Polytechnique Montréal

LOG8415 : Lab 1

Advanced Concepts in Cloud Computing

Selecting VM instances in the Cloud through benchmarking

Authors
Anis Zouatene (1963304)
Aleksandar Stijelja (1959772)
Amin Ghadesi (2121658)
Reza Rouhghalandari (2153395)

October 18, 2022

1 Abstract

Not all instances of virtual machines are the same. They provide similar usability but have different capabilities and characteristics. As such, even though two instances look similar, they will not provide the same performance. Instances are then divided into multiple categories to best suit what the user wants for his usage. To really make sure an instance is right for us, we would need to put her into a benchmark test. In this lab, we'll do exactly that for 2 types of instances that will be available on AWS EC2. AmazonWeb Services (AWS) is a leading infrastructure as a service Cloud provider. One of their products, the Amazon Elastic Compute Cloud (Amazon EC2) is a web-based service that allows customers to run application programs in the Amazon Web Services (AWS) public cloud. With EC2, we will be able to create those virtual machines.

Keywords: AWS, AWS EC2, Benchmark, VM Performance, Cloud Application

2 Introduction

As said previously, in this lab we will be focusing the majority of it in the Amazon Elastic Compute Cloud (Amazon EC2) to create our virtual machines. Since a lot of instances meet similar requirements, finding the right type might be challenging to a user. The solution to that would be to try all the instances we want to use, benchmark them by making them go through a load and finally we get the results to make our final decision on which one to take. [1]

Thus, the goal of this lab is to benchmark 2 different types of EC2 instances, being M4.large and T2.large. The total number of EC2 instances would be 9 in total and on top of it we will have to create 2 clusters in the target groups that will both contain one specific type of the 2 said previously. Of course, for each of these 2 clusters we would need an Application Load Balancer (ALB). Thereafter, in each instance we will need to deploy a Flask application, do our tests/benchmark then report the results.

In this paper, we talk about our methodology which will include virtual machines, the elastic load balancer, Flask and how we will conduct our analysis on our VM.

3 Methodology

In this section, we will go over the different sections that we worked on to be able to benchmark properly our 9 different instances. We will first go over our virtual machines (instances), our Elastic load balancer, Flask and our benchmark analysis.

3.1 Virtual Machines

As said previously, for our virtual machines we will benchmark 9 instances of 2 different types. The types being T2.large and M4.large, who each have different characteristics. As for their number, it will be 4 T2.large instances and 5 M4.large instances.

As for their characteristics, T2.large instances have double the memory in GiB for a total of 16GiB, whereas M4.large have only 8GiB. Another slight difference is in their CPU Clock Speed, where the M4.large have a 2.4Ghz and the T2.large have a 2.3Ghz.

3.2 Elastic Load Balancer

In our benchmark we will have to use the load balance that we created along with the instances. A load balancer is a load-balancing service for Amazon Web Services (AWS) deployments. ELB automatically distributes incoming application traffic and scales resources to meet traffic demands. So when it gets its user's request, it then routes its requests to their targets according to rules setup, usually forwarding them to a specific target group (clusters).

One other aspect of the load balancer is that the user can configure health checks, which monitors the health of the compute resources, so that the load balancer sends requests only to the healthy ones. As such, if a route/instance becomes unhealthy, it will stop its traffic until it becomes healthy again. In our case, our load balancer will be of the type application.

With the type being application, our load balancer will route the traffic to two distinct routes, being cluster1 and cluster2.

3.3 Flask

Flask is a web framework, it's a Python module that lets you develop web applications easily. It has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features. It offers a high compatibility with the most recent technologies, high scalability for simple applications and its database integration is simple.

All our instances will have Flask server installed and running in them listening on port 80 which have only one task of replying with a message containing the instance ID they are currently on. Since we will have to make numerous calls, the Flask server will be called multiple times to do the proper benchmark expected to determine the quality of the different type of instances. In addition, it will have 3 path routes to listen to. First being the default path "/" and the others being "/cluster1" and "/cluster2", since our instances will be divided into 2 clusters. Each path routes will return a slightly different message, but all containing the instance ID.

