TASK 7

Deploying an Application on ECS EC2 using AWS CodePipeline

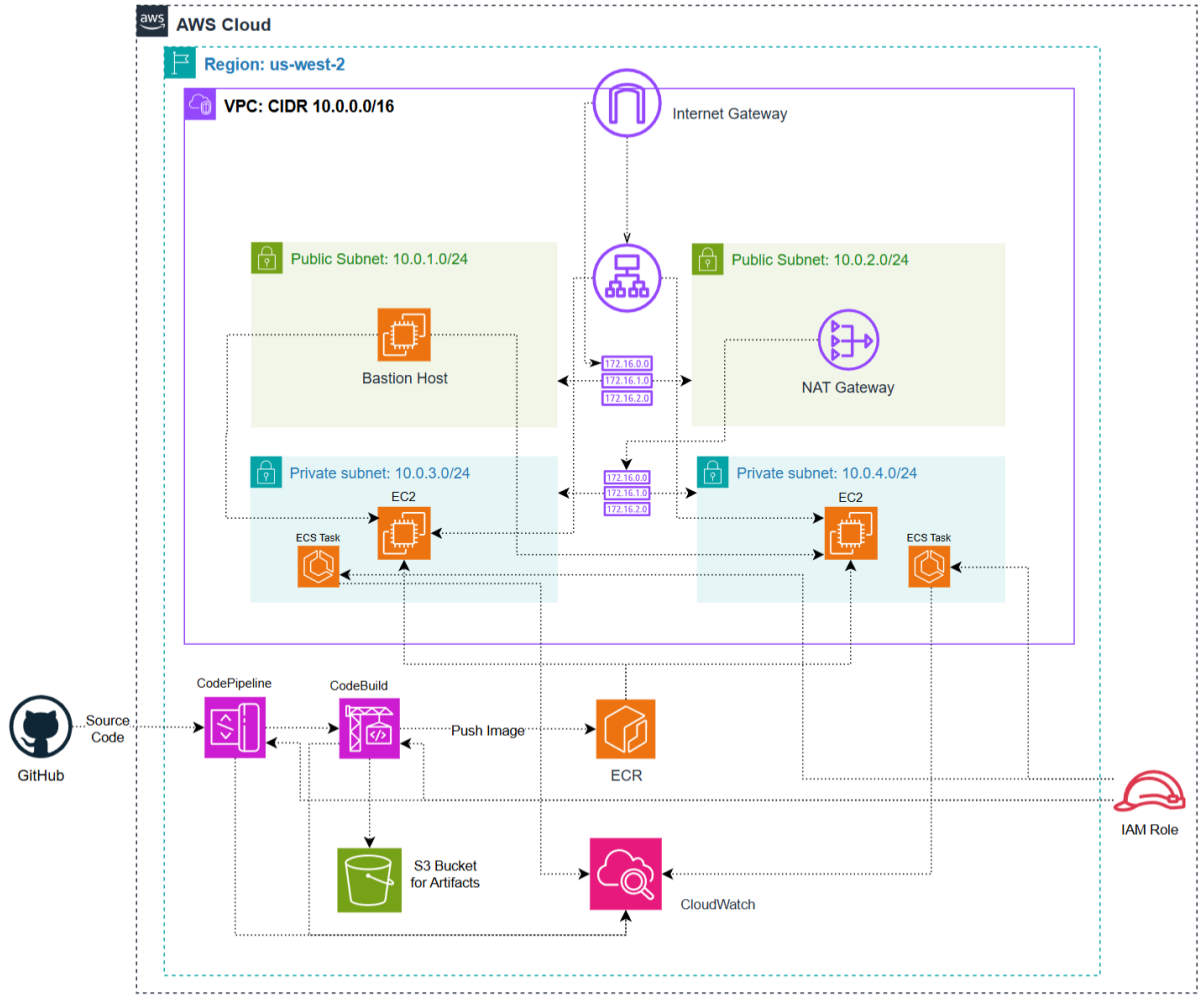
Umar Satti

Table of Contents

Task Description

Set up AWS CodePipeline on the AWS console for CI/CD of an application running on ECS EC2 and Application Load Balancer.

Architecture Diagram



Tasks

Task 1.1: Create VPC

A **Virtual Private Cloud (VPC)** is a logically isolated network environment within AWS, allowing the creation of secure and organized infrastructure.

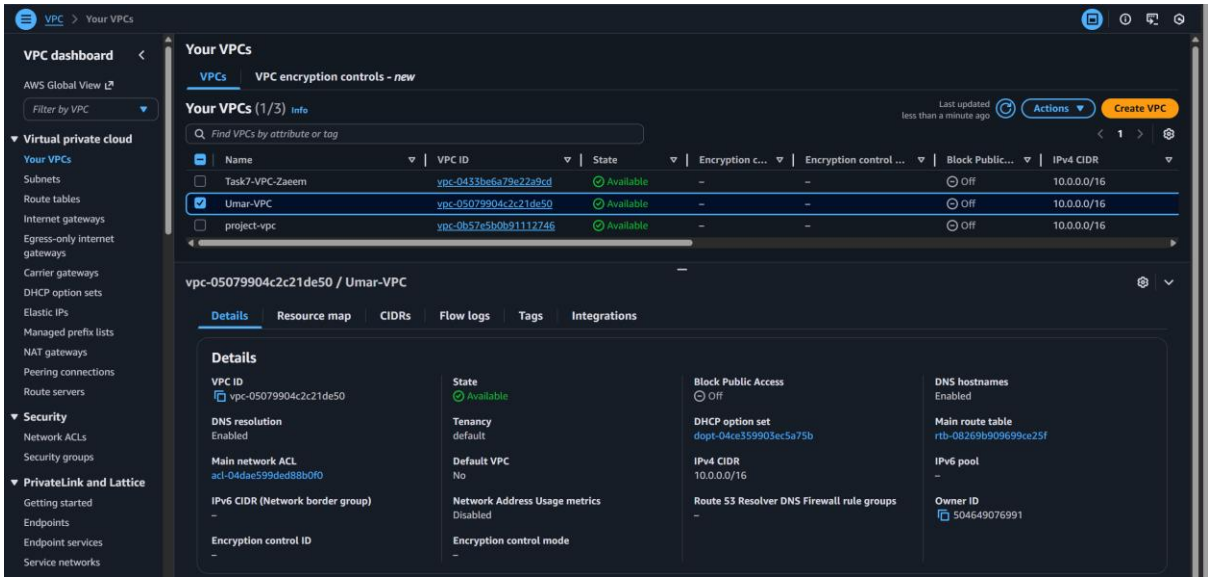
Steps:

1. Sign in to the AWS Management Console.
2. Navigate to VPC service using the search bar at the top.
3. In the VPC Console, click **Your VPCs.**
4. Click **Create VPC** button on the top right.
5. Choose **VPC only** option for 'Resources to create'.
6. Choose a name for the VPC (example: Umar-VPC) and add an IPv4 CIDR block (example: 10.0.0.0/16).
7. Leave the **Tenancy** as **Default**.

Note: Tags are already added once you choose a name. Tags are key-value pairs that are created automatically alongside AWS resources as **Name** field tag.

As shown in the image below, under **Details** tab, a VPC has been created with the following configuration:

- Name: **Umar-VPC**
- IPv4 CIDR range: **10.0.0.0/16**
- VPC ID: **vpc-05079904c2c21de50**



Defining a custom VPC (e.g., using CIDR 10.0.0.0/16) provides control over the IP address range, subnets, routing, and security configurations. This VPC serves as the foundational

network for deploying both public and private resources while maintaining clear separation and security boundaries.

Task 1.2: Create Internet Gateway

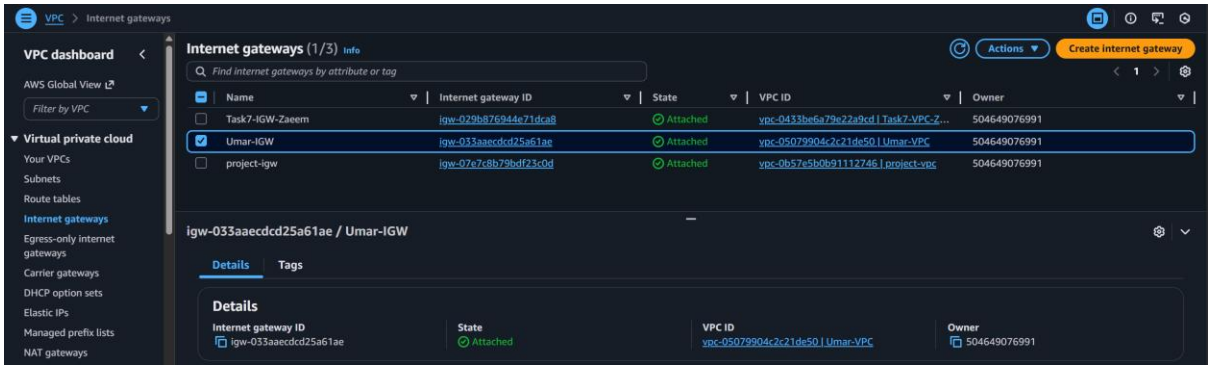
An **Internet Gateway (IGW)** enables communication between instances in the VPC and the public internet.

Steps:

1. Within the VPC Console, click **Internet gateways**.
2. Click on **Create internet gateway** button.
3. Choose a **Name** and add optional Tags**.**
4. Click **Create internet gateway** button. This creates an Internet gateway outside the VPC, and it needs to attach to the VPC created earlier.
5. Select **Actions** on the top right and click **Attach to VPC**. Choose the VPC ID (created earlier) and click **Attach internet gateway**. This attaches the Internet gateway to the VPC, allowing internet access.

As shown in the image below under **Details** tab, an internet gateway has been created with the following configuration:

- Name: **Umar-IGW**
- State: **Attached**
- VPC ID: **vpc-05079904c2c21de50**
- Internet gateway ID: **igw-033aaecdcd25a61ae**



Task 1.3: Create NAT Gateway

A NAT Gateway (Network Address Translation Gateway) provides outbound internet access to private instances without exposing them to inbound traffic.

Steps:

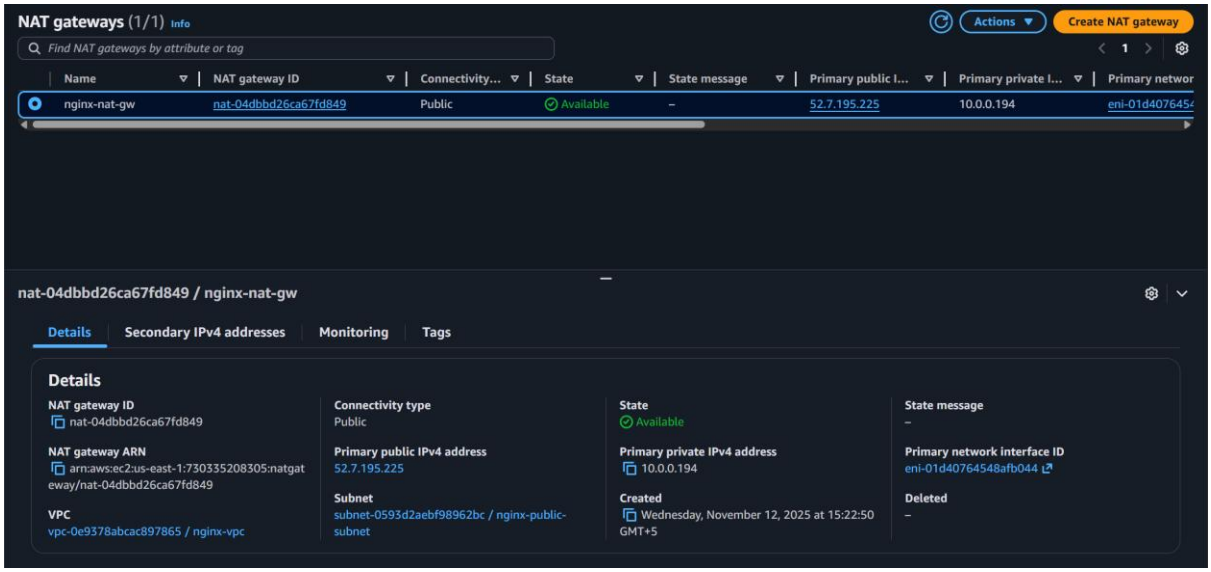
1. Within the VPC Console, click **NAT Gateways**.
2. Click on **Create NAT gateway** button on the top right.
3. Choose **Availability mode** as **Zonal**. This helps attach NAT gateway in public subnet.

4. Choose a **Name, Subnet, Connectivity type, Elastic IP allocation ID,** and optional **Tags**.
5. When done, click on **Create NAT gateway** button to create the resource.

As shown in the image below under **NAT gateways** section, a NAT gateway is created with the following configuration:

- Name: **Umar-NAT-GW**
- NAT gateway ID: **nat-03206359b658facf7**
- Connectivity type: **Public**
- VPC: **Umar-VPC**
- Primary network interface ID: **eni-081b7e632b4fd0292**

Note: When creating the NAT Gateway, click on **Allocate Elastic IP** option. This automatically generates an **Elastic IP** for NAT Gateway resource.



NAT Gateway is set up in the public subnet and assigned an Elastic IP. The private ECS EC2 instances can communicate externally without being directly reachable from the internet. This maintains a secure boundary for private subnets.

Note: For high availability and fault tolerance, create two NAT gateways, each in subsequent public subnet. For this project, only a single NAT gateway has been deployed in Public Subnet A (shown later).

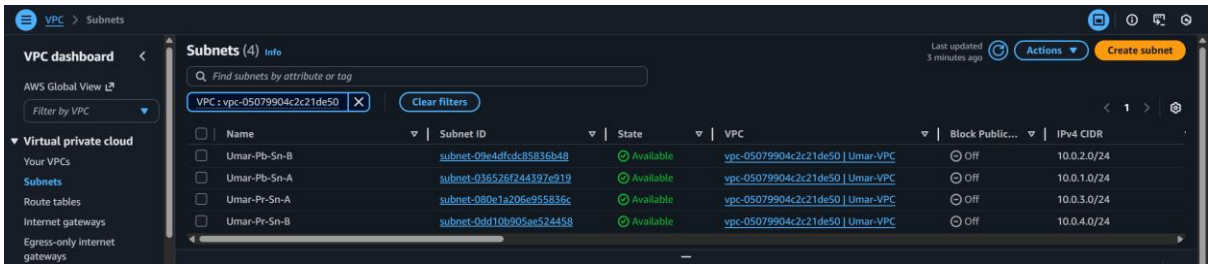
Task 1.4: Create Subnets

Steps:

1. Within the VPC Console, click **Subnets**.
2. Click on **Create subnet** button.
3. Choose a **Subnet name, Availability Zone, IPv4 VPC CIDR block, IPv4 subnet CIDR Block** and optional **Tags.**

4. Repeat this process by clicking on **Add new subnet** button at the bottom to add more subnets (Public and Private).
5. After adding the required subnets, click on **Create subnet** button at the bottom right.

