Deploying a Microservice on AWS with Autoscaling Policy Project

Contents

1. Project Overview

This Project demonstrates how to deploy a stateless microservice on **AWS** using **EC2** instances with an **Auto Scaling Policy** and an **Application Load Balancer (ALB**).
The goal is to show how **AWS** automatically adds or removes **EC2** instances based on CPU load, ensuring high availability and cost efficiency.
The project also includes monitoring using **Amazon CloudWatch** and a centralized **RDS** database to separate application data from compute resources.

1.1 Objective

The main objective is to deploy a microservice (Coupon service) on **AWS** and configure it to automatically scale up or down based on **CPU utilization**.
The system should maintain stable performance under high load and reduce resources when demand decreases.

1.2 Tools Used

- **AWS EC2-** To host the microservice instances
- **Amazon RDS (MySQL)** – To store application data
- **Application Load Balancer (ALB)** – To distribute incoming requests evenly
- **Auto Scaling Group (ASG)-** To manage scaling up and down
- **CloudWatch** – To monitor CPU usage and trigger scaling events
- **Python**- For the CPU load simulation Script.
- **SSH / Linux Terminal**- For connecting and managing EC2 Instances.

1.3 System Overview

The system consists of a central **RDS** database connected to multiple **EC2** instances running the coupon microservice.
These instances are managed by an **Auto Scaling Group (ASG)** and are fronted by an **Application Load Balancer (ALB**). When CPU usage rises above the defined threshold, new **EC2** instances are launched automatically. When CPU usage falls, the system scales down by terminating extra instances.
CloudWatch is used to monitor performance and visualize scaling events.

1.4 Microservice Description

This project included a **Spring Boot Microservice** project named **CouponService**.   The application is a **Java**-based **Restful API** designed to create and retrieve discount coupons. Its main purpose was to demonstrate how a stateless microservice can be deployed on **AWS EC2** and connected to a centralized database (**Amazon RDS**).

Source Code and Modifications:

- **Database Integration**: Updated the code to establish a live connection with a **MySQL RDS** instead of using a local or in-memory database.

- **Packaging**: The modified application was complied and packaged into a **JAR file** (**couponservice-0.0.1-SNAPSHOT.jar**) using **Maven** inside **Eclipse**.

- **Deployment Setup**: The **JAR file** was uploaded to an **Amazon S3 Bucket** so that **EC2 instances** could download it automatically during launch.

- **Security and Permissions**: An **IAM Role** (**EC2S3AccessRole**) was created and attached to the **EC2 instance** to allow secure to the **S3 bucket** without embedding credentials.

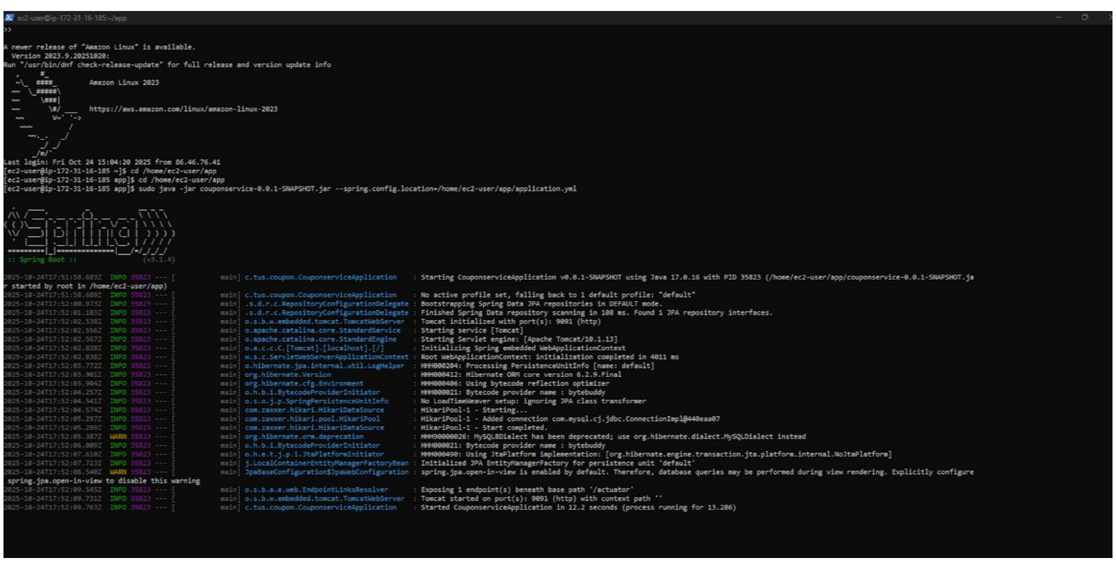

Figure 1: SpringBoot Application booted up on EC2 Instance via SSH

2. Introduction

In modern cloud computing, applications need to handle unpredictable workloads while maintaining high performance and cost efficiency.
Amazon Web Services (AWS) provides several services that make this possible, including **Elastic Compute Cloud (EC2)**, **Auto Scaling Groups (ASG)**, and **Launch Templates**. These services allow developers to deploy, manage, and scale applications dynamically. Those key concepts were demonstrated in this project through the deployment of a **Java**-based microservice on **AWS**.

2.1 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) is the main AWS service that provides virtual servers, known as **instances**, to run applications in the cloud. **[1]**
Each **EC2** instance functions like a regular computer, allowing you to install software, host web applications, or run backend services without maintaining physical hardware.
For example, in this project, an **EC2** instance was used to host the **CouponService** Spring Boot microservice.
The instance was configured to automatically download and run the JAR file from an **Amazon S3** bucket and connect to a **MySQL** database hosted on **Amazon RDS**.

EC2 offers flexibility in instance types, operating systems, and networking, making it ideal for scalable cloud deployments.

2.2 Auto Scaling

Auto Scaling is an **AWS** feature that automatically adjusts the number of running **EC2** instances based on current demand or performance metrics. When system load increases, **Auto Scaling** launches new instances to handle the traffic. When demand drops, it terminates unnecessary instances to save costs. **[2][3]**
In this project, **Auto Scaling** was used to dynamically add or remove **EC2** instances based on **CPU utilization** metrics collected by **Amazon CloudWatch**. This ensured that the **CouponService** microservice remained responsive under load while keeping resource usage efficient.

2.3 Launch Template

A **Launch Template** in **AWS** defines the configuration (Setup) used when new **EC2** instances are launched, such as the **AMI (Amazon Machine Image)**, instance type, key pair, security group, and user data scripts. Using a **Launch Template** allows **Auto Scaling Groups (ASG)** to create consistent and repeatable instances automatically. [4]

In this project, a **Launch Template** named **couponservice-template** was created.
It included the **Amazon Linux AMI**, the correct security group, and a **user data script** that started the microservice automatically when each instance booted. This approach simplified deployment and ensured every new instance behaved identically.

2.4 Vertical Scaling vs Horizontal Scaling

There are two main strategies for scaling applications: **vertical scaling** and **horizontal scaling**.

- **Vertical scaling** (also called **scaling up**) means increasing the resources of a single instance, such as upgrading from a **t2.micro** to a **t3.large** instance to gain more CPU and memory. It's simple but has limits because one machine can only grow so much. **[5]**

- **Horizontal scaling** (also called **scaling out**) means adding more instances instead of making one larger. This approach increases system capacity by distributing the workload across multiple servers. **[5]**

In this project, **horizontal scaling** was implemented using the **Auto Scaling Group (ASG)**, which launched multiple **EC2** instances behind an **Application Load Balancer (ALB)**.
When CPU usage rose above 70%, new instances were created to share the load.
When the workload decreased, the system scaled back down to one instance.
This approach demonstrated how horizontal scaling provides high availability and elasticity in cloud-based applications.

3. Configuration

This section explains how the cloud environment was set up and configured on **Amazon Web Services (AWS)** to deploy stateless microservice with automatic scaling and load balancing.
The goal of this configuration was to host a **Spring Boot**–based **Coupon Service** application that could automatically handle varying levels of traffic, maintain high availability, and recover from failures without manual intervention.

3.1 Environment Setup

The setup consisted of three main components:

1. **Launch Template**

2. **Auto Scaling Group (ASG)**

3. **Application Load Balancer (ALB)**

   All components were deployed within the same **Virtual Private Cloud (VPC)**, using two **Availability Zones (us-east-1a and us-east-1b)** for redundancy.

3.2 Launch Template – Configuration Details

The first step in implementing Auto Scaling was to create a **Launch Template** based on the existing **EC2** instance that successfully hosted the ***CouponService*** microservice earlier on. This template defines how new **EC2** instances should be launched and configured automatically whenever scaling events occur. It ensures every instance is an identical, stateless copy of the original service, maintaining consistency across deployments.

