3-tier VPC setup on AWS.

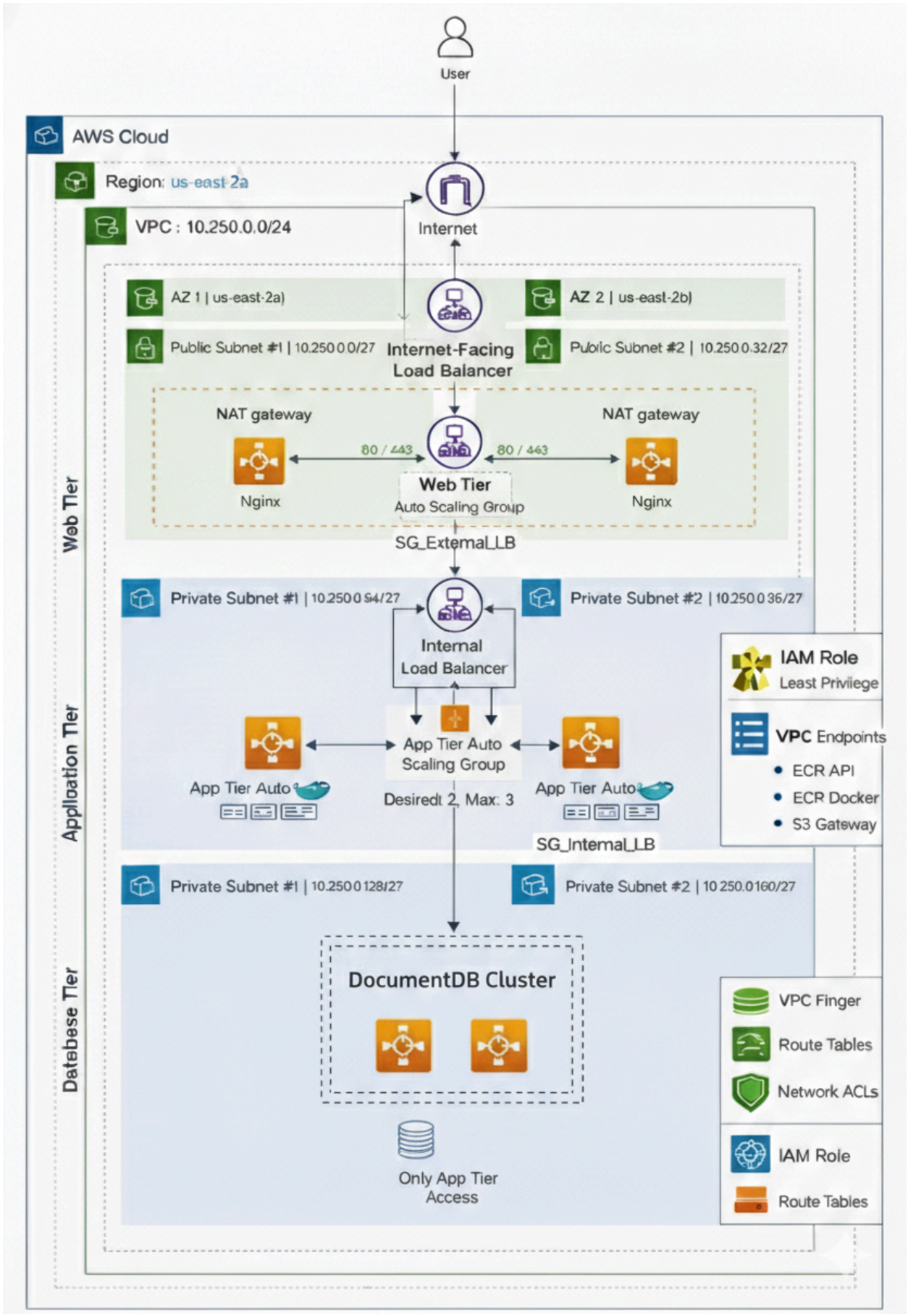
Goal:

The objective of this project is to design and deploy a **secure, highly available, and scalable three-tier application architecture on AWS** using a **custom Virtual Private Cloud (VPC)**.

The solution is architected with **clearly defined public and private subnets** to enforce strong network isolation, and leverages **Multi-Availability Zone (Multi-AZ) deployments** along with **Auto Scaling Groups (ASG)** to achieve resilience, fault tolerance, and horizontal scalability. **Layered security controls** are implemented using **Security Groups and Network ACLs** to strictly regulate inbound and outbound traffic between application tiers.

Access to AWS resources is governed through **IAM roles and policies** following the **principle of least privilege**, ensuring secure, auditable, and controlled access. The architecture is designed in alignment with **AWS production best practices**, emphasizing **security, availability, operational reliability, and future extensibility**.

Architecture Diagram:


User
AWS Cloud
Region: us-east-2a
VPC : 10.250.0.0/24
Internet
AZ 1 | us-east-2a)
AZ 2 | us-east-2b)
Public Subnet #1 | 10.250.0.0/27
Internet-Facing Load Balancer
Public Subnet #2 | 10.250.0.32/27
NAT gateway
NAT gateway
80 / 443
80 / 443
Nginx
Web Tier
Auto Scaling Group
Nginx
SG_External_LB
Private Subnet #1 | 10.250.0.64/27
Private Subnet #2 | 10.250.0.96/27
Internal Load Balancer
IAM Role
Least Privilege
VPC Endpoints
ECR API
ECR Docker
S3 Gateway
App Tier Auto Scaling Group
App Tier Auto
App Tier Auto
Desiredt 2, Max: 3
SG_Internal_LB
Private Subnet #1 | 10.250.0.128/27
Private Subnet #2 | 10.250.0.160/27
DocumentDB Cluster
VPC Finger
Route Tables
Network ACLs
IAM Role
Route Tables
Only App Tier Access
Web Tier
Application Tier
Database Tier

Tier	Purpose	Placement
Web Tier	Frontend (Nginx) + External Load Balancer	Public Subnets
Application Tier	Backend services (Docker containers) + Internal Load Balancer	Private Subnets
Database Tier	MongoDB (Amazon DocumentDB)	Private Subnets

