

Deploying a Microservice on AWS with Autoscaling Policy Project

Contents

1. Project Overview	4
1.1 Objective.....	4
1.2 Tools Used	4
1.3 System Overview	4
1.4 Microservice Description	5
2. Introduction	6
2.1 Amazon Elastic Compute Cloud (EC2).....	6
2.2 Auto Scaling.....	6
2.3 Launch Template	7
2.4 Vertical Scaling vs Horizontal Scaling.....	7
3. Configuration.....	8
3.1 Environment Setup	8
3.2 Launch Template – Configuration Details.....	8
3.2.1 Key Configuration Details.....	9
3.2.2 User Data Script.....	10
3.3 Auto Scaling Group (ASG) – Configuration Details	12
3.3.1 Select the Existing Launch Template	12
3.3.2 Select the VPC and Availability Zones for ASG	13
3.3.3 Check Health Checks	13
3.3.4 Setting up Scaling Policies and Group Size	14
3.3.5 Add Tags to the ASG.....	15
3.3.6 Showing newly created Auto Scaling Group (ASG).....	16
3.4 Application Load Balancer (ALB).....	17
3.4.1 Creating Application Load Balancer.....	17
3.4.2 Network Mapping for ALB	18
3.4.3 Security Groups and listeners	19
3.4.4 Target Group Configuration.....	20
3.4.5 Register Targets Step.....	21
3.4.6 Target Group Created	21
3.4.7 Application Load Balancer Details Screen	22
3.4.8 Verification Showing EC2 Instances.....	22
3.5 Attach Auto Scaling Group to the Application Load Balancer	23

4. Test Results	24
4.1 CPU Utilization – Before the Test.....	25
4.2 Running the Python Script.....	26
4.3 Scaling Up Result - CPU Utilization After Stress Test	26
4.3.1 Auto Scaling Group Activity Log	27
4.3.2 Verifying New Instances	28
4.4 Scale-Down Event.....	28
4.5 Scaling down Result - CPU Utilization Decreases After Stopping Stress	29
4.5.1 Auto Scaling Group Activity Log	29
4.5.2 Verifying Terminated Instances	30
5. Evaluation	31
5.1 What Worked Well.....	31
5.2 Challenges and Limitations	32
5.3 Sustainability and Cost Efficiency.....	33
6. Conclusion.....	33
7. References.....	34

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 compiled 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 access to the **S3 bucket** without embedding credentials.

```

aws-ec2-user@ip-172-31-163-109:~$
# newer release of "amazon linux" is available.
# Version 2023.9.20230826
# "curl https://aws.amazon.com/linux/ec2-user" for full release and version update info

Amazon Linux 2023

https://aws.amazon.com/linux/amazon-linux-2023

Last login: Fri Oct 24 15:04:20 2025 from 86.46.76.41
ec2-user@ip-172-31-163-109:~$ cd /home/ec2-user/app
ec2-user@ip-172-31-163-109:~/app$ cd /home/ec2-user/app
ec2-user@ip-172-31-163-109:~/app$ sudo java -jar coupon-service-0.0.1-SNAPSHOT.jar --spring.config.location=/home/ec2-user/app/application.yml

Spring
(2023.4)

ec2-user@ip-172-31-163-109:~/app$ ./coupon-service-0.0.1-SNAPSHOT.jar
Started by root in /home/ec2-user/app

main | C:\msc\coupon.CouponServiceApplication | Starting CouponServiceApplication v0.0.1-SNAPSHOTJ using Java 17-0.16 with PID 35822 (/home/ec2-user/app/coupon-service-0.0.1-SNAPSHOT.jar)
main | C:\msc\coupon.CouponServiceApplication | No active profile set, falling back to default profile: 'default'
main | s.d.r.c.RepositoryConfigurationDelegate | TomcatSwapping Spring Data JPA repositories in syncWithDefault()
main | s.d.r.c.RepositoryConfigurationDelegate | Finished Spring Data repository scanning in 108 ms. Found 1 JPA repository interfaces.
main | o.h.a.w.e.EmbeddedDatabaseTool | Tomcat initialized with port(s): 8081 (http)
main | o.apache.catalina.core.StandardEngine | Starting service [tomcat]
main | o.apache.catalina.core.StandardEngine | Starting Servlet engine: [Apache Tomcat/9.0.112]
main | o.a.s.c.g.TomcatLifecycleListener | Initializing Spring embedded WebApplicationContext
main | w.a.c.DefaultWebApplicationContext | Root WebApplicationContext: Initialization completed in 4811 ms
main | org.hibernate.Version | HHH000264: Processing PersistenceUnitInfo for 1 unit[s]
main | org.hibernate.cfg.bytecode.BytecodeProvider | HHH000041: Hibernate ORM core version 6.2.0.Final
main | org.hibernate.cfg.bytecode.BytecodeProvider | HHH000021: Using bytecode reflection optimizer
main | o.h.b.i.BYTECODEProviderInitiator | HHH000021: Bytecode provider name is 'bytebuddy'
main | o.h.b.i.BYTECODEProviderInitiator | No configuration found, ignoring JPA class transformer
main | com.zaxxer.hikari.HikariDataSource | HikariPool-1 - Starting...
main | com.zaxxer.hikari.pool.HikariPool | HikariPool-1 - Added connection com.mysql.cj.jdbc.ConnectionImpl@4ebaa8?
main | com.zaxxer.hikari.HikariDataSource | HikariPool-1 - Start completed.
main | org.hibernate.orm.deprecation | HHH000002: MySQLDialect has been deprecated; use org.hibernate.dialect.MySQLDialect instead
main | org.hibernate.orm.deprecation | HHH000003: MySQLDialect provider name is deprecated
main | o.h.b.i.BytecodeProviderInitiator | HHH000004: Using Instrumentation implementation [org.hibernate.engine.transaction.jta.platform.internal.NoJTAPlatform]
main | o.h.a.s.g.p.i.TrainPlatformInitiator | Initializing JPA IntrospectorManager for persistence unit 'default'
main | j.LocalContainerEntityManagerFactoryBean | spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure it instead.
main | o.h.a.s.g.p.i.TrainPlatformInitiator | Expected 1 endpoint(s) beneath base path "/{actuator}"
main | o.h.a.w.e.embedded.tomcat.TomcatInitServer | Tomcat started on port(s): 8081 (http) with context path "/"
main | C:\msc\coupon.CouponServiceApplication | Started CouponServiceApplication in 13.2 seconds (process running for 13.28s)
```

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

The screenshot shows the 'Create launch template' page in the AWS Console. The page is divided into two main sections: 'Launch template name and description' and 'Launch template contents'. In the first section, the 'Launch template name' is 'couponservice-template', the 'Template version description' is 'Template for CouponService microservice (Java + RDS)', and the 'Auto Scaling guidance' is selected. In the second section, the 'Application and OS Images (Amazon Machine Image)' is selected as 'Amazon Linux 2023 AMI 2023.9.2_x86_64'. The 'Summary' panel on the right shows the configuration details, including the software image, virtual server type, firewall security group, and storage instance.