3.2.1 Key Configuration Details

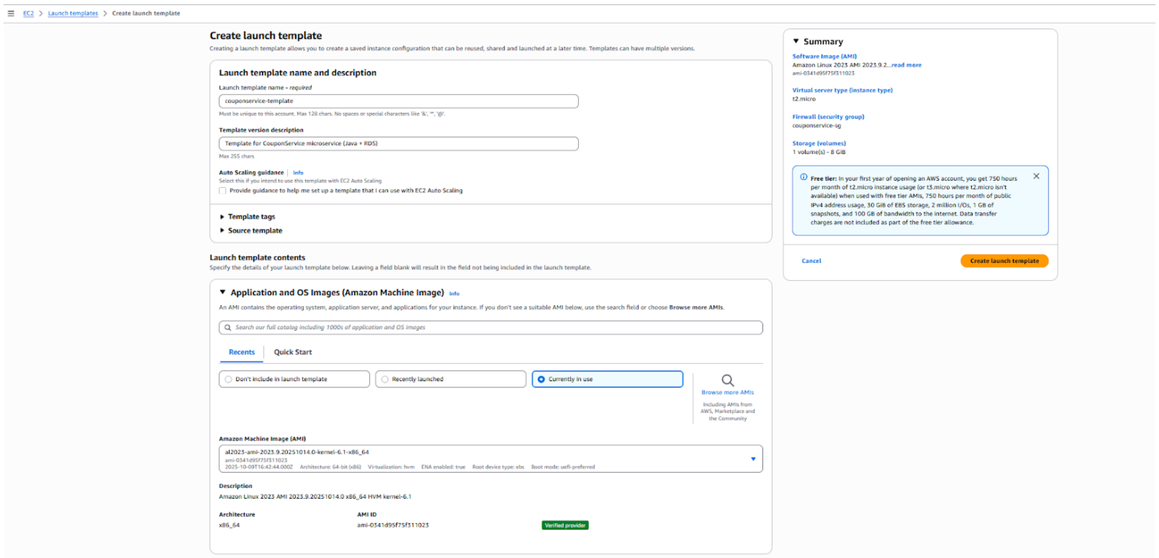

Figure 2: Launch Template Creation Page in AWS Console showing basic configuration

- **Template Name:** couponservice-template
- **Description:** Template for *CouponService* microservice (**Java** + **RDS**)
- **Amazon Machine Image (AMI):** *Amazon Linux 2023 (x86_64)* – selected for compatibility with **Java 17** and the **AWS CLI**.

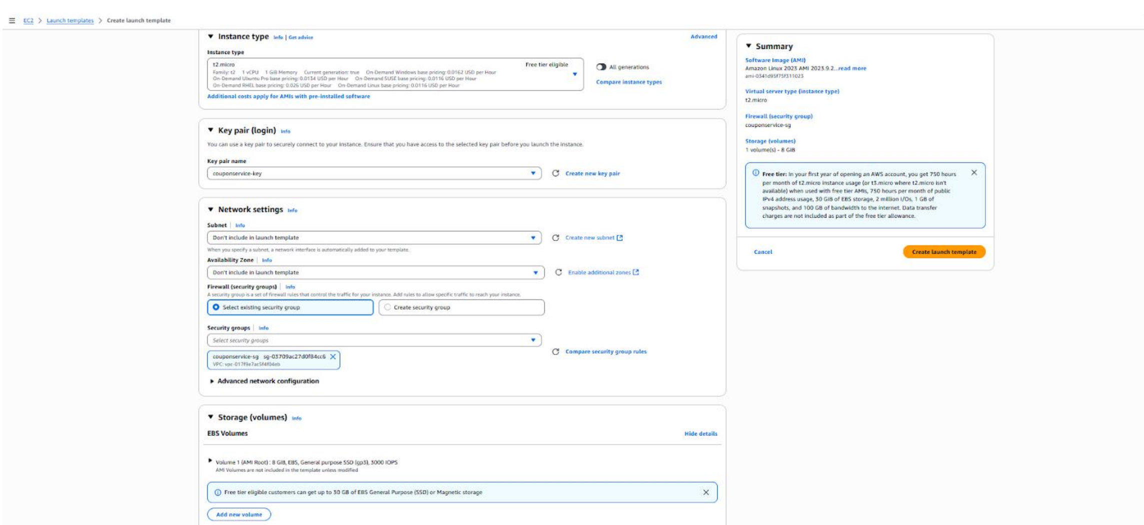

Figure 2.1: Instance type, key pair, network settings, and storage configuration for the Launch Template

- **Key Pair:** couponservice-key – enables secure **SSH** access when required.

- **Instance Type:** t2.micro (Free Tier eligible).

- **IAM Role:** EC2S3AccessRole – grants **AmazonS3ReadOnlyAccess** so each instance can automatically download the JAR file from S3 at startup.

- **Security Group:** couponservice-sg – allows inbound traffic on:

1. Port 22 (SSH)
2. Port 80 (HTTP)
3. Port 9091 (Application port for CouponService)

- **Storage:** 8 GiB (gp3 EBS volume) – adequate capacity for the Java application and log files.

3.2.2 User Data Script

A User Data script was embedded inside the Launch Template.
When a new EC2 instance launches, this script automatically performs all setup steps without any manual configuration:

```
#!/bin/bash
# Update and install dependencies
yum update -y
amazon-linux-extras install java-openjdk17 -y
yum install -y awscli -y

# Create application directory
mkdir -p /home/ec2-user/app
cd /home/ec2-user/app

# Download the application JAR from S3
aws s3 cp s3://couponservice-artifacts-aneeq/couponservice-0.0.1-SNAPSHOT.jar .

# Write Spring Boot configuration (application.yml)
cat << 'EOF' > /home/ec2-user/app/application.yml
spring:
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect

  datasource:
    url: jdbc:mysql://[redacted]amazonaws.com:3306/[redacted]
    username: admin
    password: <password>
    driver-class-name: com.mysql.cj.jdbc.Driver

server:
  port: 9091
EOF

# Run the Spring Boot service in the background and log output
cd /home/ec2-user/app
nohup java -jar couponservice-0.0.1-SNAPSHOT.jar > /home/ec2-user/app/app.log 2>&1 &
```

Figure 2.2: Full startup script

The figure (3.2) above shows the full startup script:

1. Updates the system and installs **Java 17** and the **AWS CLI**.
2. Creates the application directory under /home/ec2-user/app.
3. Downloads the **JAR file** (couponservice-0.0.1-SNAPSHOT.jar) from the **S3 bucket** (couponservice-artifacts-aneeq).
4. Generates an **application.yml file** containing the **MySQL RDS** connection details.
5. Starts the **Spring Boot** service on port 9091 in the background and logs output to app.log.

Figure 2.3: User Data section of the Launch Template showing the startup script

Launch Template allows each new **EC2** instance created by the **Auto Scaling Group (ASG)** to automatically configure itself and start the microservice during boot.
No manual deployment is required. Because all application data is stored centrally in the **RDS MySQL database**, each instance remains **stateless.** That means, it can be replaced or terminated at any time without losing data.

3.3 Auto Scaling Group (ASG) – Configuration Details

AWS will create new **EC2** instances automatically from the **Launch Template** when **CPU utilization** exceeds 70%. When the CPU usage drops below 30%, **AWS** will terminate extra instances to save resources.
The system ensures that at least one instance of the microservice remains running at all times.
This setup prepares the environment for the **Application Load Balancer (ALB)** integration in the next stage, allowing traffic to be distributed evenly across instances.

3.3.1 Select the Existing Launch Template

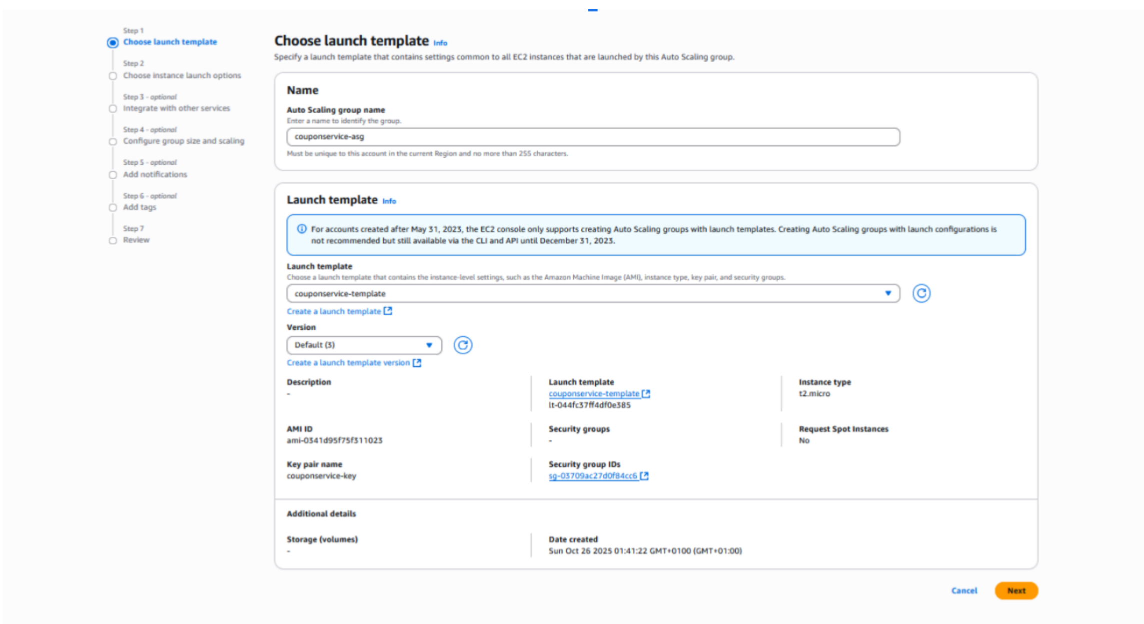

Figure 2.4: Creating the Auto Scaling Group in AWS Console by selecting the existing Launch Template (couponservice-template)

This step defines the foundation of the Auto Scaling process. The **Auto Scaling Group (ASG)** uses the launch template to ensure all newly created EC2 instances have consistent configurations. The group name “couponservice-asg” will later be referenced when connecting the system to the load balance.

