LOG8415E- Advanced Concepts of Cloud Computing

Lab 1: Cluster Benchmarking using EC2 Virtual Machines and Elastic Load Balancer (ELB)

1895231 - Anis Youcef Dilmi

2224721 - Mohammed Ridha Ghoul

1954607 - Victor Kim

2225733 - Yasser Benmansour

Presented to Professor Vahid Majdinasab

7 octobre 2023

1. Introduction

As the utilization of cloud computing continues to expand, it becomes increasingly essential to possess a comprehensive grasp of the practicality of cloud computing platforms and the efficiency of the services they provide. In this assignment, our objective is to assess these aspects within the context of the AWS cloud computing platform. To achieve this, we will deploy two clusters employing Load Balancers, each consisting of nodes housing Flask servers capable of handling HTTP requests. Following this, we will initiate a simulation of client requests directed at the Load Balancers. Lastly, we will gather and analyze all AWS data using the AWS CloudWatch tool. To ensure a thorough evaluation of AWS services' usability, we will automate all stages of this process, including deployment, simulation, data collection, and termination.

2. Flask Application Deployment Procedure

We have developed a Flask application designed to listen on port 80 and respond with "[instance id] is responding now!" In order to deploy this application on our Amazon EC2 instances, we employ a crucial AWS feature known as UserData. UserData enables us to provide a bash script that gets automatically executed whenever a new instance is launched. Our script follows these steps:

1. Apt packages are updated to ensure the latest software packages are available.
2. Pip is installed to manage our Python dependencies.
3. Pip then installs the Python web framework Flask.
4. A dedicated directory is created for our Flask application.
5. Inside the directory, "app.py" file is created.
6. The script then uses the "ec2metadata" tool to retrieve the EC2 instance ID.
7. Within the "app.py" file, it initializes the Flask application with a route on "/cluster1" (or "/cluster2," depending on the instance type and target group). This route is configured to return "[instance id] is responding now!" when accessed via port 80.
8. Finally, the script executes the Flask application, making it ready to respond to incoming requests.

3. Cluster setup using Application Load Balancer

Our script carries out the following steps to configure the cluster:

1. It establishes an AWS Security Group named "sg" that permits inbound access on ports 80 (HTTP) and 22 (SSH) and outbound access to all ports. This Security Group is utilized by all AWS EC2 Instances, Target Groups, and the Load Balancer.
2. It initiates the launch of five M4.Large EC2 instances, each equipped with 2 vCPUs and 8 GB of RAM
3. It initiates the launch of four T2.Large EC2 instances, each also featuring 2 vCPUs and 8 GB of RAM.
4. It creates an Application Load Balancer.
5. It establishes a Target Group named "cluster1-tg."
6. It establishes another Target Group named "cluster2-tg."
7. It associates all M4 instances as targets within the "cluster1-tg."
8. It associates all T2 instances as targets within the "cluster2-tg."
9. It links the target groups to the Load Balancer as listeners.
10. It defines a path forward rule directing requests to "/cluster1" toward the "cluster1-tg."
11. It defines a path forward rule directing requests to "/cluster2" toward the "cluster2-tg."

The diagram below presents the infrastructure that we have implemented.