As shown in the image below under **Subnets** section, three subnets have been created with the following configuration:



Public Subnet A:

- Name: **Umar-Pb-Sn-A**
- Subnet ID: **subnet-036526f244397e919**
- IPv4 CIDR: **10.0.1.0/24**

Public Subnet B:

- Name: **Umar-Pb-Sn-B**
- Subnet ID: **subnet-09e4dfcdc85836b48**
- IPv4 CIDR: **10.0.2.0/24**

Private Subnet A:

- Name: **Umar-Pr-Sn-A**
- Subnet ID: **subnet-080e1a206e955836c**
- IPv4 CIDR: **10.0.3.0/24**

Private Subnet B:

- Name: **Umar-Pr-Sn-B**
- Subnet ID: **subnet-04510a913a7b0fc74**
- IPv4 CIDR: **10.0.4.0/24**

In this design:

- The **Public Subnet** hosts internet-facing resources (e.g., Bastion Host and Load Balancer).
- The **Private Subnets** host internal EC2 instances running ECS tasks and containers.

This subnet structure follows the public - private architecture model, improving both scalability and network security.

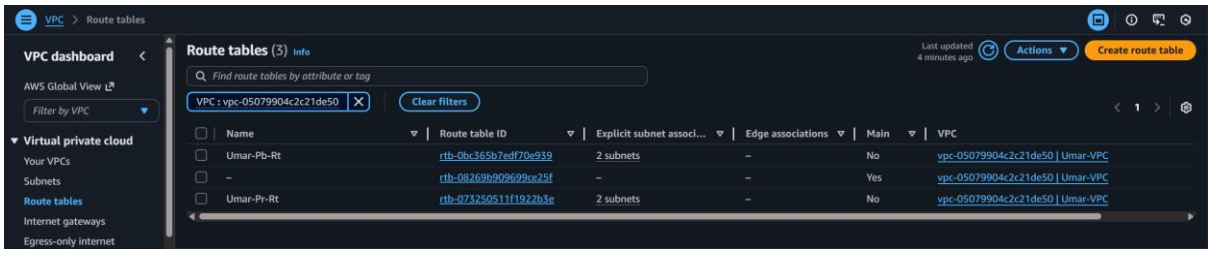
Task 1.5: Create Route Tables

Route Tables define how network traffic moves within the VPC. Each route table directs traffic either internally (within the VPC) or externally (to the internet or a NAT gateway).

Steps:

1. Within the VPC Console, click **Route tables**.
2. Click on **Create route table** button on the top right.
3. Choose a **Name, VPC**, and optional **Tags.**
4. When done, click on **Create route table** button to create the resource.

As shown in the image below under **Subnets** section, three subnets have been created with the following configuration:



Public Route Table:

- Name: **Umar-Pb-Rt**
- Route table ID: **rtb-0bc365b7edf70e939**

Private Route Table:

- Name: **Umar-Pr-Rt**
- Route table ID: **rtb-073250511f1922b3e**

In this configuration:

- The **Public Route Table** directs internet-bound traffic to the **Internet Gateway**.
- The **Private Route Table** forwards internet-bound requests through the **NAT Gateway** (set up in Task 1.3), maintaining private subnet isolation.

This ensures clear routing logic between internal and external networks.

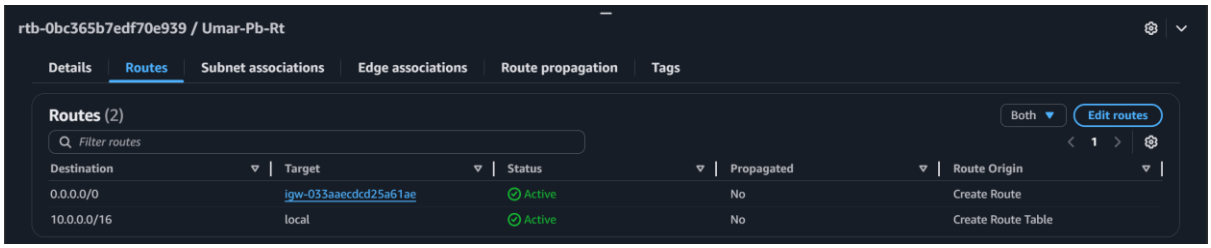
Task 1.6: Setting up Routes for Route tables

Steps:

1. Within the Route tables console, select a Route table (example: Public-Route-Table) and click **Edit routes** on the right.
2. Add a new route by choosing a **Destination** and **Target**.
3. Click **Save changes**. This adds a route to this route table.

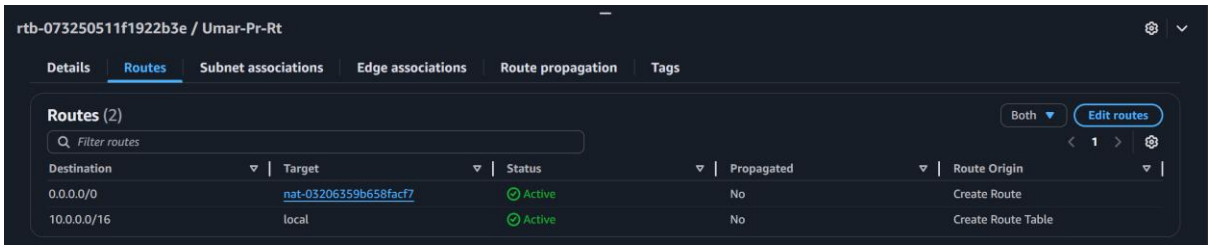
Public Routes (Umar-Pb-Rt):

- Destination: **0.0.0.0/0**
- Target: **Umar-IGW | igw-033aaecdcd25a61ae** (Internet gateway)



Private Routes (Umar-Pr-Rt):

- Destination: **0.0.0.0/0**
- Target: **Umar-NAT-GW | nat-03206359b658facf7** (NAT gateway)



This step defines how traffic flows between subnets and external networks:

- The **Public route table** sends all internet-bound traffic (0.0.0.0/0) to the **Internet Gateway**, enabling access for public resources like the load balancer DNS.
- The **Private route table** provides internet-bound traffic through the **NAT Gateway**, allowing private EC2 instances to initiate outbound connections without exposure to the internet.

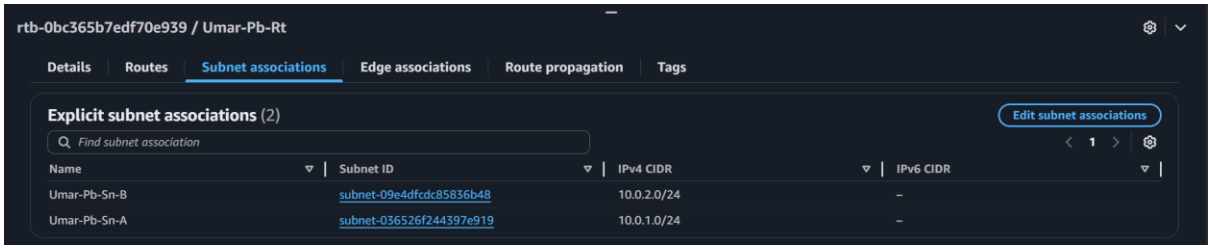
Task 1.7: Route Table Association

Steps:

1. Within the Route tables console, select a Route table (example: Public-Route-Table) and click **Subnet associations** tab under the details section.
2. Click **Edit subnet associations** and select the specific subnet (e.g. nginx-public-subnet)
3. Click **Save associations** button on the bottom right.

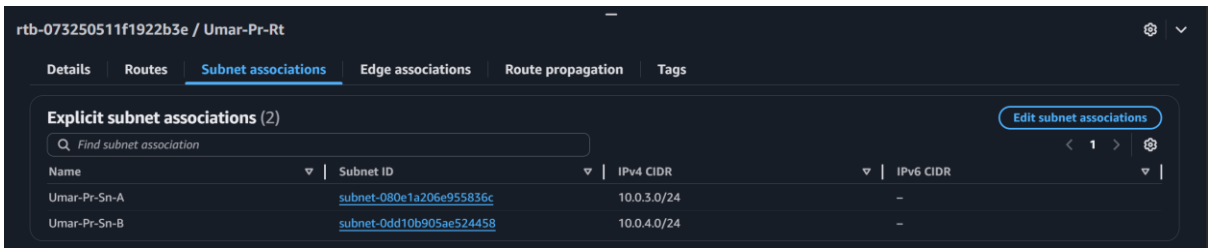
Public route table Subnet Associations

- Names: **Umar-Pb-Sn-A** and **Umar-Pb-Sn-B**
- Subnet IDs: **subnet-036526f244397e919** and **subnet-09e4dfcdc85836b48**
- IPv4 CIDR: **10.0.1.0/24** and **10.0.2.0/24**



Private route table Subnet Associations

- Names: **Umar-Pr-Sn-A** and **Umar-Pr-Sn-B**
- Subnet IDs: **subnet-080e1a206e955836c** and **subnet-0dd10b905ae524458**
- IPv4 CIDR: **10.0.3.0/24** and **10.0.4.0/24**



Each subnet must be linked to a route table to control its network traffic.
Associating subnets ensures:

- **Public Subnet** uses the Internet Gateway for external communication.
- **Private Subnets** use the NAT Gateway for secure outbound traffic.

Without this association, subnets would rely on the default main route table, potentially misrouting traffic.

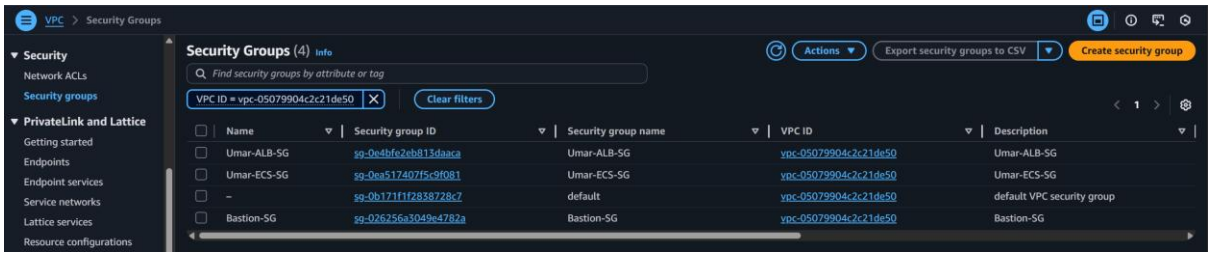
Task 1.8: Security Groups

Security Groups act as virtual firewalls that regulate inbound and outbound traffic at the instance level.

Steps:

1. Within the VPC Console, click **Security Groups**.
2. Click on **Create security gateway** button on the top right.
3. Choose a **Security group name**, **Description**, and **VPC** (nginx-vpc).

4. Add Inbound rules by selecting **Type, Protocol, Port range, Source,** and **Description** (optional).
5. Leave Outbound rules as default (All traffic).

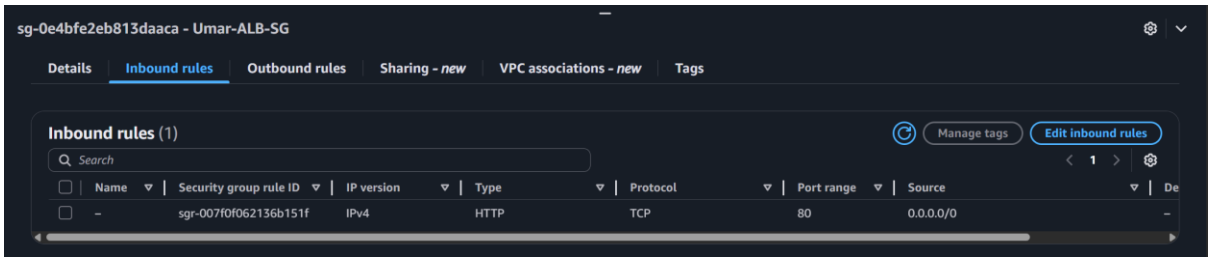


Task 1.8.1: Application Load Balancer Security group

Here is the configuration for the public subnet security group:

- Security group name: **Umar-ALB-SG**
- Security group ID: **sg-0e4bfe2eb813daaca**
- Inbound Rules:

Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0

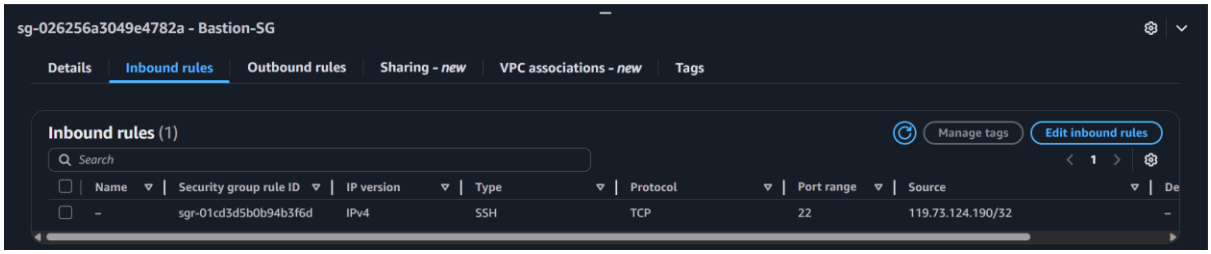


Task 1.8.2: Bastion Host Security group

Here is the configuration for the public subnet security group:

- Security group name: **Bastion-SG**
- Security group ID: **sg-026256a3049e4782a**
- Inbound Rules:

Type	Protocol	Port range	Source
SSH	TCP	22	119.73.124.190/32

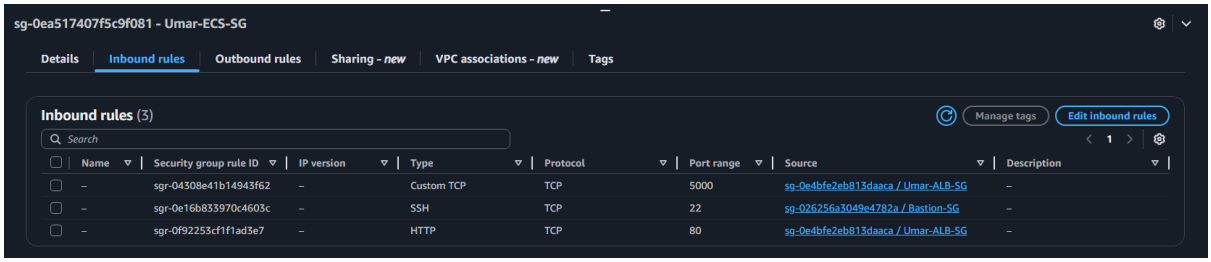


Task 1.8.3: ECS EC2 Instance Security group

Here is the configuration for the public subnet security group:

- Security group name: **Umar-ECS-SG**
- Security group ID: **sg-0ea517407f5c9f081**
- Inbound Rules:

Type	Protocol	Port range	Source
SSH	TCP	22	Bastion-SG
Custom TCP	TCP	5000	Umar-ALB-SG
HTTP	TCP	80	Umar-ALB-SG



In this configuration:

- **Umar-ALB-SG**: Allows HTTP (port 80) from any external source.
- **Bastion-SG**: Allows SSH traffic only from my IP address, preventing direct internet access.
- **Umar-ECS-SG**: Allows HTTP (port 80) / port 5000 traffic from ALB-SG and SSH traffic only from Bastion-SG.