3.3.2 Select the VPC and Availability Zones for ASG

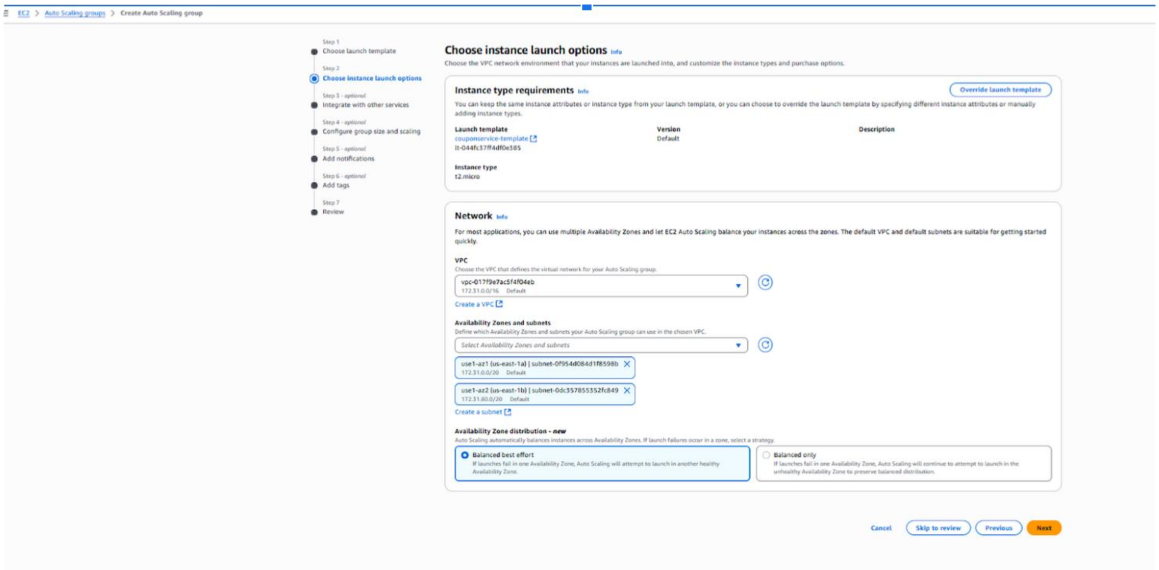

Figure 2.5: Selecting the VPC and Availability Zones (us-east-1a and us-east-1b) for the Auto Scaling Group to ensure high availability.

The **Auto Scaling Group (ASG)** is configured to deploy instances across multiple Availability Zones within the same region. This helps the application remain operational even if one zone becomes unavailable. The selected VPC defines the network boundaries for all instances launched by this group.

3.3.3 Check Health Checks

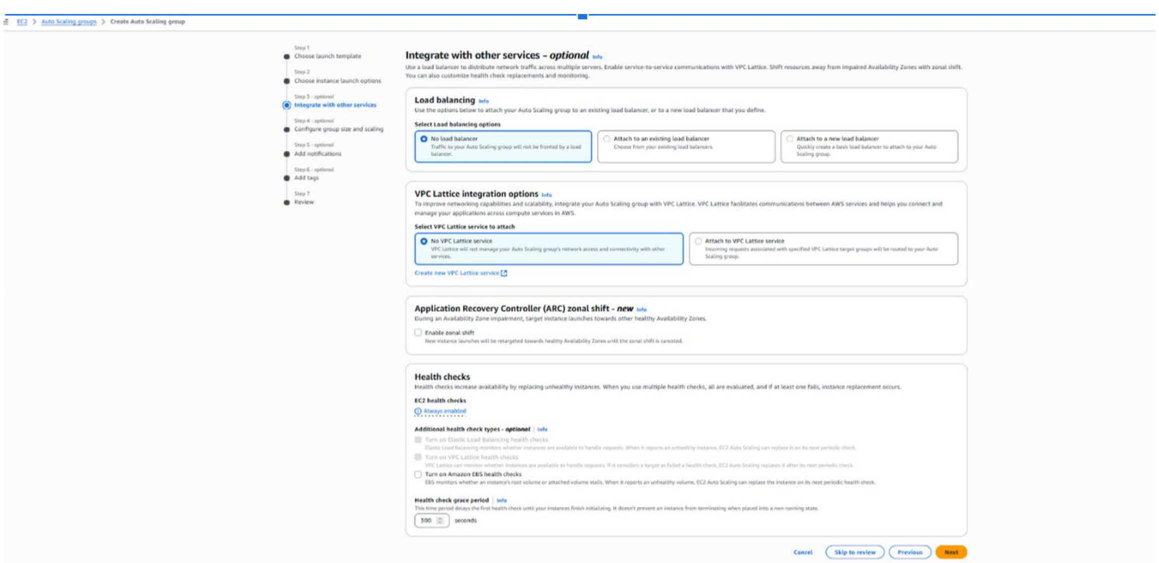

Figure 2.6: Integration with other AWS services, including options for load balancing, VPC Lattice, and health checks.

The figure (2.6, above) shows Health checks are enabled (By Default) so that **AWS** can automatically detect and replace unhealthy **EC2** instances. The load balancing option will later connect to an **Application Load Balancer (ALB)**, ensuring incoming traffic is evenly distributed. This integration enhances fault tolerance and resilience.

3.3.4 Setting up Scaling Policies and Group Size

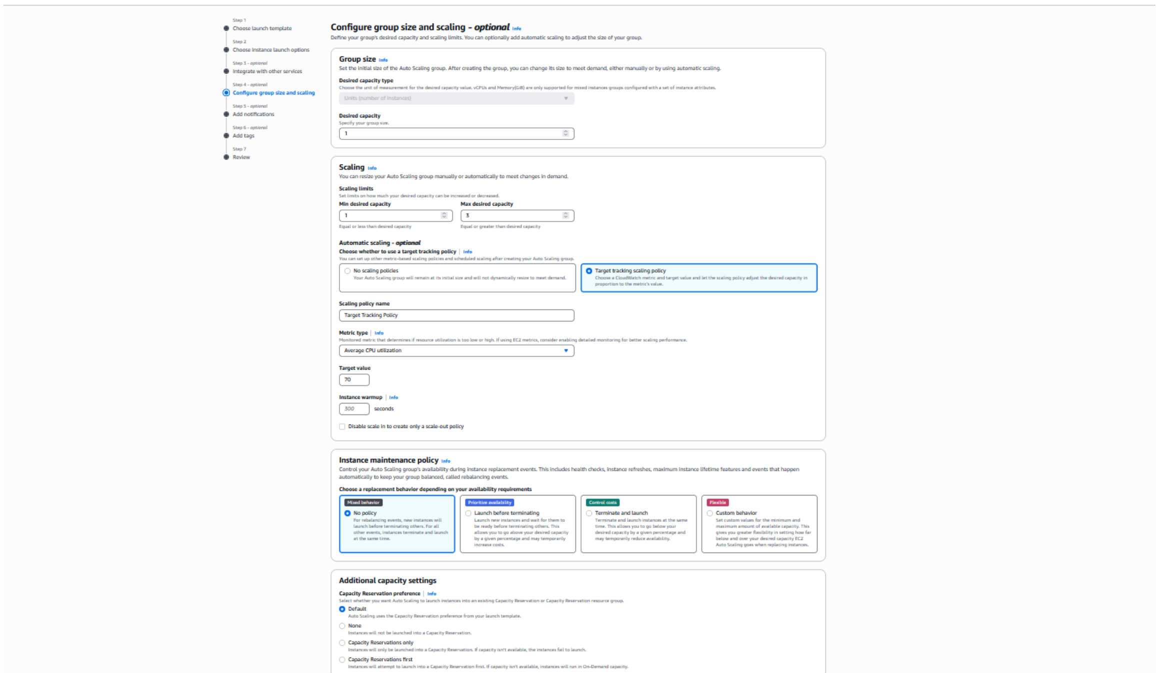

Figure 2.7: Setting up scaling policies and group size configuration for the Auto Scaling Group.

The group is configured with:

1. **Minimum capacity of 1:** There will always be at least one EC2 instance running to keep the application available.

2. **The desired capacity was also set to 1**: This indicates the target number of instances that should normally be active under regular load conditions.

3. **The maximum capacity was defined as 3**: This allows the system to scale out and launch up to three instances if the CPU utilization across running instances exceeds the 70% threshold.

A **Target Tracking Policy** based on **Average CPU Utilization (70%)** was applied. This allows the system to dynamically scale out when CPU load increases and scale in when the load decreases, maintaining optimal performance and cost
efficiency.
When demand increases (for example, heavy user traffic or CPU-intensive processing), **AWS** automatically creates additional **EC2** instances based on the **Launch Template** to balance the load. Once the workload decreases and **CPU utilization** falls below the threshold (around 30%), the **Auto Scaling Group (ASG)** gradually terminates the extra instances, returning to the desired single running instance.

3.3.5 Add Tags to the ASG

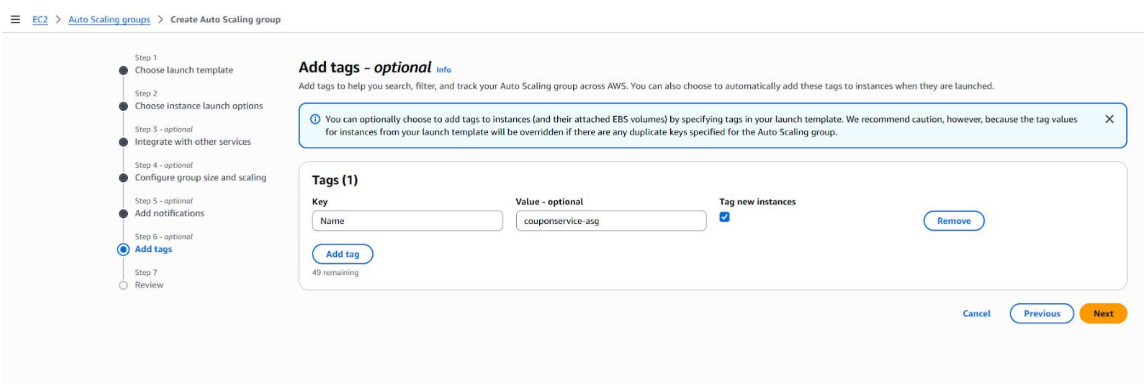

Figure 2.8: Adding a tag to the Auto Scaling Group for easy identification in the EC2 console.

‘ A tag with Key: Name and Value: ‘ **couponservice-asg** was added. This metadata helps organize **AWS** resources and makes it easier to identify all instances managed by **the Auto Scaling Group (ASG)** in large deployments.