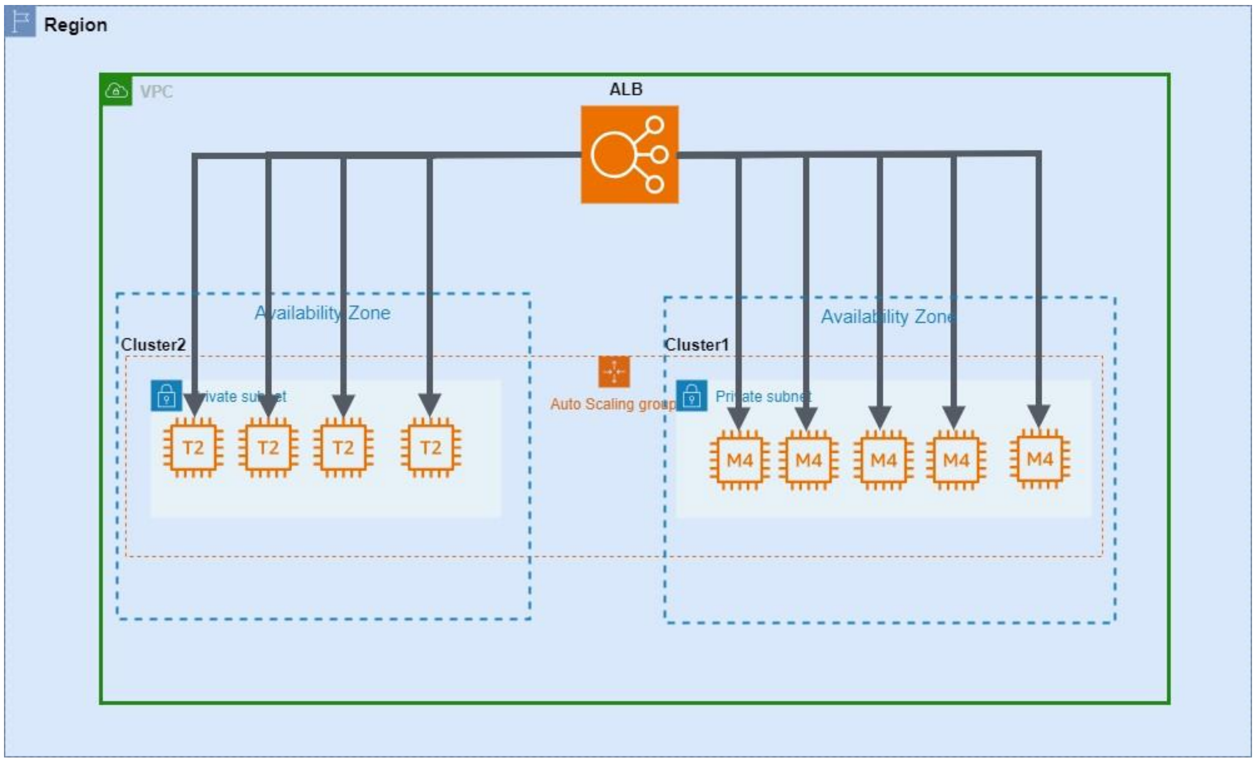

Figure 1: the infrastructure diagram

The deployment of the Flask application is automatically included in steps 2 and 3. After setting up the clusters, we execute two separate workloads in parallel: workload 1, which consists of 1000 GET requests executed sequentially, and workload 2, which comprises 500 GET requests, followed by a one-minute sleep, and then 1000 additional GET requests. We use threads to manage these workloads and gather CloudWatch metrics for "cluster1-tg," "cluster2-tg," the Load Balancer, and the EC2 instances.

4. Benchmark results

After conducting the local test scenarios, as evident from the graph displaying the NewConnectionCount metric, it becomes apparent that it takes approximately one minute for the number of new connections to gradually increase to its peak at 1600, after which it begins to decrease.

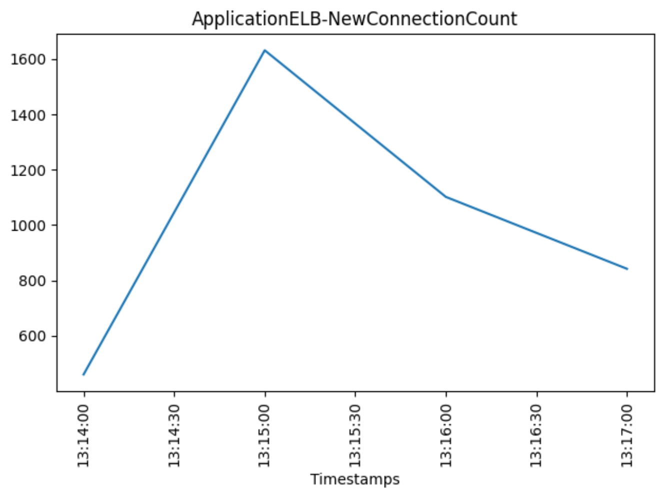

Figure 2: NewConnectionCount Metric

We can observe a similar pattern in the graph for the ProcessedBytes metric. This metric tracks the cumulative number of bytes processed by the Load Balancer. As illustrated in Figure 3, the graph exhibits a parallel trend to the previous one. In this instance, it reaches a peak of over 350,000 bytes processed.

