DevOps

LOAD-BALANCING AND AUTO-SCALING WEB APP IMPLEMENTATION

Stephen Power (20093364)

Developer Operations

Load-Balancing and Auto-Scaling Implementation

Stephen Power (20093364)

Project Title:	Load-Balanced Auto-Scaling Web Application
Student Name:	Stephen Power
Student Number:	20093364
Start Date:	November 17, 2022

Description:	This document reports the implementation of the load-balanced auto-scaling web application. The objective of this assignment is to deploy the web application and to automate it's management by using load-balancing and auto-scaling through AWS.

Contents

Architecture Diagram

This diagram represents the architecture of the assignment. A virtual private cloud (VPC) is set up as the host. An application load balancer is created within the VPC in order to balance the load over EC2 instances. An auto-scaling group is set up to configure the minimum, maximum and desired amount of instances. Web server applications are launched on the EC2 instances.

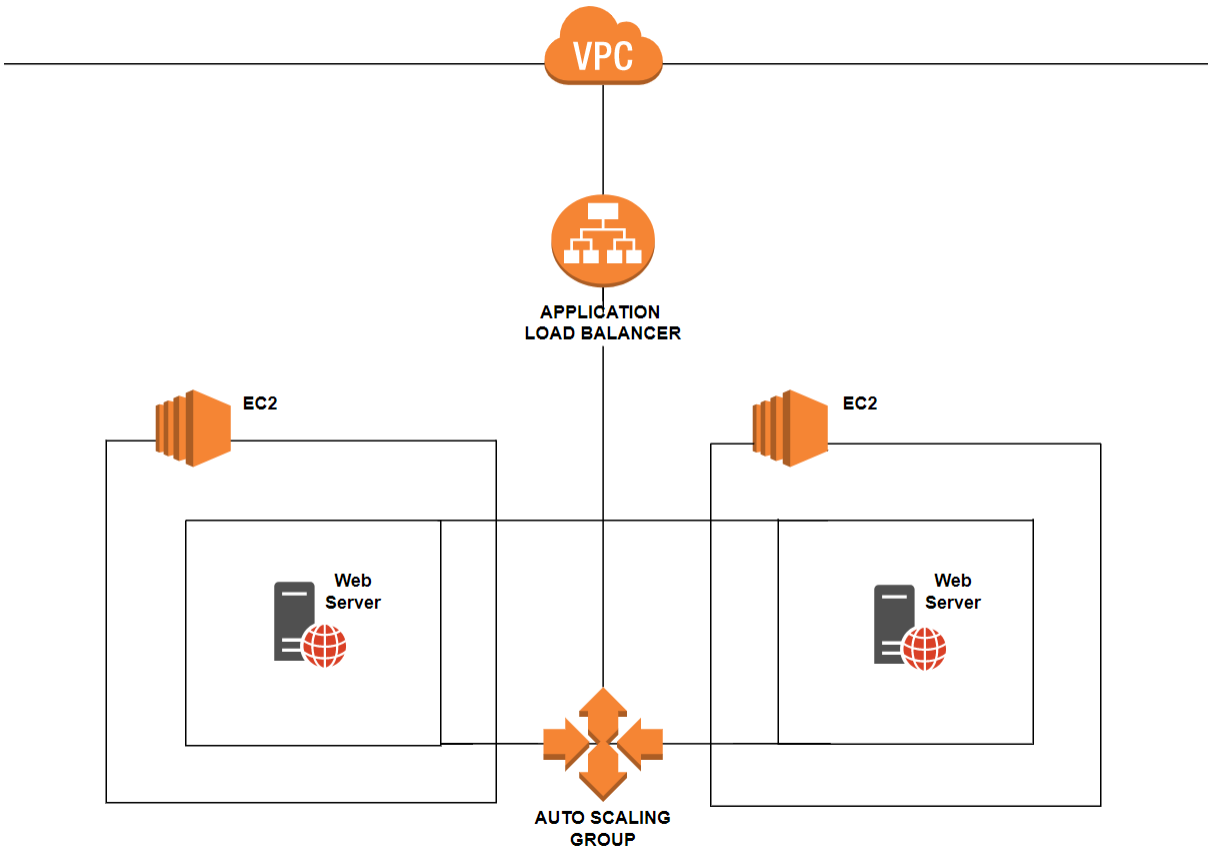

Figure 1: Architecture Diagram

1. Initial Setup

1.1 Amazon Machine Image (AMI)

A custom AMI was set up with the name 'Master web server'. The setting up of a custom AMI is required as an AMI is needed to launch an EC2 instance. This custom AMI is to be used for the master instance and auto-scaling. Once created, the image took a few minutes to become available.

