

LOG8415e  
Advanced Concepts of Cloud Computing

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

Lab 1

Autumn 2022

**By:**  
Minh-tri Do  
Jeremy Hage  
Jean-paul Khoueiry  
Jonathan Siclait

**Presented to:**  
Vahid Majdinasab

Monday, October 17, 2022

Github link : <https://github.com/jehag/LOG8415E>

## 1 Flask Application Deployment Procedure

As soon as an instance is created, the code of the `userdata.sh` file will be executed in `sudo` mode (admin, all permissions given) on the instance. This script's job is to deploy the flask app on the public IP address of the instance on port 80. This file uses Nginx, Gunicorn, Python, and Flask to serve our app. This is how the script deploys the Flask app:

1. The script pulls the python flask app and all other necessary files from our GitHub.
2. It then creates a Gunicorn service to deploy the flask app locally (`localhost:8000`). Starting the flask app service will in turn start the Flask app (`app.py` file).
3. It initializes and modifies Nginx, which acts as an internal load balancer that we configure in the default file to redirect all public requests to the EC2 instance port 80 to the port where Gunicorn has deployed our app.
4. It starts the Nginx service and the Gunicorn service on the instance to allow the flask app to answer HTTP requests.

## 2 Cluster setup using Application Load Balancer

Cluster setup can be done quite easily by running the `scripts.sh` executable file. When we run our code in `scripts.sh`, we start by pulling and running a docker image that we created for this assignment. This docker image runs the dockerfile called `requests.Dockerfile` which in turn installs multiple tools that we'll need like Terraform and Python. It also copies the `cloudwatch.tf`, `main.tf`, `variable.tf`, `userdata.sh`, `docker_script.sh`, `send_requests.py` and `benchmark.py` files into the container. The dockerfile also contains an instruction to execute the `docker_script.sh` inside the docker container at runtime. The `docker_script.sh` does multiple things. The most relevant part of this file for this particular section is near the beginning of this script. It is where we initialize (Terraform init) and create (Terraform apply) the AWS resources along with their dependencies with the help of Terraform. Our desired AWS configuration is defined in the `main.tf`, while the cloudwatch dashboard setup is defined in the `cloudwatch.tf` files. We also use the `variable.tf` file to store some recurring variables.

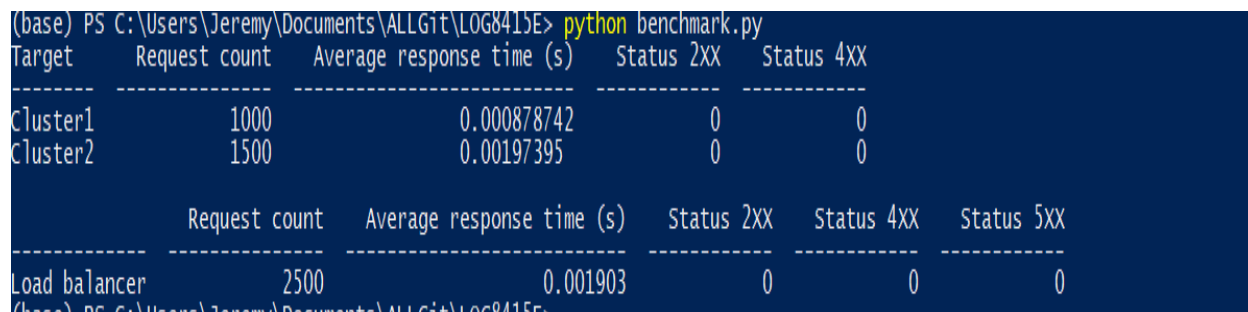
In the `main.tf` file, we use resource blocks to setup our clusters. First, we create multiple `ec2` instances for both cluster 1 and cluster 2 with their count, type, and `availability_zone` being defined in the `variable.tf` file. For their `user_data`, as explained above, we used the file `userdata.sh` so that the flask app will be available at launch (`ec2` will run `user_data` on instance creation). We also assign a security group that is defined later in this file as well as a distinct name in their tags.

Later on in the file, we announce that we will be using the default `vpc` which is why it is important to already have a default `vpc` to run this code as instructed in section 4 of this report. We then create the aforementioned security group called `not_secure_group` that allows any access from any port to any port using any protocol.