Step 1: Creation of VPC

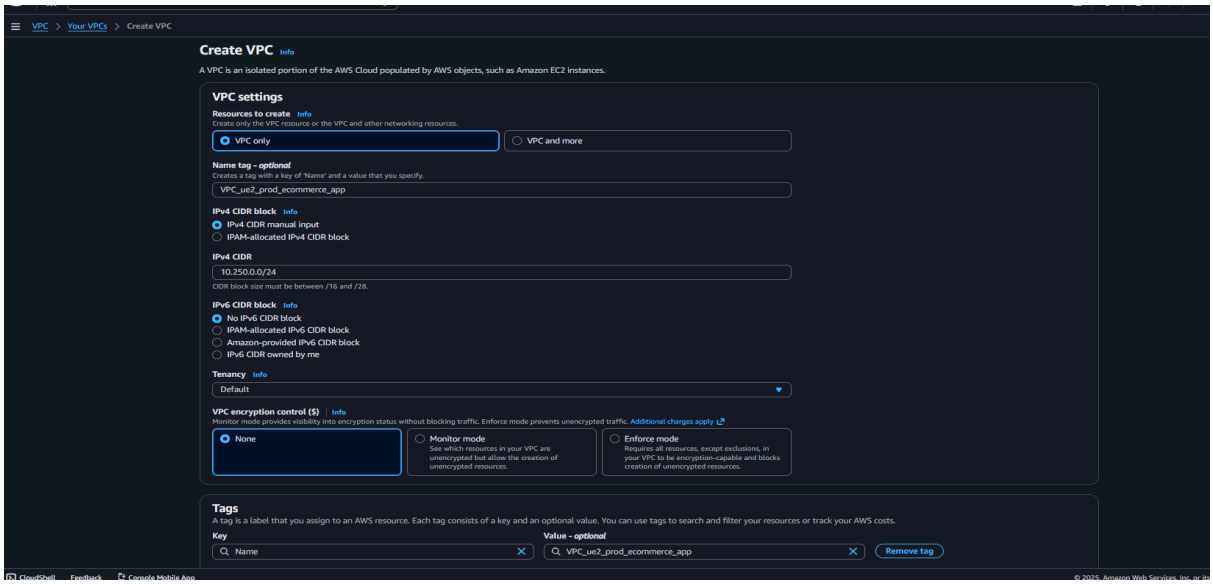
A **Virtual Private Cloud (VPC)** is an AWS networking service that enables the creation of a **logically isolated virtual network** within the AWS cloud. It allows full control over network configuration by defining a **custom IPv4 CIDR block**, which forms the foundation for subnetting, routing, and security boundaries.

In this project, a **custom VPC** is created to serve as the **core networking layer** for the three-tier architecture, enabling secure communication between application components while maintaining isolation from other AWS environments.

Parameter	Value
VPC Name	VPC_ue2_prod_ecommerce_app
Region	us-east-2
IPv4 CIDR	10.250.0.0/24

Naming Convention as per AWS recommends:

```
vpc-RegionCode-EnvironmentCode-ApplicationStackCode
```



Step 2: Subnet Design & Creation

Subnet: A subnet is the subdivision of a network. It is the range of IP addresses in your VPC

Subnet Strategy

- **2 Public Subnets** (Web Tier)
  2 Private Subnets (Application Tier)
- **2 Private Subnets** (Database Tier)
- Distributed across **2 Availability Zones** for high availability

AWS Reserved IP Addresses

In every subnet:

- First **4 IPs** and the **last IP** are reserved by AWS

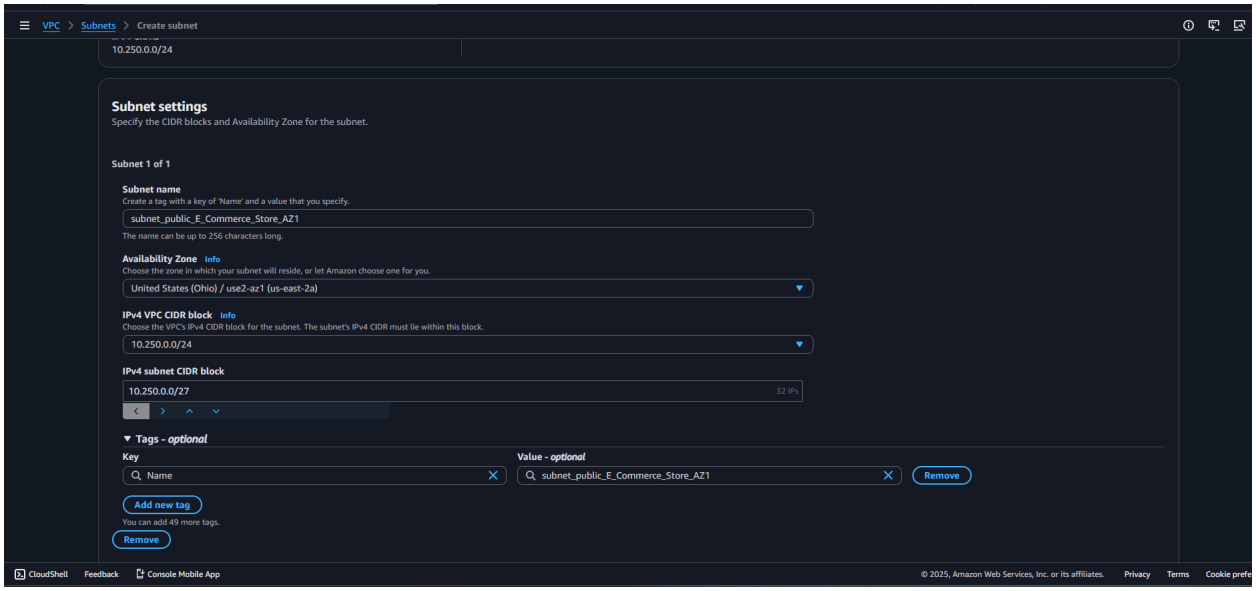
Example (10.250.0.32/27):

```
.32 → Network Address
.33 → VPC Router
```

