SAPIENZA
UNIVERSITÀ DI ROMA

CLOUD COMPUTING

Deployment and Evaluation of a Scalable Multi-Tier Web Application on AWS

FINAL PROJECT REPORT

Professor:
Emiliano Casalicchio

Students:
Abzal Aidakhmetov, 2115331
Eldar Gabdulsattarov, 2113224
Nadir Nuralin, 2113428

Academic Year 2024/2025

Contents

1 Introduction

1.1 Project Objectives

The primary objective of this project is to design, deploy, and evaluate a scalable, multi-tier web application on the Amazon Web Services (AWS) cloud platform, within the constraints of the AWS Academy Learner Lab environment. The core focus is not on the complexity of the web application itself, but rather on the implementation and analysis of a robust cloud architecture that ensures high availability and automatic scalability.
The key requirements for the project are as follows:

- **Multi-Tier Architecture:** The application must be separated into distinct presentation (frontend), application (backend), and data (database) layers.
- **High Availability:** The deployment must span at least two AWS Availability Zones (AZs) to ensure resilience against single-AZ failures.
- **Automatic Scalability:** The application tier must be configured with an Auto Scaling Group (ASG). This ASG must utilize a **Step Scaling** policy to automatically adjust the number of running instances in response to changing load conditions. The use of simpler Target Tracking scaling policies is explicitly disallowed.
- **Performance Evaluation:** A detailed analysis of the system's performance and scalability must be conducted under various workload conditions. This involves generating a synthetic load and measuring key performance indicators (KPIs) to demonstrate that the architecture scales as designed.

This report details the architectural design, implementation steps, and the results of the performance evaluation for a simple To-Do List web application built to meet these objectives.

1.2 Web App Overview

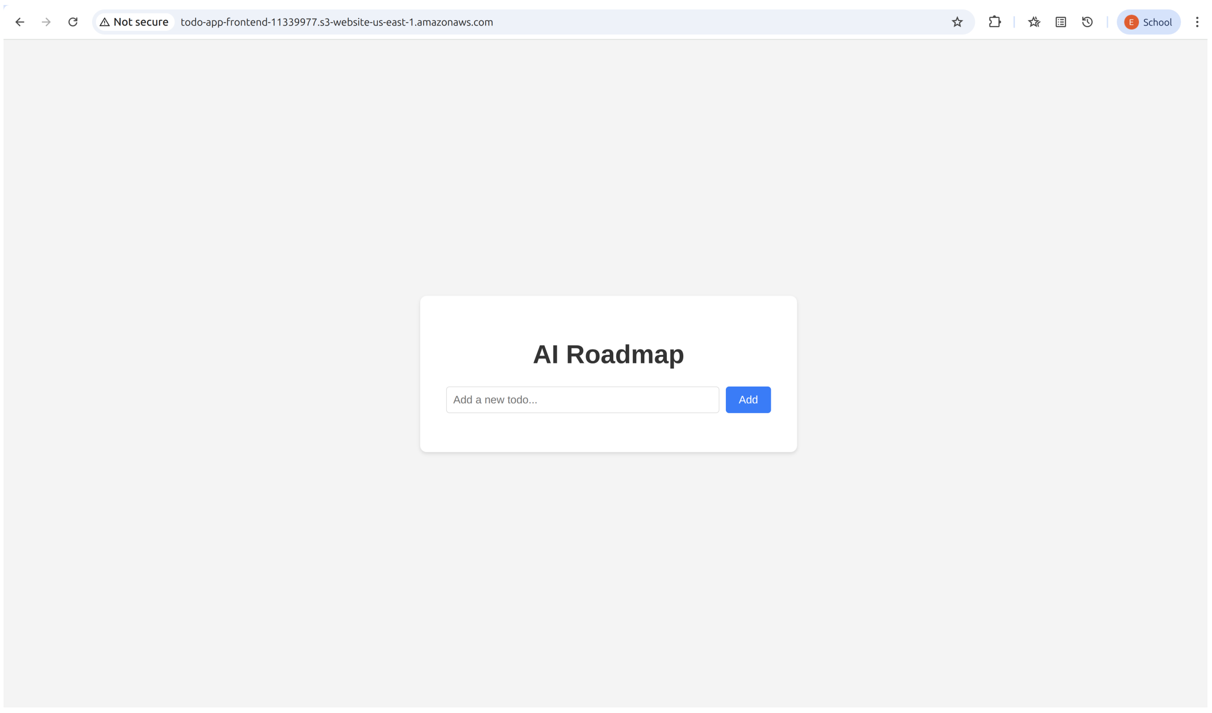

Not secure
todo-app-frontend-11339977.s3-website-us-east-1.amazonaws.com
School
AI Roadmap
Add a new todo...
Add

Figure 1: "AI Roadmap" website.

The web application developed in this project is a collaborative to-do platform that serves as an open-source AI Roadmap (Fig. 1). Its core idea is to outline the essential skills and tasks needed to become a proficient AI engineer. Users can freely add or remove items, making the roadmap a dynamic, community-driven resource. The interface is minimal and user-friendly, enabling real-time editing where all changes are instantly visible to everyone.

The application relies on a backend API and DynamoDB database to manage shared data across users. It was hosted on AWS to take advantage of its scalable and highly available infrastructure, ensuring the platform remains accessible and responsive under varying loads.

2 Solution Design and Architecture

To meet the project objectives, a highly available and scalable architecture was designed, leveraging several core AWS services. The architecture is logically separated into three tiers: a presentation tier for the user interface, an application tier for business logic, and a data tier for persistence.

2.1 Architectural Overview

The overall architecture is depicted in Figure 2. User traffic is directed to two distinct AWS services depending on the type of content requested. Requests for the static frontend assets (HTML, CSS, JavaScript) are served by Amazon S3, while dynamic API requests are routed through an Application Load Balancer to the backend application servers running on EC2.