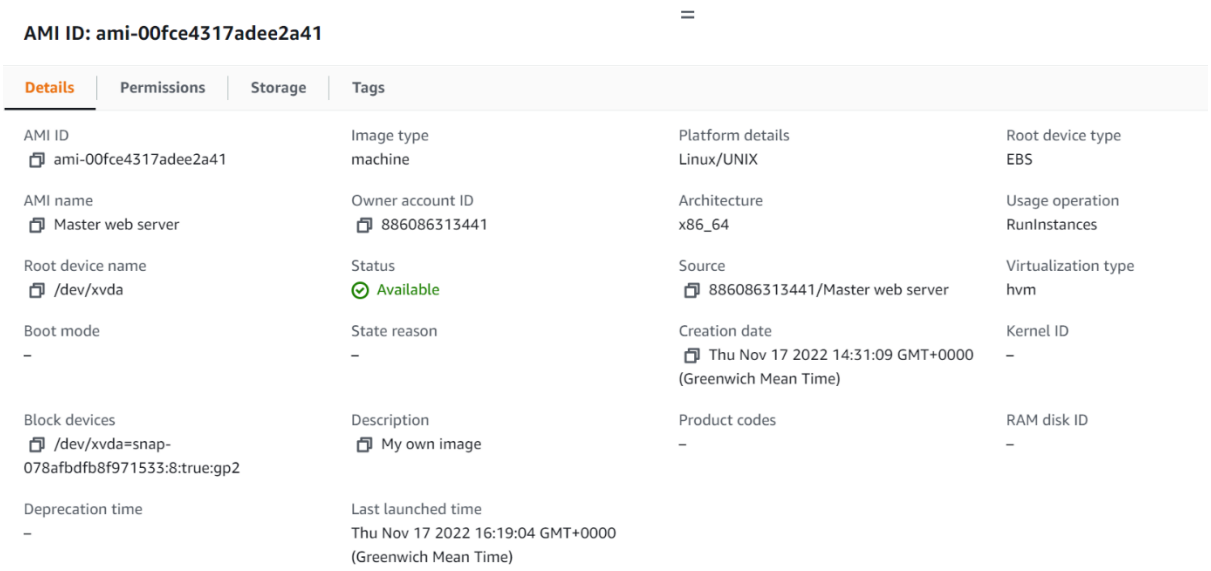

Figure 2: Custom AMI

1.2 Virtual Private Cloud (VPC)

A custom VPC was created under the name 'vpc-stephenpower'. The VPC contained 4 subnets which would be used for deploying the application. The subnets were: private1-us-east-1a, private2-us-east-1b, public1-us-east-1a, public2-us-east-1b. Appropriate security groups were also set up in the VPC to allow SSH and HTTP traffic.

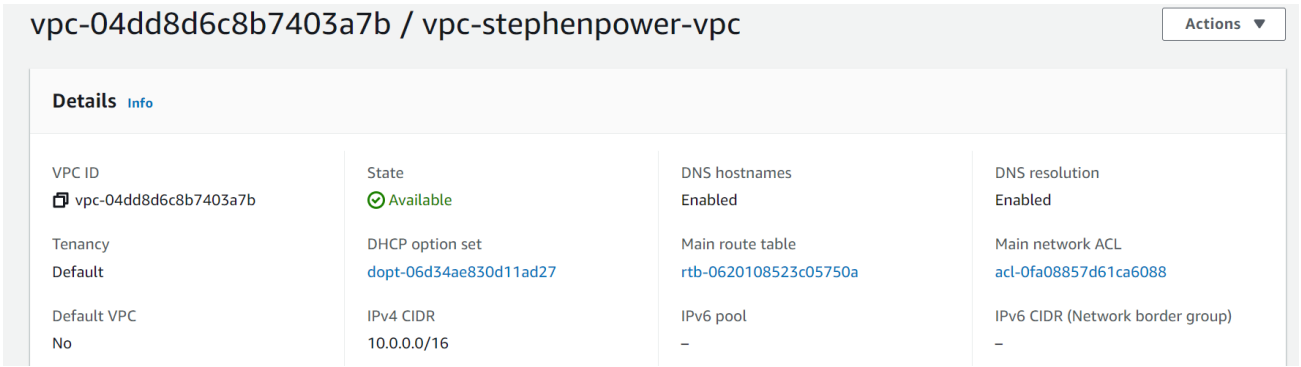

Figure 3: Custom VPC

Name	Subnet ID	State	VPC
vpc-stephenpower-subnet-private2-us-east-1b	subnet-03fcbca6eb73a17f5	Available	vpc-04dd8d6c8b7403a7b | vp...
vpc-stephenpower-subnet-private1-us-east-1a	subnet-0de6e9052b57752b4	Available	vpc-04dd8d6c8b7403a7b | vp...
vpc-stephenpower-subnet-public1-us-east-1a	subnet-019177157ec3cd488	Available	vpc-04dd8d6c8b7403a7b | vp...
vpc-stephenpower-subnet-public2-us-east-1b	subnet-04c4be72158657be3	Available	vpc-04dd8d6c8b7403a7b | vp...

Figure 4: VPC - Subnets

Security group ID	Security group name	VPC ID	Description
sg-071be317e5973d669	launch-wizard-2	vpc-04dd8d6c8b7403a7b	launch-wizard-2 create...
sg-0ddf0d6c0ba26befc	default	vpc-04dd8d6c8b7403a7b	default VPC security gr...
sg-053bdc408657a1aff	WebServerSG	vpc-04dd8d6c8b7403a7b	For the web servers in ...
sg-0827bc18c00966472	DBServerSG	vpc-04dd8d6c8b7403a7b	For the database serve...