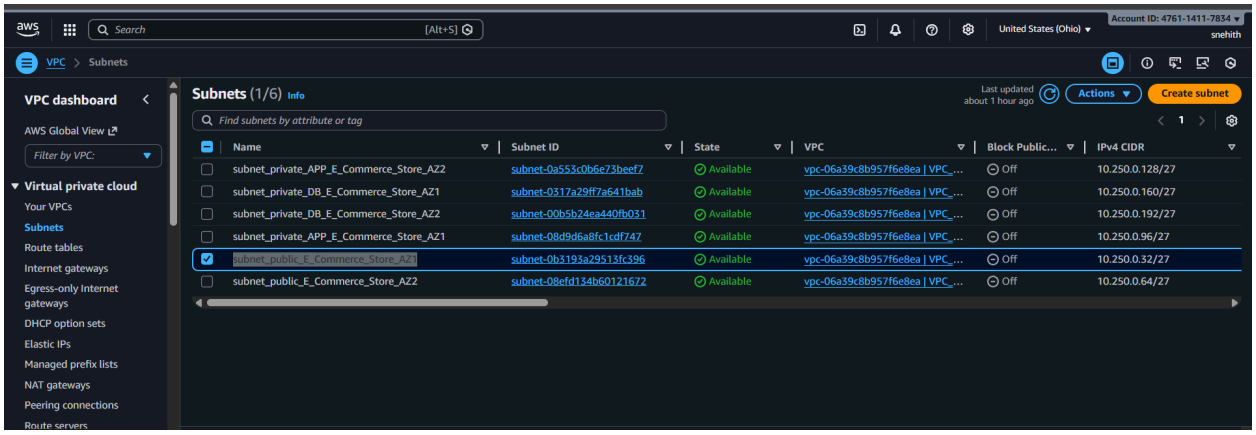
```
.34 → Amazon DNS
.35 → AWS Reserved
.63 → Broadcast
```

Subnet Allocation

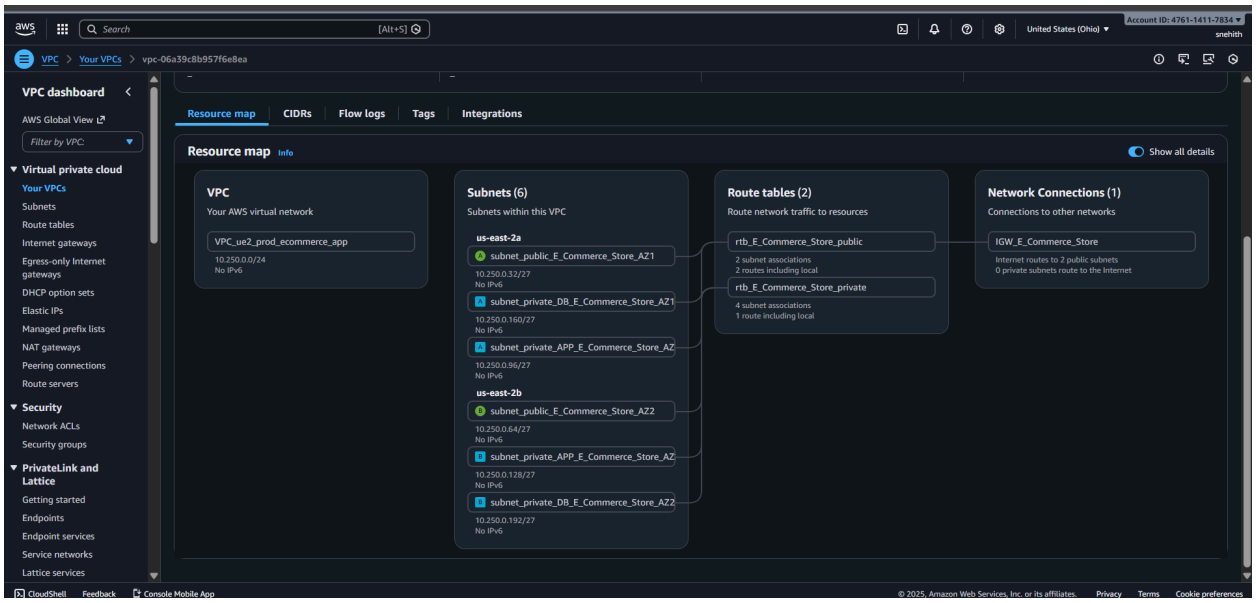
Subnet	CIDR	AZ
Public-AZ1	10.250.0.0/27	us-east-2a
Public-AZ2	10.250.0.32/27	us-east-2b
Private-App-AZ1	10.250.0.64/27	us-east-2a
Private-App-AZ2	10.250.0.96/27	us-east-2b
Private-DB-AZ1	10.250.0.128/27	us-east-2a
Private-DB-AZ2	10.250.0.160/27	us-east-2b



Create the remaining 5 subnets in the same way.



VPC Resource map:



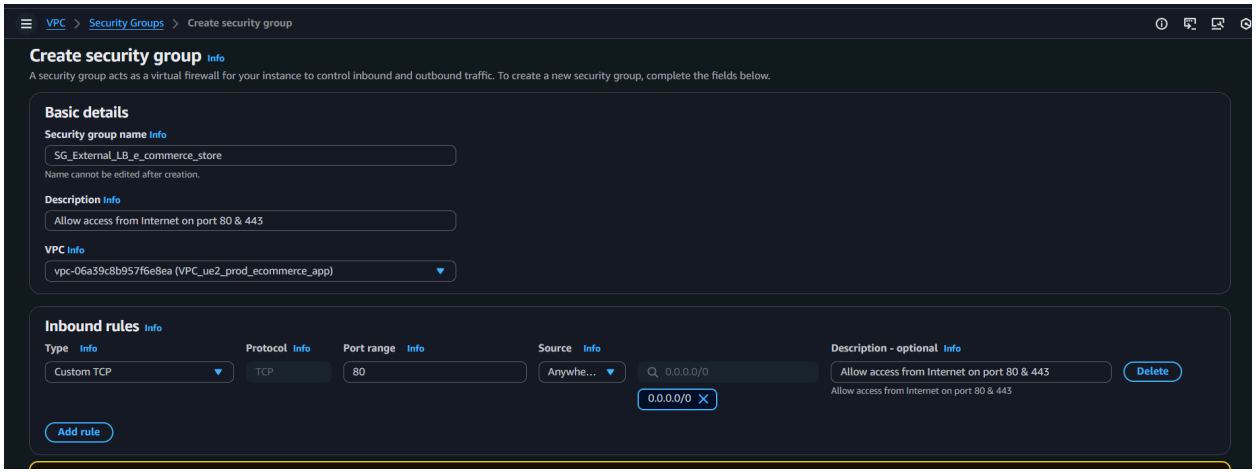
Step 3: Security Group Design

Security Groups enforce **resource-level access control**.