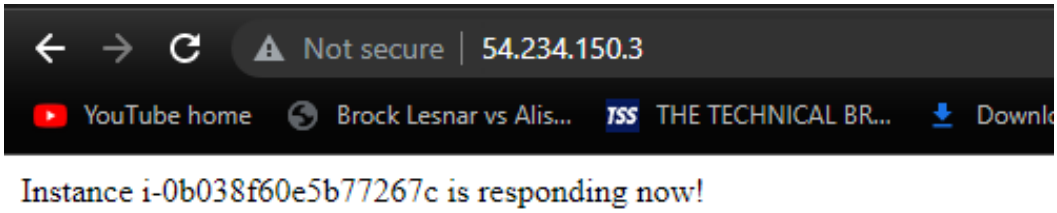


Figure 1: Default Flask message from an instance

3.4 Benchmark method

For the benchmark method, when we build our load balancer, we keep in a variable its DNS to be able to do the GET requests. These requests will be done in 2 different scenarios. The first one will be to GET 1000 requests from the ELB. The second one will be to GET 500 requests, sleep 1 minute and then GET 1000 requests. These 2 scenarios will be done via 2 threads, so 1 thread per scenario. For the clusters, the cluster1 will use both threads to be able to do both scenarios, as will the cluster2.

4 Benchmark Analysis

For this part First, we have to run the benchmark and then capture our desired metric from CloudWatch. So we describe them here.

4.1 Cloudwatch

Cloudwatch actually sends logs for services that you are consuming using logs streams and uses its dashboards to create reports on the performance of your service. it can also help you analyze the data point to understand where your application could actually break so as rightly mention here CW collects and monitors operational data in the forms of logs metrics and events and visualizes it using automated dashboards so you can get a unified view of your AWS resources application services that run in AWS.

4.2 Metric

Imagine you invest hundred dollars and you get 105 dollars. You have a profit of 5 dollars. When it comes to AWS if you have deployed service on t2.micro and the CPU reaches above 95 percent and you then try and make some changes and shoot down to 75 percent there will a considerable amount of boost in the performance that's called the performance metric. The way you measure and take a quantitative approach one a data point over a given period of time gives you a form of metrics based on which you can analyze the way your services are performing. AWS cloud watch is comprised of four pillars 2.

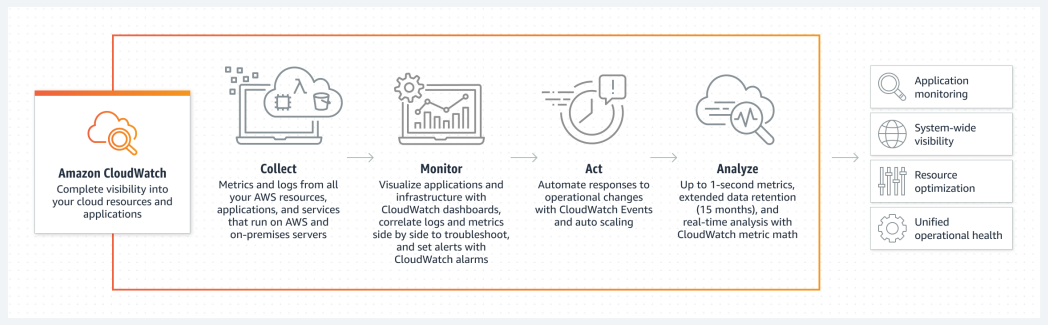


Figure 2: Image Retrieved from: https://aws.amazon.com/cloudwatch/

Name	Instance ID	Availability zone
cluster1a	i-05d4f4ea233b89f2d	us-east-1a
cluster1b	i-0e13ad0ed8951d3dc	us-east-1b
cluster1c	i-02ceee3bd54a8f70e	us-east-1c
cluster1d	i-0e5a28ccbd00342ba	us-east-1d
cluster1e	i-017eab7711a692d4e	us-east-1e
cluster2a	i-0b5230f77a4813eab	us-east-2a
cluster2b	i-0527ab13e0e76c52c	us-east-2b
cluster2c	i-04f73d8bf2d7136aa	us-east-2c
cluster2d	i-079625ab28844524c	us-east-2d

Table 1: Instances Specifications

4.3 Instances

After we run the main.py 9 instances are created 5 m4.large instances and 4 t2.large instances. you can see the instances attributes in 1.

After creating instances, the load balancer is created and then we create a target group for clusters 1 and 2 and next create listeners and rules for listeners on cluster1 and cluster2. you can find them in Table2. Now everything looks good. Instances are created, the target group is created, and all instances now belong to the target group 1 or 2. Also, the load balancer is configured and listeners and rules are created. so let's go to the final part and run the benchmark and analysis of the system. You can find some of the elb and terget1 and target2 Specifications in Table2. In target 1 and Target 2, you can find all **healthy** instances with their IDs, Name and port.

4.4 Benchmark

For this part based on the assignment, we have two functions for each scenario. In scenario1 we send 1000 get requests and in scenario2 we send 500 get requests and then wait for 60 sec and then send 1000 requests again. We repeat this for each cluster1 and then cluster2. Now we try to analyze the whole system.

4.5 Analysis

We try to find the most relative metric and analyze the system in this part. We divide this part into 3 parts. A first metric is applied in all instances.

