# LOG8415
# Advanced Concepts of Cloud Computing

William Bourret (2005931), Vlad Drelciuc (1941366),
Charles Fakih (2012512), Simon Tran (1961278)

Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada

## 1 Flask Application Deployment Procedure

We build a Flask application that simply listens on port 80 and returns "[instance_id] is responding now!". In order to deploy this app on each of our Amazon EC2 instances, we are using an AWS concept called UserData. UserData allows us to provide a bash script that will be executed automatically each time a new instance is deployed. Our script performs the following steps:

1. Update the apt packages
2. Install pip to manage our Python dependencies
3. Install Flask using pip
4. Create a flask_app directory
5. Create an app.py file
6. Grab the EC2 instance ID using ec2metadata
7. Inside app.py, initialize the Flask app with a route on "/cluster1" (or "/cluster2" depending on the instance type and the target group) and make the route return "[instance_id] is responding now! " when called on port 80
8. Execute the app

## 2 Cluster Setup

Out scripts performs the following cluster setup:

1. It creates an AWS Security Group (called custom-sec-group) that allows inbound access to ports 80 (HTTP) and 22 (SSH) and outbound access to all ports. All of our AWS EC2 Instances, Target Groups and the Load Balancer will use this Security Group.
2. It launches five M4.large EC2 instances (2 vCPU, 8 GB RAM)

3. It launches four T2.large EC2 instances (2 vCPU, 8 GB RAM)
4. It creates an Application Load Balancer
5. It creates one Target Group called: cluster1-tg
6. It creates one Target Group called: cluster2-tg
7. It attaches all M4 instances as targets of cluster1-tg
8. It attaches all T2 instances as targets of cluster2-tg
9. It attaches the target groups as listeners of the Load Balancer
10. It creates a path forward rule for route "/cluster1" towards cluster1-tg
11. It creates a path forward rule for route "/cluster2" towards cluster2-tg

The Flask application will automatically be deployed as part of steps 2 and 3. Once the clusters are setup, we then run 2 independent workloads (workload 1 = 1000 GET requests sequentially; workload 2 = 500 GET requests, then one minute sleep, followed by 1000 GET requests) in parallel using threads and collect CloudWatch metrics for: cluster1-tg, cluster2-tg, the Load Balancer and the EC2 instances.

## 3 Benchmark Results

As we can see in the NewConnectionCount metric's graph, it takes around one minute for the number of new connections to reach its peak at 2250 before decreasing.

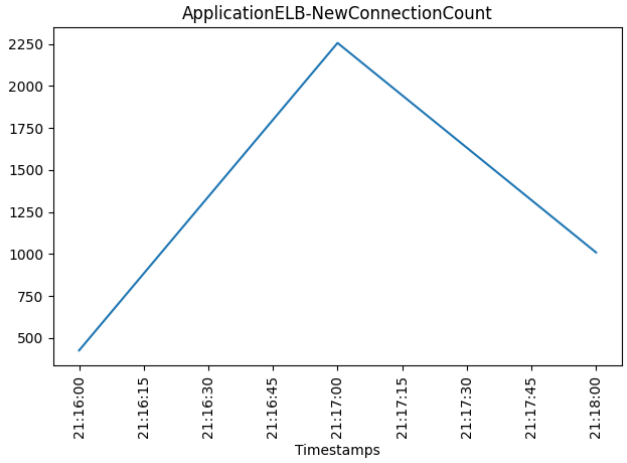


Figure 1: NewConnectionCount Metric

We can make the same observation for the ProcessedBytes metric's graph. Here, we measure the total number of bytes processed by the Load Balancer. As we can see, the graph shown at figure 1

follows the same pattern as the previous graph. In this case, it processes a maximum of over 450,000 bytes.

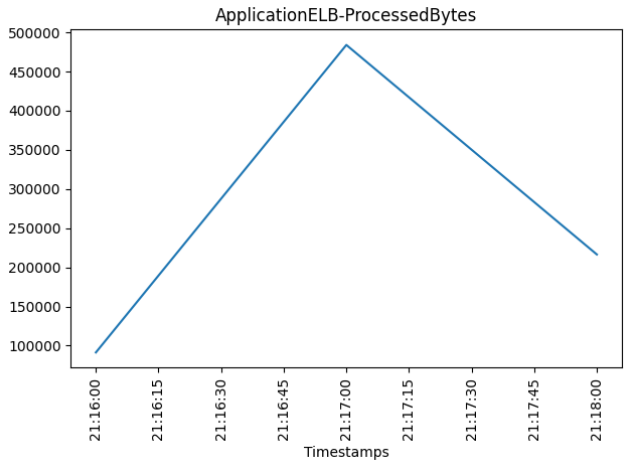


Figure 2: ProcessedBytes Metric

The TargetResponseTime metric's also follows the same pattern as the two previous graphs. At its peak, the application takes about 2 seconds to respond before decreasing. The excessive stress on the Load Balancer explains a longer response time.