This layered security model follows the principle of least privilege, ensuring only the load balancer can reach private EC2 instances

Task 1.9: Public EC2 Instance

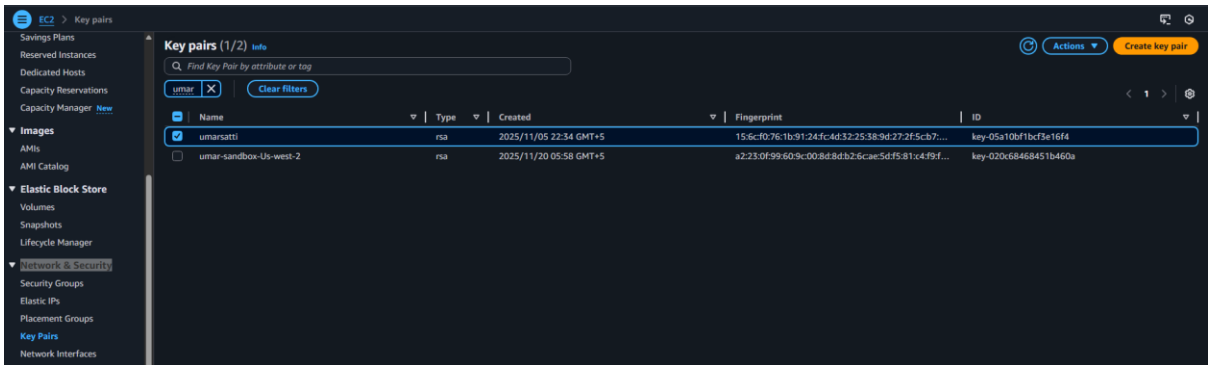
Using Key Pair

A key pair is a secure authentication mechanism that allows SSH access to EC2 instances. It consists of a public key (stored in AWS) and a private key (stored securely on the user's local machine). The private key is required to connect to an instance via SSH.

Steps:

1. Sign in to the AWS Management Console.
2. Navigate to the EC2 service using the search bar at the top.
3. On the left navigation panel, under **Network & Security**, select **Key Pairs**.
4. Click the Create key pair button on the top right.
5. Provide a **Name** (e.g., *keypair*), select **RSA** as the **key type**, and choose **.pem** as the **private key file format**.
6. Click **Create key pair** button on the bottom right.

Note: The private key file (e.g., *umarsatti.pem*) will be automatically downloaded to local machine. Store it in a secure location, as AWS does not allow re-downloading this file for security reasons. This key will later be required to SSH into EC2 instances.

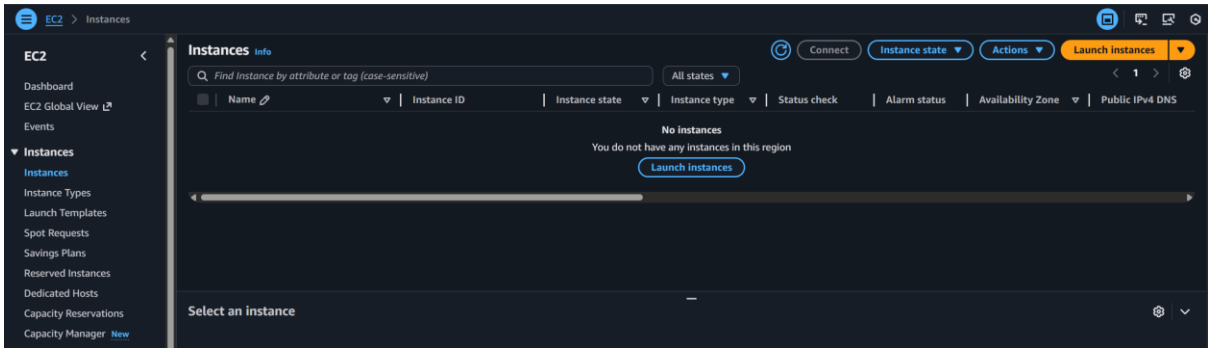


Creating a key pair establishes a secure login method for all EC2 instances in this project. It eliminates the need for password-based authentication, thereby enhancing overall security.

Task 1.9.2 Bastion Host EC2 Instance

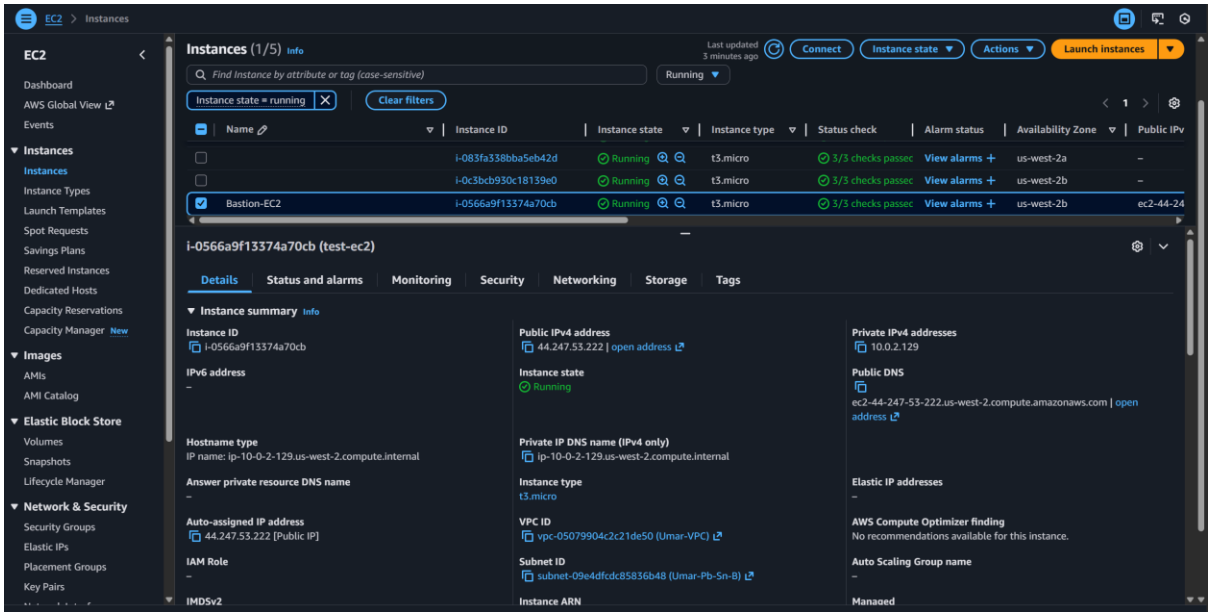
Steps:

1. In the EC2 Console, click **Instances** on the right navigation panel.
2. Click **Launch Instances** button on the top right.
3. Choose a **Name, OS Image, AMI, Architecture, Instance type,** and **Key pair**.
4. Edit **Network settings.**
5. Choose **VPC (Umar-VPC), Subnet, Availability Zone, Auto-assign public IP, and Security group.**
6. Leave the rest as default and click **Launch Instance.**



Specifications:

- Name: **Bastion-EC2**
- AMI: **Amazon Linux 2023**
- Architecture: **64-bit (x86)**
- Instance type: **t3.micro**
- Key pair: **umarsatti.pem** (for SSH login)
- VPC: **Umar-VPC**
- Subnet: **Umar-Pb-Sn-B**
- Availability zone: **us-west-1b**
- Security group name: **Bastion-SG**
- Storage: **1x8 GiB gp3**
- Instance ID: **i-0566a9f13374a70cb**



Task 1.10: ECR

Task 1.10.1: Create ECR Repository

Steps:

1. In the ECR Console, click **Repositories** under **Private registry** on the left navigation panel.
2. Click **Create repository** button on the top right.
3. Choose a **Repository name, Image tag settings,** and **Encryption settings**
4. Click **Create button**.

An ECR repository was created as shown in the image below:

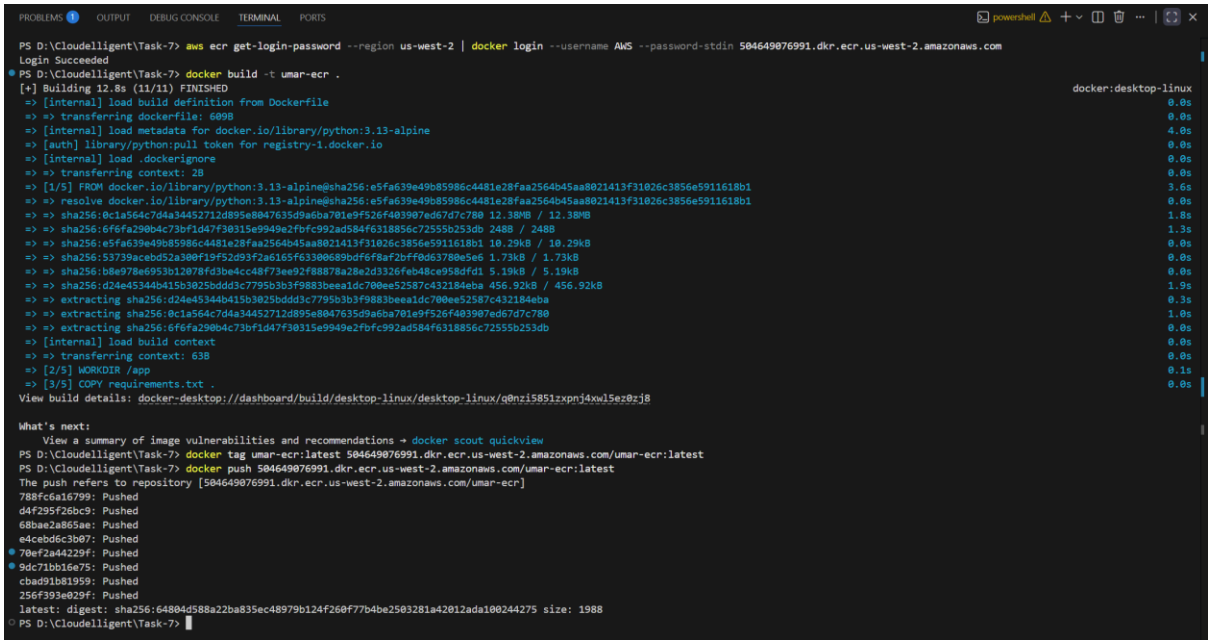
- **Repository name:** umar-ecr
- **URI:** 504649076991.dkr.ecr.us-west-2.amazonaws.com/umar-ecr
- **Tag immutability:** Mutable
- **Encryption type:** AES-256

Task 1.10.2: Push Docker Image to ECR

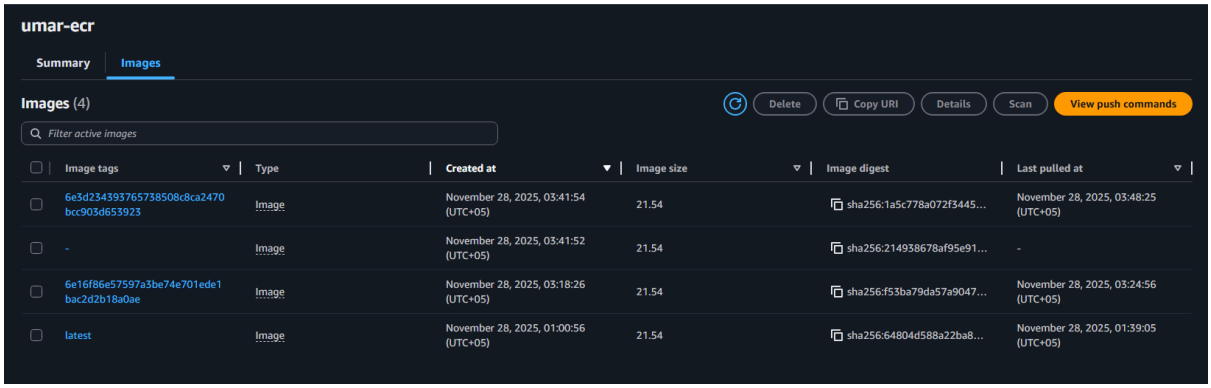
Steps:

1. In the ECR Console, click the repository created earlier (umar-ecr)
2. Click on **View push commands** button on the top right.
3. Run these commands on a local terminal in the order stated.

Note: The user must be logged in to AWS in the local terminal (using **aws configure** command) for these commands to work. Additionally, make sure **Docker Desktop** is running as well.



These set of commands shown above are used to perform account login, build docker image locally, add the latest tag, and then push this image into ECR repository.



Task 1.11: ECS Cluster

An Amazon ECS (Elastic Container Service) Cluster is a logical grouping of container instances or serverless Fargate tasks. In this project, the ECS Cluster is created using the **EC2 launch type**, meaning the tasks run on self-managed EC2 instances that register into the cluster.

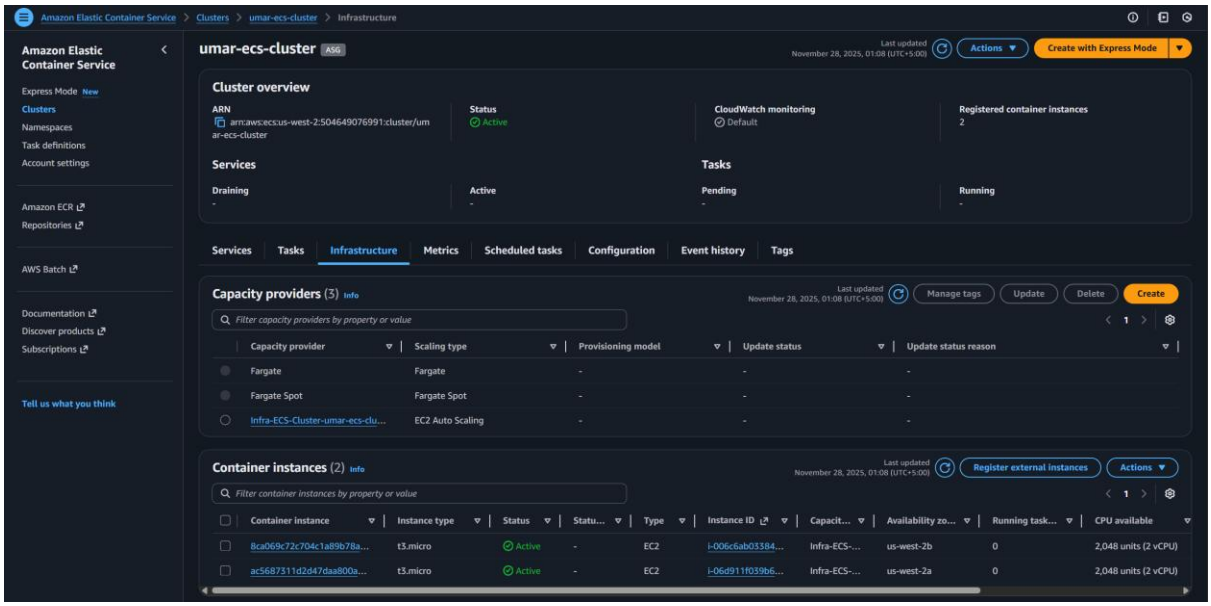
This setup allows containers to run on secure private subnets while being managed by the ECS agent installed on each EC2 instance.