Security Groups Used

Security Group	Purpose
SG_External_LB_e_commerce_store	Allow Internet traffic on 80/443
SG_web_e_commerce_store	Allow traffic only from External LB (port 80)
SG_Internal_LB_e_commerce_store	Allow traffic from Web Tier (8080/4443)
SG_APP_e_commerce_store	Allow traffic only from Internal LB

SG_DB_e_commerce_store	Allow traffic only from App Tier to DB



Step 4: Database Tier – Amazon DocumentDB

Why Amazon DocumentDB?

- MongoDB is compatible with our application
- Managed, highly available
- Multi-AZ support

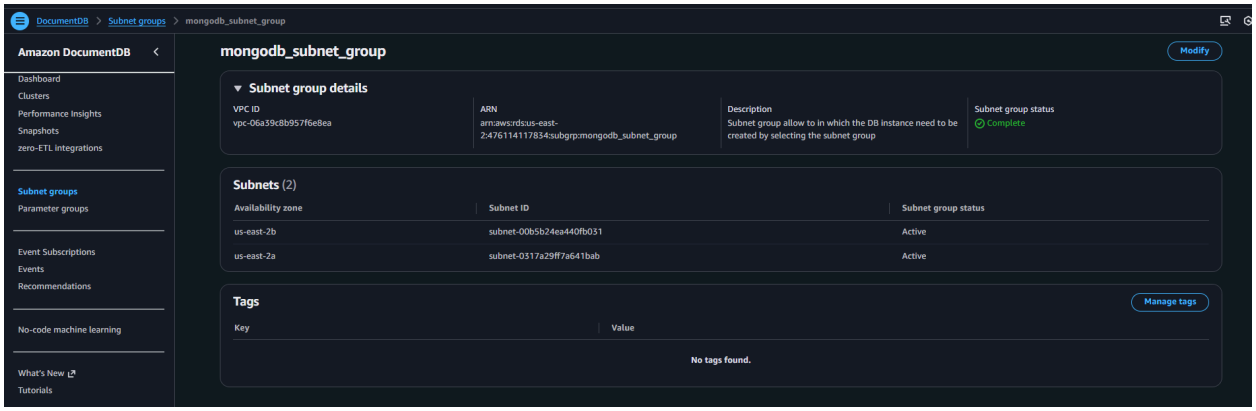
Subnet Group Creation:

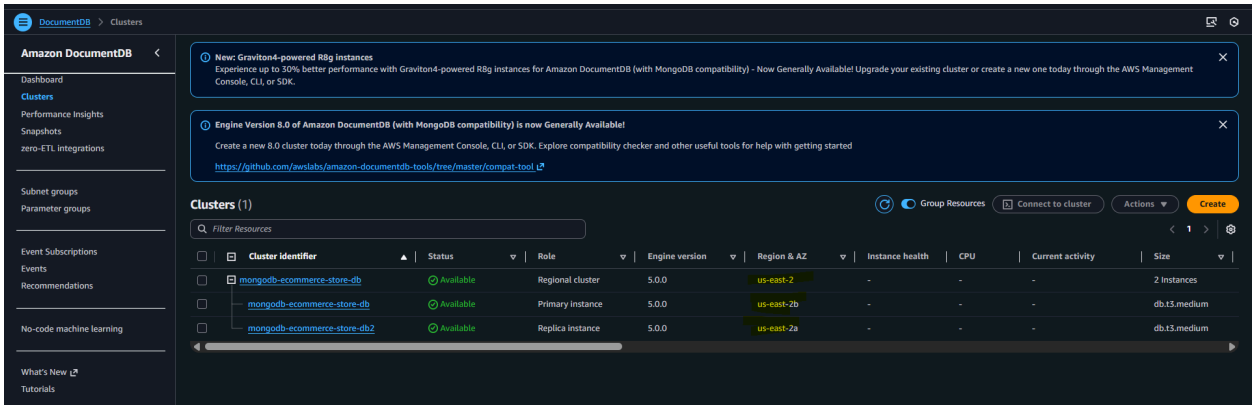
Since DocumentDB requires a subnet group:

- Subnet Group Name: mongodb_subnet_group
- Select VPC
- Add DB private subnets
- Create subnet group

Cluster Configuration

- Cluster identifier: Give a unique name
- Replica Instances: 2 (Multi-AZ)
- Username: Provide username
- Authentication: AWS Secrets Manager
- Security Group: SG_DB_e_commerce_store



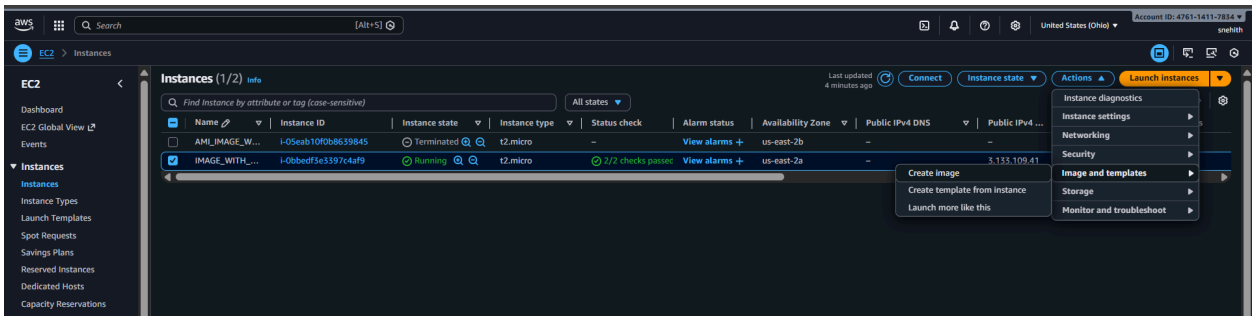


Step 5: Application Tier Setup

1. Creating the AMI (With docker and AWS CLI installed) & creating the Launch template
2. User data (On boot will pull the image from ECR and start the container)
3. Set up the ASG for the auto scaling.