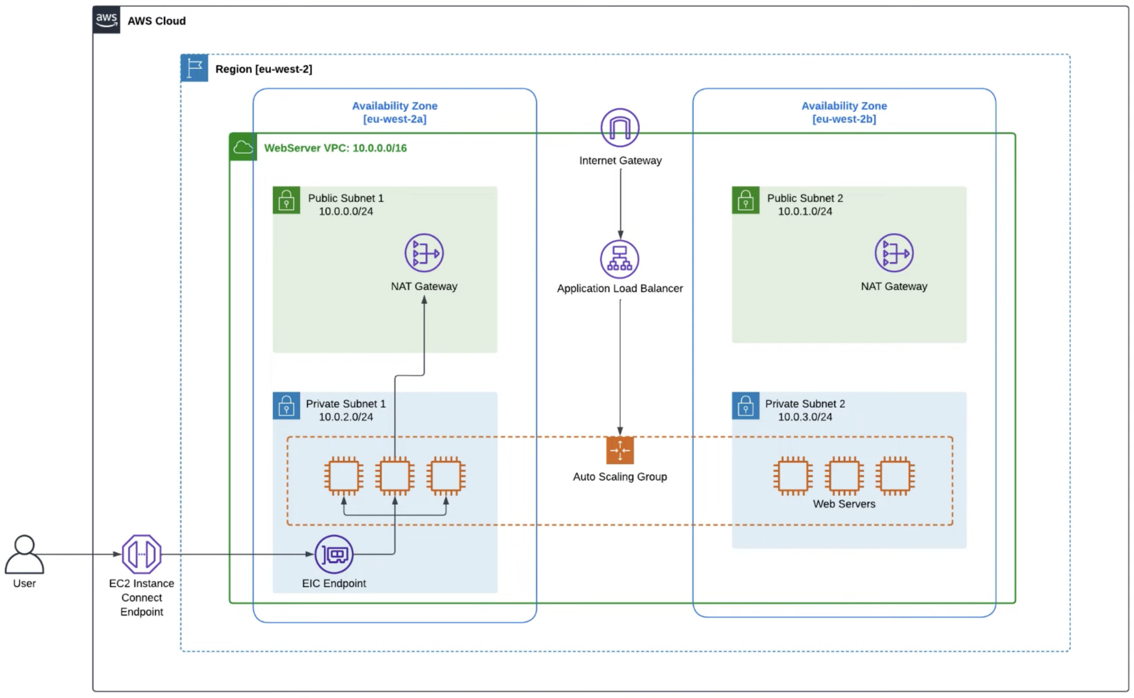

AWS Cloud
Region [eu-west-2]
Availability Zone [eu-west-2a]
Availability Zone [eu-west-2b]
WebServer VPC: 10.0.0.0/16
Internet Gateway
Public Subnet 1 10.0.0.0/24
NAT Gateway
Application Load Balancer
Public Subnet 2 10.0.1.0/24
NAT Gateway
Private Subnet 1 10.0.2.0/24
Private Subnet 2 10.0.3.0/24
Auto Scaling Group
Web Servers
User
EC2 Instance Connect Endpoint
EIC Endpoint

Figure 2: High-level architectural diagram of the multi-tier application.

2.2 AWS Services Utilized

To build and manage the architecture, a range of AWS services were integrated to support compute, storage, networking, and observability needs. The backend application was deployed on **Amazon EC2** instances, which formed the core compute layer. These instances were managed by an **Auto Scaling Group (ASG)** to maintain performance and availability as demand changed. For the frontend, we used **Amazon S3** as a simple and cost-effective way to host static files, delivering them reliably at scale.

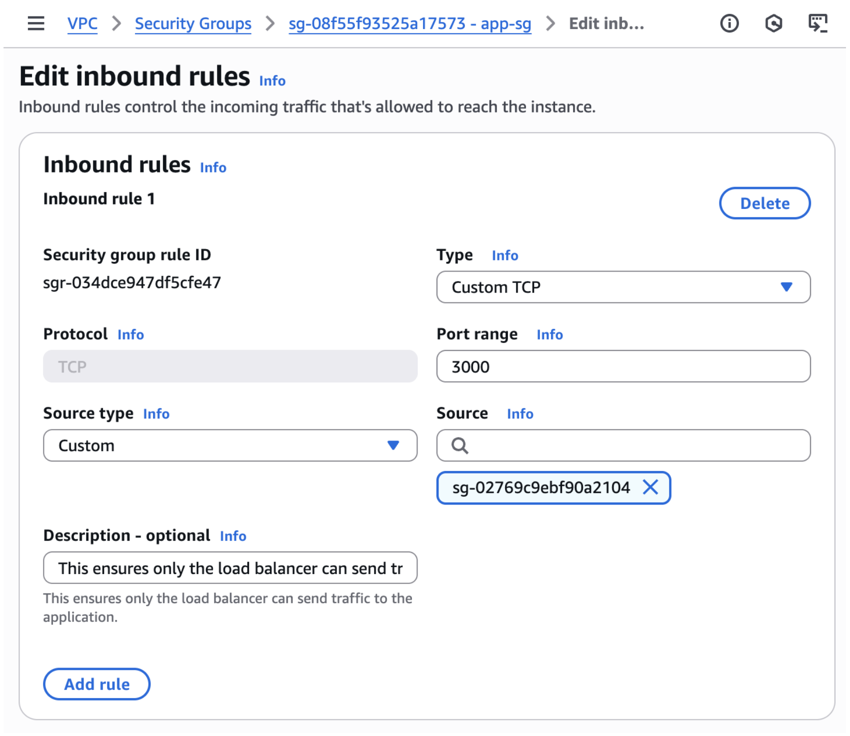
To persist application data, **Amazon DynamoDB** served as the backend database. As a serverless and fully managed NoSQL service, it allowed us to focus on development without worrying about provisioning or maintaining database servers. Incoming HTTP requests were handled through an Application Load Balancer, part of the **Elastic Load Balancing (ELB)** service, which distributed traffic across healthy EC2 instances in multiple Availability Zones. This ensured both fault tolerance and a single, resilient entry point to the backend.

The entire infrastructure was deployed within a **Virtual Private Cloud (VPC)**, offering full control over subnets, route tables, and access control policies. For performance testing, **AWS Cloud9** was used as a cloud-based development environment. It provided a reliable setup for generating load and running test scripts directly within the AWS network. Finally, **Amazon CloudWatch** was used to monitor the performance of the application and the health of the system. It collected key metrics like CPU utilization, which were used to trigger alarms and scaling actions defined in our step scaling policies.

2.3 Network Design and Security

The network was designed with security and availability as primary concerns. The VPC is configured with subnets across two Availability Zones.

- **Public Subnets:** These subnets contain the Application Load Balancer (ALB) and have a route to the internet via an Internet Gateway. This allows the ALB to be publicly accessible.
- **Private Subnets:** The EC2 instances running the backend application are placed in these subnets. They do not have a direct route to the internet. They can access external services such as Amazon Elastic Container Registry (ECR) via a NAT Gateway, but the public internet cannot initiate connections to them directly.


VPC > Security Groups > sg-08f55f93525a17573 - app-sg > Edit inb...
Info
Delete
Info
Add rule

Edit inbound rules

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

Inbound rules

Inbound rule 1

Security group rule ID: sgr-034dce947df5cfe47

Type: Custom TCP

Protocol: TCP

Port range: 3000

Source type: Custom

Source: sg-02769c9ebf90a2104

Description - optional: This ensures only the load balancer can send tr

This ensures only the load balancer can send traffic to the application.

Figure 3: A screenshot from the AWS Console showing the inbound rules for the Application Security Group (‘app-sg‘), demonstrating that it only accepts traffic from the ALB’s security group.

Security is enforced using Security Groups, which act as stateful firewalls.

- An **ALB Security Group** allows inbound HTTP traffic on port 80 from anywhere (‘0.0.0.0/0‘).
- An **Application Security Group** allows inbound traffic on the application port (3000) *only* from the ALB Security Group (Fig. 3). This ensures that application instances can only be reached through the load balancer.