Steps:

1. Navigate to the **ECS Console** using the search bar at the top of the AWS Management Console.
2. In the ECS dashboard, click **Create Cluster**.
3. Enter a **Cluster name** (example: *umar-ecs-cluster*).

4. Under **Service Connect**, leave the option turned **OFF** because no namespace is required for this deployment.
5. Under the **Infrastructure** section, choose **"Fargate and Self-managed instances"**
6. In the **EC2 Instance configuration**, click **Create a new Auto Scaling group**.
7. Configure the Auto Scaling group using the following settings:
    - **Provisioning model:** On-demand
    - **Operating system:** Amazon Linux 2023
    - **Instance type:** t3.micro
    - **EC2 instance role:** Must include the policy **AmazonEC2ContainerServiceforEC2Role.** This role allows EC2 instances to register into ECS and communicate with ECR and ECS control plane.
    - **Desired capacity:** Minimum 2 / Maximum 2
    - **SSH Key Pair:** *umarsatti.pem*
    - **Root EBS volume size:** 30 GiB
8. Under **Networking**, select:
    - **VPC:** Umar-VPC
    - **Subnets:** Choose both **Private Subnets** (Umar-Pr-Sn-A and Umar-Pr-Sn-B)
    - **Security Group:** Select **Umar-ECS-SG**
9. Ensure **Auto-assign public IP** is set to **OFF**, since these ECS EC2 instances run in private subnets.
10. Review the settings and click **Create** to provision the ECS Cluster and launch EC2 instances into the Auto Scaling group.

Once the cluster creation is completed, you should see the following under the cluster details:



- **Cluster name:** umar-ecs-cluster
- **Registered container instances:** 2 (once EC2 instances finish bootstrapping and register with ECS agent)
- **Infrastructure:** ECS EC2 launch type

- **Networking:** Instances running in private subnets for security
- **Security:** Protected by Umar-ECS-SG, allowing only ALB-SG (port 80 / 5000) and Bastion-SG (SSH)

At this stage of the setup:

- Two EC2 instances (t3.micro) running Amazon Linux 2023 are launched in the private subnets.
- These instances automatically install and run the ECS agent, registering themselves to the cluster.
- The cluster is now ready for:
    - ECS Task Definitions
    - ECS Services
    - Load Balancer configuration
    - Deployment through CodePipeline

This ECS Cluster forms the compute layer of the application deployment architecture and will later receive updated container versions through CI/CD.

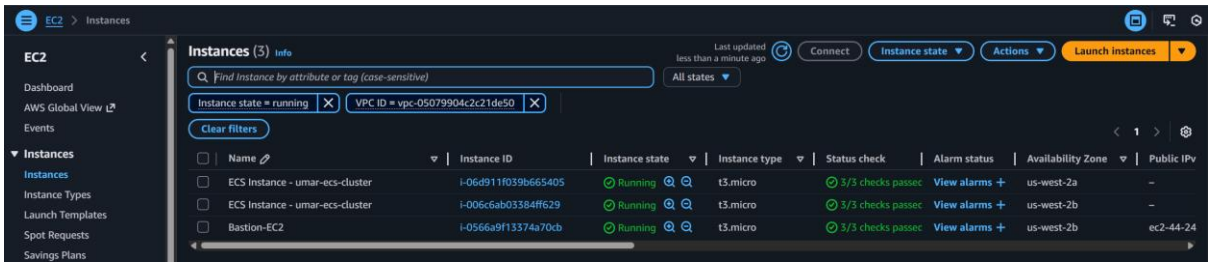
Task 1.12: Test Private EC2 Instances

After creating the ECS Cluster and Auto Scaling Group, two private EC2 instances were launched automatically within the private subnets. These instances run the Amazon ECS Agent and Docker engine, enabling them to join the ECS Cluster (*umar-ecs-cluster*) and run container tasks.

To validate that the instances were provisioned correctly and successfully registered with the ECS cluster, SSH access was tested through the Bastion Host.

Task 1.12.1: SSH into Private Instances

Since these EC2 instances are deployed in **private subnets**, direct SSH access is not possible from the public internet. Instead, SSH was performed using the **Bastion Host** located in the public subnet.



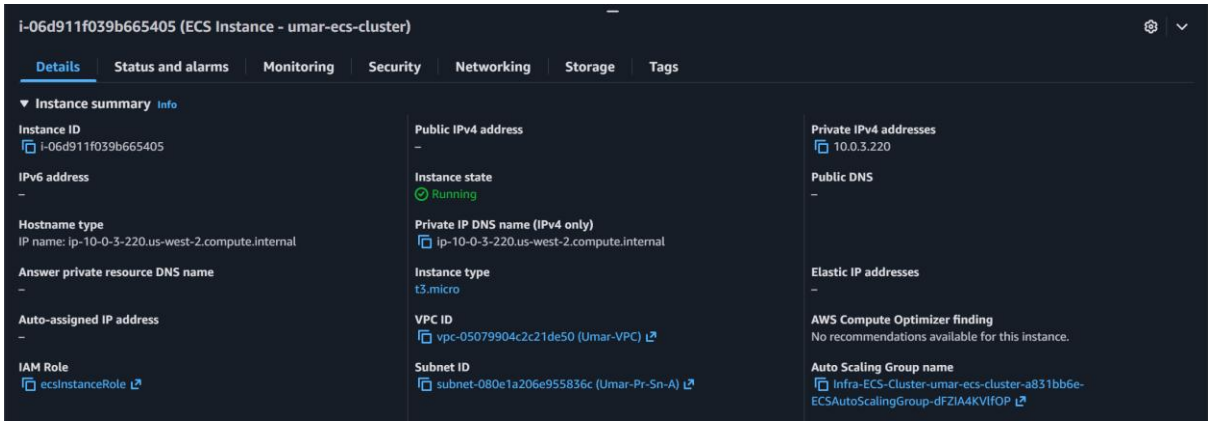
Steps:

1. SSH into the Bastion Host using the private key (umarsatti.pem):
2. ssh -i umarsatti.pem ec2-user@<Bastion-Public-IP>

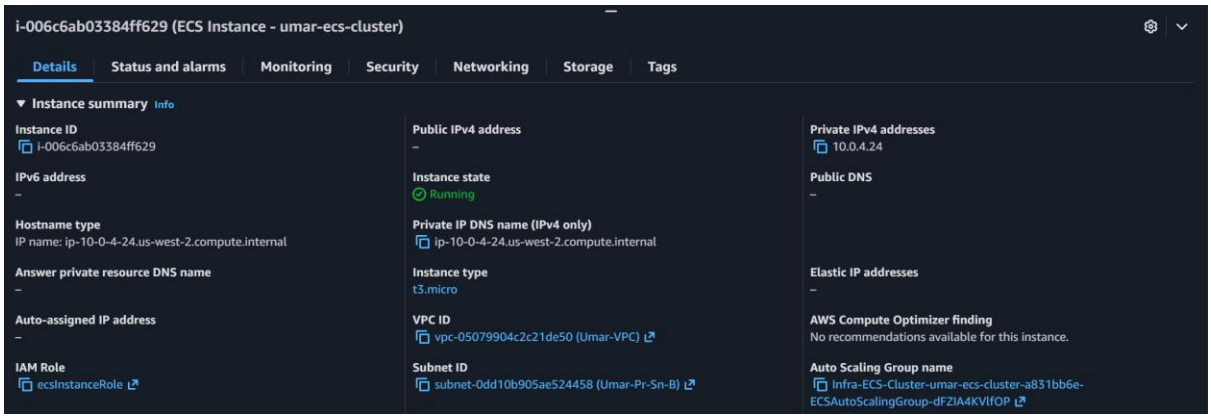
3. From the Bastion Host, SSH into each private EC2 instance using their **Private IPv4 addresses**:
4. ssh ec2-user@10.0.3.220
5. ssh ec2-user@10.0.4.24

These private IP addresses were taken from the **Details** section of each EC2 instance.

First Instance



Second Instance



Task 1.12.2: Verifying ECS Agent and Docker Installation

Once logged into each private EC2 instance, the following commands were executed to confirm that **Docker**, the **ECS Agent**, and the **ECS configuration file** were properly configured:

- sudo systemctl status docker
- sudo systemctl status ecs
- cat /etc/ecs/ecs.config

The expected outputs:

- **Docker service:** active (running)

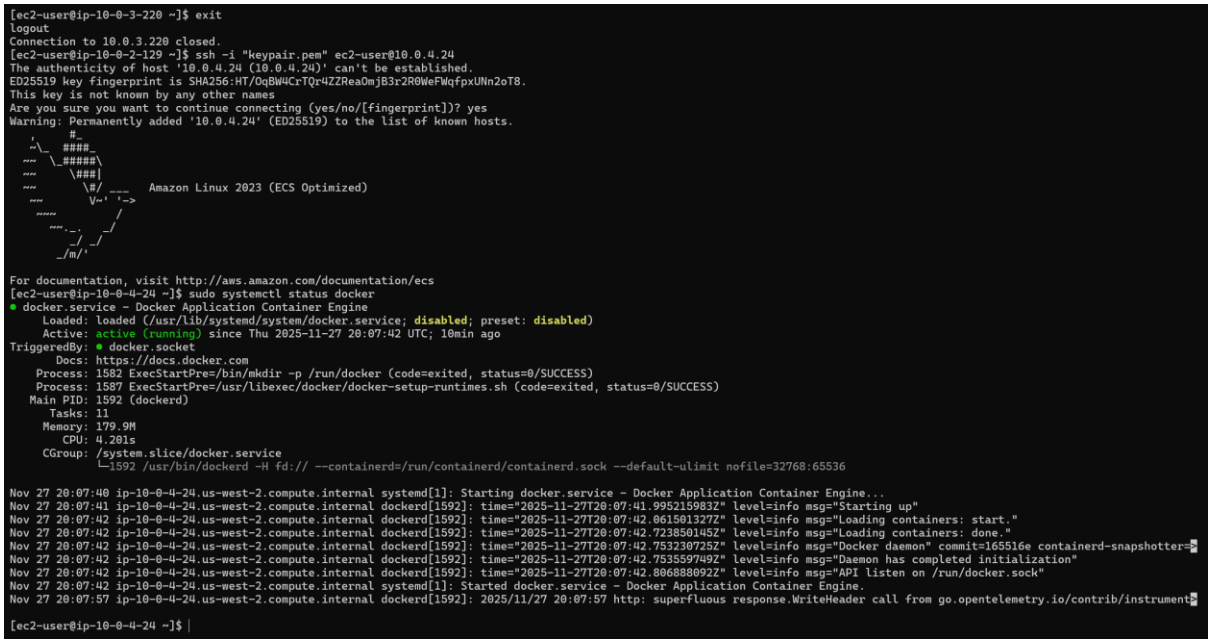
- **ECS service:** active (running)
- **ecs.config:** contains correct cluster name as **ECS_CLUSTER=umar-ecs-cluster**

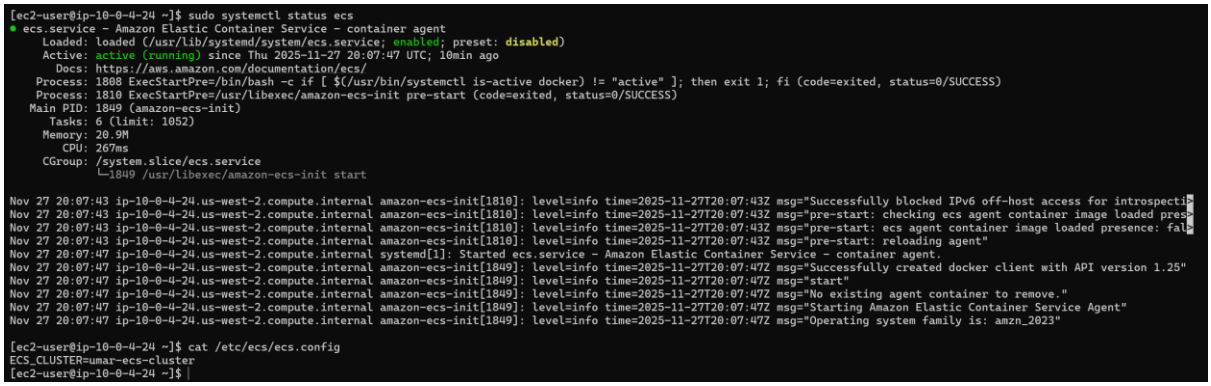
First EC2 Instance (10.0.3.220)

```
PS C:\Users\umars\Downloads> ssh -i "umarsatti.pem" ec2-user@ec2-44-247-53-222.us-west-2.compute.amazonaws.com
       #_
    ~\_  ####_        Amazon Linux 2023
   ~~  \_#####\
   ~~     \###|
   ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
    ~~       V~' '->
     ~~~         /
       ~~._.   _/
          _/ _/
        _/m/'
Last login: Wed Nov 26 22:06:05 2025 from 119.73.124.190
[ec2-user@ip-10-0-2-129 ~]$ ssh -i "keypair.pem" ec2-user@10.0.3.220
The authenticity of host '10.0.3.220 (10.0.3.220)' can't be established.
ED25519 key fingerprint is SHA256:7HjxRGWw9rz/HsgOvVVW93Snw0Tlx+6YBjPU4UsyjFk.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.0.3.220' (ED25519) to the list of known hosts.
       #_
    ~\_  ####_
   ~~  \_#####\
   ~~     \###|
   ~~       \#/ ___   Amazon Linux 2023 (ECS Optimized)
    ~~       V~' '->
     ~~~         /
       ~~._.   _/
          _/ _/
        _/m/'

For documentation, visit http://aws.amazon.com/documentation/ecs
[ec2-user@ip-10-0-3-220 ~]$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; preset: disabled)
     Active: active (running) since Thu 2025-11-27 20:07:41 UTC; 7min ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
    Process: 1581 ExecStartPre=/bin/mkdir -p /run/docker (code=exited, status=0/SUCCESS)
    Process: 1586 ExecStartPre=/usr/libexec/docker/docker-setup-runtimes.sh (code=exited, status=0/SUCCESS)
   Main PID: 1587 (dockerd)
      Tasks: 12
     Memory: 179.3M
        CPU: 4.195s
     CGroup: /system.slice/docker.service
             └─1587 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --default-ulimit nofile=32768:65536
```

```
[ec2-user@ip-10-0-3-220 ~]$ sudo systemctl status ecs
● ecs.service - Amazon Elastic Container Service - container agent
     Loaded: loaded (/usr/lib/systemd/system/ecs.service; enabled; preset: disabled)
     Active: active (running) since Thu 2025-11-27 20:07:45 UTC; 7min ago
       Docs: https://aws.amazon.com/documentation/ecs/
    Process: 1810 ExecStartPre=/bin/bash -c if [ $(/usr/bin/systemctl is-active docker) != "active" ]; then exit 1; fi (code=exited, status=0/SUCCESS)
    Process: 1812 ExecStartPre=/usr/libexec/amazon-ecs-init pre-start (code=exited, status=0/SUCCESS)
   Main PID: 1850 (amazon-ecs-init)
      Tasks: 5 (limit: 1052)
     Memory: 20.8M
        CPU: 255ms
     CGroup: /system.slice/ecs.service
             └─1850 /usr/libexec/amazon-ecs-init start

Nov 27 20:07:41 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1812]: level=info time=2025-11-27T20:07:41Z msg="Successfully blocked IPv6 off-host access for introspect
Nov 27 20:07:41 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1812]: level=info time=2025-11-27T20:07:41Z msg="pre-start: checking ecs agent container image loaded pre
Nov 27 20:07:41 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1812]: level=info time=2025-11-27T20:07:41Z msg="pre-start: ecs agent container image loaded presence: fa
Nov 27 20:07:41 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1812]: level=info time=2025-11-27T20:07:41Z msg="pre-start: reloading agent"
Nov 27 20:07:45 ip-10-0-3-220.us-west-2.compute.internal systemd[1]: Started ecs.service - Amazon Elastic Container Service - container agent.
Nov 27 20:07:45 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1850]: level=info time=2025-11-27T20:07:45Z msg="Successfully created docker client with API version 1.25"
Nov 27 20:07:45 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1850]: level=info time=2025-11-27T20:07:45Z msg="start"
Nov 27 20:07:45 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1850]: level=info time=2025-11-27T20:07:45Z msg="No existing agent container to remove."
Nov 27 20:07:45 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1850]: level=info time=2025-11-27T20:07:45Z msg="Starting Amazon Elastic Container Service Agent"
Nov 27 20:07:45 ip-10-0-3-220.us-west-2.compute.internal amazon-ecs-init[1850]: level=info time=2025-11-27T20:07:45Z msg="Operating system family is: amzn_2023"

[ec2-user@ip-10-0-3-220 ~]$ cat /etc/ecs/ecs.config
ECS_CLUSTER=umar-ecs-cluster
[ec2-user@ip-10-0-3-220 ~]$
```