Create launch template

Creating a launch template allows you to create a saved instance configuration that can be reused, shared and launched at a later time. Templates can have multiple versions.

Launch template name and description

Launch template name – required

couponservice-template

Must be unique to this account. Max 128 chars. No spaces or special characters like %, ", ' or \.

Template version description

Template for CouponService microservice (Java + RDS)

Max 255 chars

Auto Scaling guidance [info](#)

Select if you intend to use this template with EC2 Auto Scaling

☐ Provide guidance to help me set up a template that I can use with EC2 Auto Scaling

► **Template type**

► Source template

Launch template contents

Specify the details of your launch template below. Leaving a field blank will result in the field not being included in the launch template.

▼ **Application and OS Images (Amazon Machine Image)** [info](#)

An AMI contains the operating system, application server, and applications for your instance. If you don't use a suitable AMI below, use the search field or choose [Browse more AMIs](#).

Search our full catalog including 1000s of applications and OS images

Recents **Quick Start**

☐ Don't include in launch template ☐ Recently launched ☒ Currently in use

Amazon Machine Image (AMI)

ami-0341d957f10110123

Architecture: x86_64

Virtualization: hvm

Enclave: enabled

Boot mode: uefi

Boot mode: uefi preferred

Description

Amazon Linux 2023 AMI 2023.9.20231014.0 x86_64 HVM kernel:6.1

Architecture

x86_64

AMI ID

ami-0341d957f10110123

Summary

Software image (AMI)

Amazon Linux 2023 AMI 2023.9.2_x86_64

Virtual server type (Instance type)

t3.micro

Firewall (security group)

couponservice-sg

Storage (Instance)

1 volume(s) - 8 GiB

Free tier: In your first year of opening an AWS account, you get 750 hours per month of t3.micro instance usage (or t3.micro where t3.micro isn't available) when used with free tier AMIs, 750 hours per month of public IPv4 address usage, 30 GiB of EBS storage, 2 million I/Os, 1 GiB of snapshots, and 100 GiB of bandwidth to the internet. Data transfer charges are not included as part of the free tier allowance.

[Cancel](#) [Create launch template](#)

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**.

The screenshot shows the 'Create launch template' page in the AWS Console, specifically the 'Instance type', 'Key pair (login)', 'Network settings', and 'Storage (volumes)' sections. The 'Instance type' is 't3.micro'. The 'Key pair (login)' section shows a key pair named 'couponservice-key'. The 'Network settings' section shows a subnet named 'default' and a security group named 'couponservice-sg'. The 'Storage (volumes)' section shows a volume named 'Volume 1 (AMI Root)' with a size of 8 GiB and a type of 'EBS, General purpose SSD (gp3)'. The 'Summary' panel on the right shows the configuration details, including the software image, virtual server type, firewall security group, and storage instance.

Create launch template

Instance type [info](#) [Advanced](#)

Instance type

t3.micro

12 vCPUs • 8 GB • 1 EBS volume • Current generation instance • On-demand instance base pricing: 0.015625 USD per hour • On-demand instance base pricing: 0.015625 USD per hour • On-demand instance base pricing: 0.015625 USD per hour • On-demand instance base pricing: 0.015625 USD per hour

Key pair (login) [info](#)

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name

couponservice-key

☐ Create new key pair

Network settings [info](#)

Subnet

default

Don't include in launch template

When you create a subnet, a network interface is automatically added to your VPC.

Availability Zone

us-east-1a

Don't include in launch template

Firewall (security group)

couponservice-sg

Security groups

couponservice-sg

Advanced network configuration

Storage (volumes) [info](#) [View details](#)

EBS Volumes

Volume 1 (AMI Root) - 8 GiB, EBS, General purpose SSD (gp3), 3000 IOPS

AMI resources are not included in the template unless modified

Summary

Software image (AMI)

Amazon Linux 2023 AMI 2023.9.2_x86_64

Virtual server type (Instance type)

t3.micro

Firewall (security group)

couponservice-sg

Storage (Instance)

1 volume(s) - 8 GiB

Free tier: In your first year of opening an AWS account, you get 750 hours per month of t3.micro instance usage (or t3.micro where t3.micro isn't available) when used with free tier AMIs, 750 hours per month of public IPv4 address usage, 30 GiB of EBS storage, 2 million I/Os, 1 GiB of snapshots, and 100 GiB of bandwidth to the internet. Data transfer charges are not included as part of the free tier allowance.

[Cancel](#) [Create launch template](#)

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

- **Key Pair:** coupon-service-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:** coupon-service-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://coupon-service-artifacts-aneeq/coupon-service-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/
  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 coupon-service-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`.

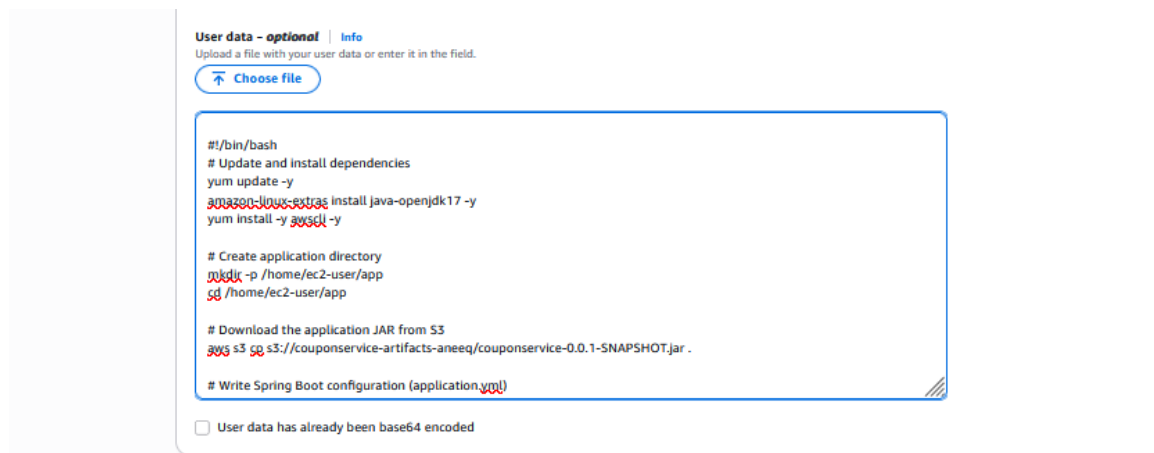


Figure 2.3: User Data section of the Launch Template showing the startup script

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

The screenshot shows the 'Choose launch template' step in the AWS console. On the left, a vertical list of steps indicates the current position: Step 1 (Choose launch template), Step 2 (Choose instance launch options), Step 3 (optional, Integrate with other services), Step 4 (optional, Configure group size and scaling), Step 5 (optional, Add notifications), Step 6 (optional, Add tags), Step 7, and Review.