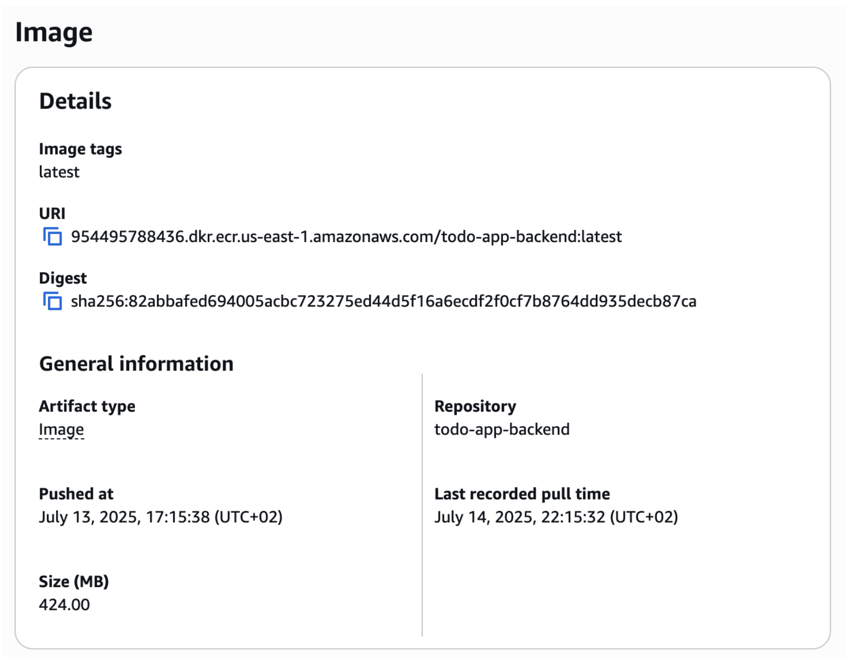
3 Implementation Details

The architectural design was implemented by configuring the required AWS services and deploying the application code. This section details the key implementation steps.

3.1 Application and Containerization

The backend application is a lightweight REST API built with Node.js and the Express framework. It exposes simple CRUD (Create, Read, Update, Delete) endpoints for managing to-do items.

To ensure a consistent and portable runtime environment, the backend application was containerized using Docker. A multi-stage 'Dockerfile' was created to build the application and package it with the Node.js 18 runtime. The final Docker image was pushed to and stored in ECR, as shown in Figure 4.



Image

Details

Image tags
latest

URI
954495788436.dkr.ecr.us-east-1.amazonaws.com/todo-app-backend:latest

Digest
sha256:82abbafed694005acbc723275ed44d5f16a6ecdf2f0cf7b8764dd935decb87ca

General information

Artifact type
Image

Repository
todo-app-backend

Pushed at
July 13, 2025, 17:15:38 (UTC+02)

Last recorded pull time
July 14, 2025, 22:15:32 (UTC+02)

Size (MB)
424.00

Figure 4: Screenshot of the 'todo-app-backend' repository in Amazon ECR, showing the pushed image.

3.2 Backend Infrastructure Deployment

The backend infrastructure was deployed manually using the AWS Management Console.

3.2.1 Launch Template Configuration

A central component of the deployment is the EC2 Launch Template, which serves as a blueprint for the application instances. Key configurations include:

- **Amazon Machine Image (AMI):** Amazon Linux 2023.
- **Instance Type:** 't2.micro'.
- **IAM Role:** The pre-configured 'LabInstanceProfile' was used to grant the instance necessary permissions to interact with other AWS services like ECR and DynamoDB.
- **User Data Script:** A startup script was provided to install Docker, log in to ECR, and pull and run the application's Docker container.

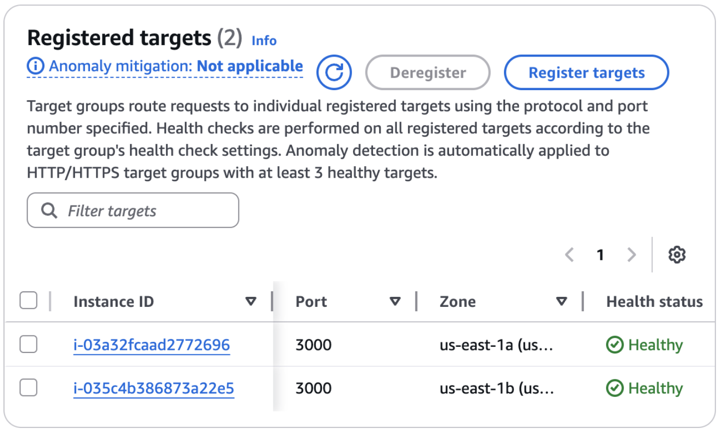
Each time an instance is launched, this template ensures that it is fully configured to serve the application without manual setup. The user data script automates the entire bootstrapping process, from installing Docker to starting the backend service. By combining the launch template with IAM roles and secure networking, we ensured reliable and repeatable instance provisioning within the ASG.

3.2.2 Auto Scaling Group and Load Balancer

The ASG was configured to manage the EC2 instances across the two private subnets.

- **Desired Capacity:** The group was initialized with a desired capacity of 2 instances to ensure high availability from the start.
- **Load Balancer Integration:** The ASG was attached to the Application Load Balancer's target group. ELB health checks were enabled to ensure the ASG could replace unhealthy instances. The health check was configured to target the root ('/') endpoint of the application.

Figure 5 shows the two initial instances running and passing the load balancer health checks, confirming a successful deployment. This setup ensured balanced traffic distribution, automatic instance replacement, and reactive scaling under load.


Registered targets (2) Info
Anomaly mitigation: Not applicable
Deregister
Register targets
Filter targets
1

Target groups route requests to individual registered targets using the protocol and port number specified. Health checks are performed on all registered targets according to the target group's health check settings. Anomaly detection is automatically applied to HTTP/HTTPS target groups with at least 3 healthy targets.

Instance ID	Port	Zone	Health status
i-03a32fcaad2772696	3000	us-east-1a (us...	Healthy
i-035c4b386873a22e5	3000	us-east-1b (us...	Healthy

Figure 5: The Target Group view in the EC2 console showing two instances in a 'healthy' state across two different Availability Zones.

3.3 Frontend Deployment

The frontend, consisting of static HTML, CSS, and JavaScript files, was deployed using Amazon S3. A bucket was created, and its public access settings and bucket policy were configured to allow public read access. The S3 bucket was then enabled for static website hosting, providing a URL endpoint to access the user interface. The 'script.js' file was updated to point its API requests to the DNS name of the Application Load Balancer.

4 Scalability and Performance Evaluation

With the application successfully deployed, a series of tests were conducted to evaluate the performance and demonstrate the functionality of the auto-scaling configuration.

4.1 Testing Methodology

- **Load Generation Tool:** The 'wrk' command-line utility was used to generate the HTTP load. This tool was chosen for its high efficiency and ability to generate a significant load from a single machine. It was run from an AWS Cloud9 instance within the same AWS region to ensure stable network performance.
- **Test Scenario:** The load test targeted the 'POST /todos' API endpoint to simulate users creating new items, as this is a realistic, database-writing workload.
- **Key Performance Indicators (KPIs):** The primary metrics monitored during the tests were:
  - **Average CPU Utilization:** The average CPU load across all instances in the ASG, monitored via Amazon CloudWatch. This was the trigger for scaling policies.
  - **Instance Count:** The number of active EC2 instances in the ASG.
  - **Request/Response Metrics:** Including average latency and socket timeouts, as reported by 'wrk'.