8.1 AMI Creation

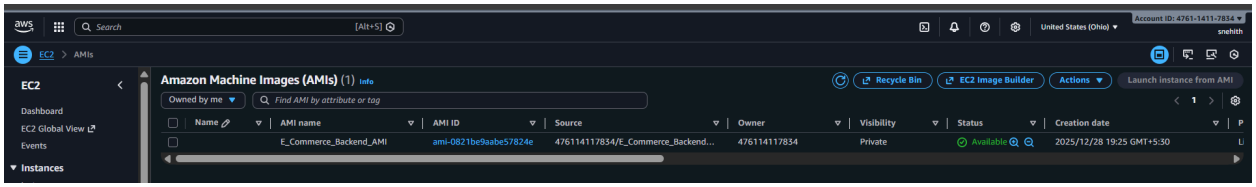
- Launch EC2
- Install:
- Docker
- AWS CLI
- Convert instance to custom AMI



Once AMI is created, create the launch template

8.2 Launch Template

Setting	Value
Template Name	Backend_Template
AMI	Backend AMI
Subnet	Do not include
Key Pair	Do not include
Security Group	SG_APP_e_commerce_store
Attach role:	E_commerce_role



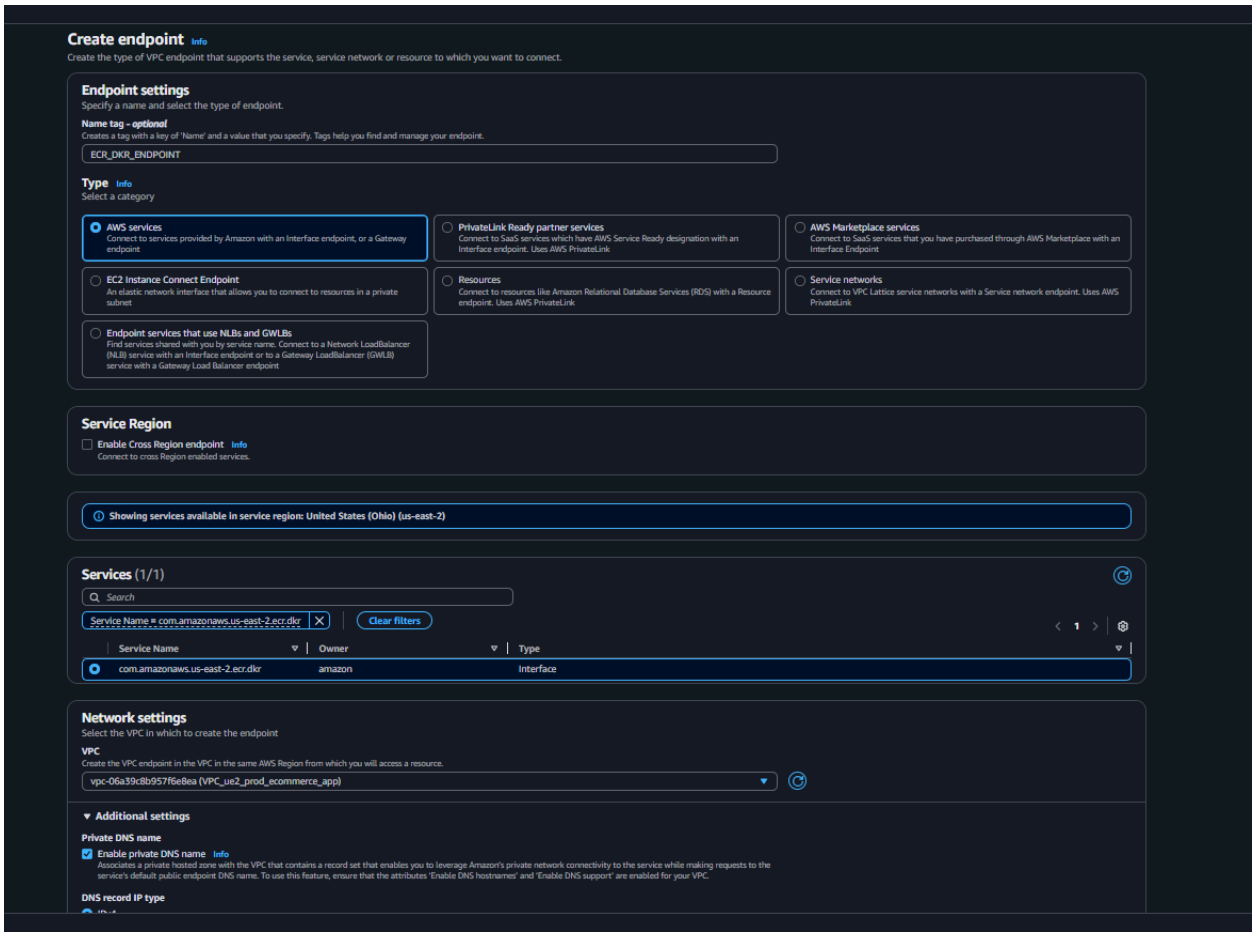
8.3 VPC Endpoints (Private Connectivity)

Since App EC2 instances are private, the following VPC endpoints are created for connecting the ECR to pull/push the image:

- ECR API Endpoint
- ECR Docker Endpoint
- S3 Gateway Endpoint

1. Creating the ECR_API_Endpoint

Setting	Value
Name	ECR_DKR_ENDPOINT
service	com.amazonaws.us-east-2.ecr.api
Security Group	SG_APP_e_commerce_store



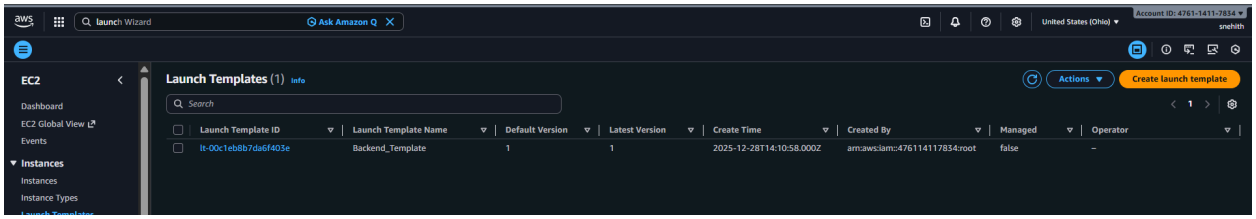
In sameway, create for the other endpoints as well.