The main content area is titled 'Choose launch template' with a subtitle 'Specify a launch template that contains settings common to all EC2 instances that are launched by this Auto Scaling group.' It contains the following fields and information:

- Name:** A text input field with the value 'couponservice-asg'. A note below states: 'Must be unique to this account in the current Region and no more than 255 characters.'
- Launch template:** A dropdown menu showing 'couponservice-template'. A note above it states: 'For accounts created after May 31, 2023, the EC2 console only supports creating Auto Scaling groups with launch templates. Creating Auto Scaling groups with launch configurations is not recommended but still available via the CLI and API until December 31, 2023.'
- Version:** A dropdown menu showing 'Default (3)'. A note below it states: 'Create a launch template version.'
- Description:** A text input field with the value '-'. A link 'Create a launch template' is next to it.
- AMI ID:** A text input field with the value 'ami-0541d957f5f11023'.
- Key pair name:** A text input field with the value 'couponservice-key'.
- Launch template:** A link to 'couponservice-template' with ID 'lt-044fc37f8d0e385'.
- Security groups:** A text input field with the value '-'. A link 'Create a launch template version' is next to it.
- Security group IDs:** A link to 'sg-01709ac27d0f84cc6'.
- Instance type:** A text input field with the value 't2.micro'.
- Request Spot Instances:** A text input field with the value 'No'.
- Additional details:** A section containing 'Storage (volumes)' with the value '-' and 'Date created' with the value 'Sun Oct 26 2025 01:41:22 GMT+0100 (GMT+01:00)'.

At the bottom right, there are 'Cancel' and 'Next' buttons.

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

EC2 > Auto Scaling groups > Create Auto Scaling group

Step 1: Choose launch template

Step 2: Choose instance launch options

Step 3: optional: Integrate with other services

Step 4: optional: Configure group size and scaling

Step 5: optional: Add notifications

Step 6: optional: Add tags

Step 7: Review

Choose instance launch options

Choose the VPC network environment that your instances are launched into, and customize the instance types and purchase options.

Instance type requirements

You can keep the same instance attributes or instance type from your launch template, or you can choose to override the launch template by specifying different instance attributes or manually adding instance types.

Launch template	Version	Description
m5.xlarge	Default	

Instance type: **t2.micro**

Network

For most applications, you can use multiple Availability Zones and let EC2 Auto Scaling balance your instances across the zones. The default VPC and default subnets are suitable for getting started quickly.

VPC
Choose the VPC that defines the virtual network for your Auto Scaling group.
us-east-1-vpc
172.31.0.0/16 Default
[Create a VPC](#)

Availability Zones and subnets
Define which Availability Zones and subnets your Auto Scaling group can use in the chosen VPC.
Select Availability Zones and subnets

us-east-1a (us-east-1a) | subnet-0f5a40b6d1f6980b
172.31.0.0/20 Default

us-east-1b (us-east-1b) | subnet-0c357855326c0d9
172.31.0.0/20 Default

[Create a subnet](#)

Availability Zone distribution - new
Auto Scaling automatically balances instances across Availability Zones. If launch failures occur in a zone, select a strategy.

☒ **Balanced best effort**
If launches fail in one Availability Zone, Auto Scaling will attempt to launch in another healthy Availability Zone.

☐ **Balanced only**
If launches fail in one Availability Zone, Auto Scaling will continue to attempt to launch in the unhealthy Availability Zone to promote balanced distribution.

[Cancel](#) [Skip to review](#) [Previous](#) [Next](#)

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

EC2 > Auto Scaling groups > Create Auto Scaling group

Step 1: Choose launch template

Step 2: Choose instance launch options

Step 3: optional: Integrate with other services

Step 4: optional: Configure group size and scaling

Step 5: optional: Add notifications

Step 6: optional: Add tags

Step 7: Review

Integrate with other services - optional

Use a load balancer to distribute incoming traffic across multiple servers. Enable service-to-service communications with VPC Lattice. Shift resources away from impaired Availability Zones with zonal shift. You can also customize health check replacements and monitoring.

Load balancing

Use the options below to attach your Auto Scaling group to an existing load balancer, or to a new load balancer that you define.

Select Load balancing options

☒ **No load balancer**
Traffic to your Auto Scaling group will not be proxied by a load balancer.

☐ **Attach to an existing load balancer**
Choose from your existing load balancers.

☐ **Attach to a new load balancer**
Quickly create a new load balancer to attach to your Auto Scaling group.

VPC Lattice integration options

To improve networking capabilities and scalability, integrate your Auto Scaling group with VPC Lattice. VPC Lattice facilitates communications between AWS services and helps you connect and manage your applications across separate services in AWS.

Select VPC Lattice service to attach

☒ **No VPC Lattice service**
VPC Lattice will not manage your Auto Scaling group's network access and connectivity with other services.

☐ **Attach to VPC Lattice service**
Routing requests associated with specified VPC Lattice target groups will be routed to your Auto Scaling group.

[Create new VPC Lattice options](#)

Application Recovery Controller (ARC) zonal shift - new

During an Availability Zone impairment, target instances shift towards other healthy Availability Zones.

☐ **Enable zonal shift**
New instance launches will be integrated towards healthy Availability Zones until the zonal shift is completed.

Health checks

Health checks monitor availability by replacing unhealthy instances. When you use multiple health checks, all are evaluated, and if at least one fails, instance replacement occurs.

EC2 health checks
[Always enabled](#)

Additional health check types - optional

☒ **Turn on Elastic Load Balancing health checks**
Elastic Load Balancing monitors whether instances are available to handle requests. When it reports an unhealthy instance, EC2 Auto Scaling can replace it on its next periodic check.

☒ **Turn on VPC Lattice health checks**
VPC Lattice can monitor whether instances are available to handle requests. If it completes a target as failed a health check, EC2 Auto Scaling replaces it after its next periodic check.

☐ **Turn on Amazon RDS health checks**
RDS monitors whether an instance is online or attached volume exists. When it reports an unhealthy volume, EC2 Auto Scaling can replace the instance on its next periodic health check.

Health check grace period
This time period delays the first health check until your instances finish initializing. It doesn't prevent an instance from terminating after placed into a new running state.

seconds

[Cancel](#) [Skip to review](#) [Previous](#) [Next](#)

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

The screenshot shows the 'Configure group size and scaling' step in the AWS Management Console. The left sidebar lists steps 1 through 7, with step 4, 'Configure group size and scaling', highlighted. The main content area is titled 'Configure group size and scaling - optional' and includes the following sections:

- Group size:** A section for setting the initial size of the Auto Scaling group. It includes a 'Desired capacity type' dropdown (set to 'On-demand (EC2)'), a 'Desired capacity' input field (set to 1), and a 'Min. group size' input field (set to 1).
- Scaling:** A section for configuring scaling policies. It includes a 'Scaling limits' section with 'Min. desired capacity' (1) and 'Max. desired capacity' (3). Below this is the 'Automatic scaling - optional' section, where 'Target tracking scaling policy' is selected. The 'Scaling policy name' is 'Target Tracking Policy', and the 'Metric type' is 'Average CPU utilization'. The 'Target value' is set to 70. The 'Instance warmup' is set to 300 seconds.
- Instance maintenance policy:** A section for controlling the Auto Scaling group's availability during instance replacement events. It includes a 'Choose a replacement behavior' section with four options: 'Replace instance' (selected), 'Launch before terminating', 'Terminate and launch', and 'Custom behavior'.
- Additional capacity settings:** A section for configuring capacity reservation preferences. It includes a 'Capacity reservation preference' section with three options: 'Default' (selected), 'None', and 'Capacity Reservations only'.