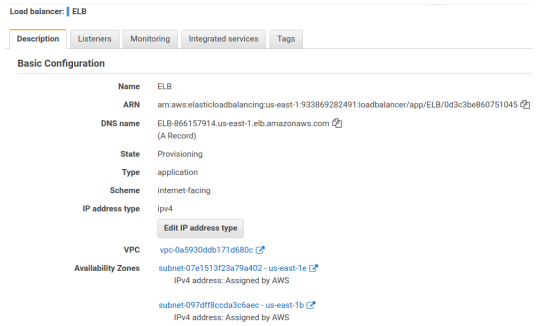
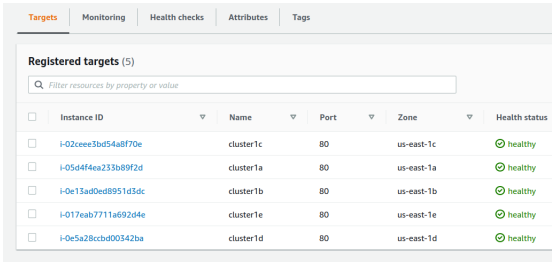
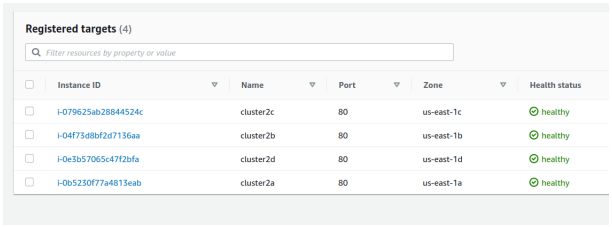
Load Balancer	
Target group 1	
Target group 2	

Table 2: ELB, target group1, target group 2 specification

the second part for that metrics is applied on clusters. and this part we can compare our two clusters and then the last part the parameters related to load balancer.

Metrics:

- All instances
- Two Clusters
- Load Balancer

4.5.1 All instances

In this section, we discuss significant metrics from all instances. We mean the metrics collect data directly from instances and not from load balancer or the whole target group. We talk about CPU utilization during the benchmark and network-in and network-out. Based on the Website[1] "T2 instances are general purpose and designed to provide moderate baseline performance and the capability to burst to significantly higher performance as required". You can find an overview of two instance types used in this homework, M4 and T2 in Table3. The main difference between a t2 instance and an m4 instance is the burstable nature of the t2 instance.

Based on AWS **CPUUtilization** references "This metric identifies the processing power required to run an application on a selected instance." So it reveals a good overview of allocated EC2 compute units for all instances from clusters 1 and 2. In Figure3 and 4 You can find it for all of them. These are the same figure but Figure4 is zoomed in to have a better look for analysis.
The blue, orange, green, red, and purple one from cluster 1, and the other are from cluster 2. The highest CPU capacity is used for the gray instance from cluster 2 but the second and third ones (orange green and brown) are from cluster 1.
For a more detailed review we capture the average CPU capacity of all instances for 15 minutes during the startup and benchmark in Figure 5. on Average CPU capacity for cluster1 will be 1.3 and for cluster 2 it will be 1.15. Clearly, cluster 1 (M4.large instance) is more efficient for what we pay for but Overall both clusters are overloaded for our needs. On Linux-on-demand, they are the almost same price (0.1 dollars per hour for M4 and 0.094 for T2)

[1]https://jayendrapatil.com/aws-ec2-instance-types/

	VPC only	EBS only	SS volume	Placement group	HVM only	Enhanced networking	price per hour (Linux-on-demand)	price per hour(Windows-on-demand)
M4	Yes	Yes		Yes	Yes	Yes	0.1	0.192
T2	Yes	Yes			Yes		0.094	0.122

Table 3: EC2 M4 and T2 comparison

but for windows-on-demand 0.192 dollars per hour for M4 and 0.122 per hour for T2.
In figures6 and 7 "NetworkPacketsIn" and "NetworkPacketsOut" metric. It is The number of packets received/send by the instance on all network interfaces. This metric is available for basic monitoring only (5-minute periods). In this graph, a maximum number of packets send and received by cluster 2 instances. the light purple, gray light green, and brown all from cluster2.
I wondered what's the difference between the metrics "NetworkIn" and "NetworkPacketsIn" and I found the answer:
NetworkIn: The number of bytes received by the instance on all network interfaces. This metric identifies the volume of incoming network traffic to a single instance.
NetworkPacketsIn: The number of packets received by the instance on all network interfaces. This metric identifies the volume of incoming traffic in terms of the number of packets on a single instance.