3.3.6 Showing newly created Auto Scaling Group (ASG)

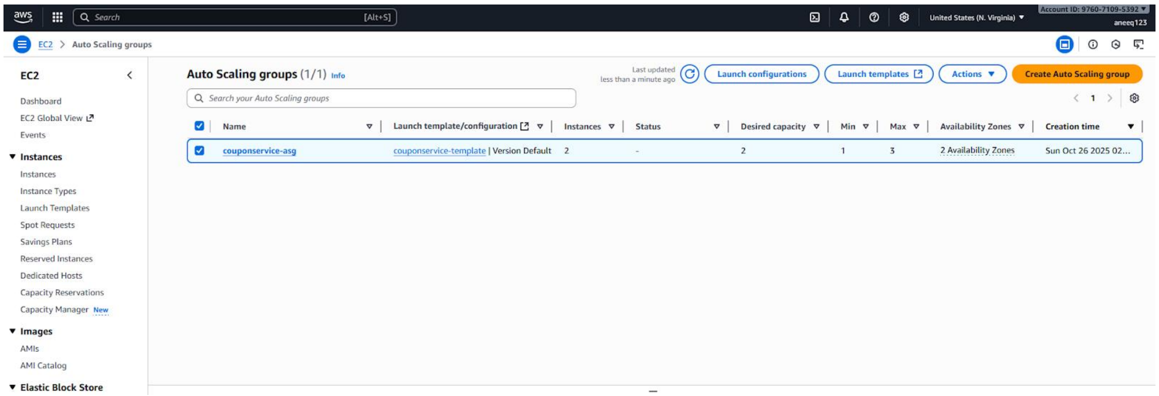

Figure 2.9: *AWS Console view showing the newly created Auto Scaling Group (ASG)*

After configuration, the **Auto Scaling Group (ASG)** is visible in the **AWS Management Console**. It shows the link to the couponservice-template, a desired capacity of one instance, and scaling limits of one to three instances across **two Availability Zones**.

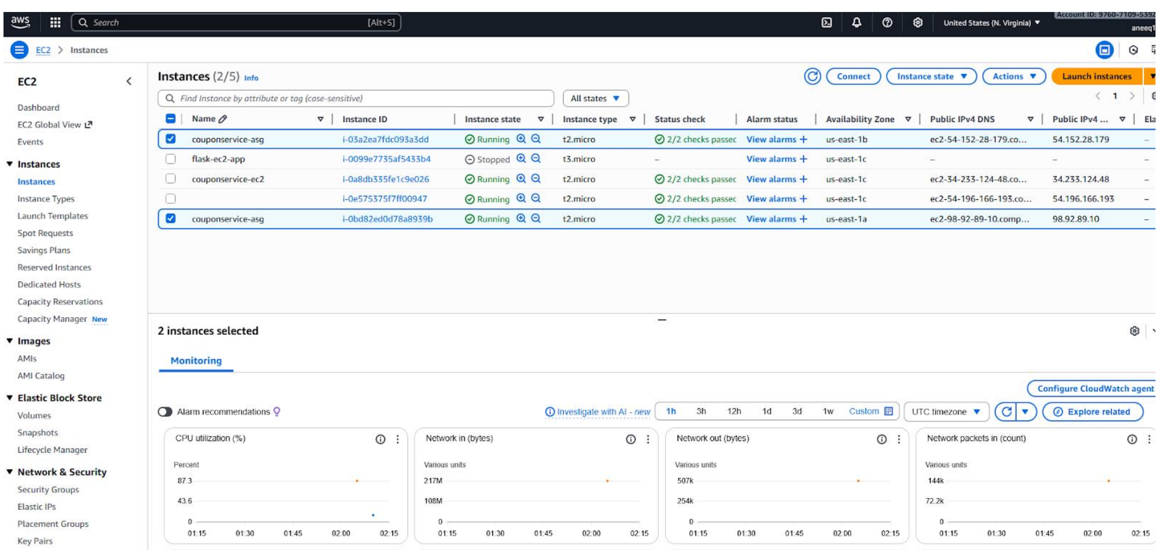

Figure 2.10: EC2 Instances page displaying multiple instances launched automatically by the Auto Scaling Group.

The Figure (2.10 above) confirms that the **ASG** has successfully launched two running **EC2** instances across separate **Availability Zones**. These instances are identical and fully configured based on the launch template. This validates that automatic scaling works as expected and that the system is ready for load balancing integration.

3.4 Application Load Balancer (ALB)

An **internet-facing Application Load Balancer (ALB)** was created to ensure that:

All user requests are routed through a single public endpoint (the **ALB** DNS name).

1. The **ALB** automatically distributes traffic across all healthy **EC2** instances managed by the **Auto Scaling Group (ASG)**.

2. The **ALB** continuously performs health checks, ensuring that only healthy backend instances receive incoming requests.

This configuration enhances **availability**, **fault tolerance**, and **scalability** for the deployed coupon service microservice.

3.4.1 Creating Application Load Balancer

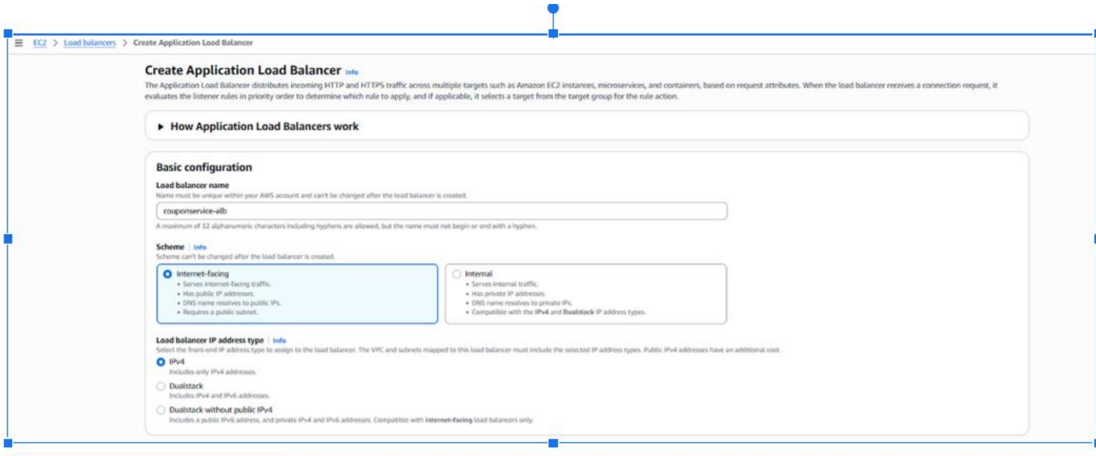

Figure 3: Creating the Application Load Balancer — Basic configuration section showing ALB name, scheme, and IP address type.

An **Application Load Balancer (ALB)** named couponservice-alb was created. The Internet-facing scheme was chosen to allow public web access, enabling users to reach the service externally via HTTP. The IPv4 address type was selected for simplicity and compatibility with the public network.

3.4.2 Network Mapping for ALB

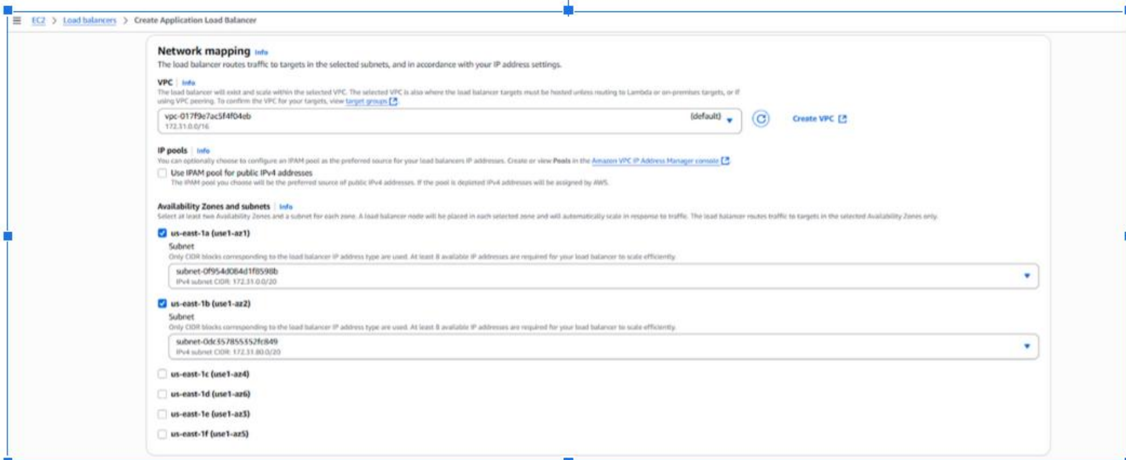

Figure 3.1: Network mapping for the ALB showing associated VPC and selected Availability Zones.

The **Application Load Balancer (ALB)** was associated with the existing **VPC** (vpc-017f9e7ac5f4f04eb) and configured to operate across two **Availability Zones**, us-east-1a and us-east-1b.
This setup ensures redundancy and high availability, as traffic can still be routed if one zone becomes unavailable.