4.2 Auto Scaling Policy Configuration

To meet the project requirements, four Step Scaling policies were configured for the ASG, triggered by CloudWatch alarms based on the average CPU utilization over a 1-minute period. Each policy defined specific thresholds that determined how many EC2 instances to add or remove depending on the system load.

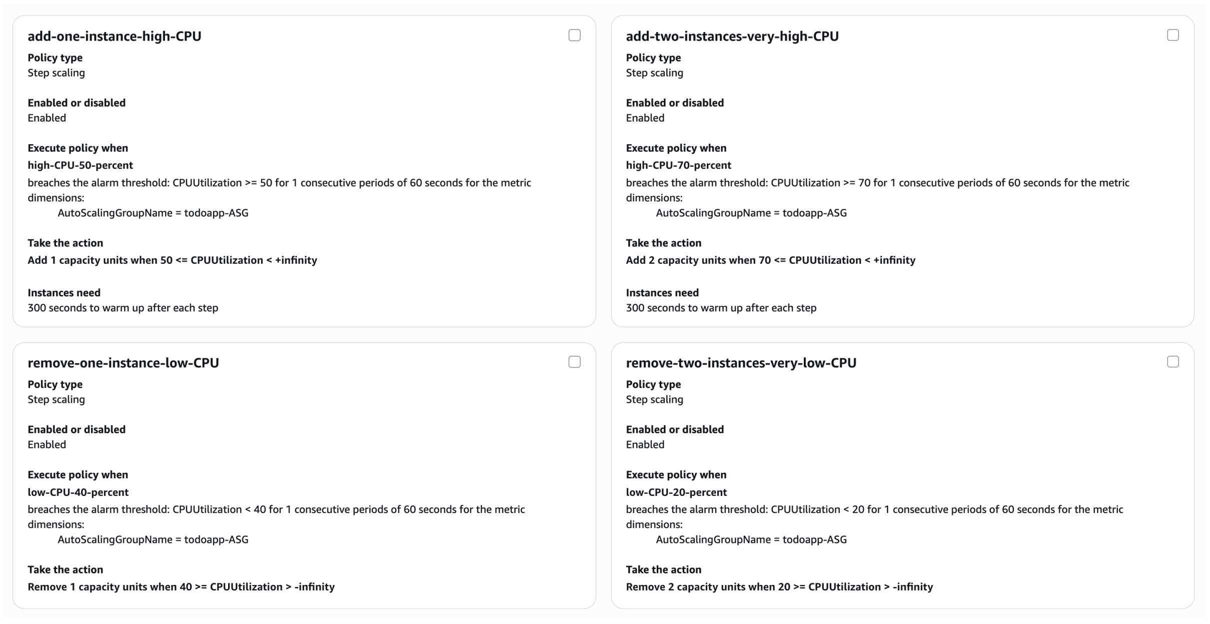
Policy Type	**Policy Name**	**Condition**	**Action**
Scale-Out Policy	High CPU	CPU Utilization $\geq$ 50% for 1 minute	Add 1 instance
	Very High CPU	CPU Utilization $\geq$ 70% for 1 minute	Add 2 instances
Scale-In Policy	Low CPU	CPU Utilization $<$ 40% for 1 minute	Remove 1 instance
	Very Low CPU	CPU Utilization $<$ 20% for 1 minute	Remove 2 instances

Table 1: Auto Scaling Step Policies based on CPU Utilization

The logic behind the scaling actions is summarized in Table 1, which outlines the CPU thresholds and the corresponding responses. For example, if the CPU load increased above 50%, an additional instance would be launched to maintain performance. In contrast, if usage dropped below a certain level, the group would scale in to reduce unnecessary resource usage. Figure 6 shows the actual configuration as seen in the AWS Console, confirming that the four scaling policies were properly configured and enabled.

4.3 Experimental Results

The evaluation was conducted in four distinct stages to observe the system's behavior under different loads.



add-one-instance-high-CPU
Policy type
Step scaling

Enabled or disabled
Enabled

Execute policy when
high-CPU-50-percent
breaches the alarm threshold: CPUUtilization >= 50 for 1 consecutive periods of 60 seconds for the metric dimensions:
AutoScalingGroupName = todoapp-ASG

Take the action
Add 1 capacity units when 50 <= CPUUtilization < +infinity

Instances need
300 seconds to warm up after each step

add-two-instances-very-high-CPU
Policy type
Step scaling

Enabled or disabled
Enabled

Execute policy when
high-CPU-70-percent
breaches the alarm threshold: CPUUtilization >= 70 for 1 consecutive periods of 60 seconds for the metric dimensions:
AutoScalingGroupName = todoapp-ASG

Take the action
Add 2 capacity units when 70 <= CPUUtilization < +infinity

Instances need
300 seconds to warm up after each step

remove-one-instance-low-CPU
Policy type
Step scaling

Enabled or disabled
Enabled

Execute policy when
low-CPU-40-percent
breaches the alarm threshold: CPUUtilization < 40 for 1 consecutive periods of 60 seconds for the metric dimensions:
AutoScalingGroupName = todoapp-ASG

Take the action
Remove 1 capacity units when 40 >= CPUUtilization > -infinity

remove-two-instances-very-low-CPU
Policy type
Step scaling

Enabled or disabled
Enabled

Execute policy when
low-CPU-20-percent
breaches the alarm threshold: CPUUtilization < 20 for 1 consecutive periods of 60 seconds for the metric dimensions:
AutoScalingGroupName = todoapp-ASG

Take the action
Remove 2 capacity units when 20 >= CPUUtilization > -infinity

Figure 6: A screenshot from the AWS Console showing the four configured Step Scaling policies for the Auto Scaling Group.

4.3.1 Test 1: Baseline Performance

A baseline test was run with 150 concurrent connections for 15 minutes to determine the capacity of the initial 1-instance configuration. The system handled an average of approximately 670 requests per second, but experienced 1135 socket timeouts, indicating that it was at its performance limit. The average CPU utilization hovered around 45-50%, just on the edge of the first scaling threshold.