Figure 5: VPC - Security Groups

1.3 Load Balancer

An application load balancer was set up with the name ‘load-balancer-sp’. The load balancer was mapped with the custom VPC so the public subnets could be used. A default target group was set up in the custom VPC under the name ‘target-group-sp’ and was selected as the designated target group for the load balancer.

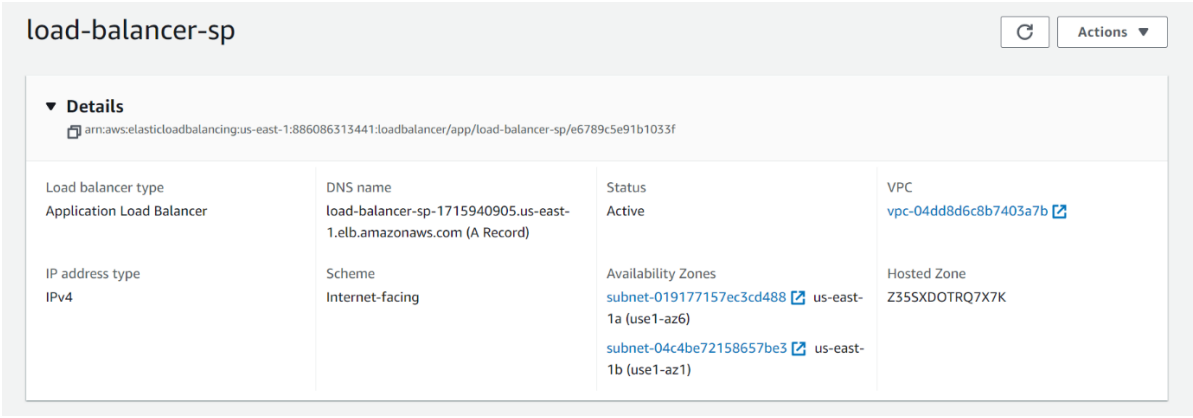

Figure 6: Load Balancer

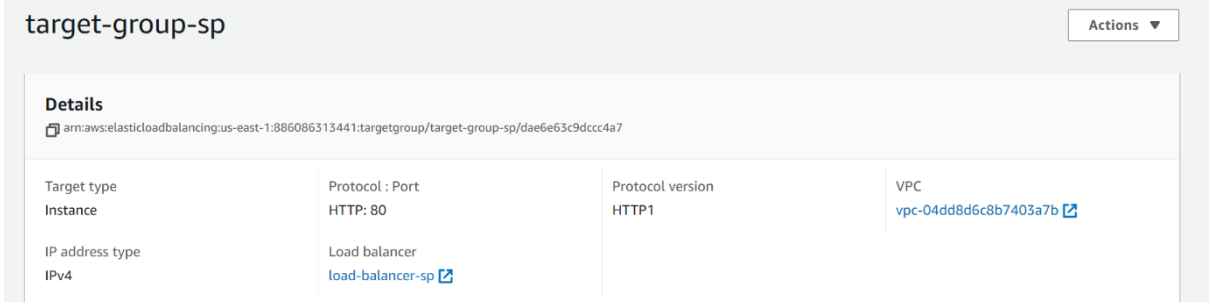

Figure 7: Target Group

1.4 Auto-Scaling

A launch template was configured with the custom AMI. The launch template was then used to create an auto-scaling group under the name 'auto-scaling-sp'. The group size was configured to have a minimum capacity of 1 and a maximum of 2. The public subnets chosen for the load balancer were also configured for the auto-scaling group.