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

The screenshot shows the 'Add tags' step in the AWS Management Console for creating an Auto Scaling Group. On the left, a vertical list of steps is shown, with 'Add tags' selected. The main area is titled 'Add tags - optional' with a blue 'info' icon. Below the title, a message states: 'Add tags to help you search, filter, and track your Auto Scaling group across AWS. You can also choose to automatically add these tags to instances when they are launched.' A blue information box contains a note: 'You can optionally choose to add tags to instances (and their attached EBS volumes) by specifying tags in your launch template. We recommend caution, however, because the tag values for instances from your launch template will be overridden if there are any duplicate keys specified for the Auto Scaling group.' Below this, a table titled 'Tags (1)' shows one tag with the key 'Name' and the value 'couponservice-asg'. To the right of the value is a checkbox labeled 'Tag new instances' which is checked, and a 'Remove' button. At the bottom left of the table is an 'Add tag' button and a note '49 remaining'. At the bottom right are 'Cancel', 'Previous', and 'Next' buttons.

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)

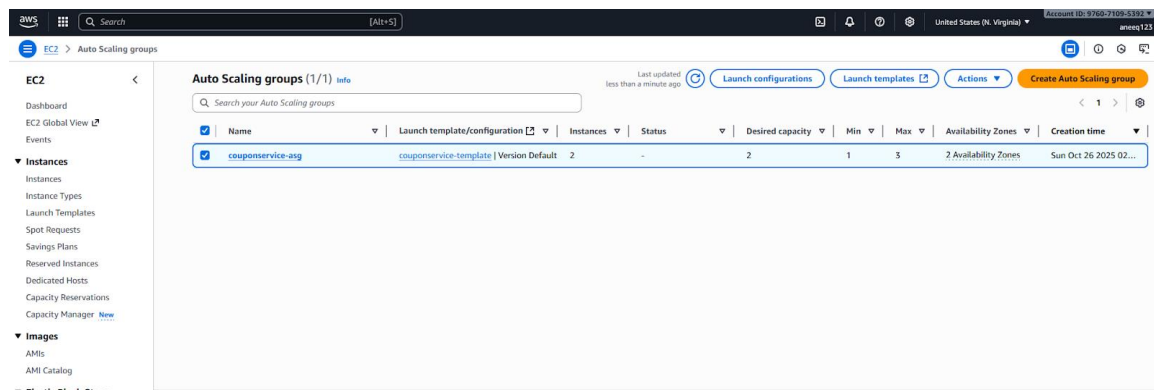


Figure 2.9: AWS Console view showing the newly created Auto Scaling Group (ASG)

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**.

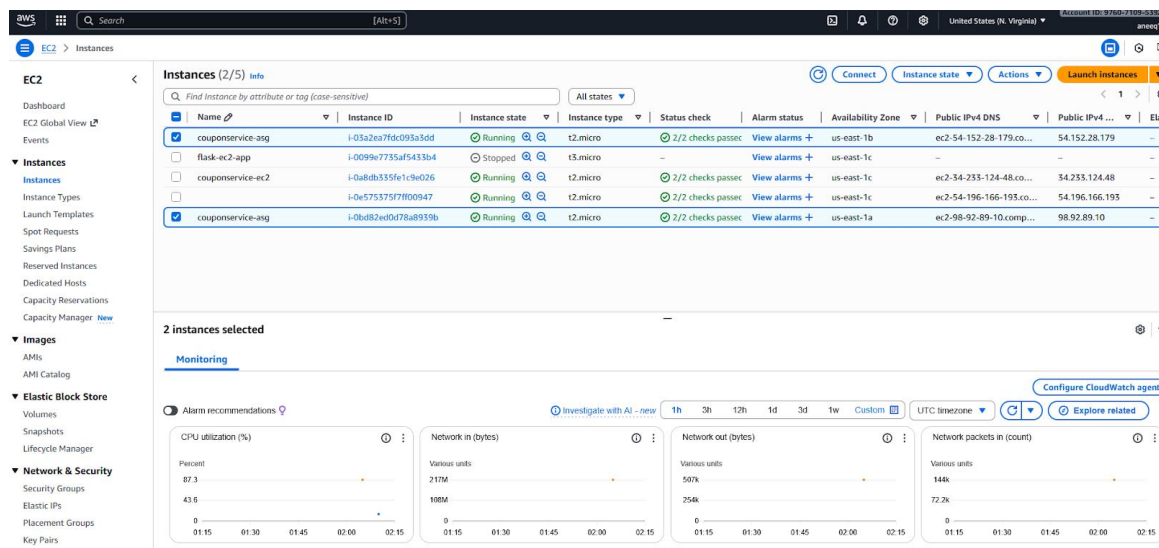


Figure 2.10: EC2 Instances page displaying multiple instances launched automatically by the Auto Scaling Group.

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

The screenshot shows the AWS Management Console interface for creating an Application Load Balancer. The breadcrumb navigation at the top indicates the path: EC2 > Load balancers > Create Application Load Balancer. The main heading is 'Create Application Load Balancer' with an 'info' icon. Below the heading is a brief description of the ALB's function. A section titled 'How Application Load Balancers work' is visible. The 'Basic configuration' section contains the following fields:

- Load balancer name:** A text input field containing 'coupon-service-alb'. A note below the field states: 'Name must be unique within your AWS account and can't be changed after the load balancer is created. A maximum of 32 alphanumeric characters including hyphens are allowed, but the name must not begin or end with a hyphen.'
- Scheme:** A section with an 'info' icon and a note: 'Scheme can't be changed after the load balancer is created.' It contains two radio button options: 'Internet-facing' (selected) and 'Internal'. The 'Internet-facing' option has a list of bullet points: 'Serves internet-facing traffic.', 'Has public IP addresses.', 'DNS name resolves to public IP.', and 'Requires a public subnet.' The 'Internal' option has a list of bullet points: 'Serves internal traffic.', 'Has private IP addresses.', 'DNS name resolves to private IP.', and 'Compatible with the IPv4 and Dualstack IP address types.'
- Load balancer IP address type:** A section with an 'info' icon and a note: 'Select the front-end IP address type to assign to the load balancer. The VPC and subnets mapped to this load balancer must include the selected IP address types. Public IPv4 addresses have an additional cost.' It contains three radio button options: 'IPv4' (selected), 'Dualstack', and 'Dualstack without public IPv4'. The 'Dualstack' option has a note: 'Includes IPv4 and IPv6 addresses.' The 'Dualstack without public IPv4' option has a note: 'Includes a public IPv6 address, and private IPv4 and IPv6 addresses. Compatible with internet-facing load balancers only.'