4.3.2 Test 2: Scale-Out Event Under Sustained Load

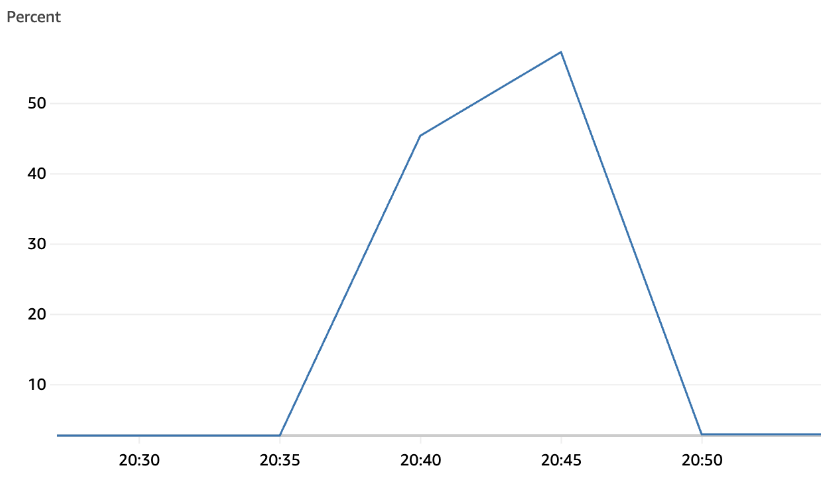
The load test was intensified to 1000 concurrent connections sustained over a 30-minute period in order to deliberately exceed the scaling threshold. As depicted in Figure 8, the average CPU utilization quickly exceeded the alarm threshold 70%. This triggered the CloudWatch alarm, which activated the `Add-1-Instance-High-CPU` policy. The ASG activity log shown in Figure 9 confirms that new instances were successfully launched in response. As a result, the total number of instances increased from 1 to 9. Despite the autoscaling activity, the average CPU utilization remained above 70% throughout the duration of the test, indicating a sustained high load on the application.

4.3.3 Test 3: Scale-In Event

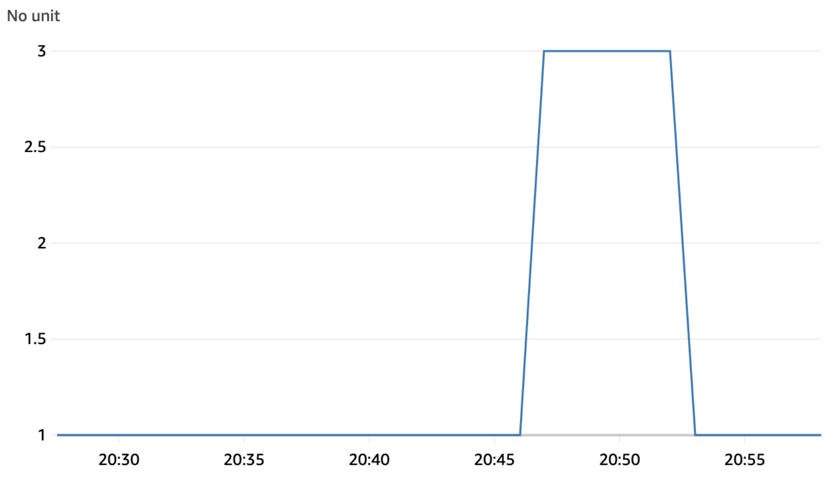
After stopping the load test, the system was monitored to observe its scale-in behavior. With no active load, the average CPU utilization dropped to nearly 0%. This triggered the `Very-Low-CPU` alarm, which prompted the ASG to terminate one of the instances. The group returned to its desired capacity of 2 instances. This behavior demonstrates the system's ability to automatically reduce its resource footprint and associated costs when the additional capacity is no longer required.

4.3.4 Test 4: High Availability Test

High availability is the test to determine whether a single server failure can bring down the website. To show that our platform can lose hardware and still serve users, we staged a controlled shutdown of one application instance. The plan was simple: shut down one of the two web servers that make up the application tier and watch the ASG repair itself while traffic continues to flow.



(a) CloudWatch graph showing the Average CPU Utilization during the baseline test, remaining stable around 45-50%.



(b) New instances come online.

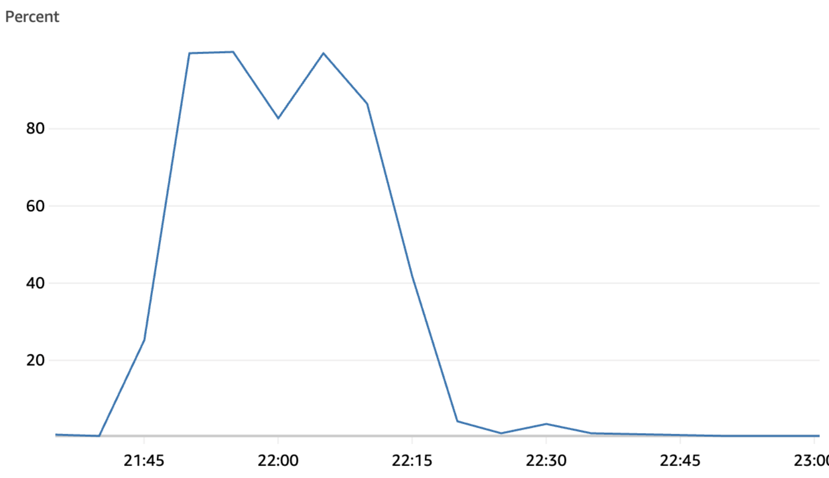
Figure 7: CloudWatch graphs showing scale-out behavior based on CPU Utilization.

Before the test, the ASG sat at its desired capacity of 2 application instances, one per Availability Zone, along with the separate Cloud9 host we used only for screenshots (Fig. 10). At T_0 we manually stopped the instance in ‘us-east-1b‘. Within seconds, its state changed to *shutting-down* (Fig. 11, the ALB marked it *unhealthy*, and traffic shifted completely to the surviving node. The sudden drop in capacity triggered the ASG’s evaluation logic; less than a minute later, a replacement launch was underway (Fig. 12).

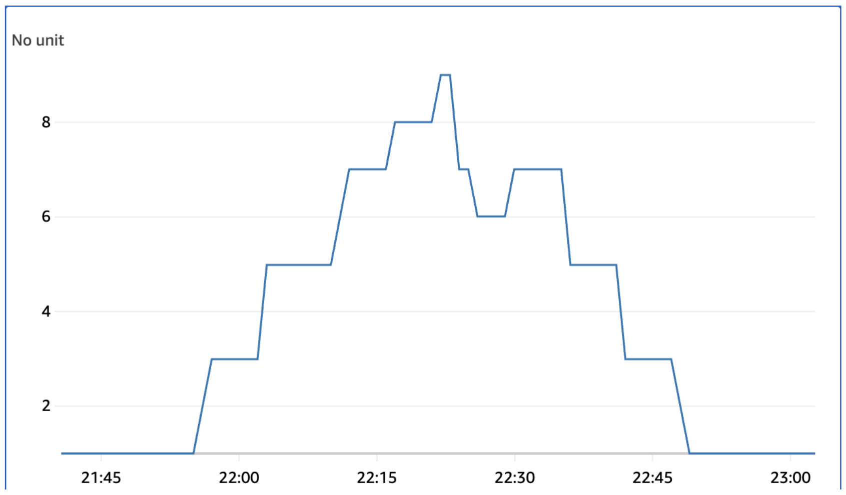
While the new EC2 instance was coming up, we kept an eye on status checks and network metrics. CPU load on the remaining server increased but held well below saturation, confirming that a single node can handle the application’s baseline traffic. Roughly two minutes after boot, the newcomer reported *Running* and cleared the 2/2 system checks (Fig. 13). A short interval later, the ALB registered two healthy targets again, re-establishing full redundancy (Fig. 14).

CloudWatch recorded the paired lifecycle events—one termination, one launch—both flagged *Successful* and completed within 5 minutes of the induced fault (Fig. 15). Throughout the experiment, we refreshed the site through the ALB’s DNS name and never observed an error or noticeable latency spike.