8.4 User Data Script (Backend EC2)

Attach below userdata:

```
#!/bin/bash
REGION="us-west-2"
ACCOUNT_ID="476114117834"
ECR_REPO="e_commerce_store"
```

```
IMAGE_TAG="latest"


sudo usermod -aG docker ubuntu

# Login to ECR
aws ecr get-login-password --region $REGION | docker login \
  --username AWS \
  --password-stdin ${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com

# Pull image
docker pull
${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/${ECR_REPO}:${IMAGE_TAG}


# Stop old container if exists
docker rm -f
${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/${ECR_REPO}:${IMAGE_TAG} ||
true



# Run container
docker run -d --name e_commerce_store_backend -p 5000:5000 \
  ${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/${ECR_REPO}:${IMAGE_TAG}
```



Creating the EC2 instance with ASG (for auto scaling feature) and attaching the Internal LB

1.1 Internal Load Balancer & ASG (App Tier)

1.1.0 Creating the target group for attaching the LB

Target Group

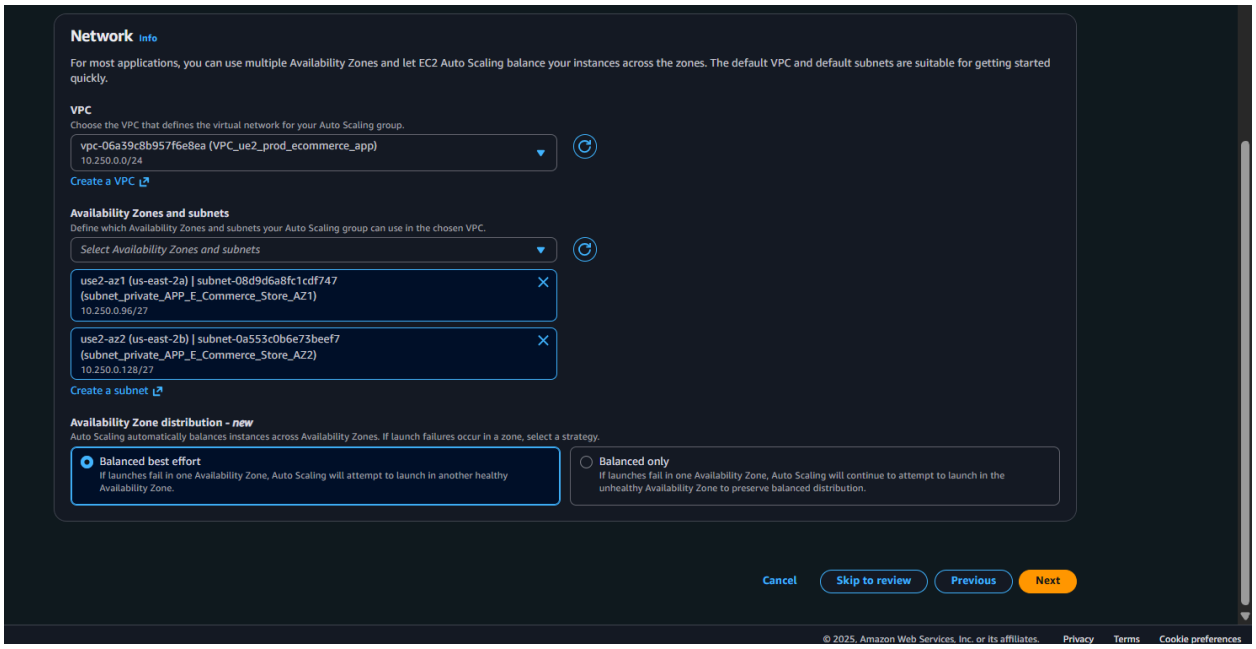
- Name: Internal_LB_Target_Group_For_APP_TIER
- Port: 5000 (Application container port)
- Subnets: App Tier

1.1.1 Creation of Internal Application Load Balancer

- Type: Internal
- Subnets: App Tier
- Security Group: SG_Internal_LB_e_commerce_store

1.2. Create an Auto Scaling group

- Name: APP_TIER_ASG_GROUP
- Select launch Template: Backend_Template
- Under Network (Select VPC and Subnet (private_APP_subnet)

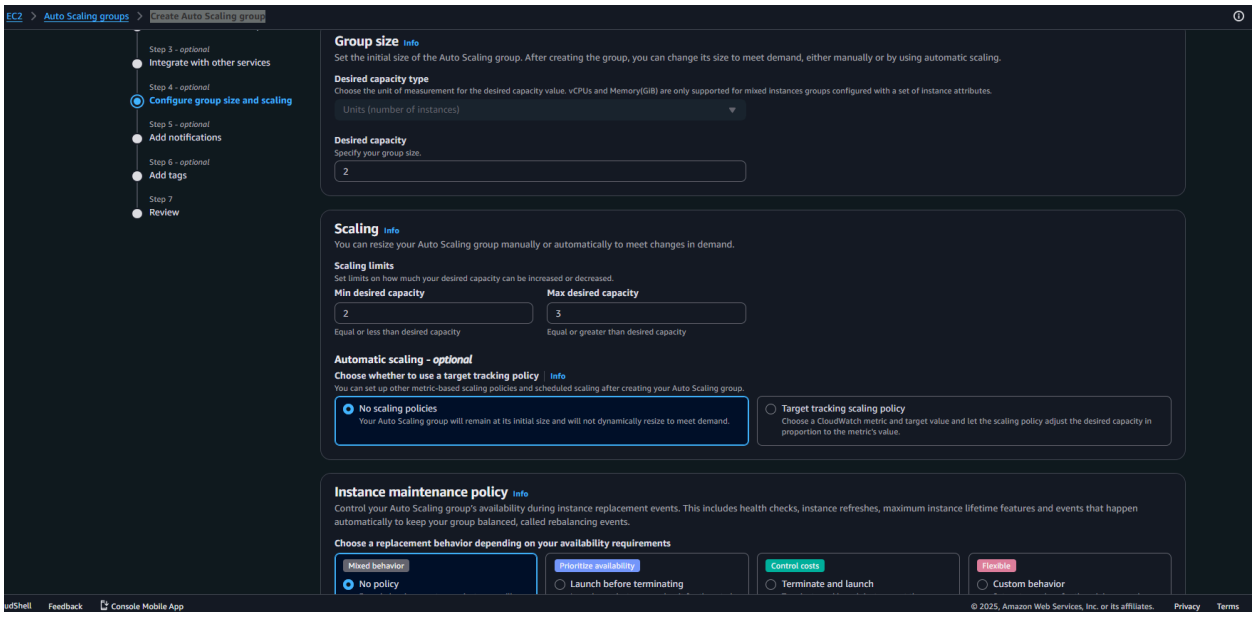


Auto Scaling Group

Parameter	Value
Desired	2
Min	2
Max	3

Subnets	Private App Subnets

Once ASG is created the it will launch the desired number of instance which we have i.e 2 (desired capacity) in the APP_Tier subnet