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

The screenshot shows the 'Create Application Load Balancer' page in the AWS Management Console. The 'Network mapping' section is active, showing the following configuration:

- VPC:** vpc-017f9e7ac5f4f04eb (Default)
- IP pools:** Use IPAM pool for public IPv4 addresses (checked)
- Availability Zones and subnets:**
 - ☒ us-east-1a (us-east-1a)
 - Subnet: subnet-0f954d0641f9590b (IPv4 subnet CIDR: 172.31.0.0/20)
 - ☒ us-east-1b (us-east-1b)
 - Subnet: subnet-08c55785552763d0 (IPv4 subnet CIDR: 172.31.80.0/20)
 - ☐ us-east-1c (us-east-1c)
 - ☐ us-east-1d (us-east-1d)
 - ☐ us-east-1e (us-east-1e)
 - ☐ us-east-1f (us-east-1f)

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

The screenshot displays the 'Create Application Load Balancer' page in the AWS Management Console. The 'Security groups' section is at the top, showing a dropdown menu with 'couponservice-sg' selected. Below this, the 'Listeners and routing' section is visible. It shows a listener for HTTP on port 80. The 'Default action' is set to 'Forward to target groups'. The 'Target group' is 'couponservice-tg' with a weight of 1 and a percent of 100%. The 'Target group stickiness' checkbox is unchecked.

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

The screenshot shows the 'Create target group' page in the AWS Management Console. The breadcrumb navigation at the top indicates the path: **EC2** > **Target groups** > **Create target group**.

The form is titled 'Target group name' and contains the following fields and options:

- Target group name:** A text input field containing 'couponservice-tg'. Below it, a note states: 'A maximum of 32 alphanumeric characters including hyphens are allowed, but the name must not begin or end with a hyphen.'
- Protocol:** A dropdown menu set to 'HTTP'. A note below reads: 'Protocol for load balancer-to-target communication. Can't be modified after creation.'
- Port:** A text input field containing '9091'. A note below reads: 'Port number where targets receive traffic. Can be overridden for individual targets during registration.'
- IP address type:** Two radio buttons are present: 'IPv4' (selected) and 'IPv6'. A note below reads: 'Only targets with the indicated IP address type can be registered to this target group. Each instance has a default network interface (eth0) that is assigned the primary private IPv4 address. The instance's primary private IPv4 address is the one that will be applied to the target.'
- VPC:** A dropdown menu set to 'vpc-017f9e7ac5f4f04eb'. A note below reads: 'Select the VPC with the instances that you want to include in the target group. Only VPCs that support the IP address type selected above are available in this list.'
- Protocol version:** Three radio buttons are present: 'HTTP1' (selected), 'HTTP2', and 'gRPC'. A note below reads: 'Send requests to targets using HTTP/1.1. Supported when the request protocol is HTTP/1.1 or HTTP/2.'
- Health checks:** A section titled 'Health checks' with a note: 'The associated load balancer periodically sends requests, per the settings below, to the registered targets to test their status.'
- Health check protocol:** A dropdown menu set to 'HTTP'.
- Health check path:** A text input field containing '/api/coupons'. A note below reads: 'Use the default path of "/" to perform health checks on the root, or specify a custom path if preferred.'

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

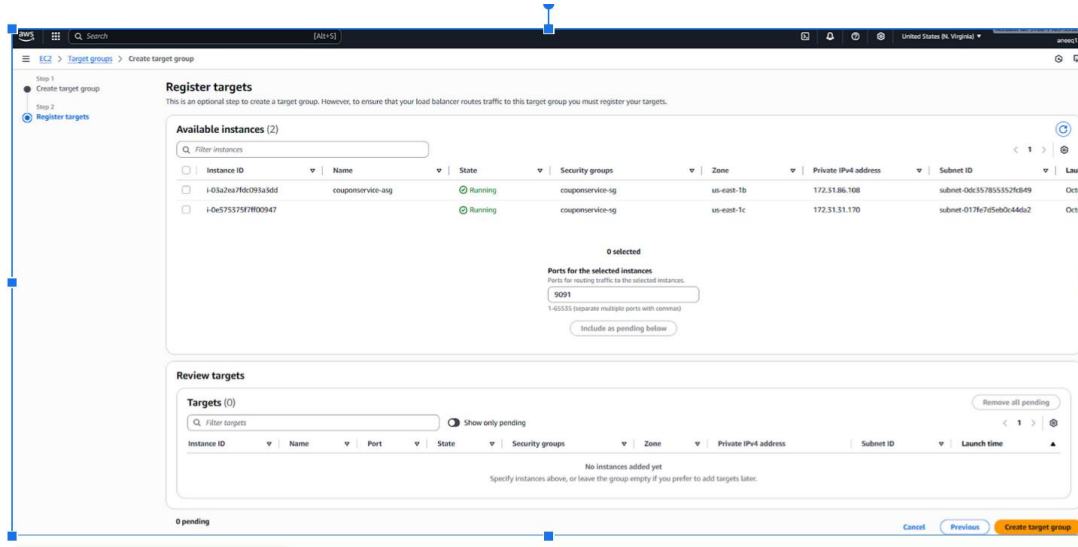


Figure 3.4: Register targets step showing EC2 instances managed by the Auto Scaling Group.

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

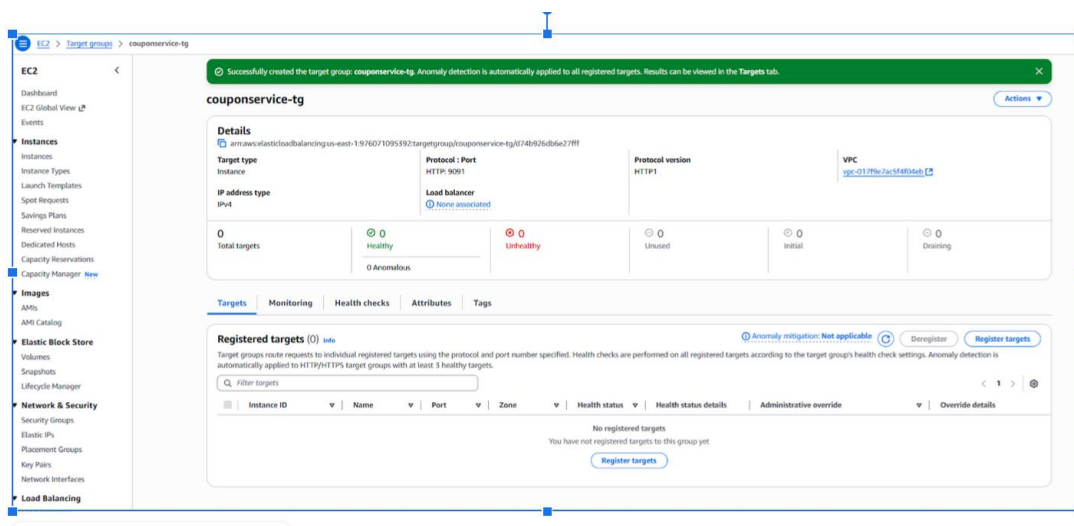


Figure 3.5: Confirmation of successfully created target group couponservice-tg.