In sum, the test demonstrates that the architecture delivers on its high availability promise, it detects a failed instance, routes traffic away instantly, provisions a replacement automatically, and restores normal



(a) CloudWatch graph showing the Average CPU Utilization during the heavy load test, going above 70%.



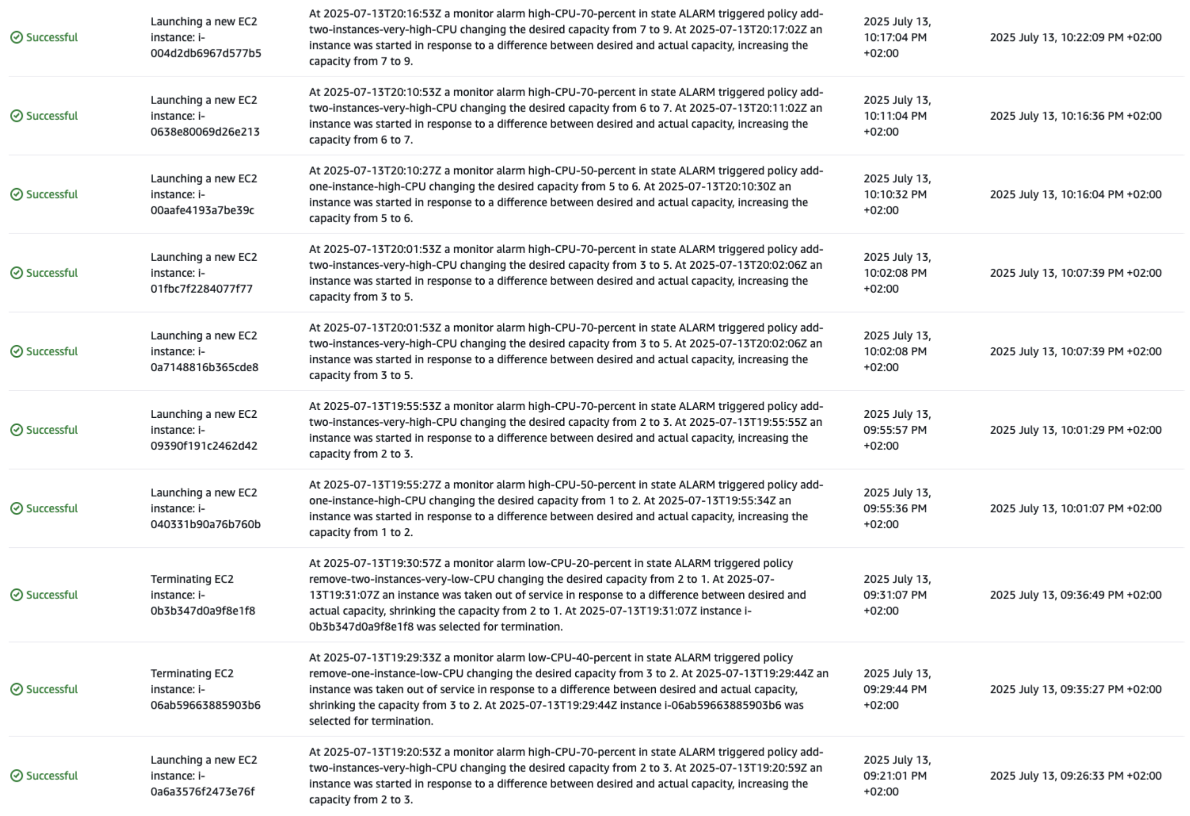
(b) Up to 9 instances show up, before scaling in.

Figure 8: CloudWatch graphs showing scale-out behavior based on CPU Utilization.

operation without any user-visible disruption.

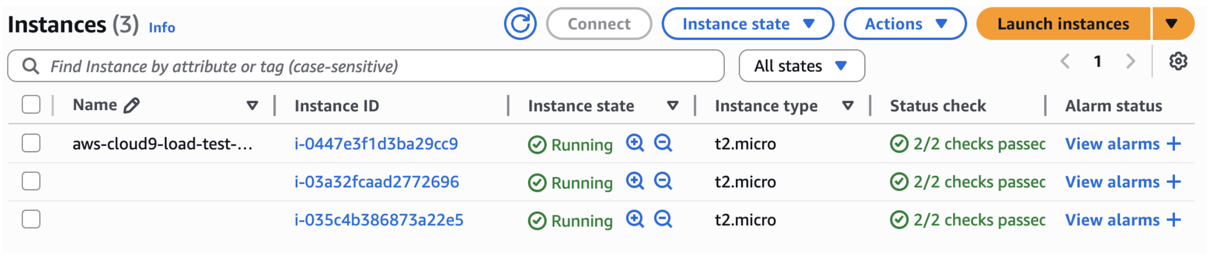
4.4 Analysis of Results

The experimental results confirm that the deployed architecture successfully meets the project's scalability requirements. The Step Scaling policies functioned as designed, automatically adding and removing compute capacity in response to real-time load metrics. The multi-AZ deployment of the ASG and the ALB ensures that this scalability is also highly available. The use of a managed database service, DynamoDB, and static hosting on S3 further enhances the overall robustness and scalability of the solution by offloading state and static content delivery from the primary compute tier.



