytes from 10.0.0.28: icmp_seq=334 ttl=127 time=0.223 ms
64 bytes from 10.0.0.28: icmp_seq=335 ttl=127 time=0.175 ms
64 bytes from 10.0.0.28: icmp_seq=336 ttl=127 time=0.194 ms
64 bytes from 10.0.0.28: icmp_seq=337 ttl=127 time=0.172 ms
64 bytes from 10.0.0.28: icmp_seq=338 ttl=127 time=0.158 ms
64 bytes from 10.0.0.28: icmp_seq=339 ttl=127 time=0.174 ms
64 bytes from 10.0.0.28: icmp_seq=340 ttl=127 time=0.181 ms
64 bytes from 10.0.0.28: icmp_seq=341 ttl=127 time=0.171 ms
64 bytes from 10.0.0.28: icmp_seq=342 ttl=127 time=0.186 ms
64 bytes from 10.0.0.28: icmp_seq=343 ttl=127 time=0.177 ms
64 bytes from 10.0.0.28: icmp_seq=344 ttl=127 time=0.186 ms
64 bytes from 10.0.0.28: icmp_seq=345 ttl=127 time=0.175 ms
64 bytes from 10.0.0.28: icmp_seq=346 ttl=127 time=0.168 ms
64 bytes from 10.0.0.28: icmp_seq=347 ttl=127 time=0.239 ms
64 bytes from 10.0.0.28: icmp_seq=348 ttl=127 time=0.168 ms
64 bytes from 10.0.0.28: icmp_seq=349 ttl=127 time=0.167 ms
64 bytes from 10.0.0.28: icmp_seq=350 ttl=127 time=0.170 ms
64 bytes from 10.0.0.28: icmp_seq=351 ttl=127 time=0.161 ms
64 bytes from 10.0.0.28: icmp_seq=352 ttl=127 time=0.191 ms
64 bytes from 10.0.0.28: icmp_seq=353 ttl=127 time=0.202 ms
64 bytes from 10.0.0.28: icmp_seq=354 ttl=127 time=0.167 ms
64 bytes from 10.0.0.28: icmp_seq=355 ttl=127 time=0.180 ms
64 bytes from 10.0.0.28: icmp_seq=356 ttl=127 time=0.170 ms
64 bytes from 10.0.0.28: icmp_seq=357 ttl=127 time=0.169 ms
```

i-0159eb3f5df6b86db (My_EC2_1)

PublicIPs: 13.61.170.60 PrivateIPs: 10.1.14.101

Successful VPC Peering Connectivity

- After updating the Security Group in the second VPC to allow ICMP traffic from **10.1.0.0/16**, I tested the ping again from **My_EC2_1**.
- This time, the ping replies started coming through continuously without stopping.
- This confirmed that the instances in both VPCs were now able to **communicate privately** using the VPC peering connection.
- With this, the cross-VPC connectivity was successfully established and verified.

Conclusion

This project helped me understand how to design and troubleshoot communication between multiple VPCs using **VPC Peering**. I learned how to set up two isolated VPCs, connect them through a private peering link, configure routing paths, and adjust security layers like NACLs and Security Groups to allow cross-VPC traffic.

By launching EC2 instances in each VPC and testing connectivity with private IPs, I saw first-hand how traffic flows through the peering connection and how AWS keeps this communication entirely **private**, without using the public internet.

Overall, this project strengthened my understanding of:

- Multi-VPC architectures
- Route table configuration for private paths
- Security controls that affect cross-VPC traffic
- Troubleshooting network issues across isolated environments

This hands-on experience gave me a deeper understanding of how real-world cloud networks communicate securely inside AWS.