3.4.3 Security Groups and listeners

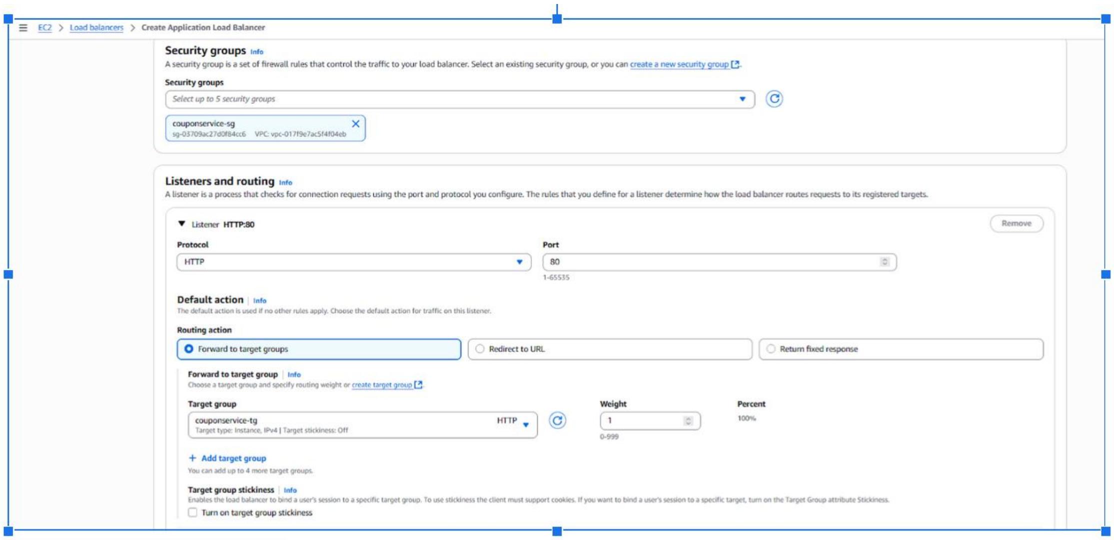

Figure 3.2: Security groups and listener configuration for HTTP traffic routing.

The couponservice-sg security group was attached to the **Application Load Balancer (ALB)** to permit HTTP traffic (port 80) from external users.
A listener was configured to forward all incoming traffic on port 80 to the target group (couponservice-tg), ensuring that user requests are routed correctly to the backend microservice instances

3.4.4 Target Group Configuration

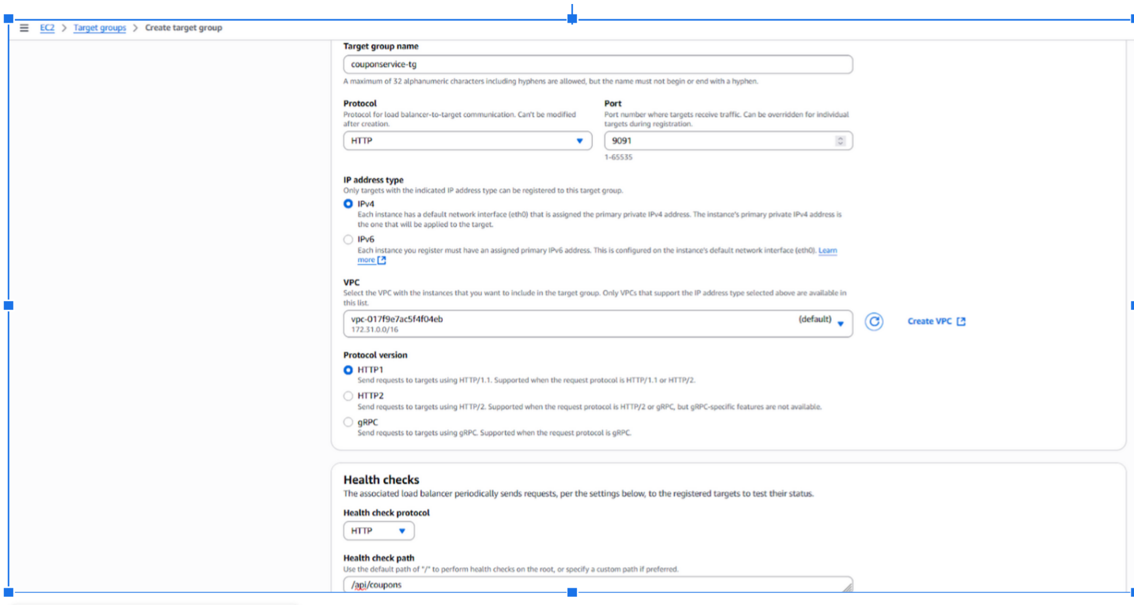

Figure 3.3: Target group configuration showing protocol version, VPC, and health check path

The target group uses HTTP (port 9091) with IPv4 under the same **VPC** (vpc-017f9e7ac5f4f04eb).
A custom health check path /api/coupons was specified, so the **Application Load Balancer (ALB)** will periodically send health requests to this endpoint to determine if instances are functioning correctly.

3.4.5 Register Targets Step

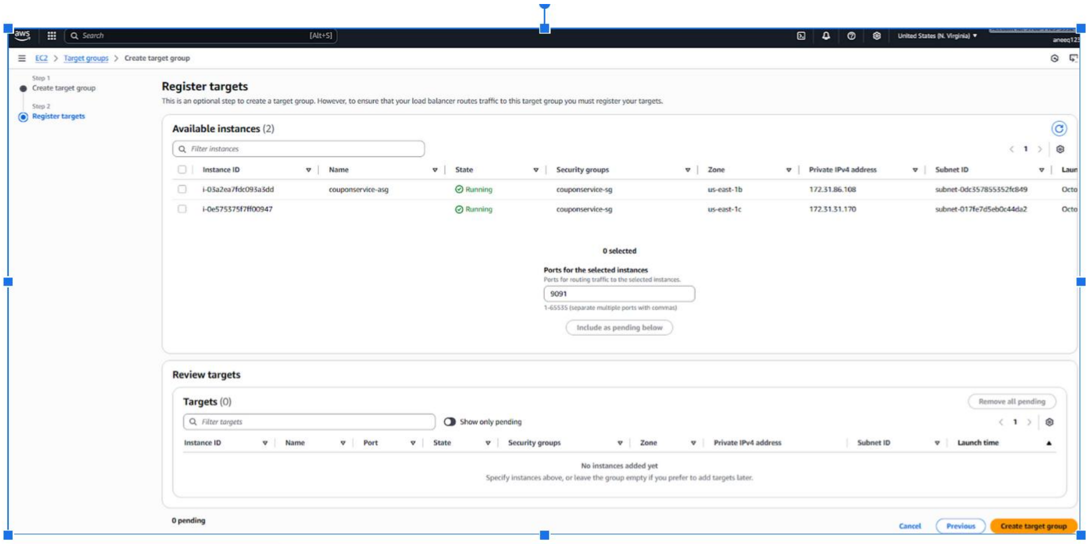

Figure 3.4: Register targets step showing EC2 instances managed by the Auto Scaling Group.

No instances were manually registered during this step because the **Auto Scaling Group (ASG)** dynamically manages instance registration and deregistration.
As scaling events occur, new **EC2** instances will automatically be added or removed from the target group.

3.4.6 Target Group Created

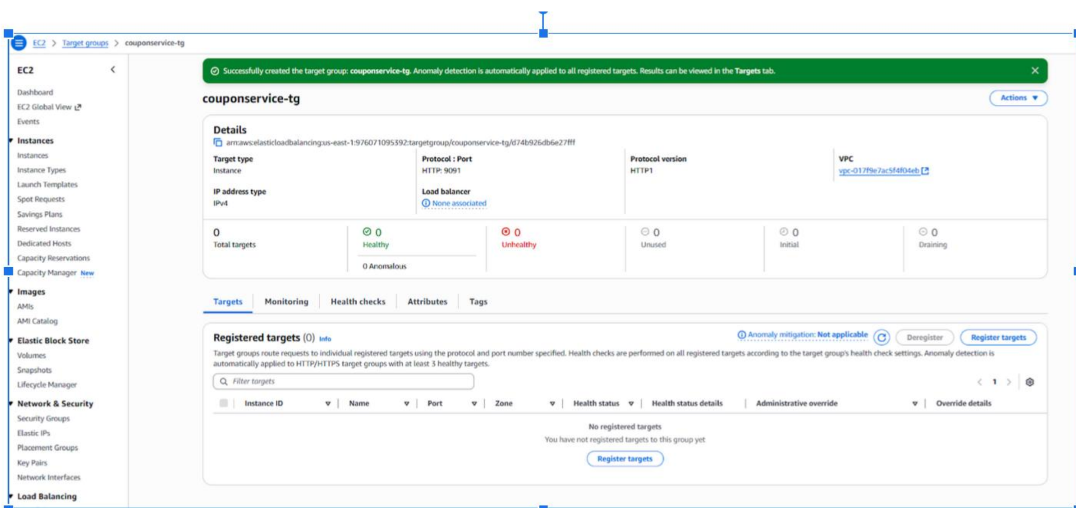

Figure 3.5: Confirmation of successfully created target group couponservice-tg.

The target group was created successfully, but no targets were manually registered. The **Application Load Balancer (ALB)** will automatically detect and register healthy instances from the ASG once scaling begins and health checks pass.

3.4.7 Application Load Balancer Details Screen

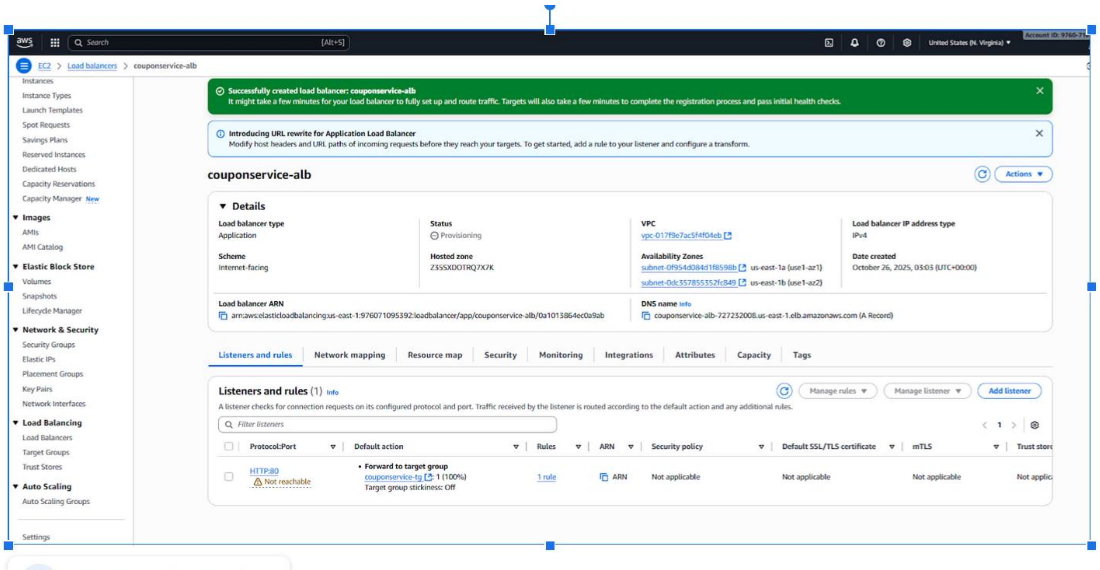

Figure: 3.6: AWS console view of the successfully created Application Load Balancer (couponservice-alb).