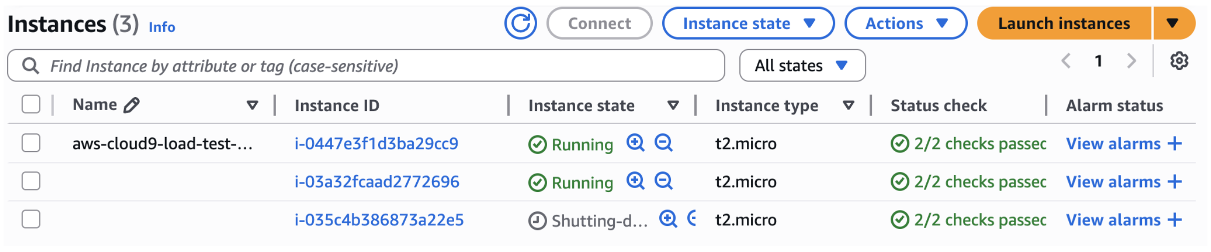
✓ Successful	Launching a new EC2 instance: i-004d2db6967d577b5	At 2025-07-13T20:16:53Z a monitor alarm high-CPU-70-percent in state ALARM triggered policy add-two-instances-very-high-CPU changing the desired capacity from 7 to 9. At 2025-07-13T20:17:02Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 7 to 9.	2025 July 13, 10:17:04 PM +02:00	2025 July 13, 10:22:09 PM +02:00
✓ Successful	Launching a new EC2 instance: i-0638e80069d26e213	At 2025-07-13T20:10:53Z a monitor alarm high-CPU-70-percent in state ALARM triggered policy add-two-instances-very-high-CPU changing the desired capacity from 6 to 7. At 2025-07-13T20:11:02Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 6 to 7.	2025 July 13, 10:11:04 PM +02:00	2025 July 13, 10:16:36 PM +02:00
✓ Successful	Launching a new EC2 instance: i-00aafe4193a7be39c	At 2025-07-13T20:10:27Z a monitor alarm high-CPU-50-percent in state ALARM triggered policy add-one-instance-high-CPU changing the desired capacity from 5 to 6. At 2025-07-13T20:10:30Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 5 to 6.	2025 July 13, 10:10:32 PM +02:00	2025 July 13, 10:16:04 PM +02:00
✓ Successful	Launching a new EC2 instance: i-01fbc7f2284077f77	At 2025-07-13T20:01:53Z a monitor alarm high-CPU-70-percent in state ALARM triggered policy add-two-instances-very-high-CPU changing the desired capacity from 3 to 5. At 2025-07-13T20:02:06Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 3 to 5.	2025 July 13, 10:02:08 PM +02:00	2025 July 13, 10:07:39 PM +02:00
✓ Successful	Launching a new EC2 instance: i-0a7148816b365cde8	At 2025-07-13T20:01:53Z a monitor alarm high-CPU-70-percent in state ALARM triggered policy add-two-instances-very-high-CPU changing the desired capacity from 3 to 5. At 2025-07-13T20:02:06Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 3 to 5.	2025 July 13, 10:02:08 PM +02:00	2025 July 13, 10:07:39 PM +02:00
✓ Successful	Launching a new EC2 instance: i-09390f191c2462d42	At 2025-07-13T19:55:53Z a monitor alarm high-CPU-70-percent in state ALARM triggered policy add-two-instances-very-high-CPU changing the desired capacity from 2 to 3. At 2025-07-13T19:55:55Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 July 13, 09:55:57 PM +02:00	2025 July 13, 10:01:29 PM +02:00
✓ Successful	Launching a new EC2 instance: i-040331b90a76b760b	At 2025-07-13T19:55:27Z a monitor alarm high-CPU-50-percent in state ALARM triggered policy add-one-instance-high-CPU changing the desired capacity from 1 to 2. At 2025-07-13T19:55:34Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 July 13, 09:55:36 PM +02:00	2025 July 13, 10:01:07 PM +02:00
✓ Successful	Terminating EC2 instance: i-0b3b347d0a9f8e1f8	At 2025-07-13T19:30:57Z a monitor alarm low-CPU-20-percent in state ALARM triggered policy remove-two-instances-very-low-CPU changing the desired capacity from 2 to 1. At 2025-07-13T19:31:07Z 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 2025-07-13T19:31:07Z instance i-0b3b347d0a9f8e1f8 was selected for termination.	2025 July 13, 09:31:07 PM +02:00	2025 July 13, 09:36:49 PM +02:00
✓ Successful	Terminating EC2 instance: i-06ab59663885903b6	At 2025-07-13T19:29:33Z a monitor alarm low-CPU-40-percent in state ALARM triggered policy remove-one-instance-low-CPU changing the desired capacity from 3 to 2. At 2025-07-13T19:29:44Z 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 2025-07-13T19:29:44Z instance i-06ab59663885903b6 was selected for termination.	2025 July 13, 09:29:44 PM +02:00	2025 July 13, 09:35:27 PM +02:00
✓ Successful	Launching a new EC2 instance: i-0a6a3576f2473e76f	At 2025-07-13T19:20:53Z a monitor alarm high-CPU-70-percent in state ALARM triggered policy add-two-instances-very-high-CPU changing the desired capacity from 2 to 3. At 2025-07-13T19:20:59Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 July 13, 09:21:01 PM +02:00	2025 July 13, 09:26:33 PM +02:00

Figure 9: A screenshot of the Auto Scaling Group's "Activity" tab, clearly showing a new instance being launched in response to the CloudWatch alarm.



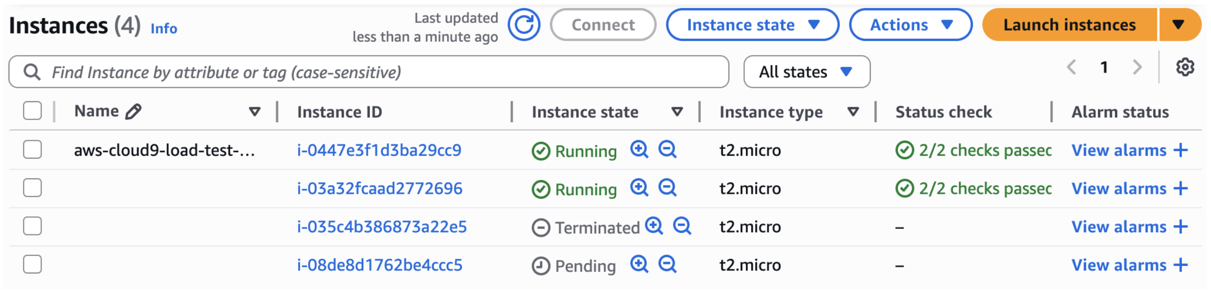
Name	Instance ID	Instance state	Instance type	Status check	Alarm status
aws-cloud9-load-test-…	i-0447e3f1d3ba29cc9	✓ Running	t2.micro	✓ 2/2 checks passec	View alarms +
	i-03a32fcaad2772696	✓ Running	t2.micro	✓ 2/2 checks passec	View alarms +
	i-035c4b386873a22e5	✓ Running	t2.micro	✓ 2/2 checks passec	View alarms +

Figure 10: Initial state of EC2 instances before failure injection.



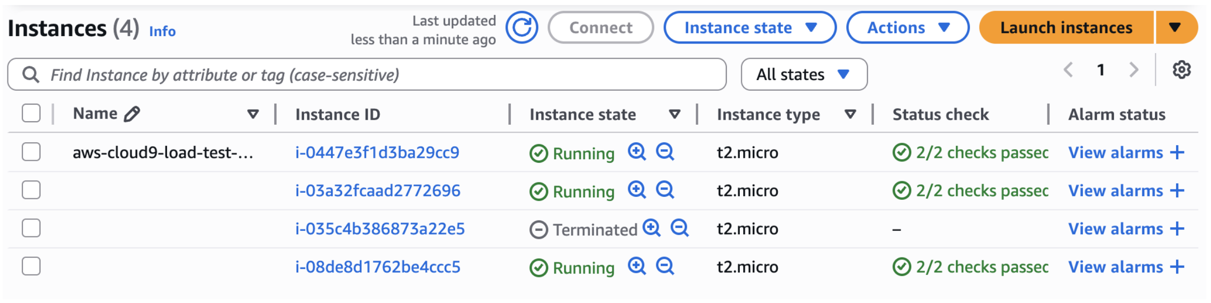
Name	Instance ID	Instance state	Instance type	Status check	Alarm status
aws-cloud9-load-test-…	i-0447e3f1d3ba29cc9	✓ Running	t2.micro	✓ 2/2 checks passec	View alarms +
	i-03a32fcaad2772696	✓ Running	t2.micro	✓ 2/2 checks passec	View alarms +
	i-035c4b386873a22e5	Shutting-d…	t2.micro	✓ 2/2 checks passec	View alarms +

Figure 11: One instance being shut down.