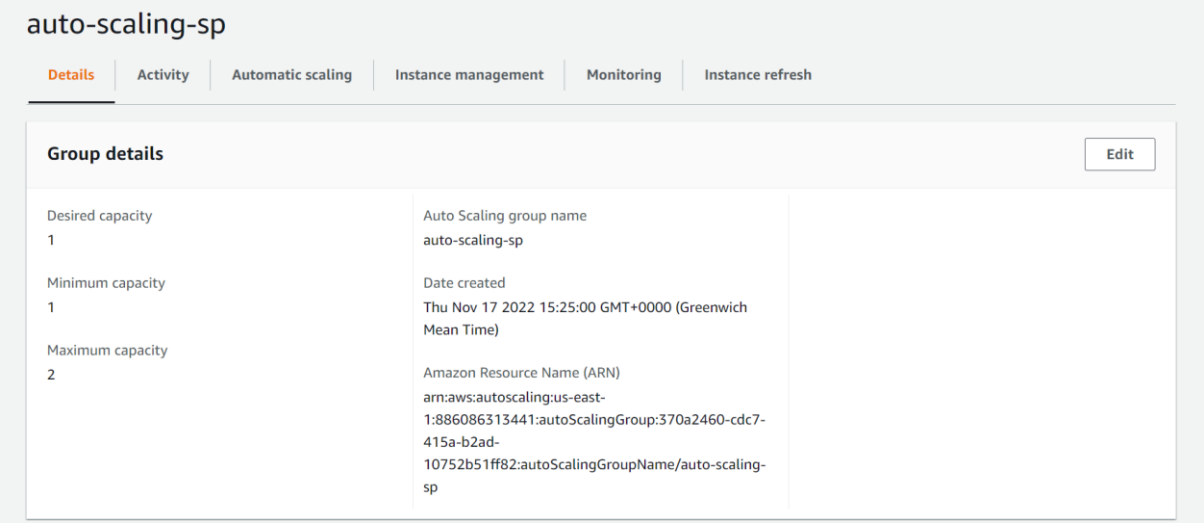

Figure 8: Auto Scaling Group

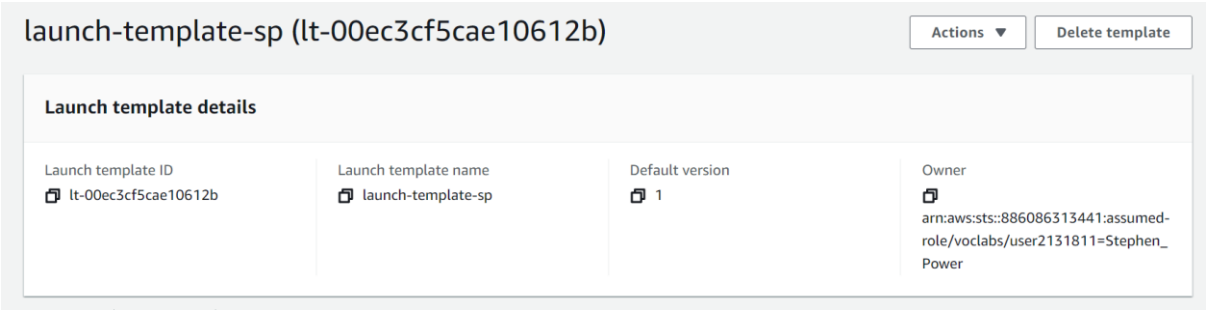

Figure 9: Launch Template

1.5 Scaling Policies

Two dynamic scaling policies were set up for the auto-scaling group:

1. Scale in on low CPU
2. Scale out on high CPU

Both policies used simple scaling based on CloudWatch alarms. The alarms were triggered using the CPU utilisation metric. If the CPU utilisation is greater than 50, the high CPU alarm will trigger and the 'Scale out on high CPU' policy will activate. Conversely, if the CPU utilisation is equal to or less than 50, the low CPU alarm will activate the 'Scale in on low CPU' policy. These scaling policies were chosen as when CPU utilisation is over 50, more resources are needed.

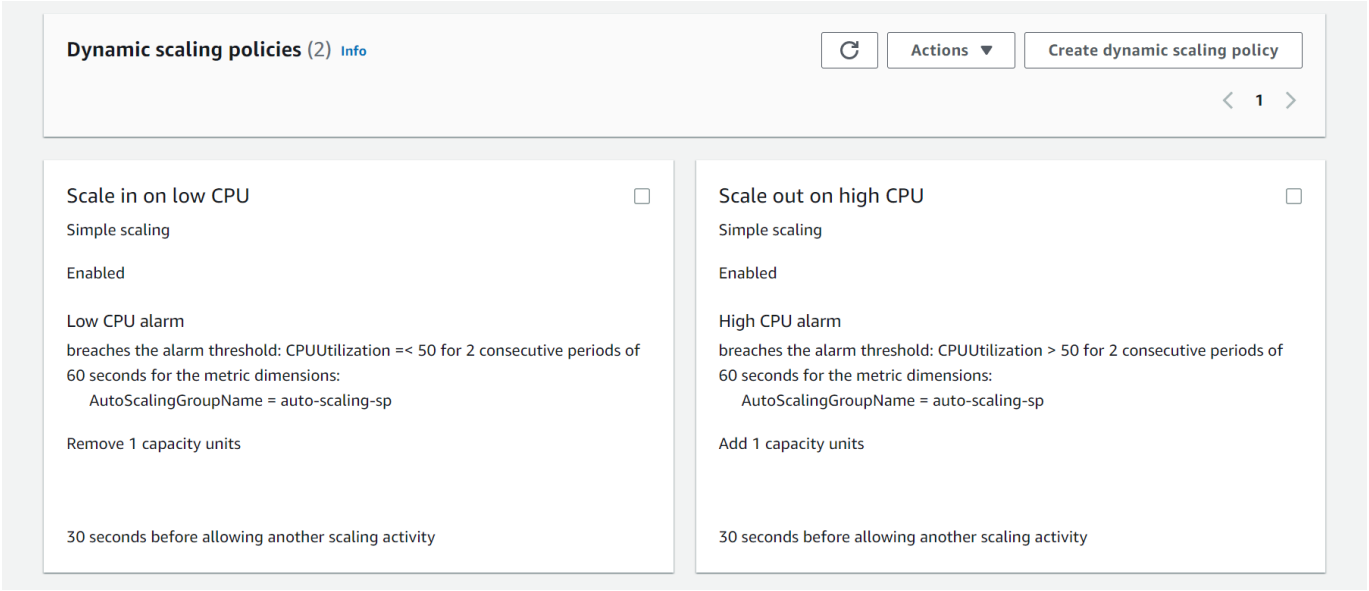

Figure 10: Scaling Policies

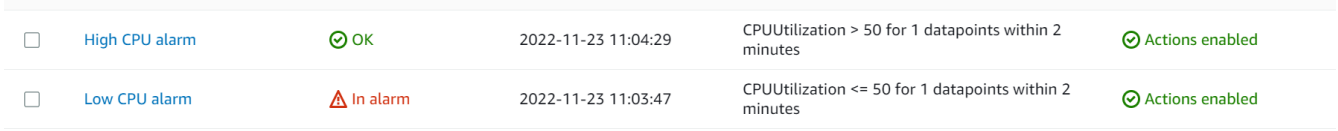

Figure 11: CloudWatch Alarms

2. Deployment of Application

2.1 Web Application

A web application was launched on completion of the initial setup. The web application could be viewed by copying the DNS name of the load balancer and inputting it into a web browser search. The web application simply contained the ID of the EC2 instance. The application was dynamic as if the instance changed, the application would also change to display the current instance ID.