Then, we create a Application Load Balancer (ALB) called alb that will later be used to route requests to our 2 clusters. Once it is created we create two target groups: One called cluster1 and another called cluster2 that we link to port 80 with the HTTP protocol and our default vpc. To get uptime data in the cloudwatch dashboard, we added a health check on the path /clusterX on port 80 for each target group. After that, we add an ALB listener with 2 rules called cluster1\_rule and cluster2\_rule that redirects all traffic directed to either cluster to their respective cluster (/cluster1 redirects to cluster1 and /cluster2 redirects to cluster2). To make each instance get requests from the right target group, we added a target\_group\_attachment resource that links each instance to their respective target group, in accordance with their instance type.

### 3 Results of the benchmark

To benchmark our clusters, in the docker\_script.sh file, we run send\_requests.py which sends 2 groups of requests to each cluster. The first cluster receives 1000 threaded requests while the second one receives 500 threaded requests at the same time as the first cluster, followed by 1000 threaded requests 60 seconds later. We chose to thread the request groups to send all the requests to both clusters at the same time. The requests themselves are also threaded to speed up the process. After completing both request threads, we run benchmark.py to fetch useful cloudwatch metrics related to the requests we just made to our system. We present our results thoroughly in the Annex with graphs but here is a screenshot illustrating what the output of the benchmark looks like in the terminal.



```
(base) PS C:\Users\Jeremy\Documents\ALLGit\LOG8415E> python benchmark.py
```

Target	Request count	Average response time (s)	Status 2XX	Status 4XX
Cluster1	1000	0.000878742	0	0
Cluster2	1500	0.00197395	0	0

	Request count	Average response time (s)	Status 2XX	Status 4XX	Status 5XX
Load balancer	2500	0.001903	0	0	0

```
(base) PS C:\Users\Jeremy\Documents\ALLGit\LOG8415E>
```

fig 1 : Benchmarking results in the terminal

Before heading into the analysis of the difference between each cluster, we need to specify the difference between the 2. Cluster1 contains 4 m4 large instances, which are supposed to provide an overall great balance between computing, memory, and network resources. On the flip side, Cluster2 contains 5 T2 large instances which are burstable performance instances that can temporarily sustain higher CPU performance. The first observation we can make for both clusters is that they each received and returned an answer to every request that was sent (1000 requests to Cluster1 and 1500 requests to Cluster2). The second observation made is that even if the cluster1 received fewer requests, it still had a greater average request count per instance in a 60-second period, since it has fewer instances, and that the 1500 requests sent to cluster 2 were split up into 2 groups, and both groups had a 60-second interval between them. This made it so that Cluster1 had a maximum average of 250 requests per second, while Cluster2 had a maximum

average of 200 requests per second. The most interesting observation is the average response time of each cluster. Even when Cluster1 had fewer instances than Cluster2, it still ended up with an average response time 2 times smaller than its counterpart. While this difference in response time could have been due to the difference in the number of requests sent to each cluster, it is still a very good indicator that M2 instances are better suited for this kind of application. Lastly, we can see that the cloudwatch dashboard shows that every instance in both cluster1 and 2 is unhealthy. This could be due to 2 reasons. The health check might not be working properly, and/or the hosts themselves might not have been available at that time due to the massive number of requests they received. Further testing could have been done to identify the problem, but since it was not the main purpose of the assignment, we chose to simply include a note mentioning this anomaly.

## 4 Instructions to run the code

To run the code, it is important to already have AWSCLI and docker installed as well as AWS credentials set up in your workspace (this can be verified by doing: `aws sts get-caller-identity`). Your AWS account should also be clean of instances, target groups, and load balancers. It must however have a default vpc. This should technically be the initial state of your account if you just opened it for the first time. The only thing left to do then is to double-click on the scripts.sh file and the code should run perfectly.

