LOG8415
Advanced Concepts of Cloud Computing

Cluster Benchmarking using EC2 Virtual Machines and Elastic Load Balancer (ELB)

Lab 1

Fall 2023

By:

Samy Cheklat – 1937812
Cassy Charles – 1947025
Ming Xiao Yuan - 1949477

Presented to:
Vahid Majdinasab

Saturday, October 7th 2023

Our Github link for the laboratory is: https://github.com/ikrash3d/TP1-LOG8415-TF

1 Introduction

Amazon Web Services (AWS) stands out as a leading provider of Infrastructure as a Service (IaaS) in the cloud computing domain. Among their offerings, Amazon Elastic Compute Cloud (Amazon EC2) is a web-based service that empowers users to execute application programs within the AWS public cloud infrastructure. This service enables developers to quickly provision virtual machines (VMs) across AWS data centers globally, delivering essential compute capacity for various IT projects and cloud workloads.

The primary objectives of this laboratory assignment encompass the creation of two clusters, each consisting of five EC2 instances, and the implementation of an Application Load Balancer (ALB) for each cluster. Within each individual EC2 instance, a Flask application is to be deployed, and Docker will be employed to conduct test scenarios locally. The ensuing report will comprehensively detail the experimental procedure, including the deployment of the Flask Application on each EC2 instance, the configuration of the cluster setup with ALB, the presentation of benchmark results, and a conclusive description of how to execute the script.

2 Deployment of Flask application

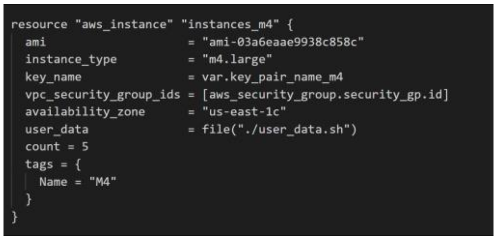
To simplify the deployment of our application, we've opted to utilize Terraform. This tool, categorized as infrastructure as code, enables us to articulate cloud resources through configuration files. Initially, we ensured that our Flask application and its dependencies were enumerated in a requirements.txt file with their respective versions. With the "pip install" command, it's possible to fetch these dependencies and install them.

To facilitate this capability, we employed a Dockerfile. Upon the successful creation of our EC2 virtual machines, where we granted traffic access on port 80 within the security group, we proceeded to connect to the instance for deploying Flask. Upon launching our Docker image on the instance, a command facilitates the automatic execution of our app.py program, enabling its exposure on port 5000.

3 Cluster setup using Application Load Balancer

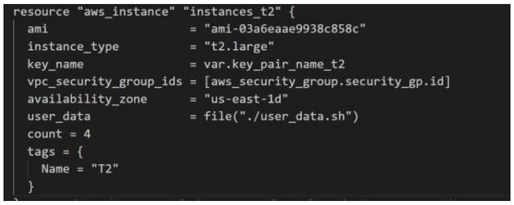
We established two distinct sets of clusters for our Application Load Balancer, denoted as cluster1 and cluster2. The initial cluster is configured to connect with 5 m4.large instances, while the latter corresponds to 4 t2.large instances. The creation of these instance clusters provides a seamless means of distributing incoming traffic based on whether the request was directed to /cluster1 or /cluster2.We specify an ALB instance for the Application Load Balancer, configuring it as non-internal to ensure its exposure to the internet.

Following this, a listener is declared to actively await incoming HTTP requests. Rules were defined for each listener.In configuring the ALB for a particular cluster, the chosen specifications involve opting for an Internet-facing and IPv4 setup, serving as the gateway for client requests directed to the ALB. This type of load balancer, facing the internet, directs client requests to designated targets. Following this, the three availability zones associated with the specific cluster are chosen, and the listener is configured to operate on port 80. Subsequently, a target group is incorporated into the ALB, comprising the 5 EC2 instances belonging to that cluster. These steps are then replicated for the second cluster.



```
resource "aws_instance" "instances_m4" {
  ami                    = "ami-03a6eaae9938c858c"
  instance_type          = "m4.large"
  key_name               = var.key_pair_name_m4
  vpc_security_group_ids = [aws_security_group.security_gp.id]
  availability_zone      = "us-east-1c"
  user_data              = file("./user_data.sh")
  count = 5
  tags = {
    Name = "M4"
  }
}
```

Figure 1: Setup for our instances of type m4.large



```
resource "aws_instance" "instances_t2" {
  ami                    = "ami-03a6eaae9938c858c"
  instance_type          = "t2.large"
  key_name               = var.key_pair_name_t2
  vpc_security_group_ids = [aws_security_group.security_gp.id]
  availability_zone      = "us-east-1d"
  user_data              = file("./user_data.sh")
  count = 4
  tags = {
    Name = "T2"
  }
```

Figure 2: Setup for our instances of type t2.large

4 Benchmark results

Two threads (cluster_1 and cluster_2) are created to simulate concurrent generation of HTTP requests.