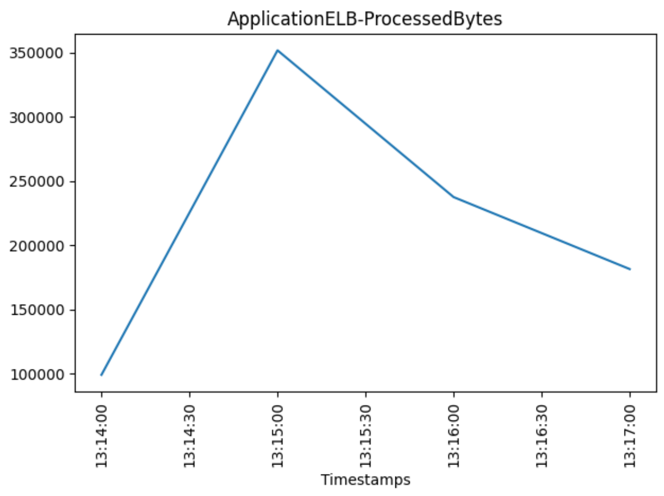

Figure 3: ProcessedBytes Metric

The TargetResponseTime metric exhibits a consistent pattern akin to the two prior graphs. When at its pinnacle, it necessitates approximately one second for the application to provide a response before subsequently diminishing. This prolonged response time can be attributed to the heightened load exerted on the Load Balancer.

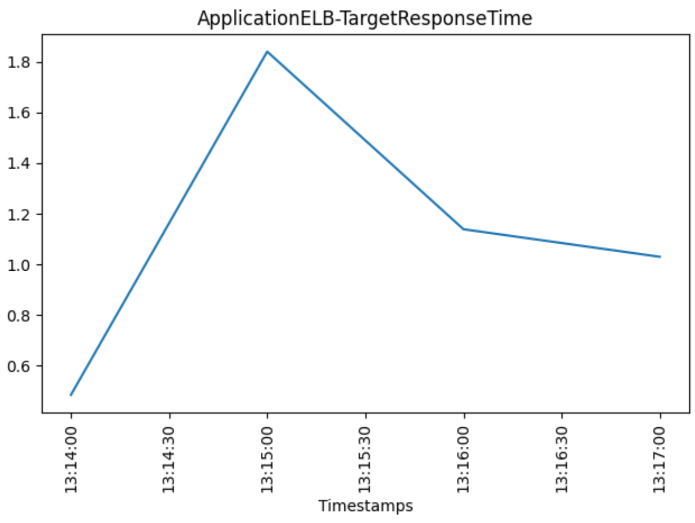

Figure 4: TargetResponseTime Metric

The RequestsCountPerTarget graph serves as a tool for evaluating the request distribution among the various target groups. A close examination of this graph enables us to distinguish between the unique workloads assigned to each cluster.

For instance, we can readily discern that the workload involving 1000 GET requests is explicitly channeled toward cluster 1, which comprises M4 instances. This allocation manifests as a singular peak in the graph. Conversely, the second workload, encompassing 500 GET requests, a pause, and then 1000 GET requests, is intended for cluster 2, housing T2 instances. Notably, this workload is visualized in orange on the graph and is characterized by the presence of two distinct peaks, rather than a solitary one. This distinction highlights the diverse workload patterns associated with each cluster.

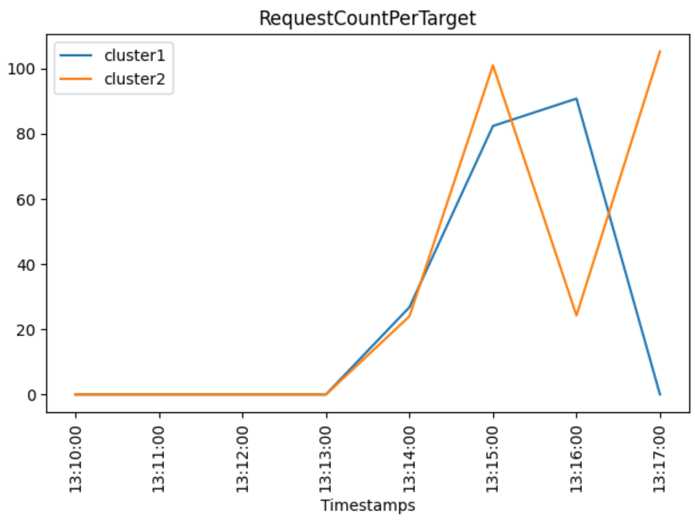

Figure 5: RequestCountPerTarget Metric

By referring to the CPUUtilization graph, we gain insights into the comparative average CPU utilization across each instance. A conspicuous observation emerges as we note that four instances exhibit notably higher average CPU utilization percentages, which we have identified as our T2 instances. In contrast, the remaining five instances correspond to our M4 cluster.