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

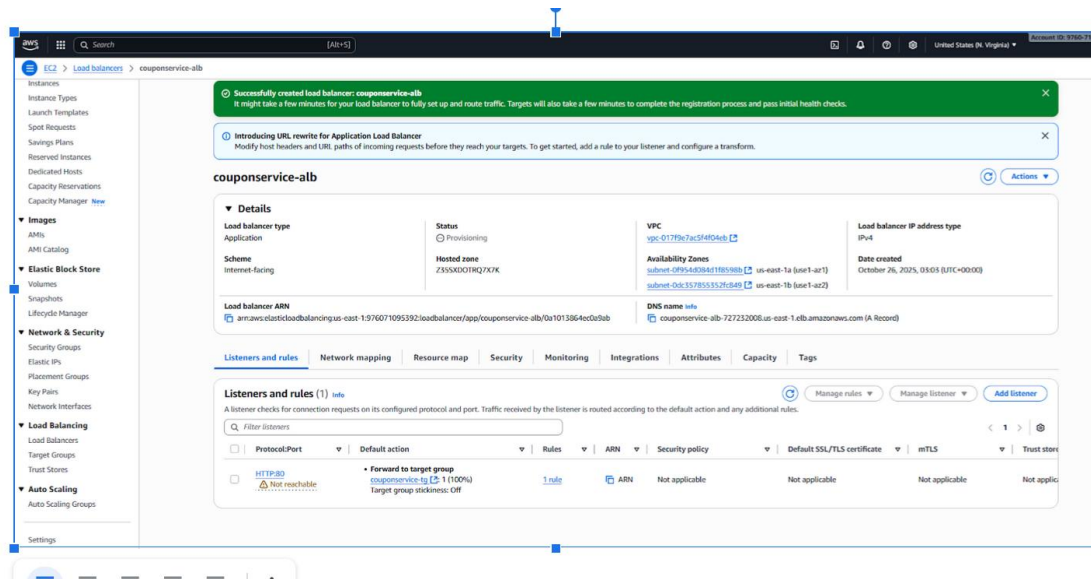


Figure 3.6: AWS console view of the successfully created Application Load Balancer (couponservice-alb).

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

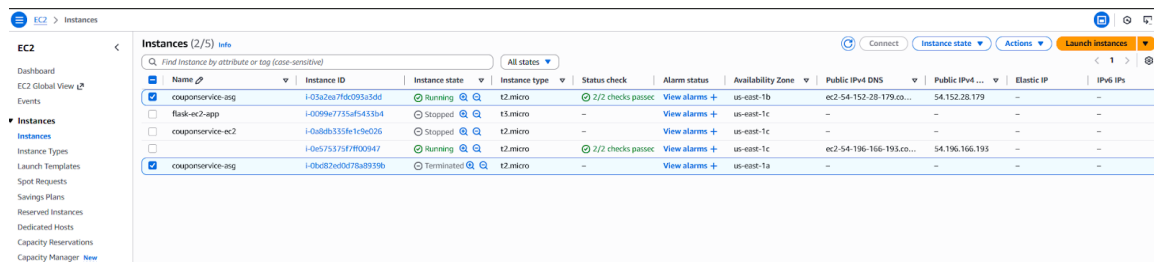


Figure 3.7: Verification showing EC2 instances managed by ASG and health status summary.

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.

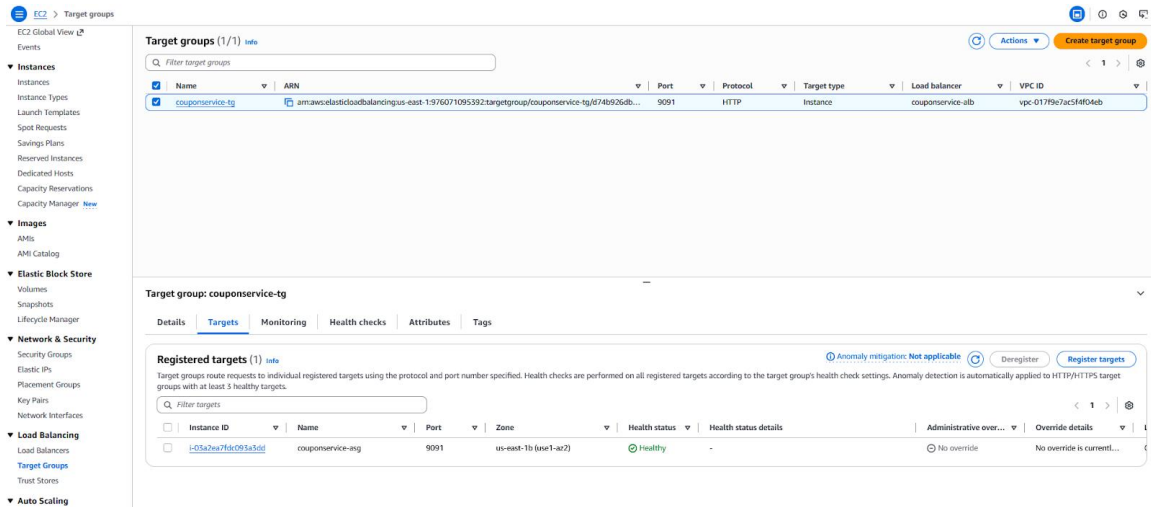


Figure 3.8: Target Group (couponservice-tg) showing registered EC2 instance with Healthy status after updating the health check path to /actuator/health.

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

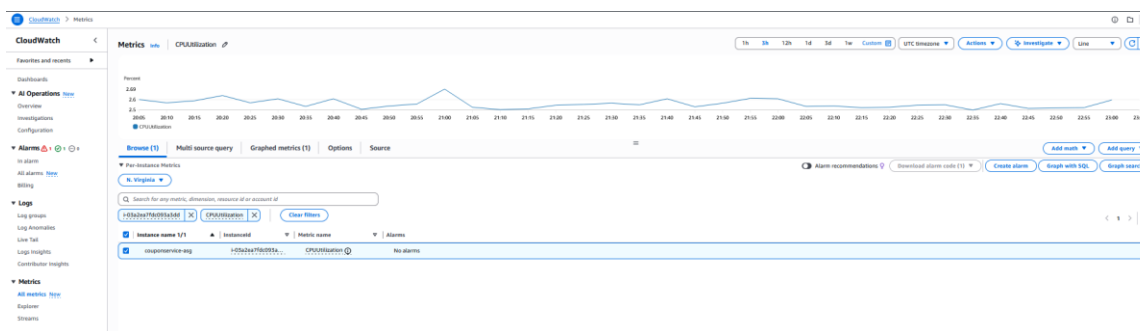


Figure 4.1: CloudWatch metric (CPUUtilization) before applying CPU stress

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

```
Run "/usr/bin/dnf check-release-update" for full release and version update info
#
# Amazon Linux 2023
#
https://aws.amazon.com/linux/amazon-linux-2023

Last login: Sun Oct 26 10:19:45 2025 from 86.46.76.41
[ec2-user@ip-172-31-86-108 ~]$ python3 --version
Python 3.9.23
[ec2-user@ip-172-31-86-108 ~]$ ls
app
[ec2-user@ip-172-31-86-108 ~]$ nano burn_cpu.py
[ec2-user@ip-172-31-86-108 ~]$ ls
app burn_cpu.py
[ec2-user@ip-172-31-86-108 ~]$ nano burn_cpu.py
[ec2-user@ip-172-31-86-108 ~]$ nano burn_cpu.py
[ec2-user@ip-172-31-86-108 ~]$ python3 burn_cpu.p^C
[ec2-user@ip-172-31-86-108 ~]$ python3 burn_cpu
python3: can't open file '/home/ec2-user/burn_cpu': [Errno 2] No such file or directory
[ec2-user@ip-172-31-86-108 ~]$ ls
app burn_cpu.py
[ec2-user@ip-172-31-86-108 ~]$ python3 burn_cpu.py
Starting CPU burn with 4 threads...
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
```