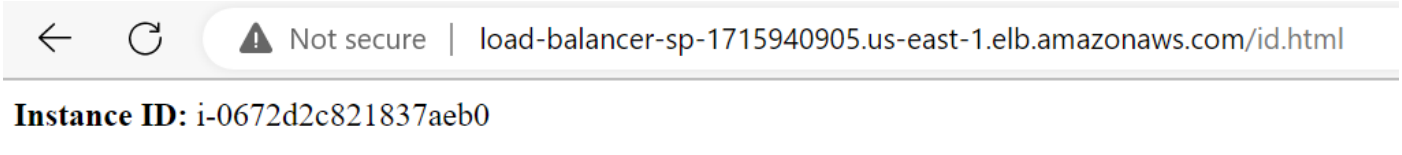

Figure 12: Web Application

2.2 Distributed Load

A simple infinite while loop (while true; do x=0; done) was placed on an instance through an SSH connection to show that the load is not just reliant on one server. Once the loop was initiated, the load balancer scaled out and another server was automatically set up. Both servers could be seen running using the 'top' command in the SSH connection, as well as displaying both instance id's on the web application.

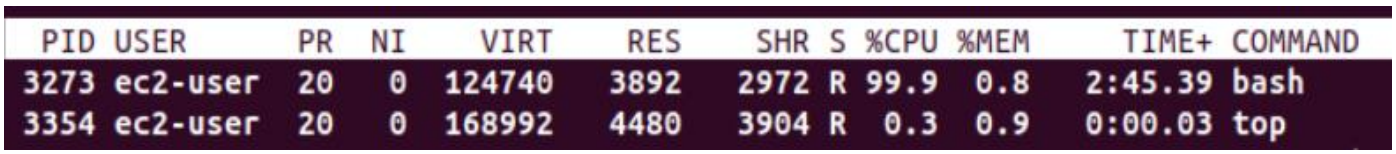

Figure 13: Multiple Instances

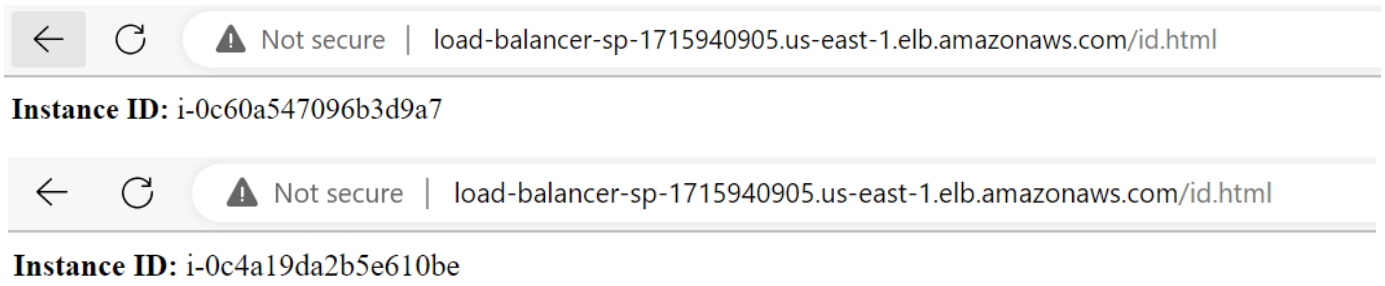

Figure 14: Web Page (Multiple Instances)

2.3 Custom Metrics

A bash script was used to monitor custom metrics on the EC2 instances. The script monitored the following metrics: percentage of used memory, the number of total and port 80 TCP connections, the percentage of I/O wait time, the number of HTTPD processes running and the total number of processes running. A cron job was created in order to push these custom metrics to CloudWatch.