The **Application Load Balancer (ALB)** entered the Provisioning state and was assigned a public DNS name.
It was linked with the couponservice-tg target group through a listener on port HTTP:80, configured to forward requests to backend instances once health checks are passed.

3.4.8 Verification Showing EC2 Instances

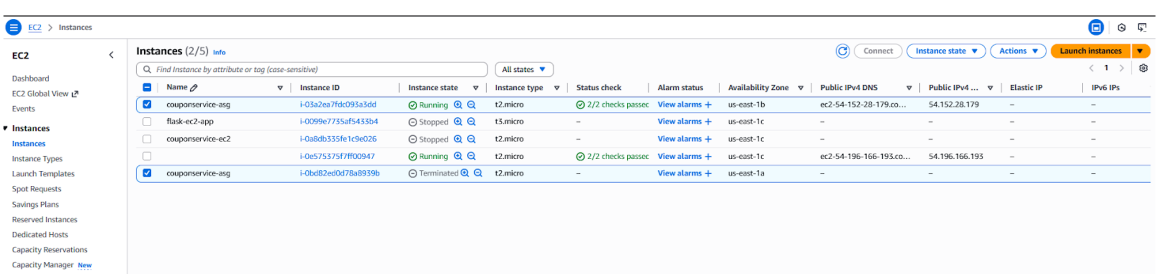

Figure 3.7: *Verification showing EC2 instances managed by ASG and health status summary.*

To validate functionality, the **EC2** Instances tab was reviewed. The **Auto Scaling Group (ASG)** launched multiple instances, and one was terminated automatically as part of lifecycle management.
This confirmed that both **Auto Scaling** and **Load Balancing** mechanisms are functioning correctly, maintaining healthy **EC2** instances and routing requests efficiently.

3.5 Attach Auto Scaling Group to the Application Load Balancer

In this step, the **Auto Scaling Group (ASG)** was connected to **the Application Load Balancer (ALB)** so that incoming traffic could be evenly distributed across all **EC2** instances managed by the ASG. Initially, the **Target Group (couponservice-tg)** was configured with the following parameters:

- **Target type:** Instance
- **Protocol / Port:** HTTP : 9091
- **VPC:** Same as the ASG
- **Health check path:** /api/coupons

However, the initial health checks failed because the endpoint /api/coupons returned a **404 (Not Found)** error.
After connecting to one of the instances via **SSH**, it was verified that the **Spring Boot microservice** was running on port **9091** and responding successfully at /actuator/health with an **HTTP 200** status. The Target Group health check settings were then updated as follows:

Protocol: HTTP

Path: /actuator/health

Port: Traffic port (9091)

Success code: 200

After applying these changes, the instance health status changed from Unhealthy to Healthy, confirming that the correct endpoint was being monitored as shown figure (4.6) below.

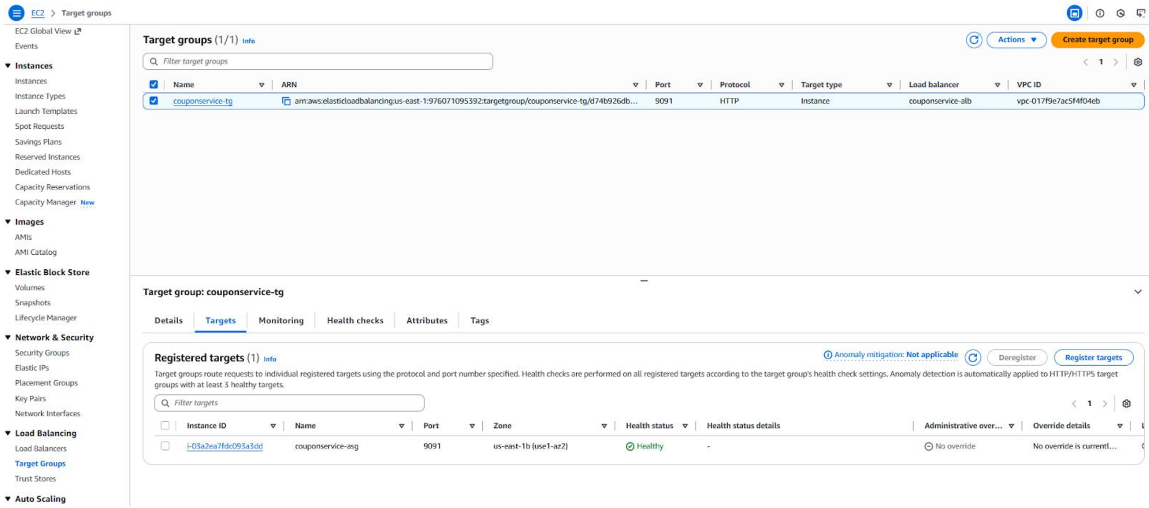

Figure 3.8: Target Group (couponservice-tg) showing registered EC2 instance with Healthy status after updating the health check path to /actuator/health.

4. Test Results

This section demonstrates how the Auto Scaling configuration was tested by applying CPU load on the running **EC2** instances and observing the resulting scaling behaviour through **Amazon CloudWatch** and the **Auto Scaling Group (ASG) activity logs**.

The purpose of this test was to verify that the system automatically launches additional instances when **CPU utilization increases** (**scale-up**) and terminates them when demand decreases (**scale-down**).

```
import threading
import math
import time

def burn():
    # tight loop doing pointless CPU work
    while True:
        math.sqrt(123456789)

def main(worker_threads=4):
    print(f"Starting CPU burn with {worker_threads} threads...")
    for i in range(worker_threads):
        t = threading.Thread(target=burn)
        t.daemon = True  # so threads die if main thread exits
        t.start()
        print(f"Thread {i+1} started")

    # keep the main thread alive so the process doesn't exit
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("Stopping load... bye")

if __name__ == "__main__":
    # You can bump this number higher (8, 12, 16) to push harder
    main(worker_threads=4)
```

Figure 4: Python script (burn_cpu.py) used to simulate CPU load across multiple threads.

A **Python script** (Figure 4, above) named **burn_cpu.py** was used to simulate high CPU usage. This script repeatedly performs mathematical operations in multiple threads, consuming processor resources to generate artificial load. The stress test was executed via **SSH** directly on one of the **EC2** instances.

4.1 CPU Utilization – Before the Test

Figure 4.1: CloudWatch metric (CPUUtilization) before applying CPU stress

Before running the **Python** stress script, **CPU utilization** was low, averaging between 2–3%, as shown in the CloudWatch graph (Figure 5.1 above).
This indicates that the system was idle and operating within normal performance levels under minimal load.

4.2 Running the Python Script

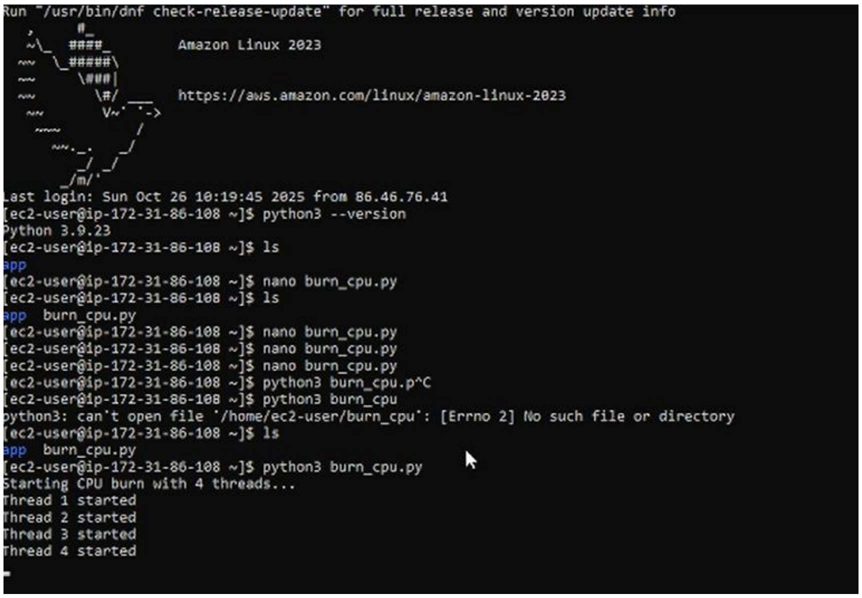

Figure 4.2: Executing burn_cpu.py inside the EC2 instance via SSH.

The command **python3 burn_cpu.py** was executed on the **EC2 instance**.
The output confirmed that four worker threads were started simultaneously, triggering an increase in CPU activity.
Within a few minutes, the CPU load began rising sharply in the **CloudWatch** dashboard.