Second EC2 Instance (10.0.4.24)





Both private EC2 instances show:

- Docker is installed and running.
- The ECS Agent is active and communicating with AWS ECS.
- The ECS configuration file correctly references the cluster name **umar-ecs-cluster**.

This confirms that the instances have successfully registered into the ECS cluster and are ready to run ECS tasks as part of the deployment pipeline.

The ECS Cluster now has two active container instances visible under the **"Registered container instances"** section of the ECS Console.

Task 1.13: ECS Task Definition

An ECS Task Definition is a blueprint that describes the container(s) that will run on ECS. It includes information such as the Docker image, CPU/memory allocation, network mode, IAM permissions, and port mappings. The ECS Service will later use this task definition to launch and maintain running tasks on the EC2 instances within the cluster.

In this project, a task definition is created to run the Flask application stored in the ECR repository.

Steps:

1. Navigate to the **ECS Console** and select **Task Definitions** from the left navigation panel.
2. Click the **Create new Task Definition** button.
3. Select **EC2 launch type** since the application will run on self-managed EC2 instances in the ECS cluster.
4. Configure the task definition with the settings defined below.

Task Definition Configuration

- **Task definition family:** Umar-ecs-ec2-task-definition

Infrastructure Requirements

- **Launch type compatibility:** Amazon EC2 Instances
- **Operating System:** Linux / x86_64
- **Network mode:** bridge
- **Task size:** 0.25 vCPU / 0.5 GB Memory
- **Task Execution Role:** ecsTaskExecutionRole
- This **IAM role** includes the required permissions
    - **AmazonEC2ContainerRegistryReadOnly**
    - **AmazonECSTaskExecutionRolePolicy**

These policies allow ECS to:

- Pull container images from Amazon ECR
- Write logs to Amazon CloudWatch
- Perform basic ECS agent operations

Container Configuration

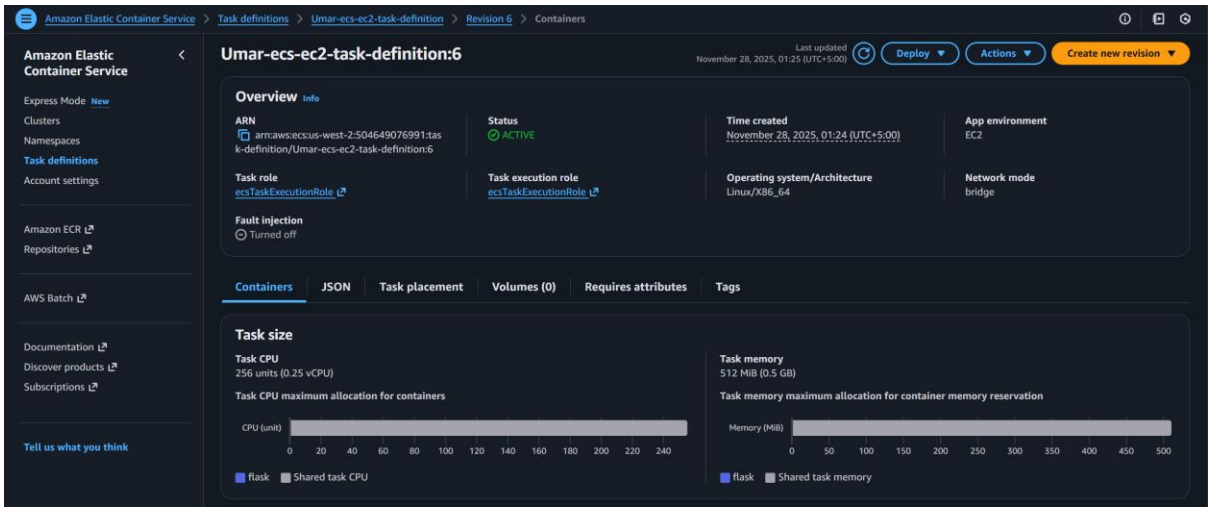
Click **Add container** and configure the container using the following details:

- **Container name:** flask
- **Image URI:** 504649076991.dkr.ecr.us-west-2.amazonaws.com/umar-ecr

This is the Docker image stored in the ECR repository created earlier.

- **Port Mappings**
    - **Host port:** 5000
    - **Container port:** 5000
    - **Protocol:** TCP
    - **Port name:** flask-port

This allows traffic from the EC2 instance (host) to be forwarded to the Flask container on port 5000. All other container settings (environment variables, logging, health checks, etc.) were left at **default values** for this task definition revision.



The final task definition successfully defines:

- The required container image stored in Amazon ECR
- Resource allocations suitable for a small Flask application
- Network configuration using bridge mode
- Proper IAM execution role permissions
- Required port mappings for the application

Once created, this task definition becomes available for use in the next step: creating an ECS Service that will run and maintain the container tasks on the EC2 instances in the **umar-ecs-cluster**.

Task 1.14: Application Load Balancer and Target Groups

To properly distribute incoming traffic across the ECS EC2 instances running the Flask application, an **Application Load Balancer (ALB)** is configured. The ALB operates at Layer 7 (HTTP/HTTPS) and forwards requests to targets (EC2 instances) registered in a target group. This ensures scalability, high availability, and health monitoring for the application.

The ALB is deployed in the **public subnets**, while the EC2 instances running the ECS tasks remain in the private subnets. This ensures secure and controlled access to the backend containers.

Task 1.14.1: Create Application Load Balancer

Steps:

1. Navigate to the **EC2 Console**, then select **Load Balancers** from the left navigation panel.
2. Click **Create Load Balancer** and select **Application Load Balancer**.
3. Configure the ALB with the settings defined below.

Basic Configuration

- **Load balancer name:** (as created in your screenshot)
- **Scheme:** Internet-facing
- **IP address type:** IPv4

Network Mapping

- **VPC:** Umar-VPC
- **Subnets:** Umar-Pb-Sn-A (Public Subnet A) and Umar-Pb-Sn-B (Public Subnet B)

These public subnets allow the ALB to receive traffic from the internet and forward requests privately to backend ECS tasks.

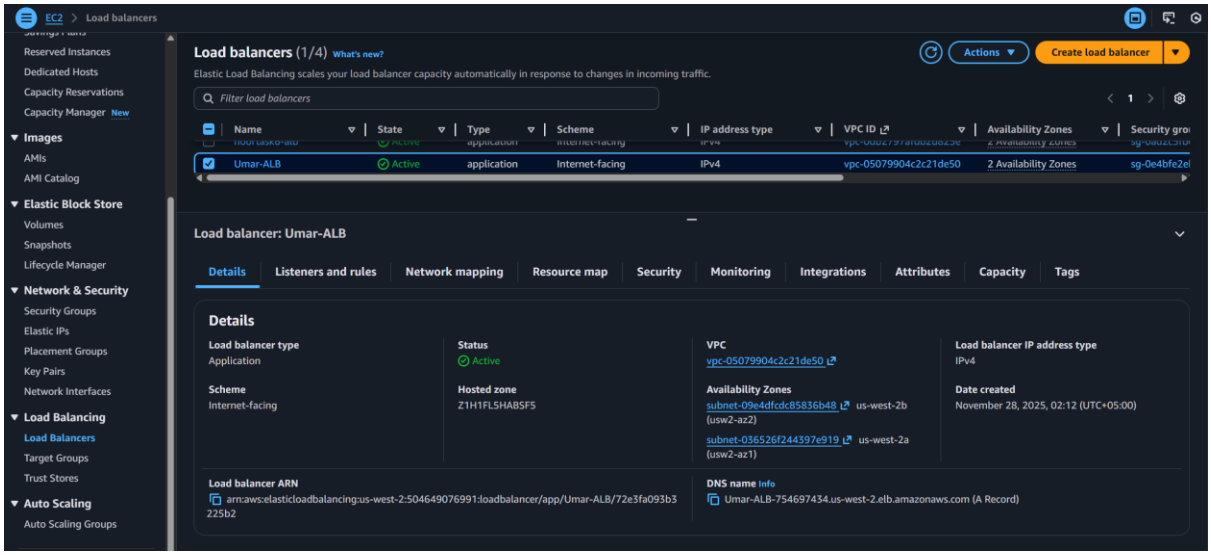
Security Group

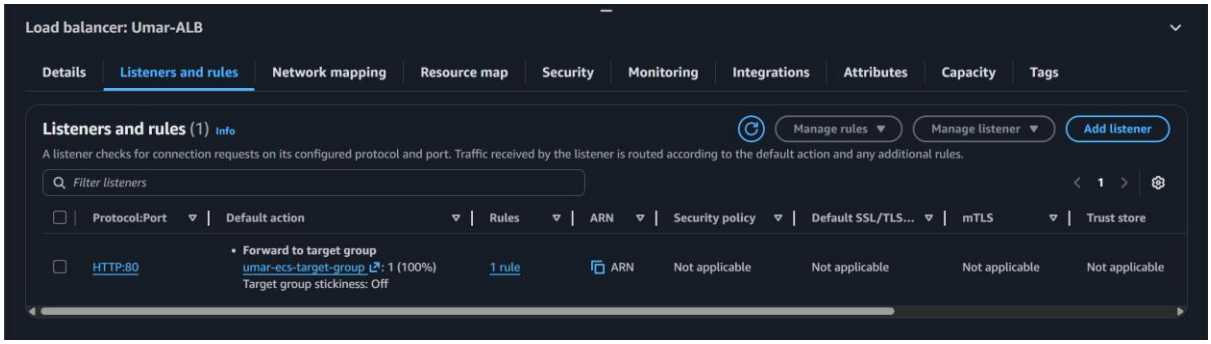
- Attach the ALB security group: **Umar-ALB-SG**. Allows inbound HTTP traffic on port 80 from all IP addresses.

Listeners:

- **Protocol:** HTTP
- **Port:** 80

A default action will be added later to forward traffic to the target group once it is created.





Task 1.14.2: Create Target Group

The target group defines how requests are routed to backend EC2 instances. ECS tasks will run on the EC2 instances in bridge mode, listening on port 5000.

Steps:

1. In the EC2 Console, select **Target Groups** from the left navigation panel.
2. Click **Create target group**.
3. Configure the settings defined below.

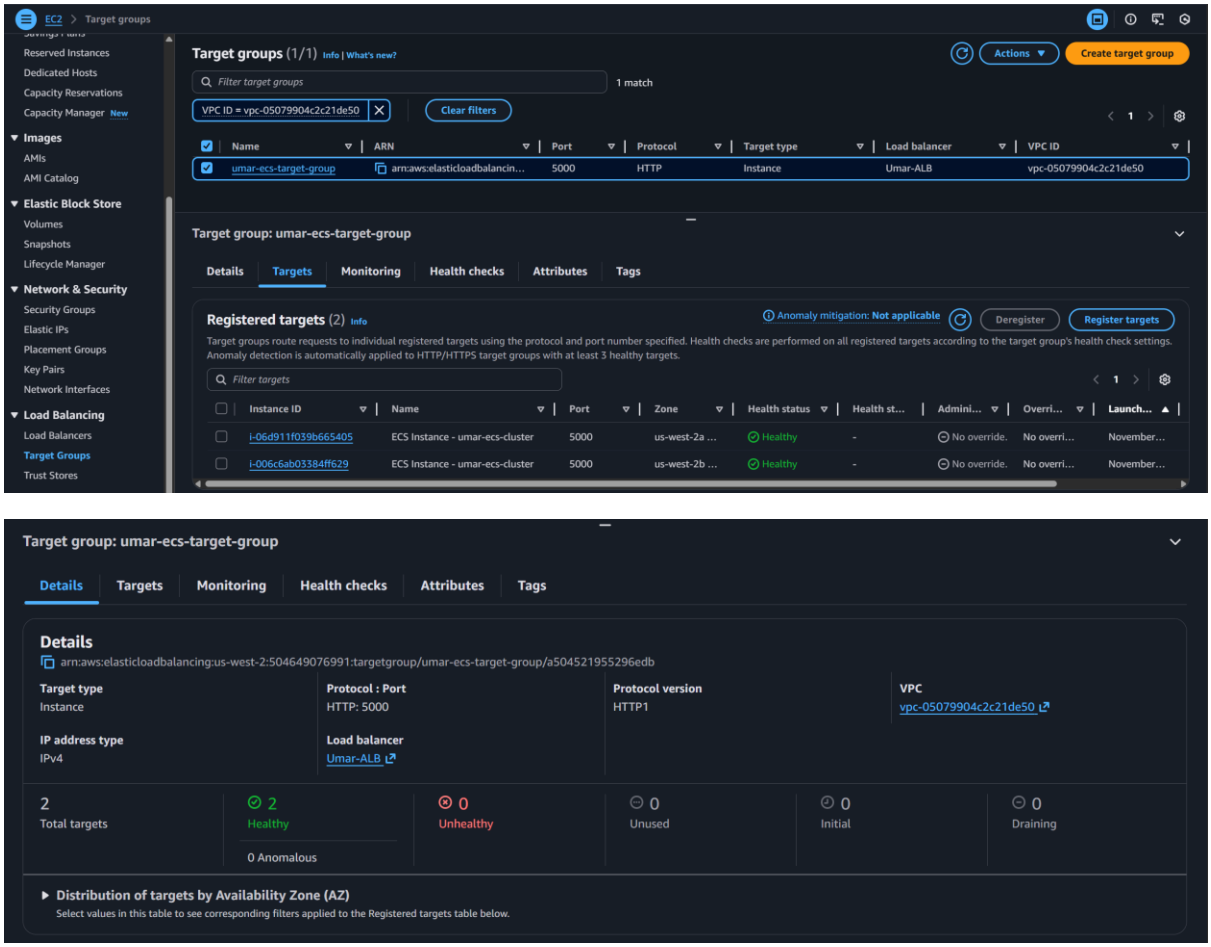
Target Group Settings

- **Target type:** Instance
- **Name:** umar-ecs-target-group
- **Protocol:** HTTP
- **Port:** 5000
- **IP address type:** IPv4
- **VPC:** Umar-VPC
- **Protocol version:** HTTP1

Continue to the next step and register the EC2 targets.