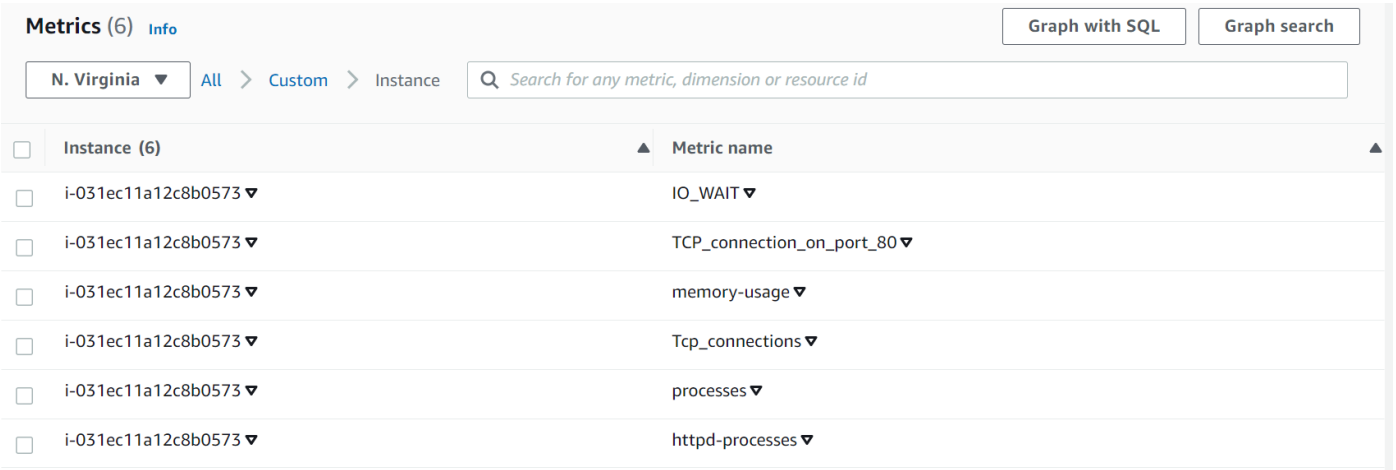

Figure 15: Custom Metrics

```
#!/bin/bash
INSTANCE_ID=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)
USEDMEMORY=$(free -m | awk 'NR==2{printf "%.2f\t", $3*100/$2 }')
TCP_CONN=$(netstat -an | wc -l)
TCP_CONN_PORT_80=$(netstat -an | grep 80 | wc -l)
IO_WAIT=$(iostat | awk 'NR==4 {print $4}')
PROCESSES=$(expr $(ps -A | grep -c .) - 1)
HTTPD_PROCESSES=$(ps -A | grep -c httpd)

aws cloudwatch put-metric-data --metric-name memory-usage --dimensions Instance=$INSTANCE_ID --namespace "Custom" --value $USEDMEMORY
aws cloudwatch put-metric-data --metric-name Tcp_connections --dimensions Instance=$INSTANCE_ID --namespace "Custom" --value $TCP_CONN
aws cloudwatch put-metric-data --metric-name TCP_connection_on_port_80 --dimensions Instance=$INSTANCE_ID --namespace "Custom" --value $TCP_CONN_PORT_80
aws cloudwatch put-metric-data --metric-name IO_WAIT --dimensions Instance=$INSTANCE_ID --namespace "Custom" --value $IO_WAIT
aws cloudwatch put-metric-data --metric-name processes --dimensions Instance=$INSTANCE_ID --namespace "Custom" --value $PROCESSES
aws cloudwatch put-metric-data --metric-name httpd-processes --dimensions Instance=$INSTANCE_ID --namespace "Custom" --value $HTTPD_PROCESSES
```

Figure 16: Monitoring Script

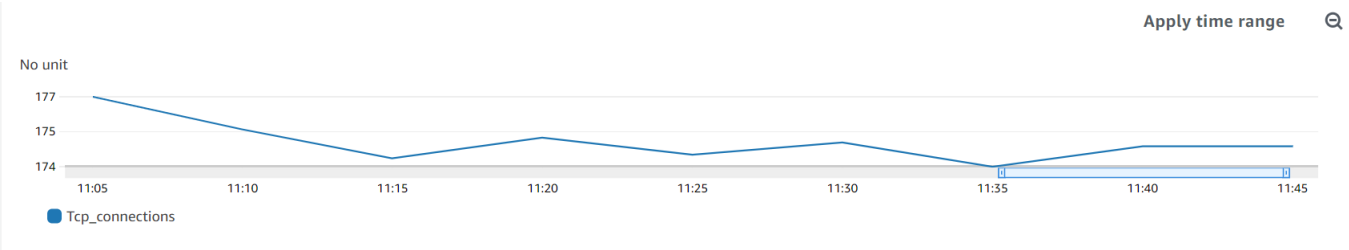

Figure 17: TCP Connections Monitoring (CloudWatch)

3. Additional Functionality

3.1 Secure Load Balancer

The load balancer was initially set up to accept HTTP traffic (port 80). Additionally, a self-signed certificate was created for the load balancer using the AWS Certificate Manager so it could now also accept HTTPS traffic (port 443). The self-signed certificate was generated using OpenSSL.