4.3 Scaling Up Result - CPU Utilization After Stress Test

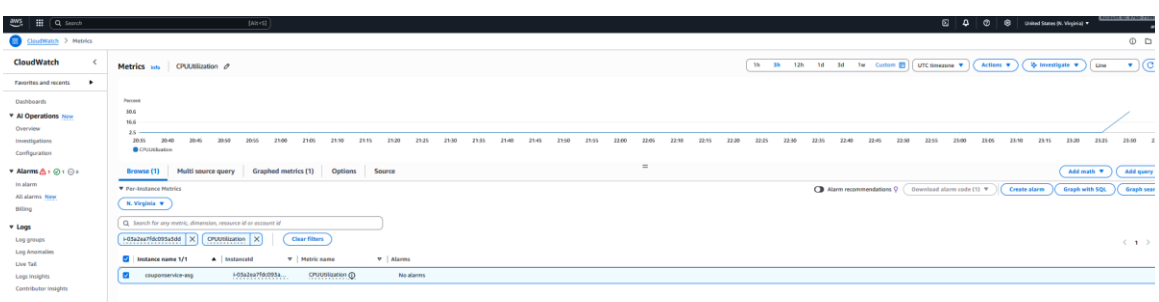

Figure 4.3: *CloudWatch metric showing CPU spikes after initiating the stress test.*

After applying load, the **CPU utilization exceeded 70%**, crossing the threshold defined in the **Target Tracking Policy**.
This triggered the **Auto Scaling Group (couponservice-asg)** to initiate a **scale-up event**, automatically launching new EC2 instances to handle the increased demand.

4.3.1 Auto Scaling Group Activity Log

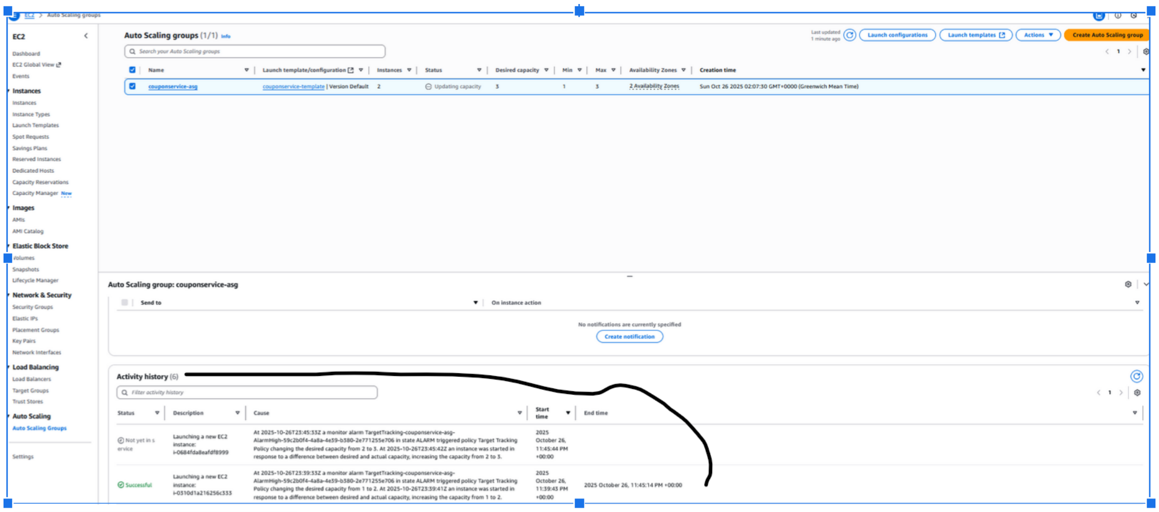

Figure 4.4: ASG activity history showing automatic instance creation triggered by CPU utilization.

The **Auto Scaling Group (ASG)** activity log confirms that multiple scaling events occurred:

- The desired capacity increased from 1 to 2, and later 2 to 3 instances.
- Each event was triggered by the monitoring alarm (TargetTracking-couponservice-asg-AlarmHigh).
- New **EC2** instances were launched in **us-east-1a** and **us-east-1b Availability Zones**, improving redundancy and load distribution.

4.3.2 Verifying New Instances

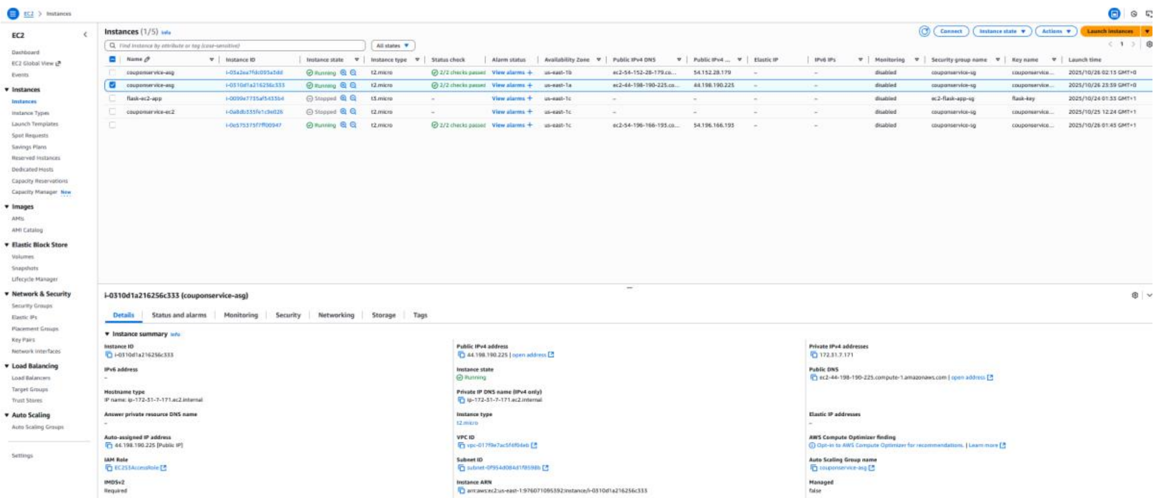

Figure 4.5: EC2 Instances dashboard showing new running instances launched by the Auto Scaling Group.

After scaling up, new instances appeared in the **EC2** console, each automatically launched using the couponservice-template. All instances passed the health checks and became available to serve traffic through the **Application Load Balancer(ALB)**.

4.4 Scale-Down Event

After verifying the scale-up process, the next test examined how the **Auto Scaling Group (ASG)** responded when CPU utilization dropped back to normal levels.
This was done by manually stopping the **Python** stress script (**burn_cpu.py**), which reduced the CPU workload on the active **EC2** instances.

4.5 Scaling down Result - CPU Utilization Decreases After Stopping Stress

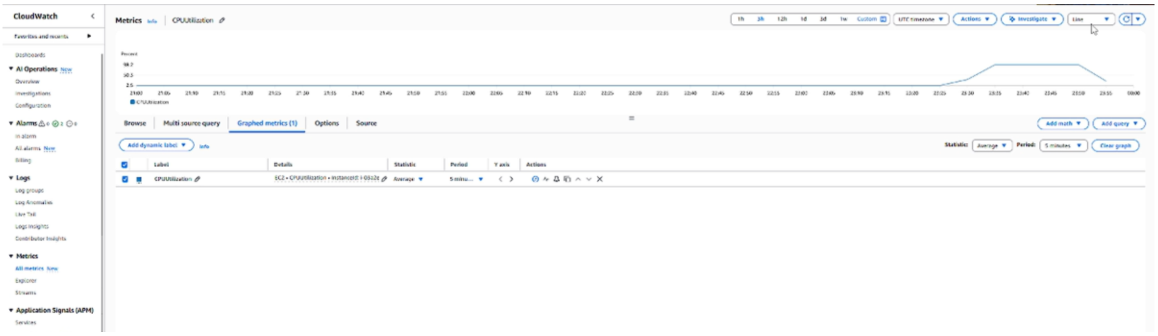

Figure 4.6: CloudWatch metric showing CPU utilization dropping after the Python stress script was stopped.

Once the load generator was stopped, the CPU usage gradually decreased below the 30% threshold, as shown in the **CloudWatch** graph.
This decrease in utilization signaled to the **Auto Scaling Group (ASG)** that the system no longer required multiple instances.

4.5.1 Auto Scaling Group Activity Log

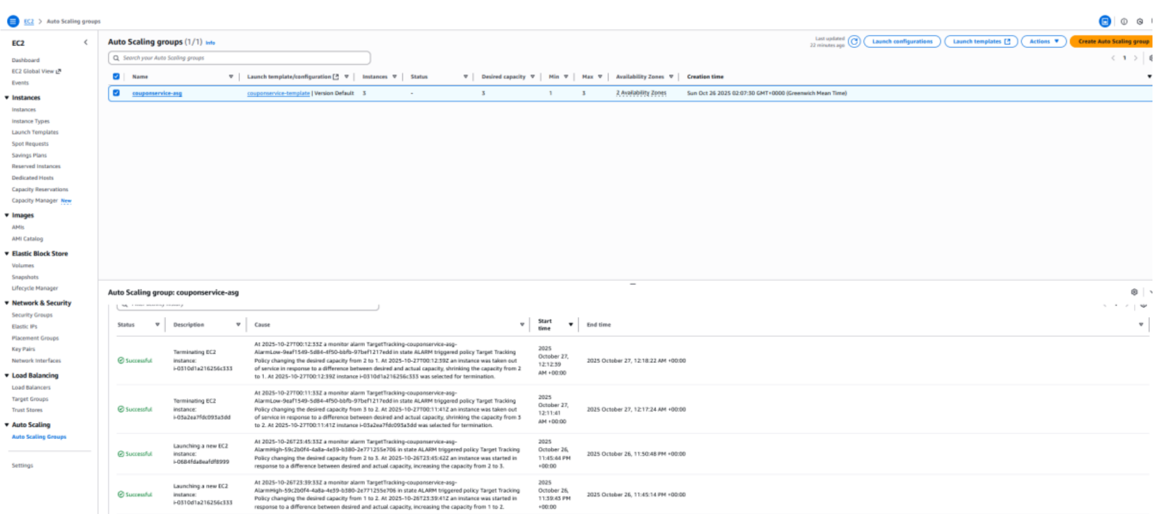

Figure 4.7: Auto Scaling Group activity history confirming automatic instance termination after load reduction.