	Cluster 1	Cluster 2
Instance Type	m4.large	t2.large

Table 1: Specifications of the two clusters

According to the number per request bar chart, M4 instances generate 1000 requests to Cluster 1. T2 instances generates 500 initial requests to Cluster 2, sleeps, and then generates an additional 1000 requests. Requests are sent to specific endpoints (/cluster1 and /cluster2) indicating different clusters through the load balancer. The intentional sleep in cluster_2 introduces a change in load over time, allowing observation of the system's ability to adapt to varying workloads.

The bar chart representing the average response time per target group indicates a marginal speed advantage, in seconds, for M4 instances over T2 instances. However, the disparity is minimal, signifying a balanced load distribution between the two target groups. This observation suggests that neither target group experiences significantly higher congestion than the other, affirming the effectiveness of the load balancing.

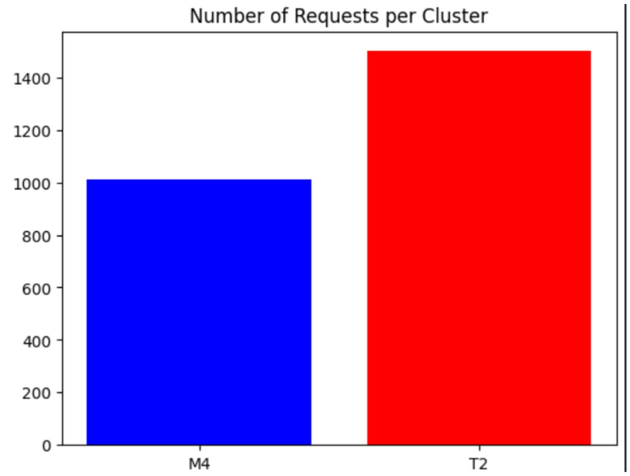


Figure 3: Number of requests per cluster

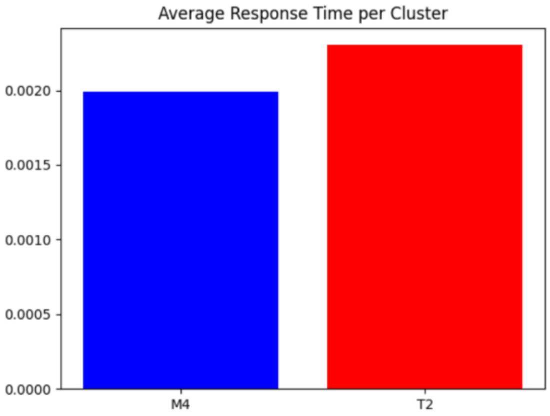


Figure 4: Average response time per cluster

Moreover, a table was generated to exhibit the metrics of the load balancer, encompassing statistics like the active connection count, total processed bytes and total request count within both M4 and T2 instances.

Total number of requests	2500.0
Total bytes processed	1269000.0
Average number of connections	28.0
Number of healthy instances M4	5.0
Number of healthy instances T2	4.0

Table 2: Load balancer metrics

Examining the metrics table of the load balancer reveals that the ALB effectively handles information for 2500 requests, and all our instances seem to operate effectively with 1269000 for the number of total bytes processed and 25 connections.

5 Instructions to run the code

Prerequisite:

- Clone the source code on your local machine.
- Ensure that **Docker** is installed on your local machine.
- Ensure that **Terraform** is installed on your local machine.
- In the folder **/infrastructure** within the cloned project, generate an SSH Key. Name it "**my_terraform_key**". When creating the key **DO NOT GIVE IT A PASSPHRASE**.
- To generate an SSH Key use the following command: **ssh-keygen -t rsa -b 4096 -C "your_email@example.com" -f my_terraform_key**

- In the folder /infrastructure, you should have a file named “**my_terraform_key**” and “**my_terraform_key.pub**”.

To execute the code:

- Open **Docker Desktop**.

- Launch a bash terminal in the folder **/scripts** within the cloned project.

- Launch the “**run.sh**” script using the following command: **bash run.sh**

- Follow the instructions provided by the script.

- Make yourself a coffee while the script runs (most important step).

- The results are displayed in **/benchmark/metrics** within the cloned project.

6 Annexe

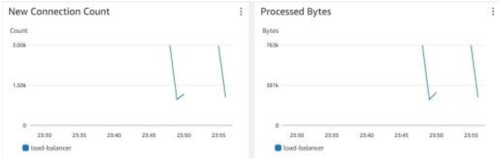


Figure 5: New connection count and bytes processed

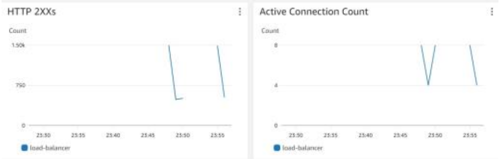


Figure 6: HTTP 2XXs and active connection count

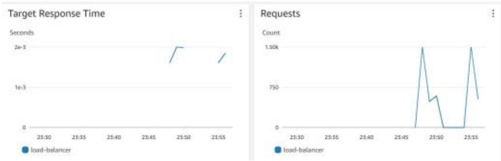


Figure 7: Target response time and requests