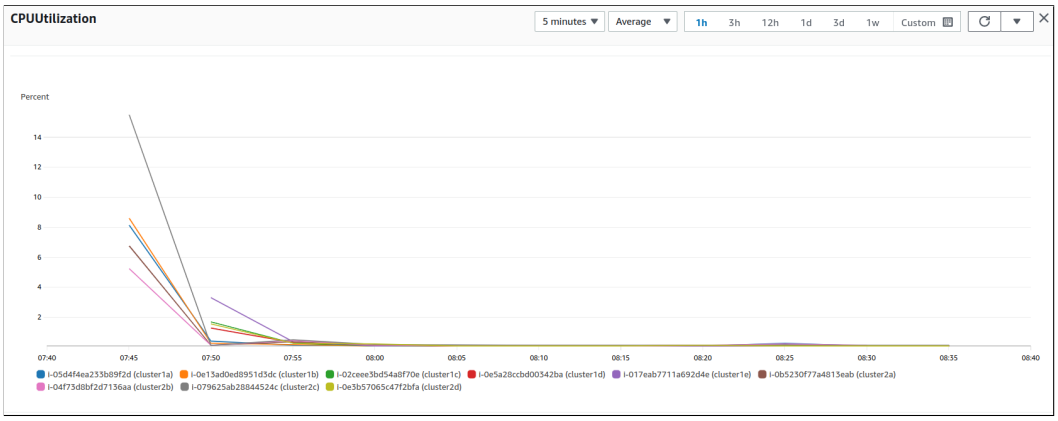


Figure 3: CPU utilization of all instances V1

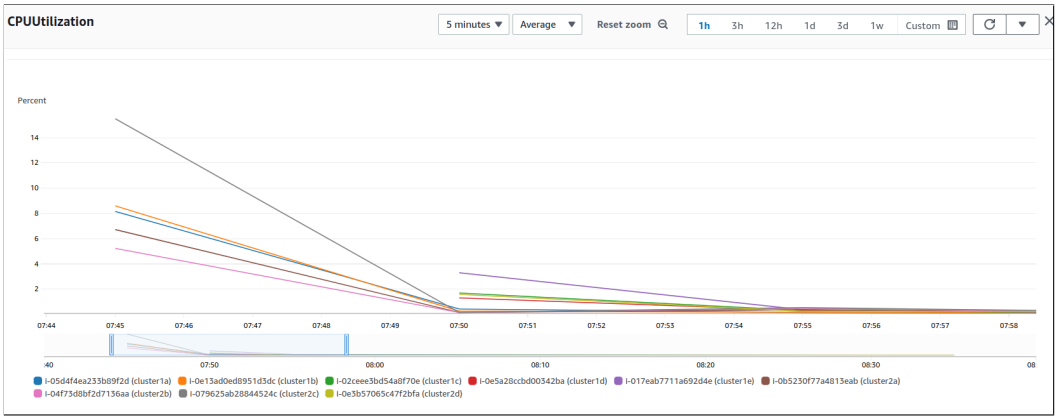


Figure 4: CPU utilization of all instances V2

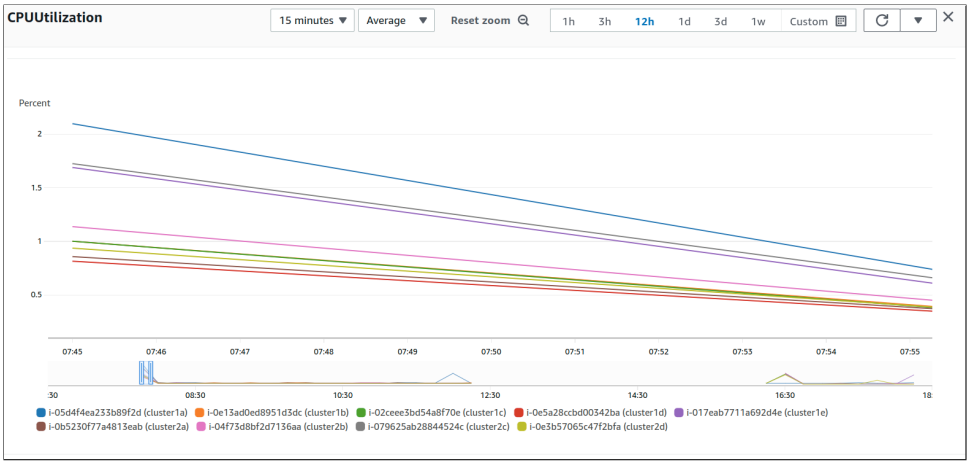


Figure 5: CPU utilization of all instances V3

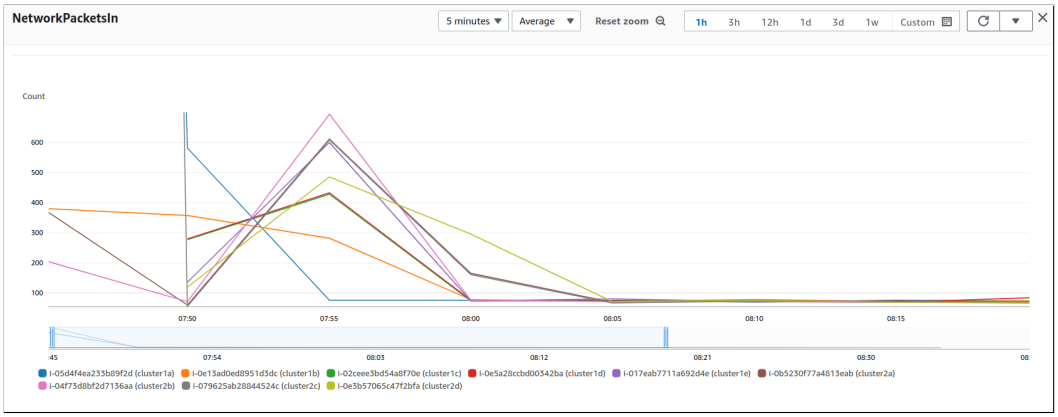


Figure 6: Network packets in of all instances

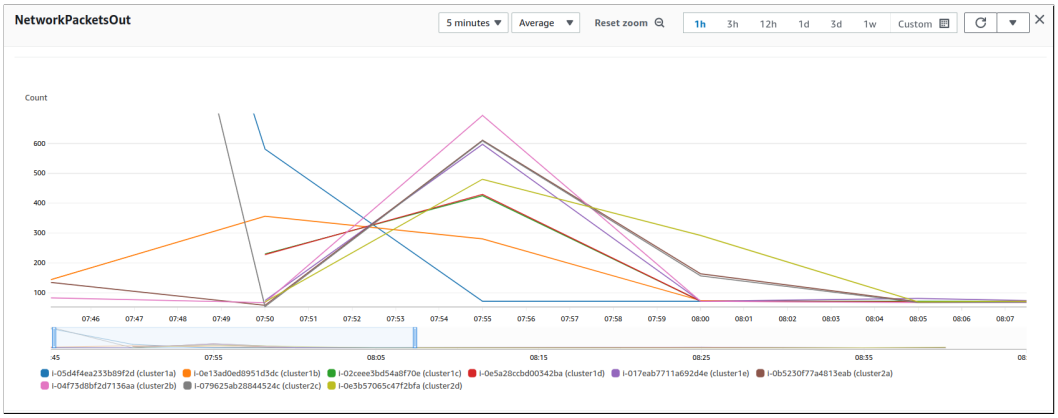


Figure 7: Network packets out of all instances

4.5.2 Clusters

In this section, we compare the significant metrics from two clusters. In Figure8 you can find the average response time of the target group1 and group 2 in one graph which means the time after the request leaves the load balancer until a response from the target is received.
They almost have the same response time. cluster 2 has a little bit higher time but it can be ignored. So we can claim here type of instance is not effective at least in target response time.