Web Tier Setup

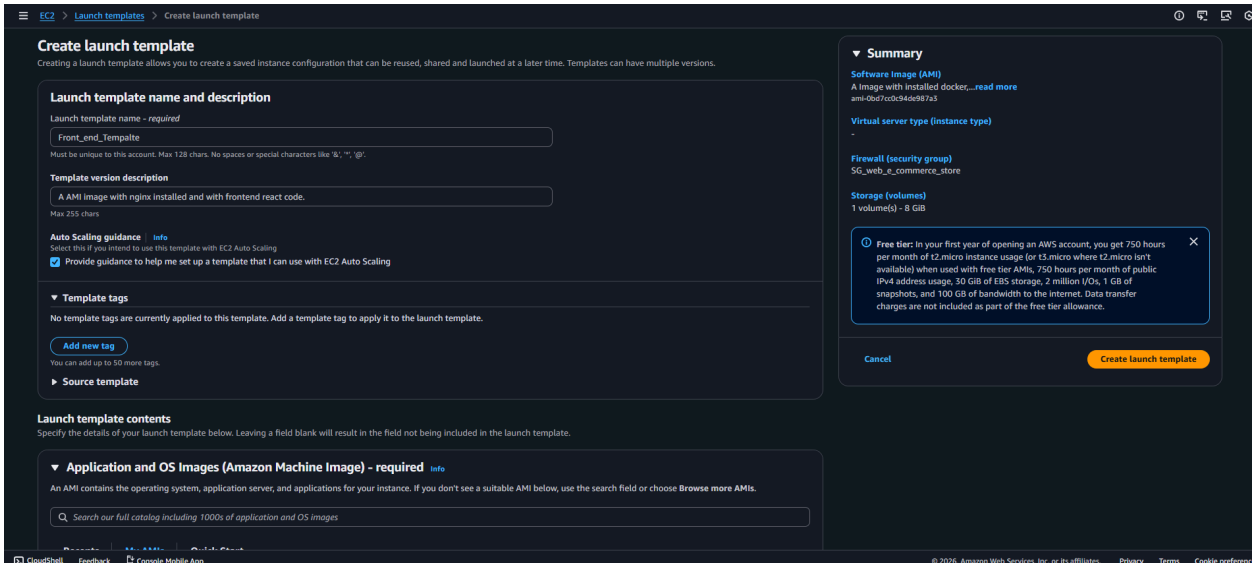
Creating the Web-tier with External Load Balancers

Web AMI

- Install Nginx
- Place frontend code
- Update Nginx configuration
- Create a custom AMI

Nginx config file :

.



External Load Balancer

- Type: Internet Facing
- Port: 80
- Subnets: Public Subnets
- Security Group: SG_External_LB_e_commerce_store

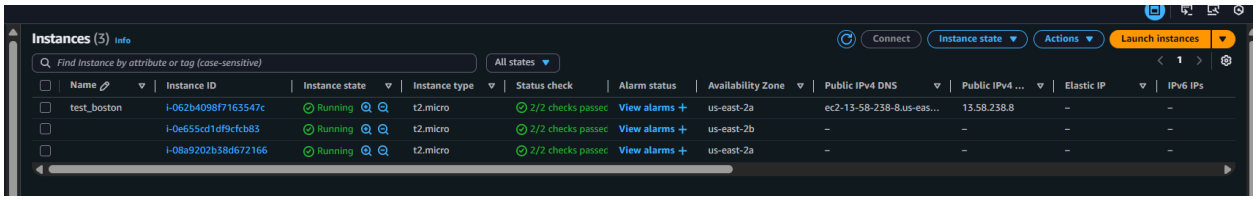
Target Group

- Port: 8081
- VPC: select VPC

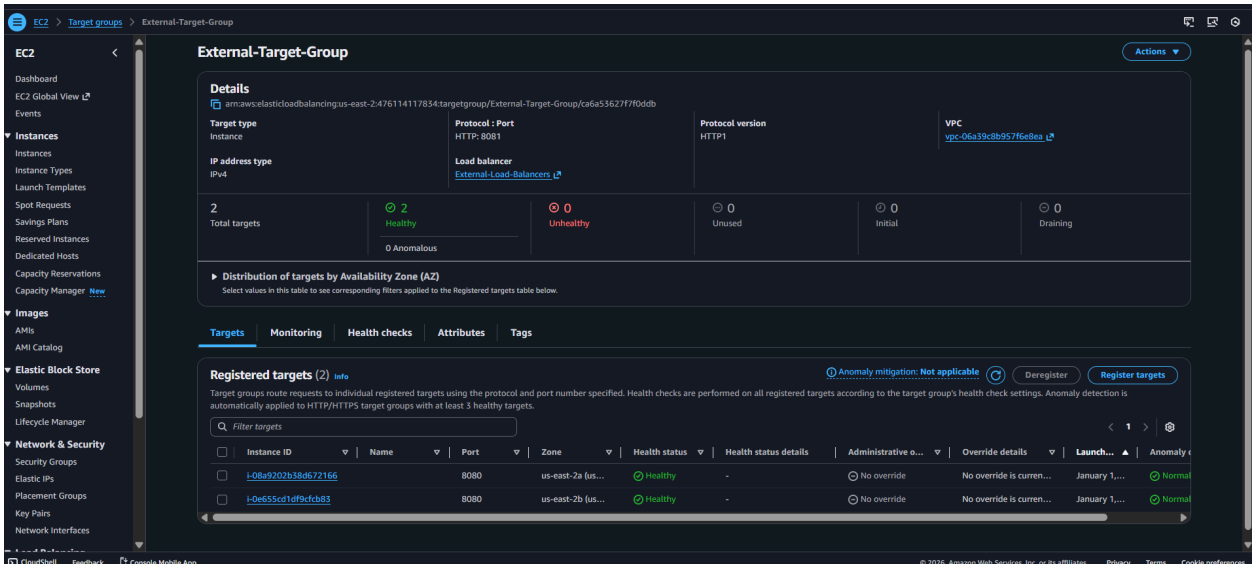
Creation of web_tier_ASG

- Name: Web_Tier_ASG
- Launch template:Front_end_tempate(select)
- Select VPC and public subnets
- Attach External target group: : External-Target-Group