Figure 18: Secure Load Balancer

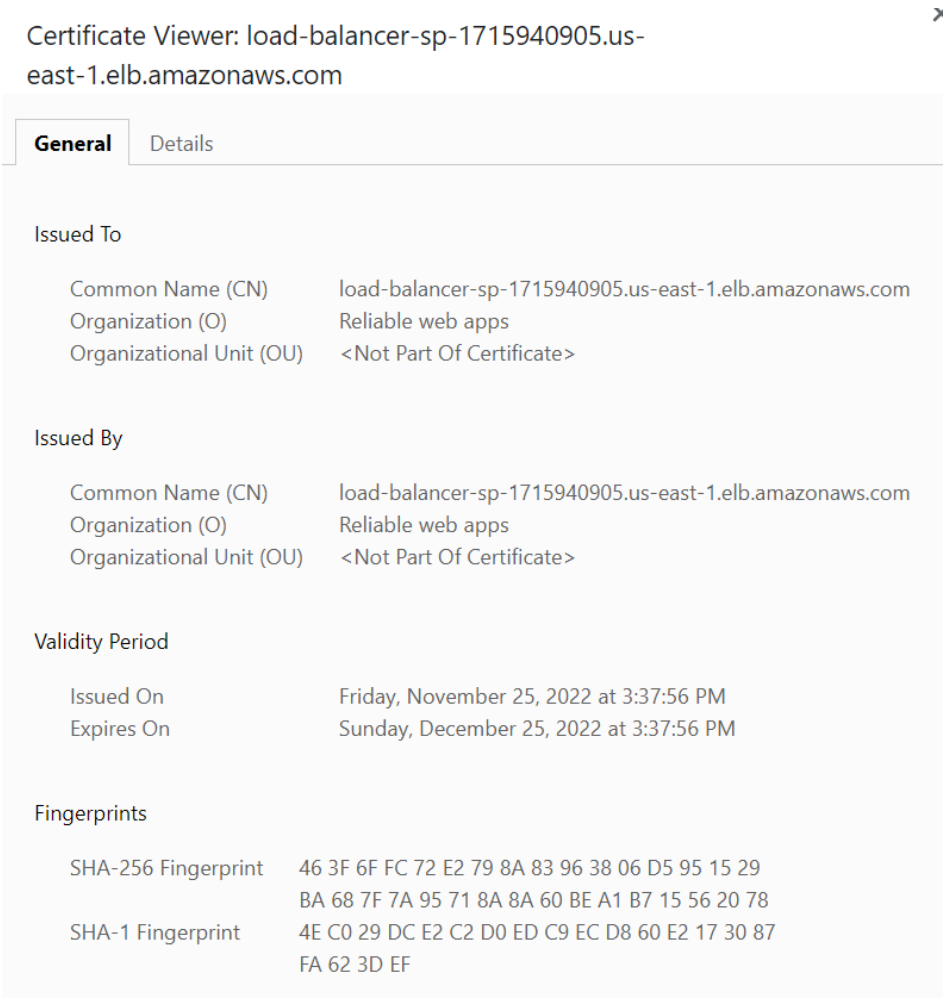

Figure 19: Self-Signed Certificate for the Load Balancer

3.2 Simple Queue Service (SQS)

A message queuing service was set up under the name ‘SQS-SP’. Messages could be sent and received in a queue which would be used for auto-scaling. Two CloudWatch Alarms were set up:

1. High messages visible alarm
2. Low messages visible alarm

If the amount of messages in the queue was greater than 10, the high messages visible alarm would activate and if the amount of messages visible in the queue were equal to 10 or less, the low messages visible alarm would activate. Two dynamic scaling policies were set up using these alarms to carry out the auto-scaling. The auto-scaling group would scale out if the high messages visible alarm was active and it would scale in if the low messages visible alarm was active.

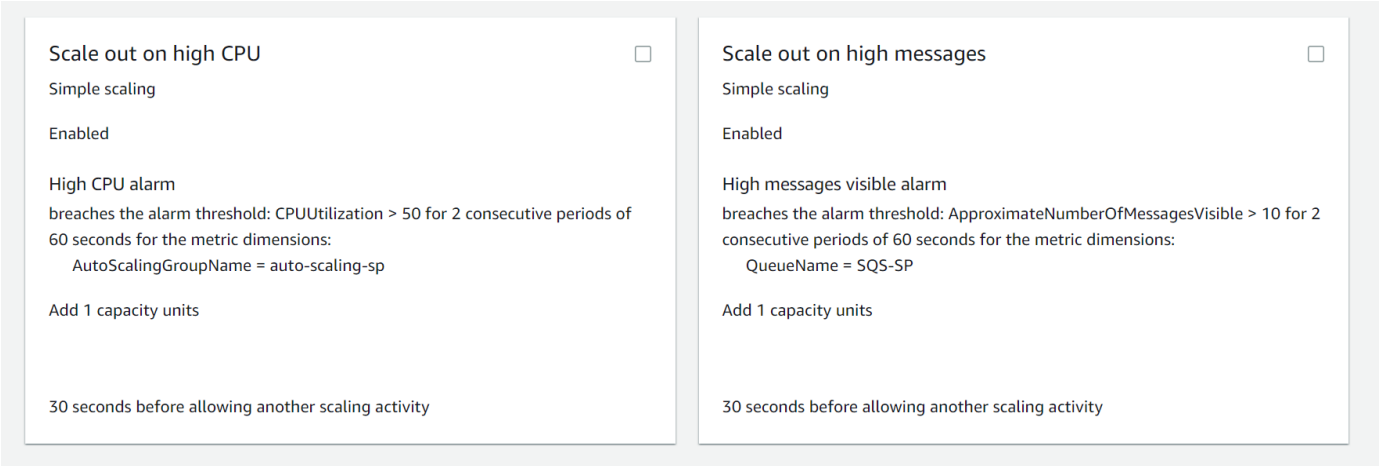

Figure 20: SQS Scaling Policies

Name	State	Last state update	Conditions
Low messages visible alarm	OK	2022-11-25 20:05:21	ApproximateNumberOfMessagesVisible <= 10 for 1 datapoints within 2 minutes
High messages visible alarm	In alarm	2022-11-25 20:04:19	ApproximateNumberOfMessagesVisible > 10 for 1 datapoints within 2 minutes