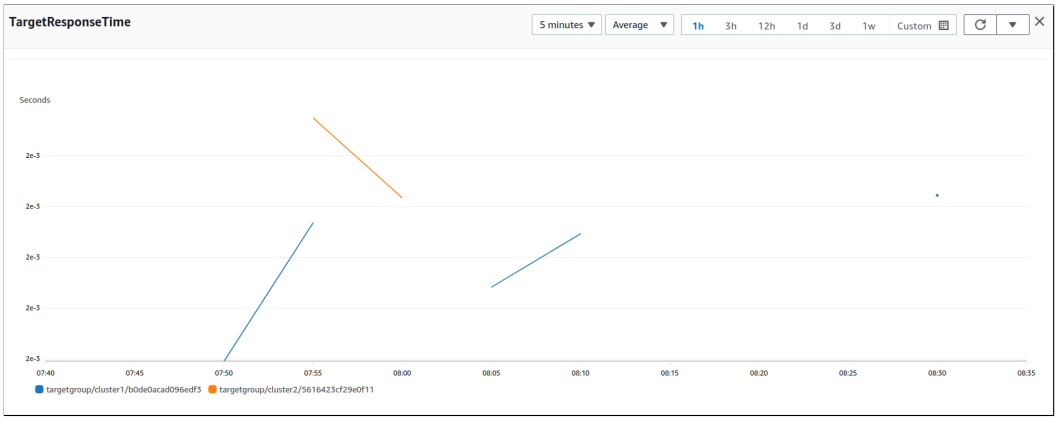


Figure 8: Target response time of Cluster1 and Cluster2

The request count and request count per target are analysed in graph 9 and 10. RequestCountPerTarget is for The average number of requests received by each target in a target group but requestcount is The number of requests processed over IPv4 and IPv6. This metric is only incremented for requests where the load balancer node was able to choose a target. In both one cluster 1 have higher request count. It means it handles it better.
Also in Graph11 metric "HTTPCode-Target-2XX-Count" defines The number of HTTP response codes generated by the targets. This does not include any response codes generated by the load balancer. Cluster 1 is clearly doing better. Generally, Cluster1 acts better in this section.

4.5.3 Load Balancer

In this section, we discuss metrics related to the loading balancer. Graph12 represents the request count through load balancer. The maxmimum value is