Instances (4) Info
Last updated less than a minute ago
Connect
Instance state
Actions
Launch instances
Find Instance by attribute or tag (case-sensitive)
All states
1

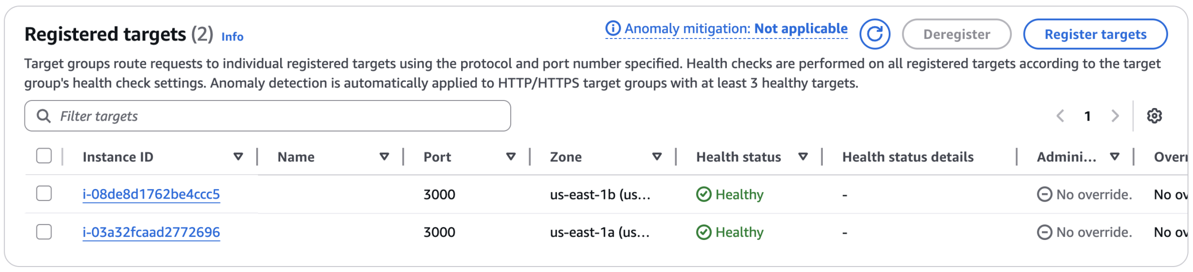
Name	Instance ID	Instance state	Instance type	Status check	Alarm status
aws-cloud9-load-test-…	i-0447e3f1d3ba29cc9	Running	t2.micro	2/2 checks passec	View alarms +
	i-03a32fcaad2772696	Running	t2.micro	2/2 checks passec	View alarms +
	i-035c4b386873a22e5	Terminated	t2.micro	–	View alarms +
	i-08de8d1762be4ccc5	Pending	t2.micro	–	View alarms +

Figure 12: New instance appearing in ’pending’ state.


Instances (4) Info
Last updated less than a minute ago
Connect
Instance state
Actions
Launch instances
Find Instance by attribute or tag (case-sensitive)
All states
1

Name	Instance ID	Instance state	Instance type	Status check	Alarm status
aws-cloud9-load-test-…	i-0447e3f1d3ba29cc9	Running	t2.micro	2/2 checks passec	View alarms +
	i-03a32fcaad2772696	Running	t2.micro	2/2 checks passec	View alarms +
	i-035c4b386873a22e5	Terminated	t2.micro	–	View alarms +
	i-08de8d1762be4ccc5	Running	t2.micro	2/2 checks passec	View alarms +

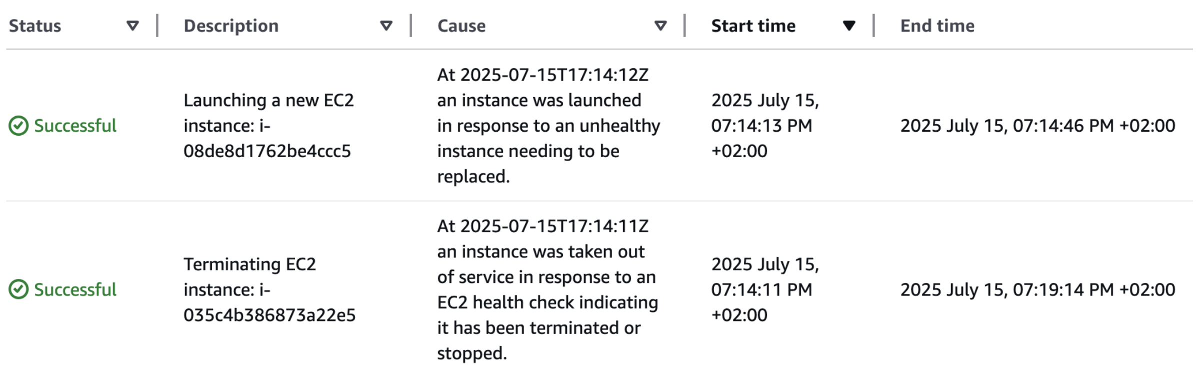
Figure 13: Replacement instance reaches ’running’ state.


Registered targets (2) Info
Anomaly mitigation: Not applicable
Deregister
Register targets
Filter targets
1

Target groups route requests to individual registered targets using the protocol and port number specified. Health checks are performed on all registered targets according to the target group's health check settings. Anomaly detection is automatically applied to HTTP/HTTPS target groups with at least 3 healthy targets.

Instance ID	Name	Port	Zone	Health status	Health status details	Admini…	Overr
i-08de8d1762be4ccc5		3000	us-east-1b (us…	Healthy	-	No override.	No ov
i-03a32fcaad2772696		3000	us-east-1a (us…	Healthy	-	No override.	No ov

Figure 14: Load balancer shows both instances as healthy.



Status	Description	Cause	Start time	End time
Successful	Launching a new EC2 instance: i-08de8d1762be4ccc5	At 2025-07-15T17:14:12Z an instance was launched in response to an unhealthy instance needing to be replaced.	2025 July 15, 07:14:13 PM +02:00	2025 July 15, 07:14:46 PM +02:00
Successful	Terminating EC2 instance: i-035c4b386873a22e5	At 2025-07-15T17:14:11Z an instance was taken out of service in response to an EC2 health check indicating it has been terminated or stopped.	2025 July 15, 07:14:11 PM +02:00	2025 July 15, 07:19:14 PM +02:00

Figure 15: Lifecycle events log: CloudWatch logs confirm successful.

5 Conclusion

This project successfully achieved the goal of designing, deploying, and validating a highly available and automatically scalable multi-tier web application on AWS. The final architecture, composed of an Application Load Balancer, an Auto Scaling Group with EC2 instances in private subnets, a DynamoDB database, and an S3-hosted frontend, adheres to cloud architecture best practices for resilience and scalability.

The key outcome of the project was the successful demonstration of the Step Scaling policies. The performance evaluation confirmed that the system could automatically provision new compute resources in response to sustained increases in CPU load and subsequently de-provision those resources when they were no longer needed. This elastic behavior is fundamental to building cost-effective and performant cloud-native applications.

Throughout the implementation, several technical challenges were encountered, which provided valuable learning experiences. These included:

- **Docker Architecture Mismatch:** An early ‘exec format error‘ revealed that Docker images built on an ARM-based local machine are not compatible with the x86-based EC2 instances, necessitating the use of a cross-platform build flag (‘–platform linux/amd64‘).
- **Node.js Version Incompatibility:** Application crashes were traced back to an outdated Node.js version in the initial Docker image, which did not support methods used by its dependencies. Updating the base image from ‘node:14‘ to ‘node:18‘ resolved the issue.
- **AWS Learner Lab IAM Restrictions:** The inability to create IAM users and roles required adapting the implementation to use the provided temporary credentials and the pre-configured ‘LabInstanceProfile‘ role, highlighting the importance of working within established security boundaries.
- **Network and Health Check Debugging:** Initial instance health check failures required a methodical debugging process, involving the analysis of security group rules, system logs, and application logs to pinpoint and resolve the root causes.

Overcoming these challenges was instrumental in reaching the final, working deployment. The project serves as a practical demonstration of the principles and services required to build robust, self-scaling systems on the AWS platform.