Register Targets

- Select the two private ECS EC2 instances created earlier.
- Register both instances. Each will receive traffic on port 5000 once the ECS service deploys containers.



The Application Load Balancer and target group have been successfully created with the following features:

- ALB deployed in public subnets to receive external traffic.
- Listener on HTTP port 80 directing all traffic to the target group.
- Target group configured to forward traffic to private EC2 instances on port 5000.
- Security groups ensure only the ALB can communicate with the ECS EC2 instances.

With this configuration completed, the environment is prepared for the next step of creating the ECS Service that will link the task definition to the target group and run the application behind the ALB.

Task 1.15: ECS Service

An ECS Service is responsible for running and maintaining a specified number of task instances simultaneously within the ECS Cluster. It ensures that tasks remain healthy, restarts them if they fail, and integrates with the Application Load Balancer to distribute traffic across the running containers.

In this project, the ECS Service uses the task definition created earlier and deploys two container tasks across the EC2 instances in the cluster. The service also interacts with the ALB and target group to provide external access to the application.

Steps:

1. Navigate to the **ECS Console** and select the existing cluster named **umar-ecs-cluster**
2. Click the **Services** tab and then choose **Create**.
3. Configure the service using settings defined below.

Service details

- **Task definition Family:** Umar-ecs-ec2-task-definition
- **Revision:** Latest. This defines the container image, CPU/memory resources, network mode, and port mappings.
- **Service name:** Umar-ecs-ec2-task-definition-service

Environment

- **Existing cluster:** umar-ecs-cluster
- **Compute options:** Capacity provider strategy
    - **Capacity provider:** Auto Scaling Group generated during cluster creation
    - **Base:** 0
    - **Weight:** 1

Deployment configuration

- **Scheduling strategy:** Replica
- **Desired tasks:** 2
- **Availability Zone rebalancing:** Turn on
- **Deployment options**
    - **Deployment strategy:** Rolling update
    - **Minimum running tasks:** 50%
    - **Maximum running tasks:** 150%

Load Balancing

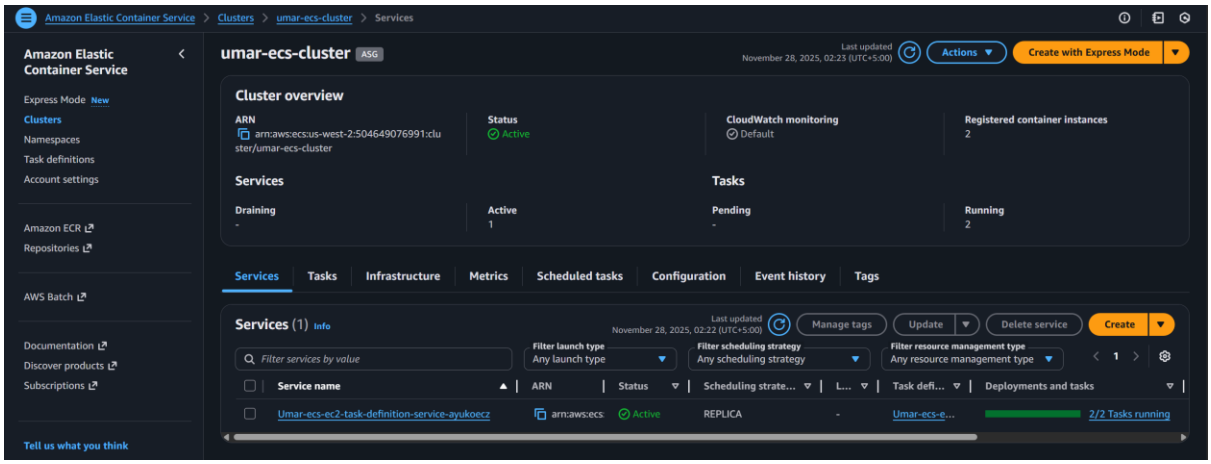
The ECS Service integrates with the application's existing ALB:

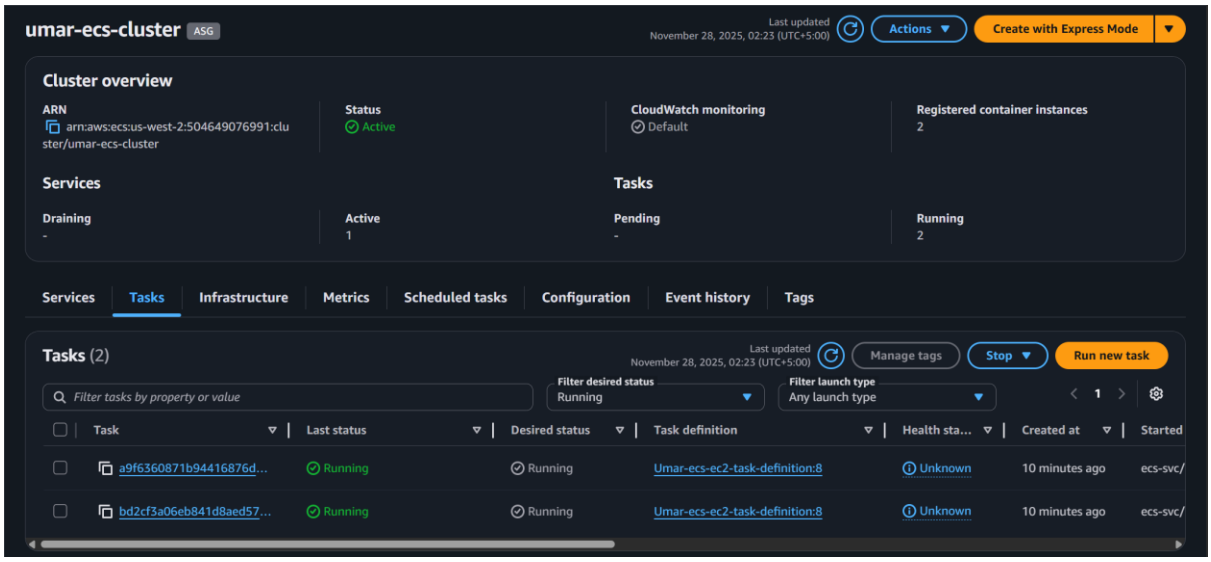
- **Load balancer type:** Application Load Balancer
- **Listener:** HTTP : 80
- **Target group:** umar-ecs-target-group (port 5000)

ECS automatically registers and deregisters tasks with the target group, enabling dynamic load balancing across the EC2 instances.

Once the service is created:

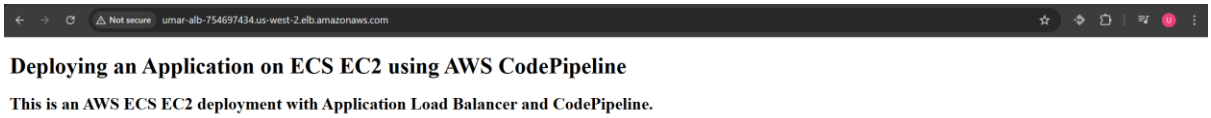
- ECS deployed **two running tasks** across the private EC2 instances in **bridge mode**, exposing port 5000.
- Each running task was automatically registered to the target group associated with the ALB.
- The ALB forwards HTTP traffic (port 80) to the target group, which sends traffic to the containers on port 5000.
- ECS maintains task health and ensures two tasks remain running at all times.





With the ECS Service successfully configured, the backend application environment is ready for CI/CD integration through CodeBuild and CodePipeline.

Verify that application is working using ALB DNS Name



Task 1.16: CodeBuild

In this task, an AWS CodeBuild project was created to automate the process of building Docker images, pushing them to Amazon ECR, and preparing artifacts for CodePipeline. The build environment uses a GitHub repository as the source, enabling continuous delivery whenever new commits are pushed.

CodeBuild Configuration

Project Setup

- **Project name:** Umar-EC2-ECS-CodeBuild
- **Project type:** Default

Source

- **Source provider:** GitHub
- The build project is directly connected to the GitHub repository used for the Flask application
- **GitHub repository:** Umarsatti1/Task-7-Deploying-an-Application-on-ECS-EC2-using-AWS-CodePipeline.

Environment Configuration

- **Provisioning model:** On-Demand
- **Environment image:** Managed image
- **Compute type:** EC2
- **Runtime:** Container
- **Operating system:** Amazon Linux
- **Image:** aws/codebuild/amazonlinux-x86_64-standard:5.0 (latest at time of creation)

Additional Configuration

- **Privileged mode:** Enabled
- Required so CodeBuild can run Docker commands (e.g., build and push Docker images).
- **VPC:** Umar-VPC (created earlier)
- **Subnets:** Private Subnets A and B
- **Security group:** Umar-ECS-SG
    - Ensures CodeBuild can communicate with ECR and other internal services securely within the VPC.
- **Environment Variables**
    - **ACCOUNT_ID:** AWS account ID
    - **REGION:** us-west-2
    - **REPOSITORY_NAME:** name of the ECR repo (created earlier)
    - These variables are referenced inside the buildspec YAML file.

Buildspec (YAML) Configuration

A buildspec file was created to automate:

- Authentication to Amazon ECR
- Docker image build
- Docker image tagging
- Docker image push to ECR
- Generation of imagedefinitions.json, which is required by CodePipeline and ECS deployments

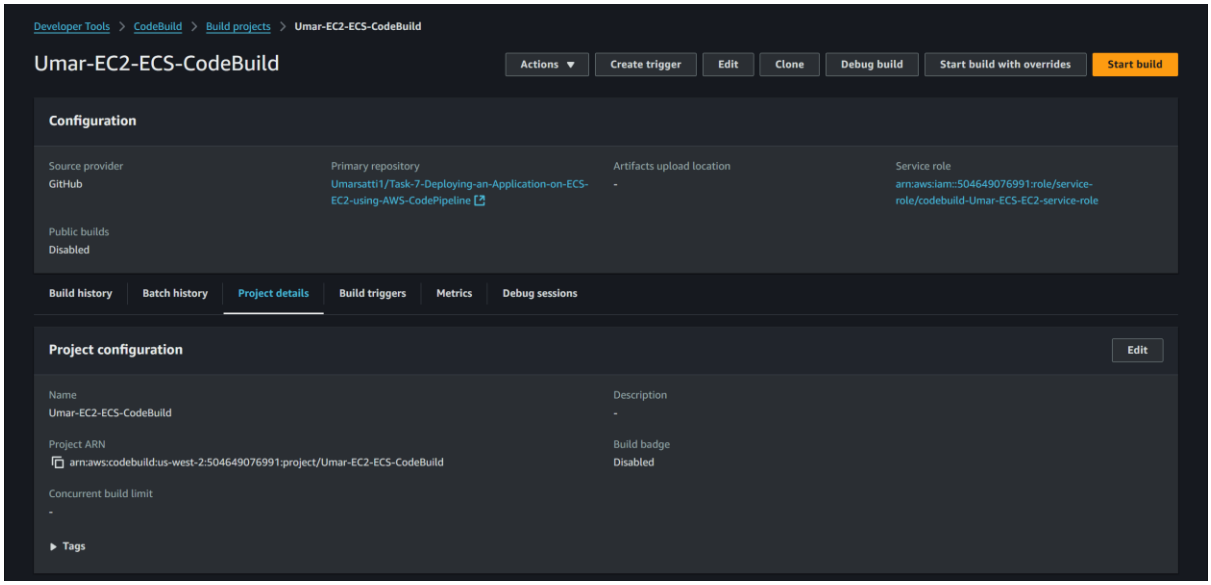
This ensures that any update to the GitHub repository triggers an image rebuild and push process.

Artifacts

- **Output file:** imagedefinitions.json

Logging

- **CloudWatch Logs:** Enabled
- **Log group name:** Umar-EC2-ECS-CodeBuild



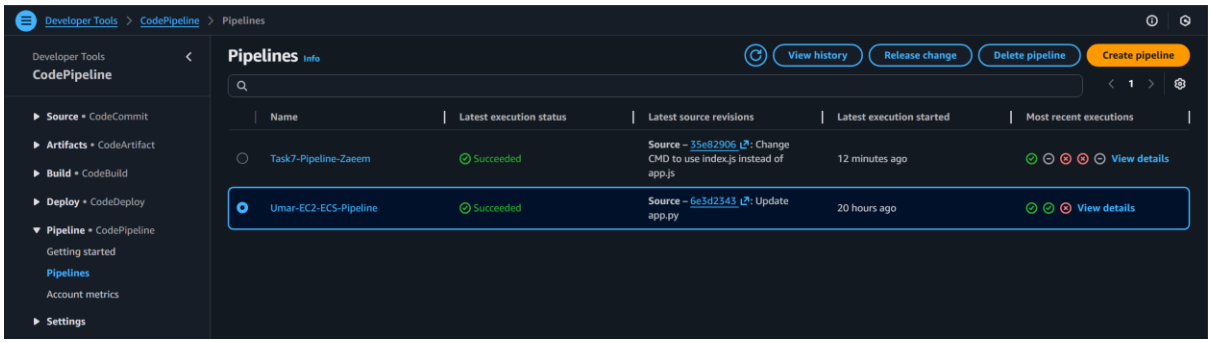
The CodeBuild project is now fully configured to:

- Listen to changes in the GitHub repository
- Build and push updated Docker images to ECR
- Generate deployment artifacts for CodePipeline

This setup completes the CI portion of the CI/CD process for the ECS EC2-based architecture.

Task 1.17: CodePipeline

In this task, an AWS CodePipeline was created to automate the continuous integration and continuous deployment (CI/CD) process for the ECS EC2-based application. This pipeline integrates GitHub (source), CodeBuild (build), and ECS (deploy), ensuring that any code changes pushed to the repository automatically trigger a rebuild of the Docker image, upload it to ECR, and deploy the updated container to the ECS cluster.



- In the **CodePipeline** page, click **Create pipeline** button on the top right.
- Use the following settings to create the CodePipeline.

Step 1: Create Pipeline

- **Creation option:** Build custom pipeline

Step 2: Pipeline Settings

- **Pipeline name:** Umar-EC2-ECS-Pipeline
- **Execution mode:** Queued (default)
- **Service role:** AWSCodePipelineServiceRole-us-west-2-Umar-EC2-ECS-Pipeline

This IAM role includes all required CodePipeline permissions.