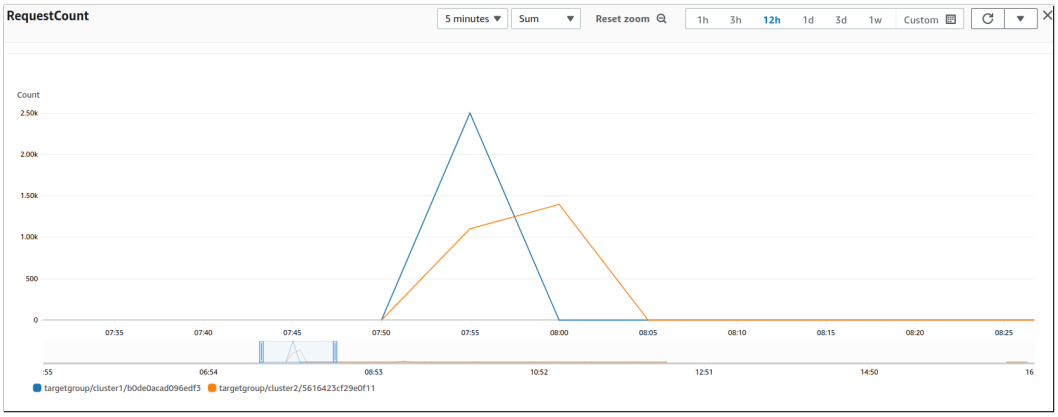


Figure 9: Request count of Cluster1 and Cluster2

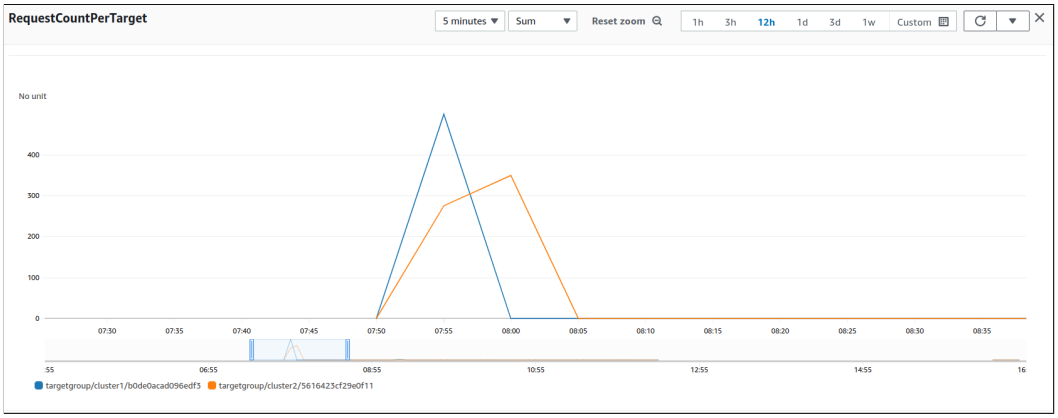


Figure 10: Request count per target of Cluster1 and Cluster2

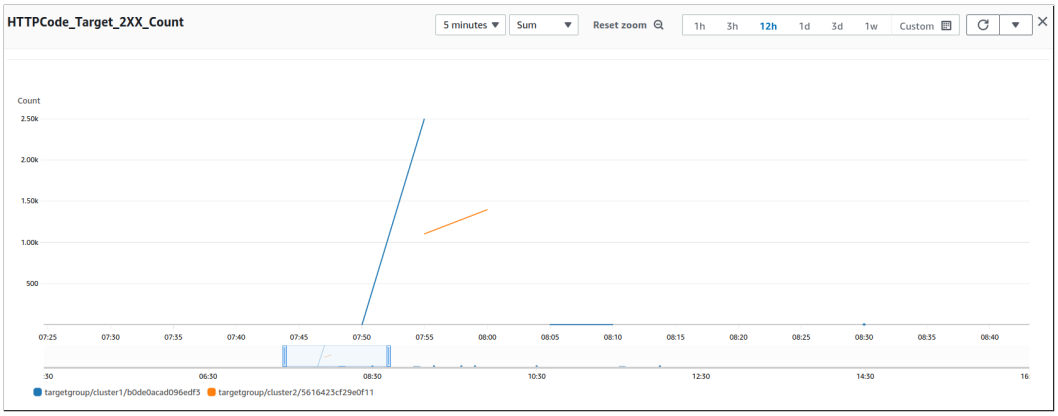


Figure 11: HTTP code Target count of Cluster1 and Cluster2

3.5k requests during running the benchmark. Graph**??** shows processed byte The total number of bytes processed by the load balancer over IPv4 and IPv6 (HTTP header and HTTP payload) and the maximum value was 1.4M bytes. Graph15 shows The number of