ASG created the two web_tier instances

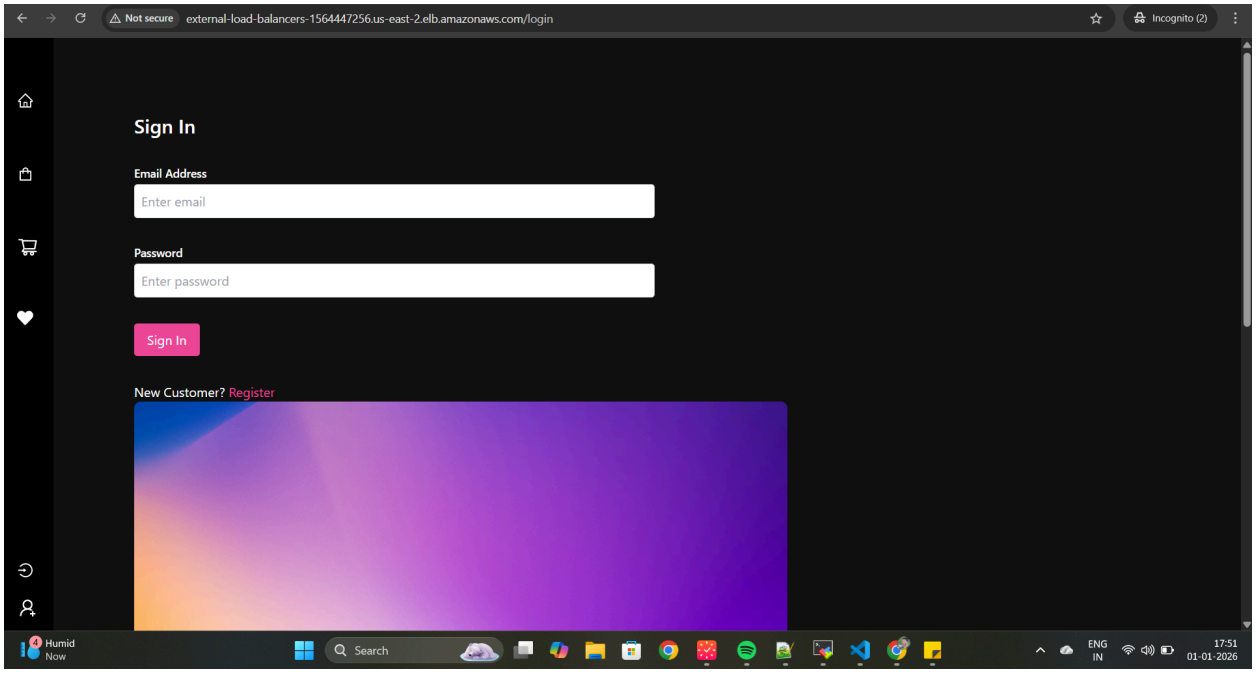


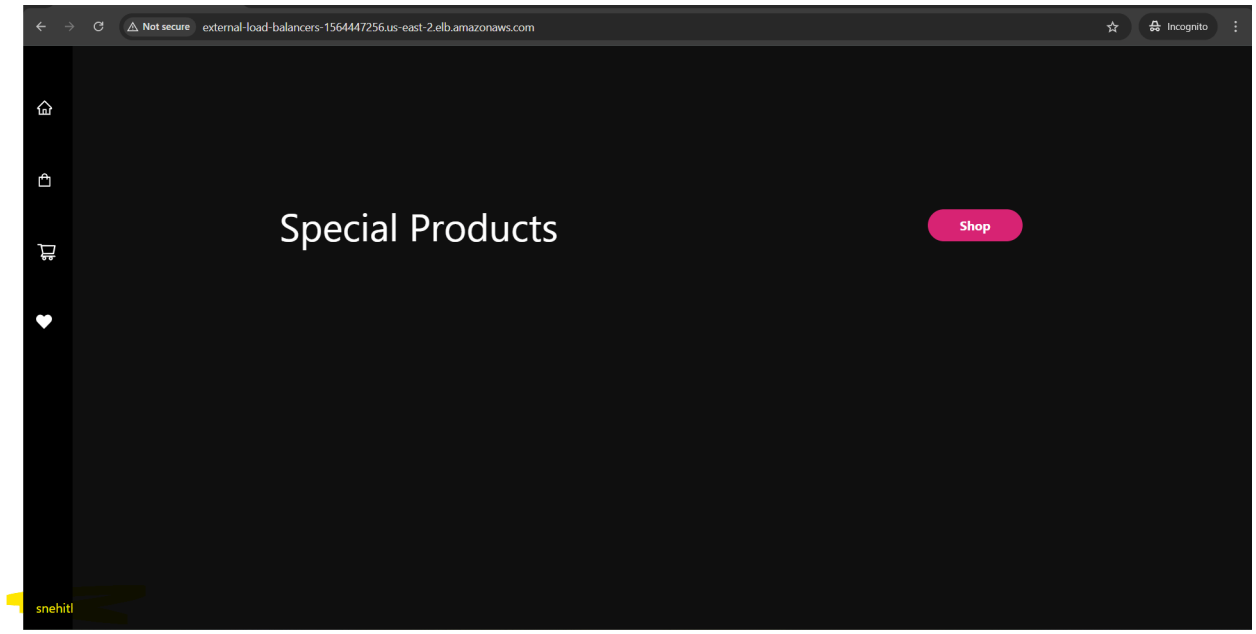
Target Group status:



11. Application Access & Validation

- Access the application using the **External Load Balancer DNS**
- Frontend loads successfully
- Backend connectivity verified
- User registration & login tested





Note: Image loading issue due to missing proxy configuration.

12. Conclusion

This project successfully demonstrates:

- Secure 3-tier AWS architecture
- Proper public/private subnet isolation
- High availability using Multi-AZ
- Scalable compute using ASG
- Secure database deployment in private subnets

The core goal of **secure, production-ready network architecture** is fully achieved.

Next Steps

- Fix frontend proxy/image loading
- Addition of security (WAF) and caching for less latency (CDN : cloudfront)
- Enable CloudWatch alarms
- Provisioning the resources using the terraform