The **Auto Scaling Group** (**ASG**) activity log recorded multiple **termination events**, showing that:

- The system automatically **reduced capacity** from three instances back down to one.
- Each termination was triggered by the **TargetTracking policy (AlarmLow)**.
- The scaling action occurred approximately **4–5 minutes** after the **CPU utilization** dropped below the threshold, reflecting the normal cooldown delay configured in **AWS**.

This confirmed that the scaling policies were functioning correctly in both directions, scaling up during high CPU load and scaling down once utilization normalized.

4.5.2 Verifying Terminated Instances

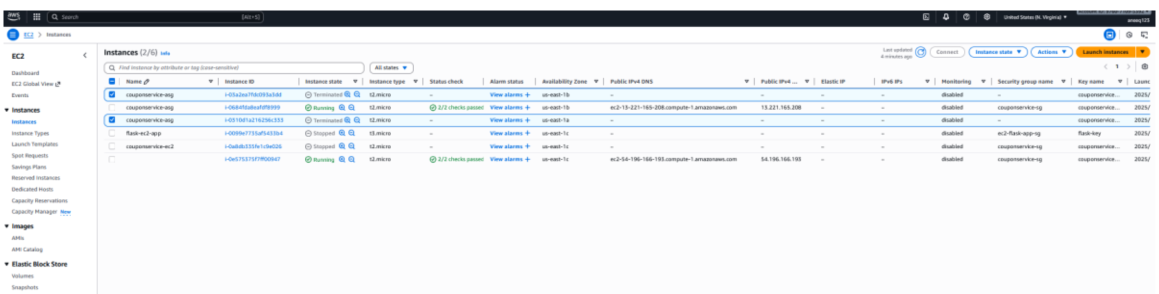

Figure 4.8: EC2 Instances dashboard showing terminated instances after the scale-down event.

In the **EC2** console, two previously active instances were marked as "**terminated**," leaving a single healthy instance running under the **Auto Scaling Group** (**ASG**). This shows that **AWS** efficiently managed resources by automatically shutting down extra **EC2** instances once the CPU load and demand decreased.

5. Evaluation

The project successfully demonstrated the deployment and automation of a **stateless Spring Boot microservice (CouponService)** on AWS using **EC2**, **Auto Scaling Groups (ASG)**, and an **Application Load Balancer (ALB)**.
The main objective is to show how AWS automatically scales application instances based on CPU load, was fully achieved.

5.1 What Worked Well

- **End-to-End Automation**: The Auto Scaling configuration worked as expected. When **CPU utilization** exceeded the 70% threshold, the **Auto Scaling Groups (ASG)** automatically launched new EC2 instances from the **Launch Template**, and when CPU usage dropped below 30%, the extra instances were terminated.

- **Self Configuring Instances**: Each new instance successfully booted using the embedded **User Data script**, which installed **Java 17**, downloaded the **JAR file** from **S3**, wrote the **application.yml** configuration, and started the service automatically.

- **Load Balancing and Health Checks**: The **Application Load Balancer(ALB)** distributed incoming requests evenly across multiple instances. After updating the health check path from /api/coupons to /actuator/health, all targets became healthy and stable.

- **Cloud Integration:** The microservice connected seamlessly to the centralized **RDS MySQL** database, proving that the application layer was stateless and capable of horizontal scaling.

5.2 Challenges and Limitations

- **Scaling Delays**: One noticeable issue was the delay during scale-down events. Based on the Auto Scaling Activity History (See figure 5, below), it took approximately 20–30 minutes after the CPU load dropped for **AWS** to terminate the extra instances. The system launched additional instances between 23:39 pm and 23:50 pm, but it did not start terminating them until around 12:11 am–12:18 am. This delay may have been caused by the default cooldown period or the **CloudWatch** metric evaluation time window.

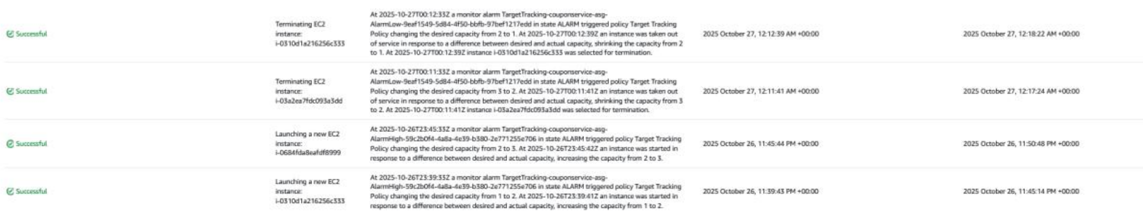

Figure 5: Auto Scaling Activity History showing successful scale-up and delayed scale-down events for the

- **CloudWatch Alarms**: Although CPU-based alarms were configured (See Figure below). one for **high CPU usage (>70%)** to trigger scale-up, and another for **low CPU usage (<49%)** to trigger scale-down but no visible notifications or alerts were triggered during scaling events. The scaling actions still occurred, but the missing alerts made it harder to track system behaviour in real time.

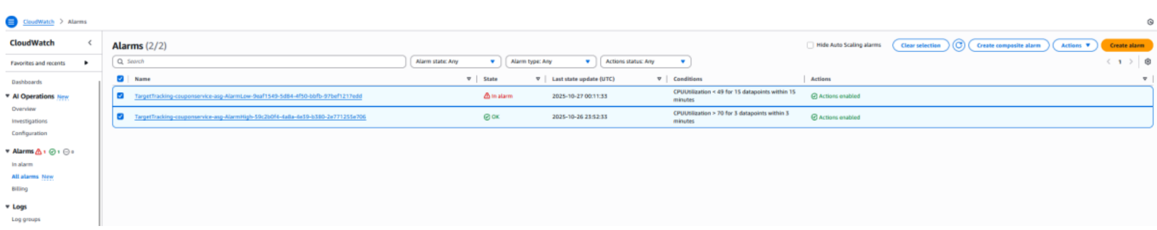

Figure 5.1: CloudWatch (CPU-Based Alarms Configured)

- **Initial Setup Difficulties**: During the early setup phase, the CouponService microservice failed to connect to the **RDS** instance. The issue was traced to the **application.yml** configuration file, it still contained a local database reference (localhost) instead of the **RDS** endpoint URL. Once the correct **RDS** endpoint, username, and password were inserted, the microservice connected successfully.

5.3 Sustainability and Cost Efficiency

The project demonstrated how **AWS** Auto Scaling contributes to sustainable cloud computing and cost optimization. By dynamically adjusting the number of running **EC2** instances according to real-time CPU usage, the system:

- Minimises energy consumption by avoiding idle compute resources when demand is low. **[7]**

- Reduces operational costs because you only pay for active instances and **AWS** automatically terminates excess capacity. **[8]**

- Improves efficiency by maintaining optimal performance under varying workloads through automated scaling of capacity. **[9]**

6. Conclusion

This project showed how to deploy and manage a small stateless microservice on **AWS** using an **Auto Scaling Group (ASG**), an **Application Load Balancer (ALB**), and **CloudWatch** monitoring. During testing, the setup automatically created new **EC2** instances when CPU usage increased and removed them when the load dropped, proving that **AWS** scaling works effectively.

The **ASG** handled changes in traffic without any manual steps, maintaining good performance and uptime. There were some small issues, such as slow scale-down times (around 20–30 minutes) and **CloudWatch** alarms not sending visible notifications. These delays are likely due to **AWS** cooldown settings or **CloudWatch's** evaluation periods, and the missing alerts could be fixed by setting up proper **SNS** or email notifications.

Overall, the project demonstrated the main benefits of cloud computing, scalability**, automation, reliability, and cost savings**. **AWS** automatically adjusted the number of running instances based on demand, saving costs by avoiding idle resources. This also supports **sustainable cloud practices**, as the system only uses the resources it needs.

7. References

List all external sources:

1. Amazon Web Services (2025) Amazon EC2 documentation. Available at: https://docs.aws.amazon.com/autoscaling/ (Accessed: 28 October 2025). = EC2

2. Amazon Web Services (2025) What is Auto Scaling? Available at: https://docs.aws.amazon.com/ec2/ (Accessed: 28 October 2025). = Auto Scaling

3. Amazon Web Services (2025) Amazon CloudWatch User Guide. Available at: https://docs.aws.amazon.com/cloudwatch/ (Accessed: 28 October 2025) = Auto Scaling

4. Amazon Web Services (2025) Launch template – EC2 Auto Scaling User Guide. Available at: https://docs.aws.amazon.com/autoscaling/ec2/userguide/LaunchTemplates.html (Accessed: 28 October 2025). = Launch Templates

5. Amazon Web Services (2024) Scalability on AWS – Architecture Center. Available at: https://aws.amazon.com/architecture/ (Accessed: 28 October 2025). = Vertical Scaling Vs Horizontal Scaling

6. Amazon Web Services (2025) Amazon CloudWatch User Guide. Available at: https://docs.aws.amazon.com/cloudwatch/ (Accessed: 28 October 2025)

7. AWS for Engineers (2024) 5 Principles for Designing Sustainable AWS Architectures. Available at: https://awsforengineers.com/blog/5-principles-for-designing-sustainable-aws-architectures/ (Accessed: 28 October 2025) = Minimises energy consumption

8. Amazon Web Services (2025) Auto Scaling Benefits – EC2 User Guide. Available at: https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-benefits.html (Accessed: 28 October 2025 = Reduces operational costs

9. Amazon Web Services (2025) What is Amazon EC2 Auto Scaling? Available at: https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html (Accessed: 28 October 2025). = Improves efficiency