HTTP redirection codes that originate from the load balancer the same as the request count in this scenario.
The Graph 14 is kind of different from other graphs in this part. it calculates the time elapsed, in seconds, after the request leaves the load balancer until a response from the target is received. This is equivalent to the target-processing-time field in the access logs.

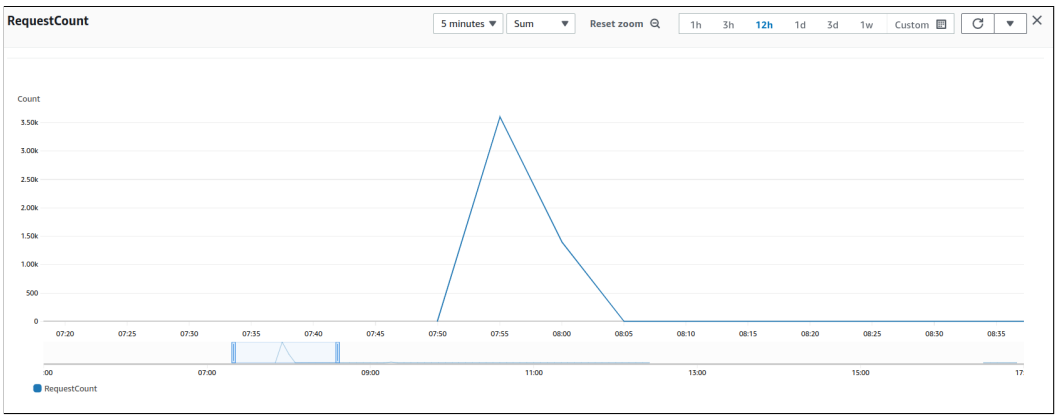


Figure 12: Request count (load balancer)

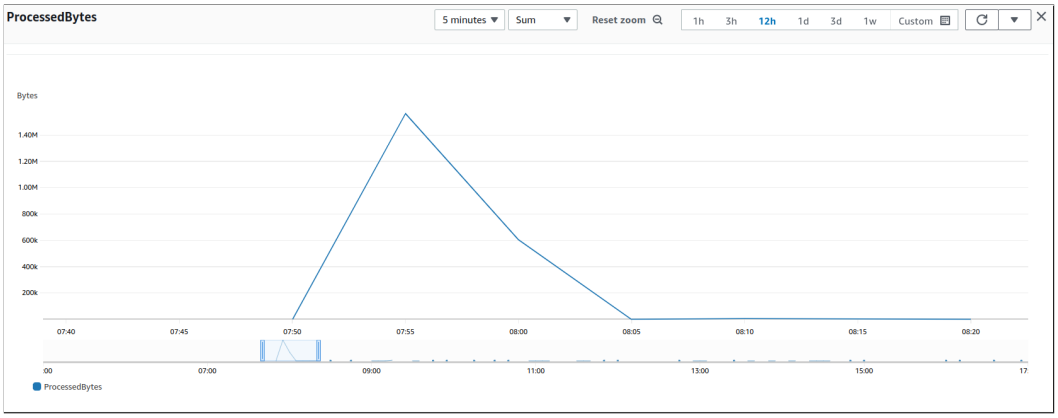


Figure 13: Processed bytes(load balancer)