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

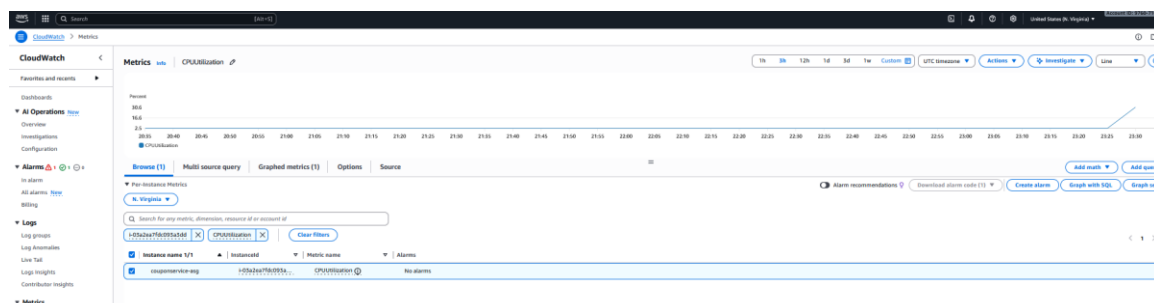


Figure 4.3: CloudWatch metric showing CPU spikes after initiating the stress test.

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

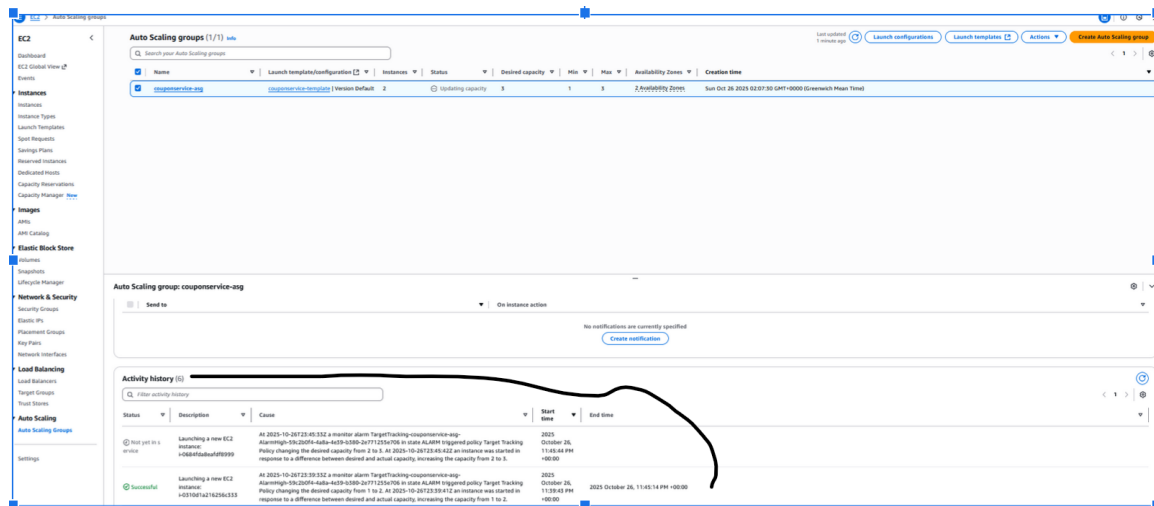


Figure 4.4: ASG activity history showing automatic instance creation triggered by CPU utilization.

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

Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IP v4 DNS	Public IP v4	Elastic IP	IPv6 IP	Monitoring	Security group name	Key name	Launch time
i-0310d1a216256c333	Running	t2.micro	2/2 checks passed	View alarms	us-east-1a	ec2-54-198-198-225.us-east-1b...	54.198.198.225	-	-	disabled	couponservice-arg	couponservice...	2025/10/26 02:13 GMT+0
load-w2-app	Stopped	t2.micro	-	View alarms	us-east-1c	-	-	-	-	disabled	w2-load-app-arg	load-key	2025/10/24 01:03 GMT+1
couponservice-w2	Stopped	t2.micro	-	View alarms	us-east-1c	-	-	-	-	disabled	couponservice-arg	couponservice...	2025/10/25 12:24 GMT+1

i-0310d1a216256c333 (couponservice-arg)	
Instance summary	
Instance ID	i-0310d1a216256c333
Public IP address	54.198.198.225 (open address)
Instance state	Running
Private IP DNS name (IPv4 only)	ip-172-31-7-177.ec2.internal
Instance type	t2.micro
VPC ID	vpc-017f9e7a3059b8ad
Subnet ID	subnet-0f5e4d84d19019b1
Instance IAM	arn:aws:iam::137907108533:instance-profile/i-0310d1a216256c333
Auto-assigned IP address	54.198.198.225 (Public IP)
Launch Role	EC2-Role
Instance IAM	arn:aws:iam::137907108533:instance-profile/i-0310d1a216256c333

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

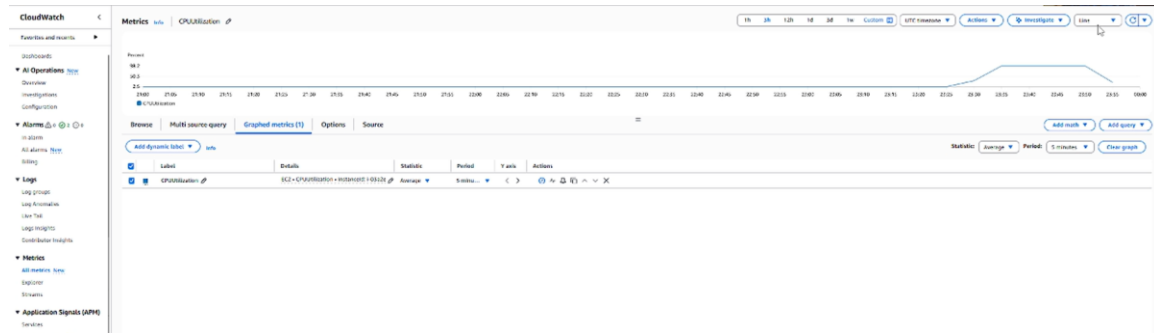


Figure 4.6: CloudWatch metric showing CPU utilization dropping after the Python stress script was stopped.