Our working hypothesis is twofold. First, the M4 instances likely exhibit lower average CPU utilization due to two factors:

i) a greater number of instances sharing the computational load (5 vs. 4), potentially distributing the load more efficiently, and

ii) the load being distributed over an extended time frame, as evidenced by the 60-second pause between the two series of GET requests. This elongated measurement period contributes to a lower average CPU utilization as the total observation duration is extended.

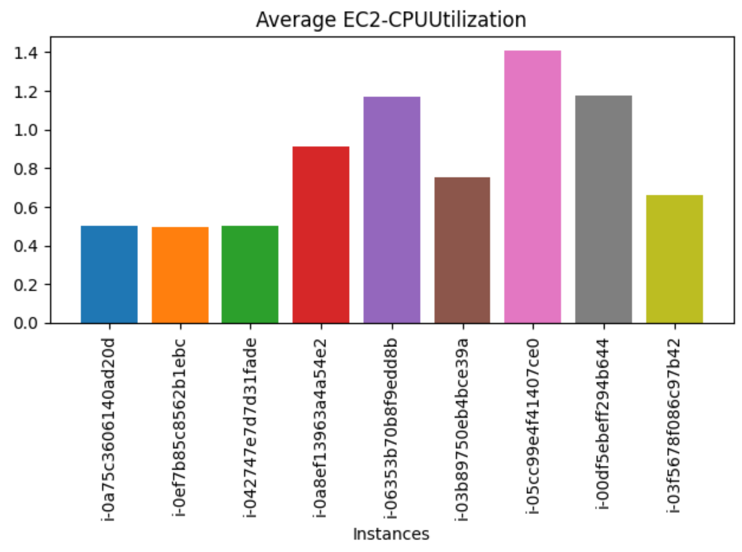

Figure 6: Average EC2 CPUUtilization Metric

The NetworkOut graph yields findings that parallel those observed in the CPUUtilization graph. Here, we scrutinize the average network outbound requests across each instance. Once more, it becomes evident that four instances stand out with a significantly higher volume of outbound requests. Notably, these instances align with those previously identified as having elevated CPU utilization in the CPU Utilization graph.

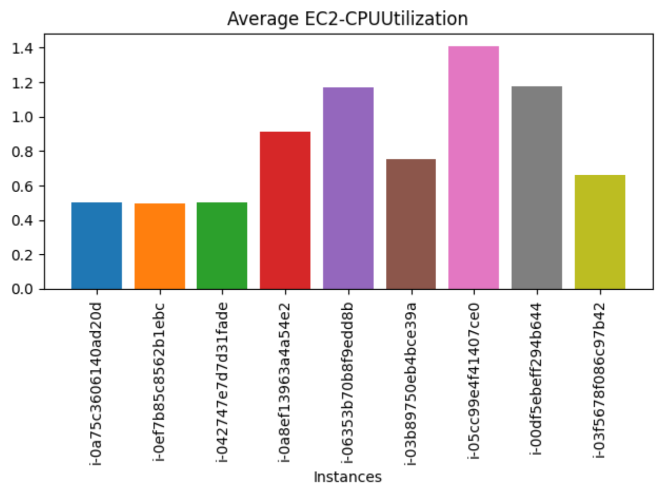

Figure 7: Average EC2 NetworkOut Metric

5. Instructions to run the code

To execute the code, you will need to follow these steps:

1. Ensure that the code is present on your local machine.
2. Have Python 3 installed locally on your machine.
3. Install Docker locally on your machine.
4. Make sure you have an AWS account.
5. Update the environment variables within the ".aws_creds" file with the accurate values for AWS ACCESS KEY ID, AWS SECRET ACCESS KEY, and AWS SESSION TOKEN.

Once these configurations are complete, the final step involves running the "run.sh" bash script when creating an instance. This script handles the installation of Python dependencies and the execution of the main program. Throughout this process, log messages will be visible in the console, providing information at each step of execution. You can view the results directly in the console or locate them in the "json" folder for further examination.

Link to our Github repo: git@github.com:youcefdilmi88/log8415e-tp1.git