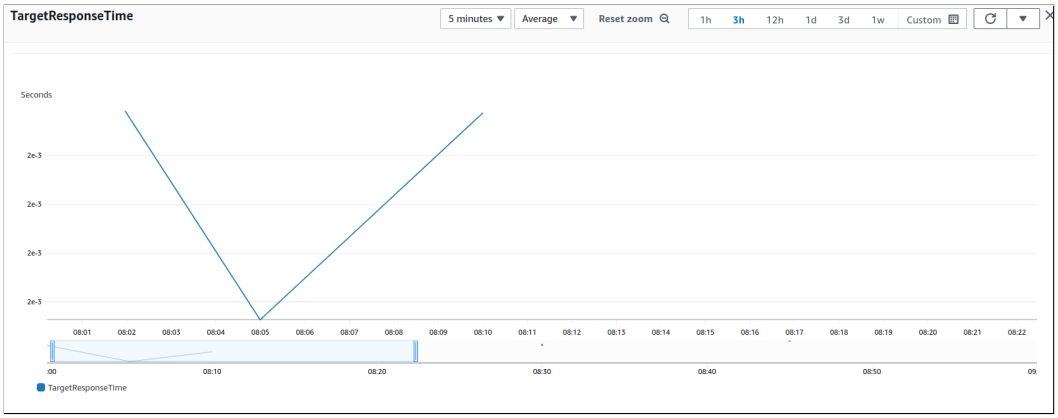


Figure 14: Target response time (load balancer)

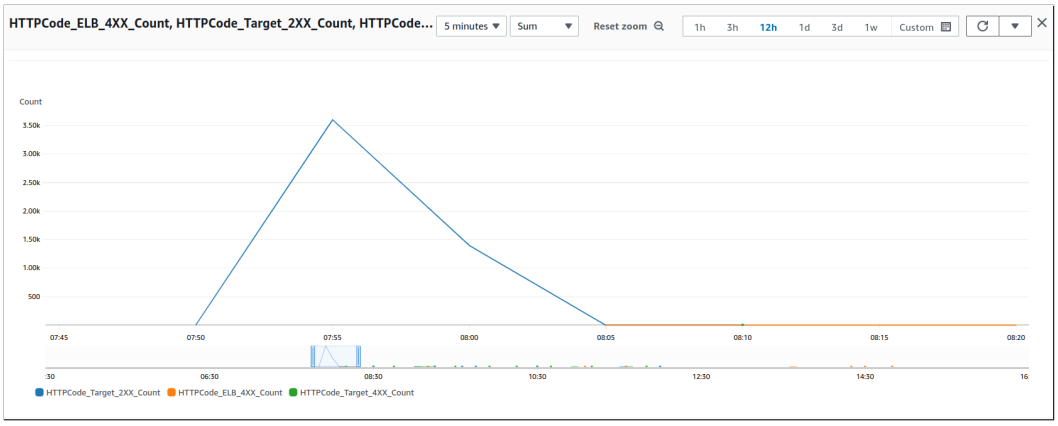


Figure 15: HTTPCode ELB 4XX Count (load balancer)

5 How to run the code

To run the program, you need to go configure your AWS credentials in the "credentials" file which is in the "Assignment 1" folder of the git repository. The following attributes:

- awsAccessKeyId
- secretAccessKey
- sessionToken

After these credentials have been modified, once you are in the Assignment 1" folder of the git repository, all you have to do is run the "script.sh" file. It will build a docker image with all the necessary requirements installed as well as the necessary files for the program. Then the docker will go ahead and run the "main.py".

It will then create everything you need and install/setup Flask into each instance, alongside the load balancer, the listener and target groups for both cluster. A key pair and security group will also be created.

Once all those are setup, it will go ahead and call the benchmark function which for each target group will setup two threads, each accomplishing one of the two scenario, as explained in section 4.4. After that, the analysis function from the "analysis.py" file will pull some metrics using Cloudwatch client. The results will be written in "results.txt" file for better reading.

The final step of the run, after analysis is done, is the program terminating everything including the load balancer, the target groups and the instances, as well as the key pair and security group.

For the sake of better understanding and clarity, although we do automate pulling some metrics, we actually did the analysis using the Cloudwatch dashboard and doing screenshots of the graph for pertinent metrics.

Also, if you want to go straight to running without need to build docker etc.. (this was for the bonus point), you can simply do "python3 main.py" in the terminal when you are in the "Assignment 1" folder and it will run the program.

6 References

References

[1] S. A. Abtahizadeh, *LOG8415: Lab 1 Selecting VM instances in the Cloud through benchmarking*, LOG8415: Advanced Concepts of Cloud Computing, 2019.

[2] Amazon Web Services. (2019) Amazon EC2. https://aws.amazon.com/ec2/

[3] Amazon Web Services. (2019) Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/

[4] Dyouri,A. (2020). How To Make a Web Application Using Flask in Python 3. Digital Ocean. https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3

[5] Amazon. (n.d). CloudWatch metrics for your Application Load Balancer. AWS. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-cloudwatch-metrics.html

[6] Amazon. (n.d). CloudWatch. AWS boto3. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/cloudwatch.html