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

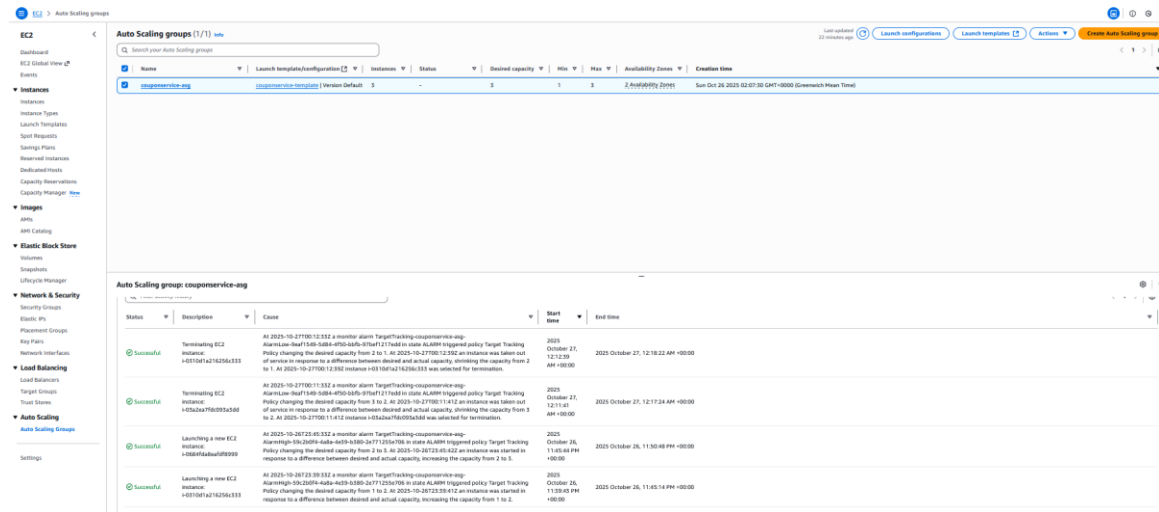


Figure 4.7: Auto Scaling Group activity history confirming automatic instance termination after load reduction.

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

Search

Instances

EC2

Instances (2/6) info

Find instance by attribute or tag name-vmwv04

All states

Dashboard

EC2 Global View

Events

Instances

Instance Types

Launch Templates

Spot Requests

Scaling Plans

Reserved Instances

Dedicated Hosts

Capacity Reservations

Capacity Manager

Images

Elastic Block Store

Volumes

Instances

Instance Profiles

Network Interfaces

Subnets

VPCs

VPN Connections

VPN Gateways

Virtual Private Clouds

Virtual Private Gateways

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

Virtual Private Networks

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)**.

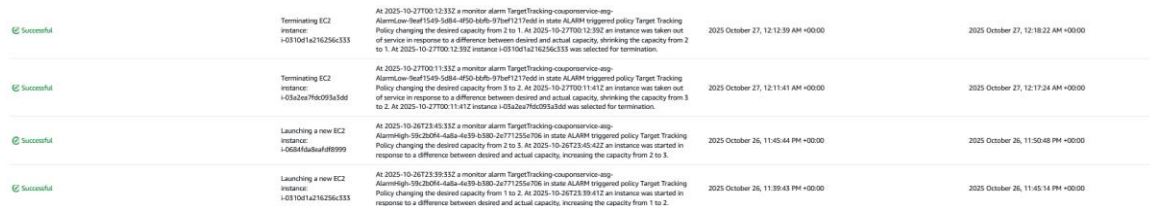
The main objective is to show how AWS automatically scales application instances based on CPU load, was fully achieved.

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

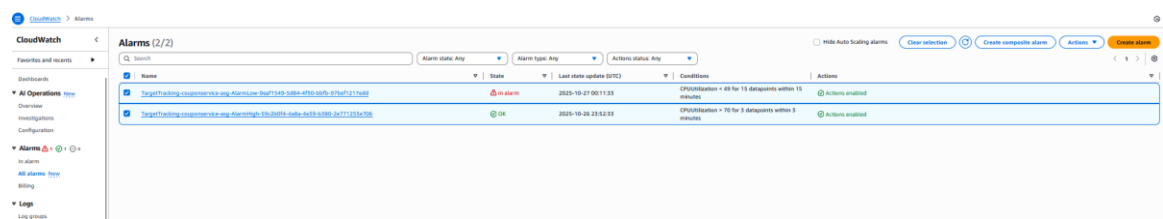
- **Scaling Delays:** One noticeable issue was the delay during scale-down events. Based on the Auto Scaling Activity History (See figure 5, below), it took approximately 20–30 minutes after the CPU load dropped for **AWS** to terminate the extra instances. The system launched additional instances between 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.



Successful	Terminating EC2 instance: i-0310d1a216256c333	At 2025-10-27T00:12:33Z a monitor alarm TargetTracking-coop-service-asp AlarmLow-Bfa1549-5d8d-4f9d-bd9f-97bf1217d8d in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-10-27T00:12:30Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 2 to 1. At 2025-10-27T00:12:39Z instance i-0310d1a216256c333 was selected for termination.	2025 October 27, 12:12:39 AM +0000	2025 October 27, 12:18:22 AM +0000
Successful	Terminating EC2 instance: i-03a2ea79a093a3d6f	At 2025-10-27T00:11:33Z a monitor alarm TargetTracking-coop-service-asp AlarmLow-Bfa1549-5d8d-4f9d-bd9f-97bf1217d8d in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 3 to 2. At 2025-10-27T00:11:41Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2025-10-27T00:11:41Z instance i-03a2ea79a093a3d6f was selected for termination.	2025 October 27, 12:11:41 AM +0000	2025 October 27, 12:17:24 AM +0000
Successful	Launching a new EC2 instance: i-0684f6a5a4a4f8999	At 2025-10-26T23:45:33Z a monitor alarm TargetTracking-coop-service-asp AlarmHigh-59c2b0f4-4da4-4c39-b380-2a777255e706 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 3. At 2025-10-26T23:45:42Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 October 26, 11:45:44 PM +0000	2025 October 27, 11:50:46 PM +0000
Successful	Launching a new EC2 instance: i-0310d1a216256c333	At 2025-10-26T23:39:33Z a monitor alarm TargetTracking-coop-service-asp AlarmHigh-59c2b0f4-4da4-4c39-b380-2a777255e706 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2025-10-26T23:39:41Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 October 26, 11:39:43 PM +0000	2025 October 26, 11:45:14 PM +0000

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.



Name	State	Last state update (UTC)	Conditions	Actions
TargetTracking-coop-service-asp	In alarm	2025-10-27 00:11:33	CPUUtilization < 49 for 15 datapoints within 15 minutes	Actions enabled
TargetTracking-coop-service-asp	OK	2025-10-26 23:52:03	CPUUtilization > 70 for 3 datapoints within 15 minutes	Actions enabled

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