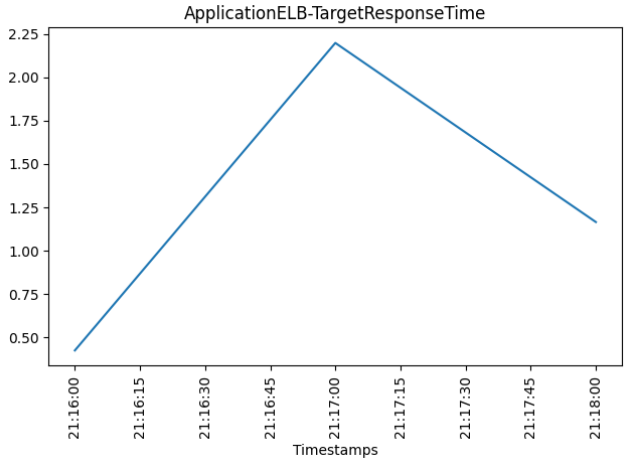


Figure 3: TargetResponseTime Metric

With the RequestsCountPerTarget graph, we compare the number of requests for each target group. Looking at the graph, we can identify the different workloads on each cluster. For example, we know the workload of 1000 GET requests is directed to the cluster 1 (M4 instances), making a single and slightly higher peak. The second workload (500 GET requests + pause + 1000 GET requests) targeted at the cluster 2 (T2 instances) can be observed in orange on the graph, making 2 peaks instead of one.

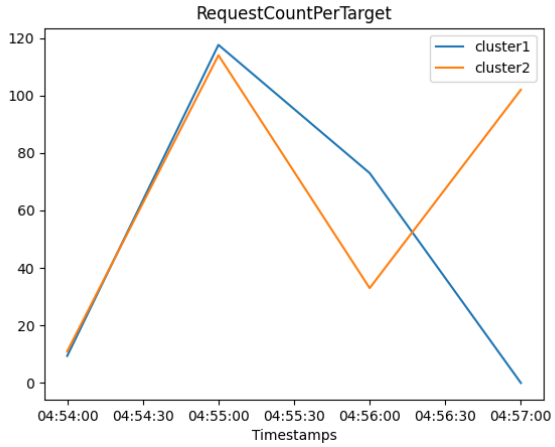


Figure 4: RequestCountPerTarget Metric

With the CPUUtilization graph, we can compare the average CPU utilization on each instance. We can clearly see that 4 instances have a much higher average CPU utilization percentage and we identified these as our T2 instances. The remaining 5 instances correspond to our M4 cluster. Our hypothesis is that the M4 instances have a lower utilization on average because i) there are more instances to share the load (5 vs 4) and ii) the load is spread over a longer period of time (due to the 60 second pause between the 2 series of GET requests), lowering the average as the total measuring period is longer.

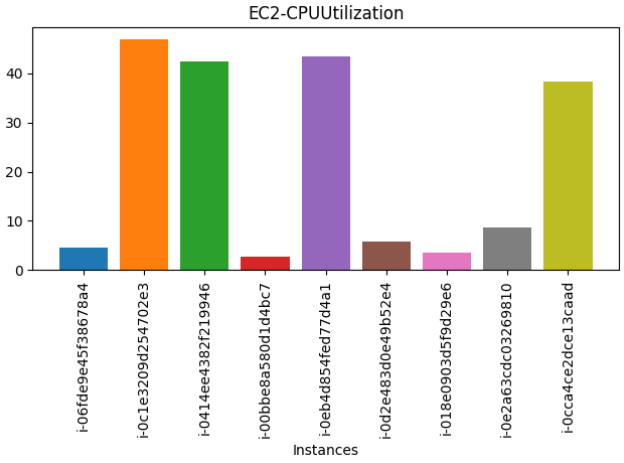


Figure 5: Average EC2 CPUUtilization Metric

The NetworkOut graph gives results that are similar to the CPUUtilization one. , we can compare the average network out requests on each instance. Once again, we observe that 4 instances have much more out requests. These instances are the same as the instances that showed a high CPU usage in the CPU Utilization graph.

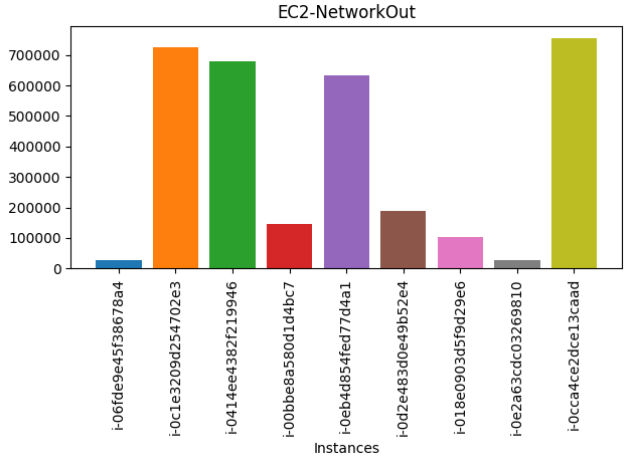


Figure 6: Average EC2 NetworkOut Metric

# 4 How to Run the Code

To run the code, you will need to:

1. Install and run Docker Engine on your local machine
2. Navigate to the project folder: **cd tp1**
3. Update the environment variables inside **.env** with the correct values for AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY and AWS_SESSION_TOKEN
4. Make **scripts.sh** executable: **chmod +x scripts.sh**
5. Then simply execute: **./scripts.sh**