Step 3: Add Source Stage

- **Source provider:** GitHub
- **Connection:** umarsatti (same connection used in CodeBuild)
- **Repository:** Task-7
- **Branch:** main
- **Output artifact format:** CodePipeline default
- **Enable automatic retry on stage failure:** Enabled
- **Webhook:** Start pipeline on push and pull request events

This ensures that any commit to the main branch automatically triggers the pipeline.

Step 4: Add Build Stage

- **Build provider:** AWS CodeBuild
- **Project name:** Umar-EC2-ECS-CodeBuild
- **Build type:** Single build
- **Region:** US West (Oregon)
- **Input artifacts:** SourceArtifact
- **Enable automatic retry:** Enabled

The build stage handles Docker image creation, ECR push, and generates imagedefinitions.json.

Step 5: Test Stage

- **Skipped** (not required for this project)

Step 6: Add Deploy Stage

- **Deploy provider:** Amazon ECS
- **Region:** US West (Oregon)
- **Input artifacts:** BuildArtifact
- **Cluster:** umar-ecs-cluster
- **Service:** Umar-ecs-ec2-task-definition-service-ayukoecz
- **Automatic rollback:** Enabled

This stage updates the ECS service with the new Docker image. ECS performs a rolling update: launching new tasks, ensuring they are healthy, then draining and stopping old tasks.

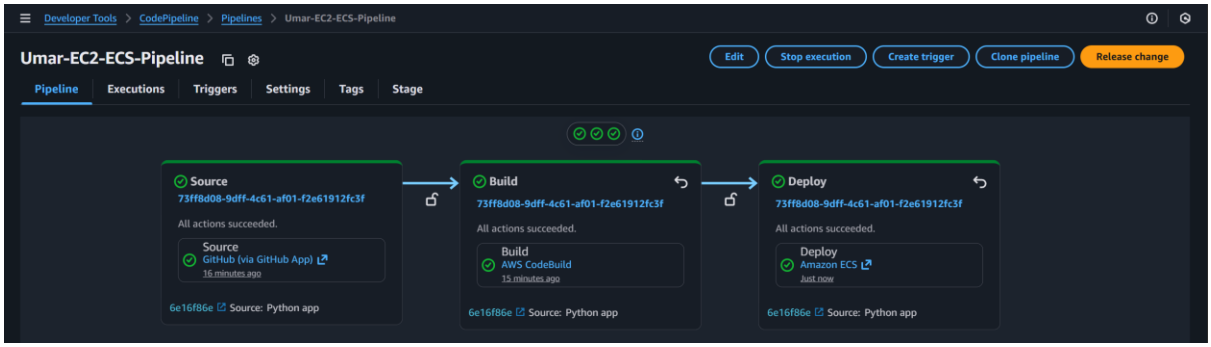
Step 7: Review and Create

- All configuration details were reviewed and the pipeline was created successfully.

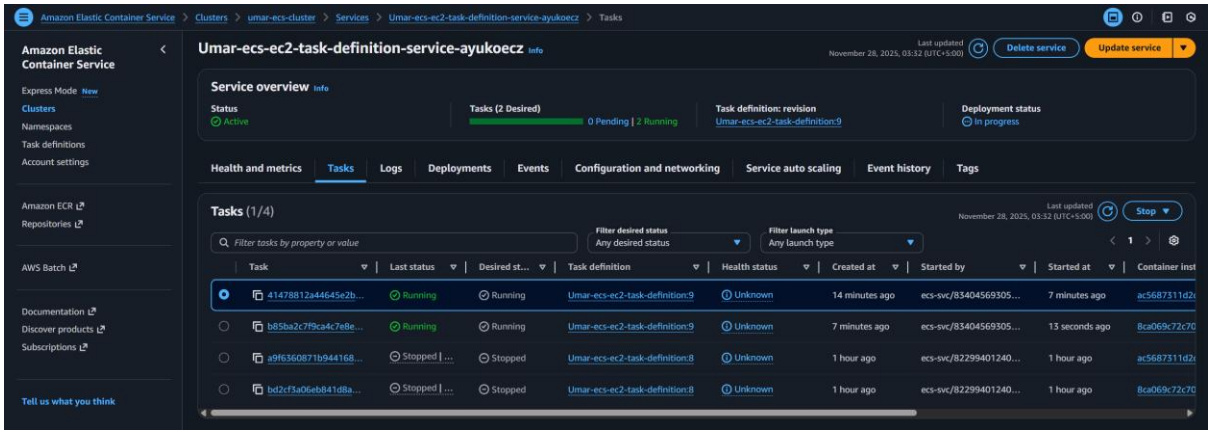
After creating the pipeline:

- The pipeline automatically executed all stages in sequence (Source → Build → Deploy).
- ECS registered **new tasks**, deployed the updated container image, and **stopped the old tasks** once the new ones became healthy.
- Events logs confirmed successful registration of new tasks and deregistration of the old ones.
- The ALB DNS endpoint successfully served the updated Flask application, verifying that the deployment process works end-to-end.

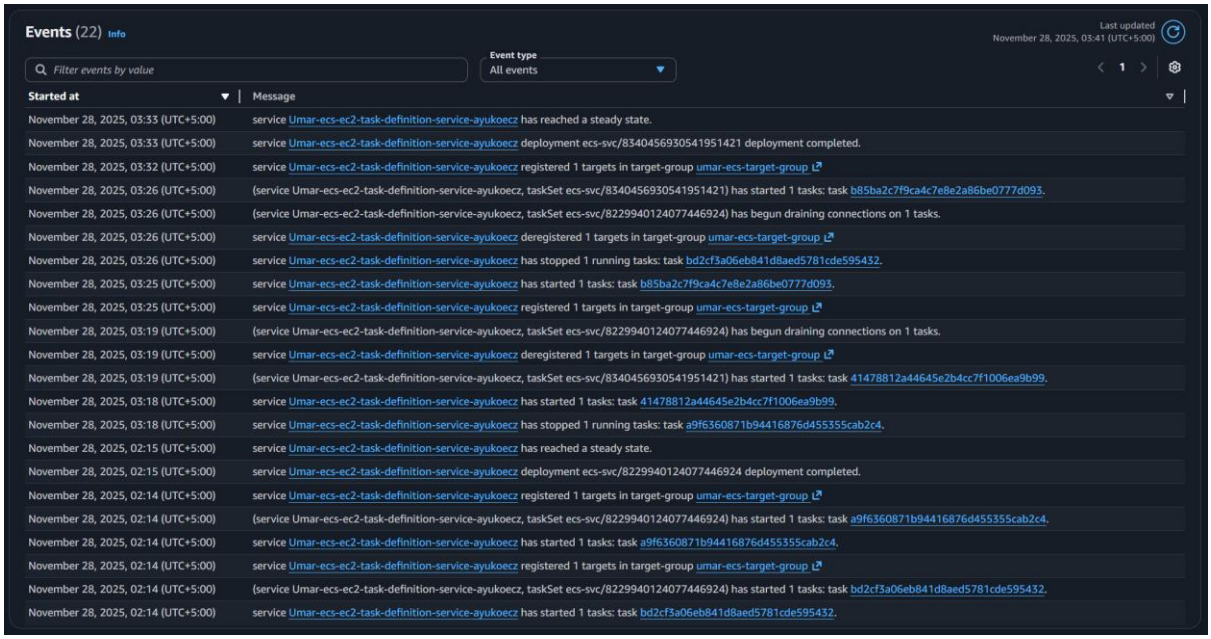
Results after creating CodePipeline and running it



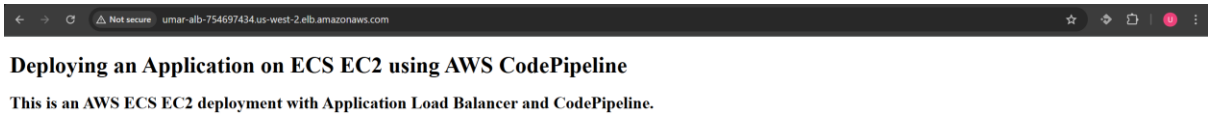
New tasks are created while the old ones are stopped



Events showing that new tasks were registered and old ones were stopped



ALB DNS Name to access website



Task 1.18: End-to-End Testing

This final task verifies the complete CI/CD workflow from source code change to automated deployment on ECS using the CodePipeline and CodeBuild setup created earlier. The goal is to confirm that updates to the application are automatically detected, built, pushed to ECR, deployed through ECS, and reflected instantly through the Application Load Balancer (ALB).

Steps Performed

1. Update the Application

A small change was made to the **app.py** file. This change ensures that the application output visibly reflects the deployment of a new version.

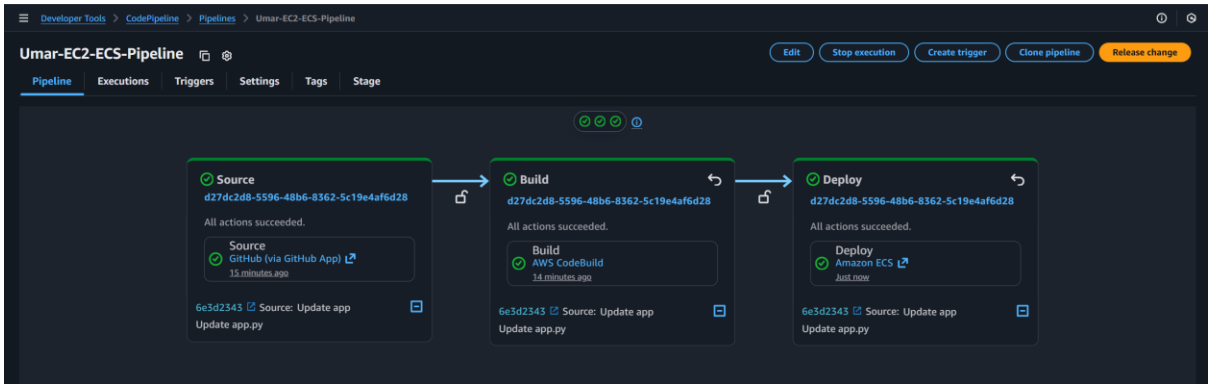
2. Push Changes to GitHub

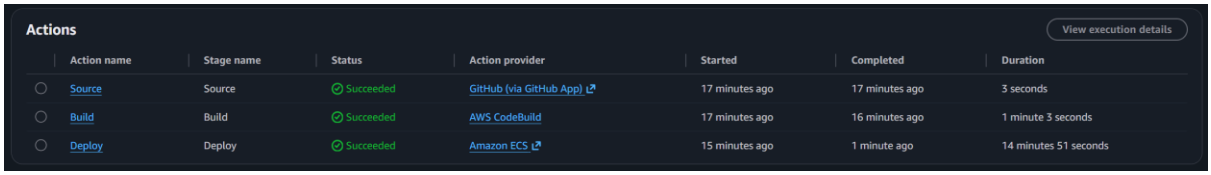
Once the update was committed and pushed to the GitHub repository, the webhook triggered **CodePipeline** automatically.

3. Pipeline Execution

Upon receiving the commit event:

- **Source Stage** retrieved the latest code from GitHub
- **Build Stage** (CodeBuild):
  - Built a new Docker image
  - Tagged it with the commit identifier and "latest"
  - Pushed it to the Amazon ECR repository
  - Generated the imagedefinitions.json artifact
- **Deploy Stage** (ECS):
  - Updated the ECS service with the new image
  - Initiated a rolling deployment

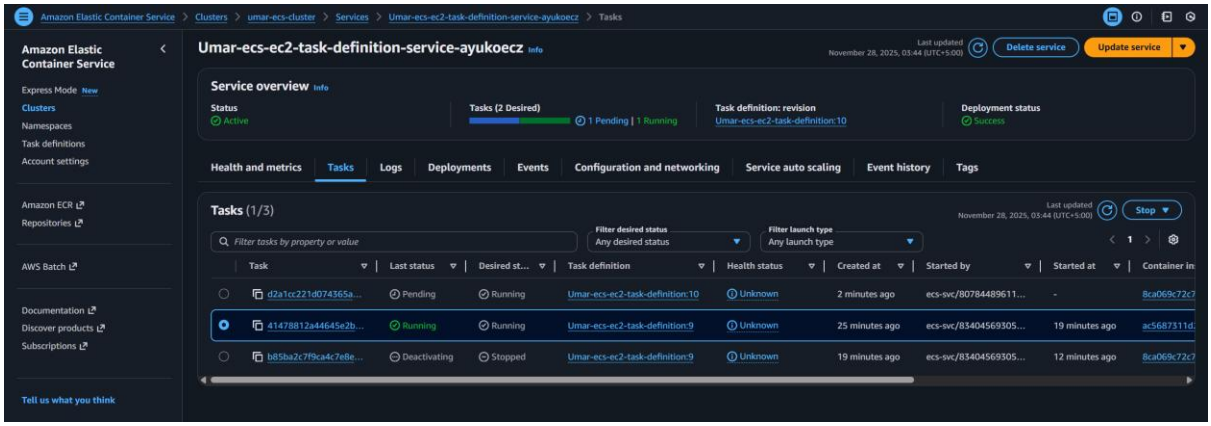


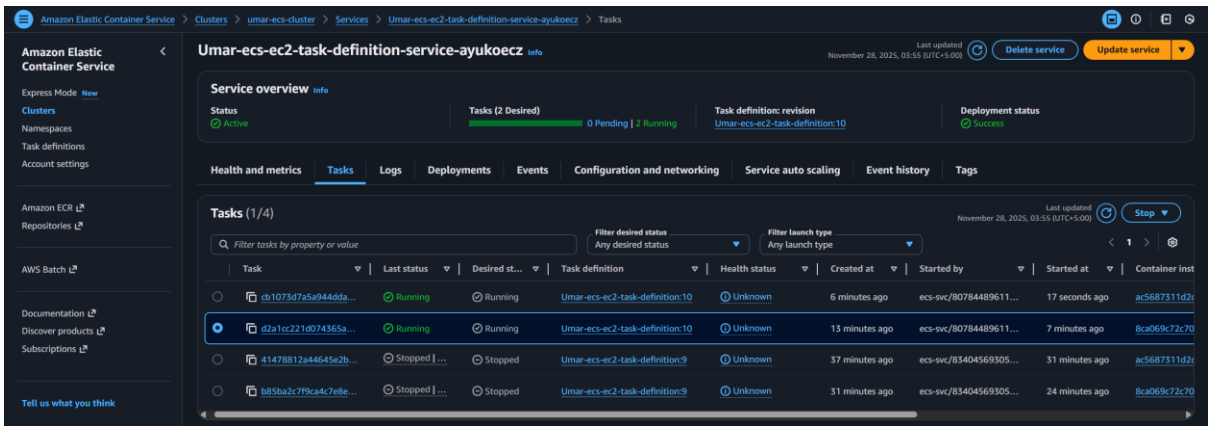


4. ECS Rolling Deployment

ECS handled the deployment automatically:

- New tasks were launched using the **latest task definition revision**
- Once the new tasks became healthy, ECS **deregistered and stopped** the old tasks
- Target group health checks confirmed the new tasks were ready to serve traffic

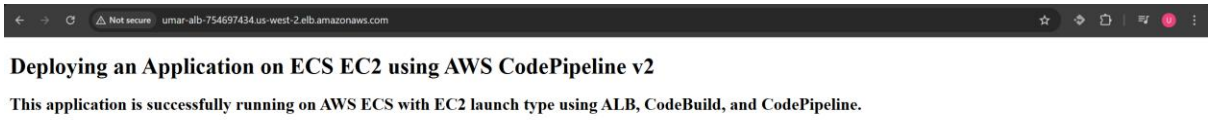




5. Verification Using ALB DNS

Finally, the Application Load Balancer URL was accessed.
The application displayed the updated output from the modified app.py file, confirming that:

- The new Docker image was built and pushed successfully
- ECS deployed the new version
- The ALB routed traffic to the updated tasks



Result

The end-to-end CI/CD workflow is functioning correctly.