Figure 21: SQS CloudWatch Alarms

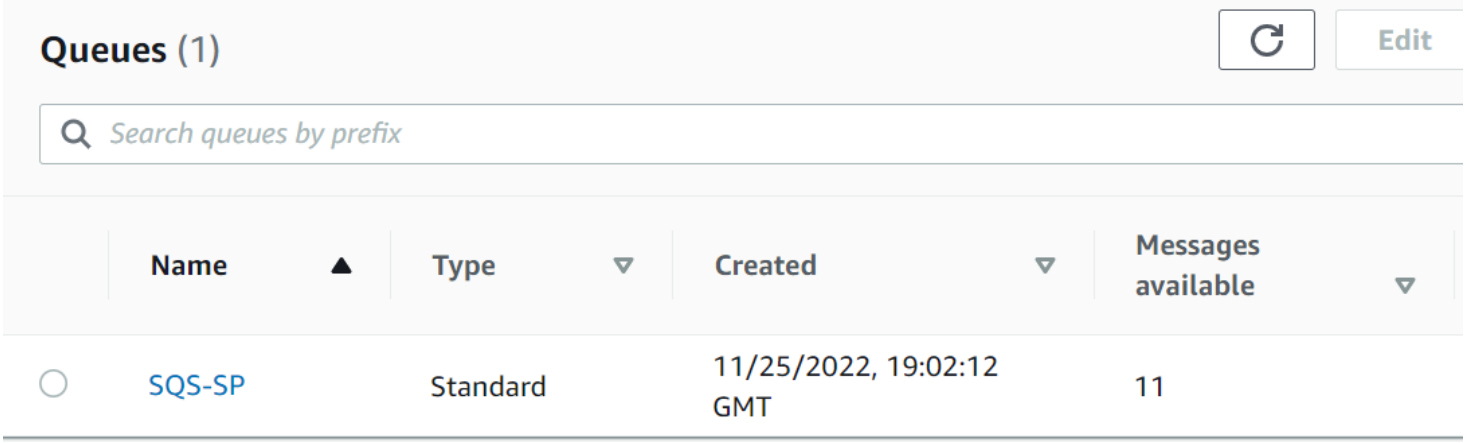

Figure 22: SQS-SP (11 Messages Avalable)

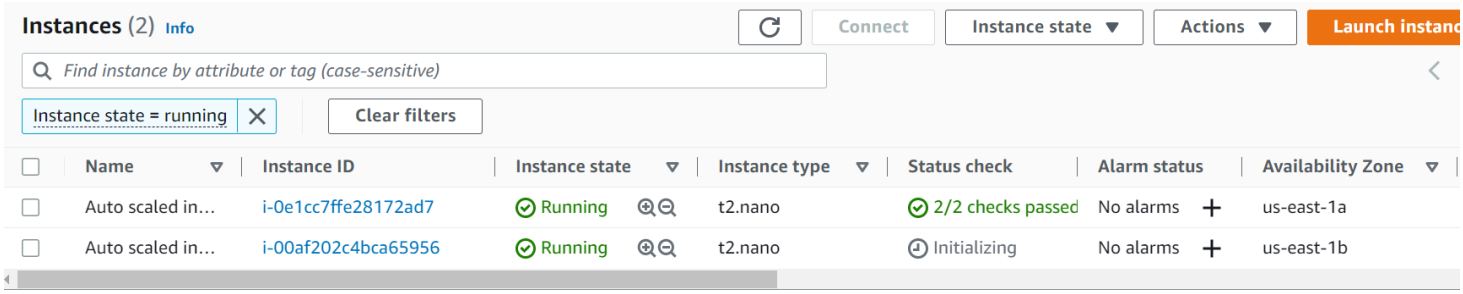

Figure 23: SQS Scaling Out

4. Conclusion

It was clear to see the benefits of load balancing and auto-scaling through setting these up for a web application using AWS. AWS elastic load balancing distributes traffic across multiple EC2 instances, in multiple availability zones. Traffic is routed to instances based on their health status which is determined by health checks. More memory, disk space, and faster CPU speeds is a result of balancing an application on multiple instances. The goal of load balancing is to increase performance.

A group of EC2 instances can be grouped together in an auto-scaling group. The minimum, maximum, and desired amount of instances in the group can be specified. The auto-scaling group is used with the load balancer to scale in or scale out when needed. Scaling policies that use CloudWatch alarms can be set up to automatically determine when a scaling group should scale in or scale out. Auto-scaling policies were set up for EC2 and SQS metrics in this assignment which gave an insight into how these services work.

Working with different AWS services in this assignment such as VPC, EC2, CloudWatch, SQS, and Certificate Manager has resulted in a strong learning experience. Learning the theory about these services at first, but then actually working with these services practically for the assignment really helped in fully understanding how these services function and what they are used for.