- Any code changes trigger a new pipeline execution
- Docker images are rebuilt and published to ECR automatically
- ECS deploys new tasks via rolling updates with no downtime
- The ALB serves the updated application immediately

This validates that the complete AWS DevOps pipeline from code commit to deployment is fully operational.

Task 1.19: IAM Roles, CloudWatch Logs, and S3 Bucket

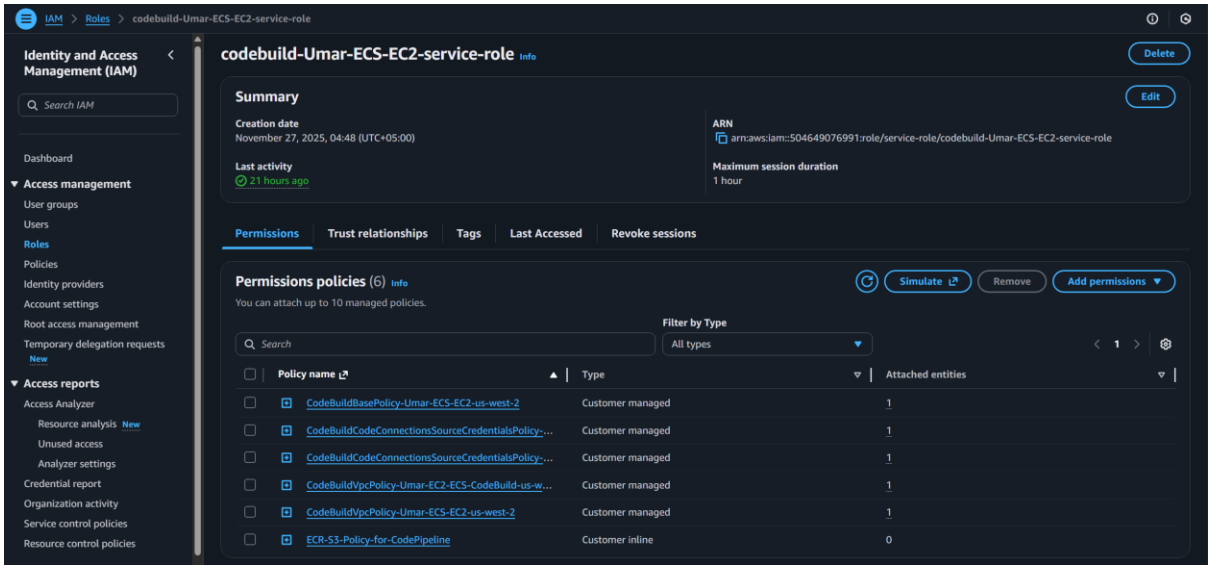
This task documents the IAM roles, CloudWatch log groups, and S3 artifact storage created as part of the automated CI/CD pipeline for ECS.

Task 1.19.1: IAM Roles and Permission Policies

Several IAM roles were created and automatically associated with CodeBuild, CodePipeline, and ECS during earlier tasks. These roles ensure the pipeline has the required permissions to build Docker images, push them to ECR, interact with GitHub via CodeConnections, update ECS services, and store pipeline artifacts in S3.

Below is a summarized explanation of what each role does (instead of listing full JSON policy documents).

1. CodeBuild Role: codebuild-Umar-ECS-EC2-service-role

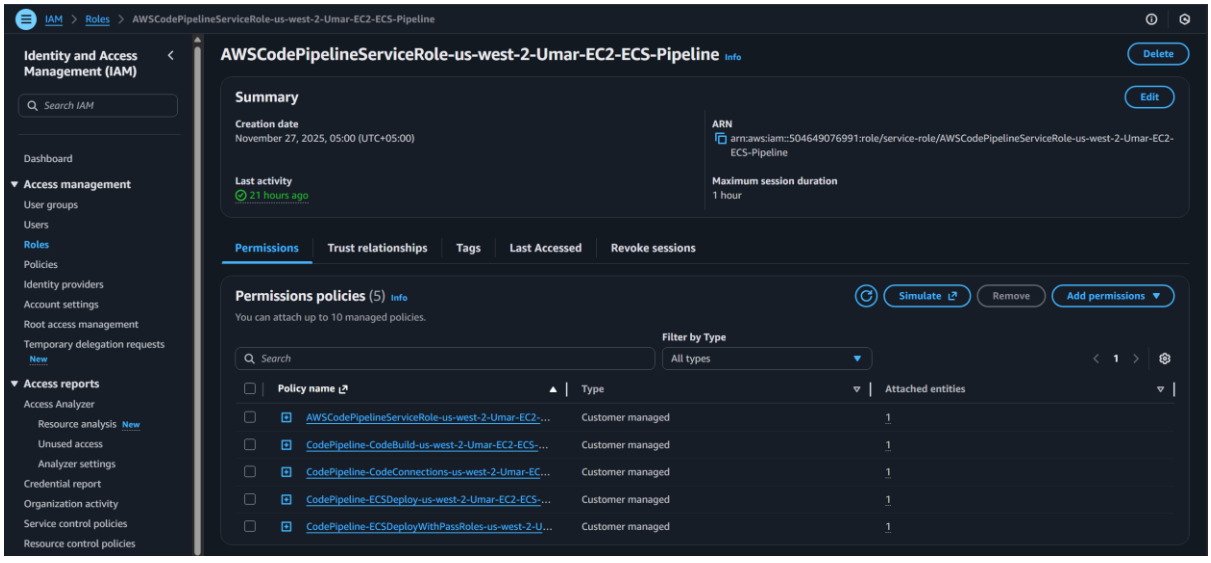


This role includes policies that allow CodeBuild to:

- Write build logs to CloudWatch
- Read and upload artifacts to the S3 bucket used by CodePipeline
- Authenticate with GitHub via CodeConnections
- Create and manage network interfaces within the VPC
- Build and push Docker images to Amazon ECR

These permissions enable the build process (docker build and docker push) to run inside private subnets securely.

2. CodePipeline Role: AWSCodePipelineServiceRole-us-west-2-Umar-EC2-ECS-Pipeline



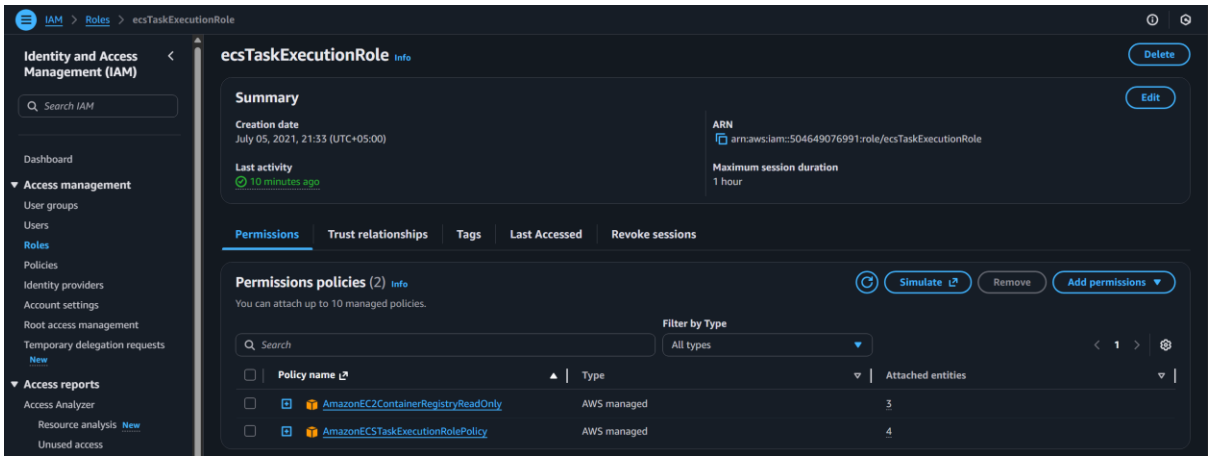
This role contains policies that allow CodePipeline to:

- Access the S3 artifact bucket (read/write objects)

- Trigger CodeBuild builds
- Use CodeConnections to pull source code from GitHub
- Register new task definitions and update the ECS service
- Pass the ecsTaskExecutionRole to ECS tasks during deployment

These permissions allow CodePipeline to orchestrate the entire CI/CD workflow from source → build → deploy.

3. ECS Task Execution Role: ecsTaskExecutionRole



This role is used by ECS tasks to:

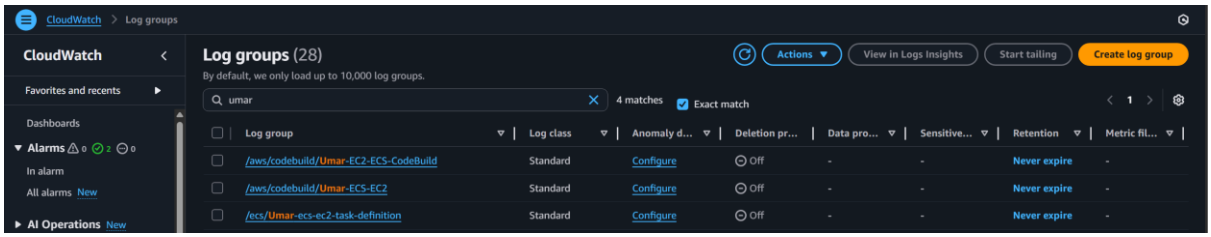
- Pull container images from Amazon ECR
- Write container logs to CloudWatch
- Access any additional runtime services your application may require

This role ensures the ECS service can run containers securely.

Task 1.19.2: CloudWatch Log Groups Created

The following CloudWatch log groups were created automatically through CodeBuild and ECS:

- /aws/codebuild/Umar-EC2-ECS-CodeBuild
- /aws/codebuild/Umar-ECS-EC2
- /ecs/Umar-ecs-ec2-task-definition



These log groups store:

- Build logs (Docker image build logs)
- Application logs generated by ECS tasks
- ECS deployment and container output
- Errors and debugging information

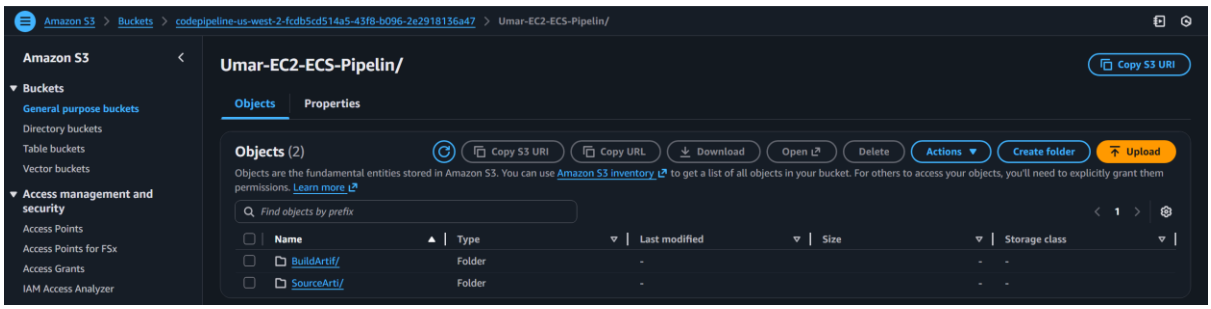
Task 1.19.3: S3 Bucket Pipeline Artifacts

A dedicated S3 bucket was created by CodePipeline:

- **Bucket name:** codepipeline-us-west-2-fcdb5cd514a5-43f8-b096-2e2918136a47

Inside the bucket, the pipeline creates the following folders:

- **SourceArtifact/:** contains zipped copies of the GitHub source code
- **BuildArtifact/:** contains the imagedefinitions.json file generated by CodeBuild



These artifacts are essential for:

- Passing source code into CodeBuild
- Passing the latest image metadata into ECS during deployment

Task 1.20: Troubleshooting and Lesson Learned

During the setup of the ECS-based CI/CD pipeline using CodeBuild, CodePipeline, ECR, and ALB, several issues were encountered that affected container deployment, Docker builds, and ECR authentication. This section documents each problem, its root cause, and the implemented solution, serving as a reference for anyone following this deployment guide.

Issue 1: Incorrect Host Port Mapping in ECS Task Definition

Problem Description:
During ECS service creation, the task definition was configured with the wrong ***Host port*** mapping. As a result, the application failed to register healthy targets in the ALB target group.

Root Cause:
The task definition was set to use a dynamic host port instead of a fixed port.
Since the application listens specifically on port 5000, using a dynamic port caused mismatches between the container port, host port, and target group configuration.
Additionally, the Target Type was mistakenly set to ***IP*** instead of ***Instances***, which is not suitable for EC2 launch type services.

Solution:

- Set Container port = 5000 and Host port = 5000 (no dynamic port mapping).
- Updated the target group to use Target type "**Instances**". This ensured that ECS correctly registered EC2 instances as targets and routed traffic to the container's fixed port.

Issue 2: ECR Login Failure During Build Stage

Problem Description:
CodeBuild was unable to authenticate with Amazon ECR, displaying errors when attempting to run "View push commands" or push Docker images.

Root Cause:
The CodePipeline IAM Service Role did not have the necessary ECR and S3 permissions required for CodeBuild to authenticate, upload layers, or push images to an ECR repository. Without these permissions, CodeBuild cannot obtain an authorization token or upload Docker layers.

Solution:
A custom policy was added to the CodePipeline service role, granting the following permissions:

- ECR actions (token retrieval, image upload, layer uploads)
- S3 object read/write access (required for pipeline artifacts)

This resolved the authentication issue and allowed Docker images to be pushed to ECR successfully.

Issue 3: Docker Build Failing in CodeBuild Environment

Problem Description:
While attempting to build the Docker image inside CodeBuild, the following error occurred: ***"Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?"***

Root Cause:
Docker cannot run in a CodeBuild environment unless Privileged Mode is enabled. By default, CodeBuild containers do not have access to the Docker daemon, so any docker build or docker push commands fail.

Solution:
During the creation of the CodeBuild project, the Privileged option under "**Additional configuration**" was enabled. This allowed the CodeBuild container to run Docker commands, build the container image, and push it to ECR.