## 5 Annex

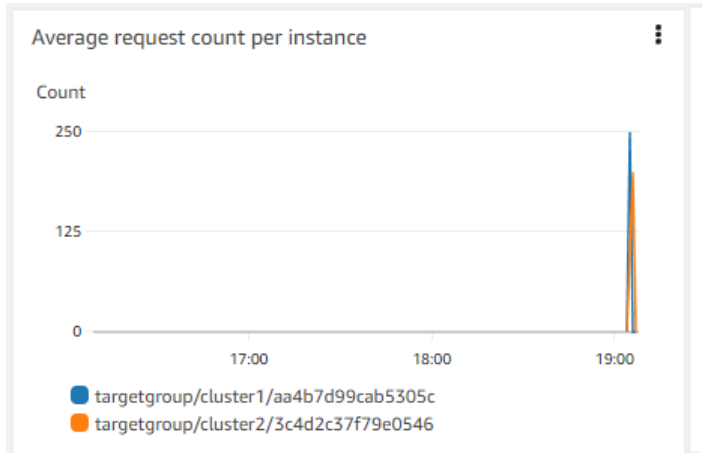


fig 2 : Average Request Count per Instance

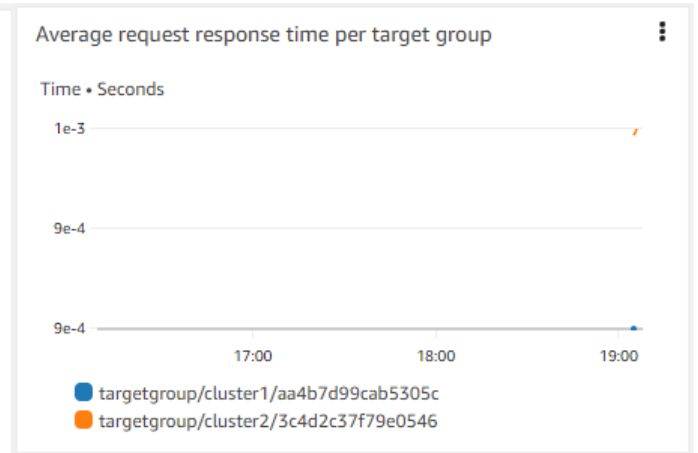


fig 3 : Average Request Response Time Per TargetGroup



fig 4 : Average Request Response Time

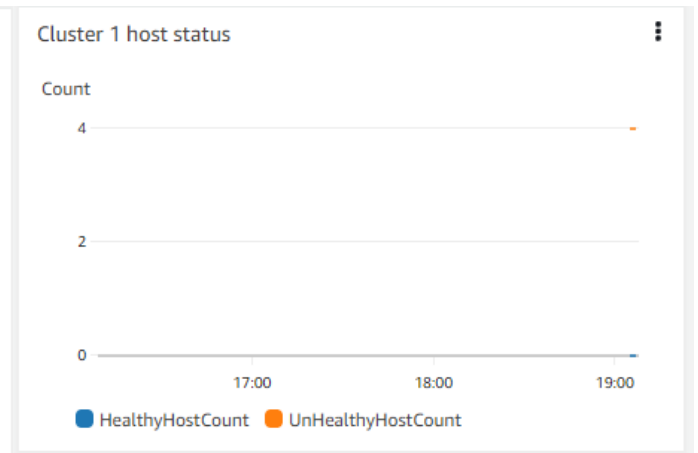


fig 5 : Cluster1 Host Status

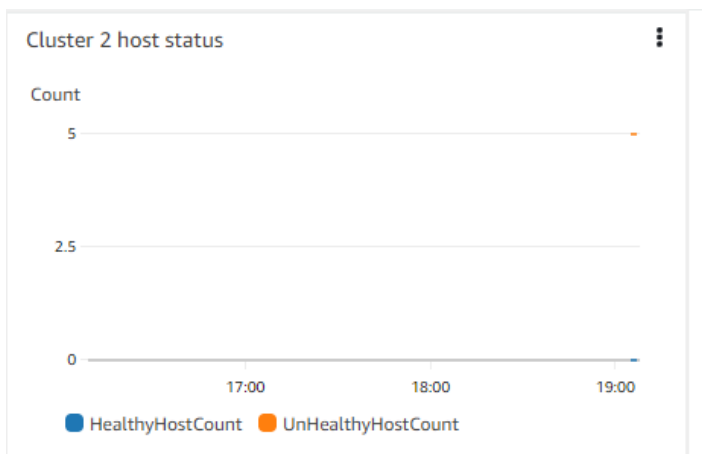


fig 6 : Cluster2 Host Status

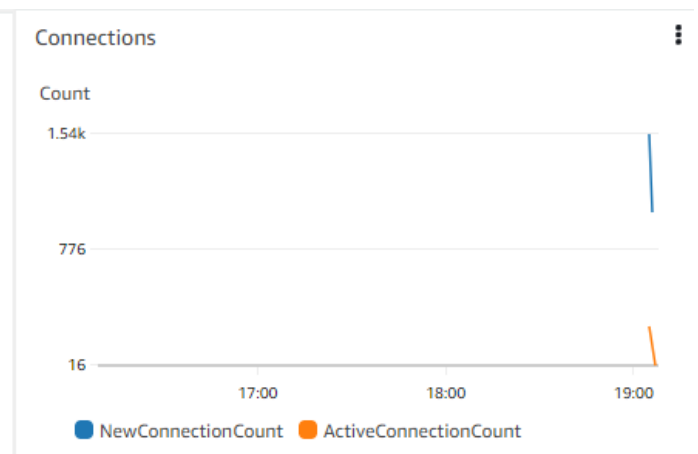


fig 7 : Connections

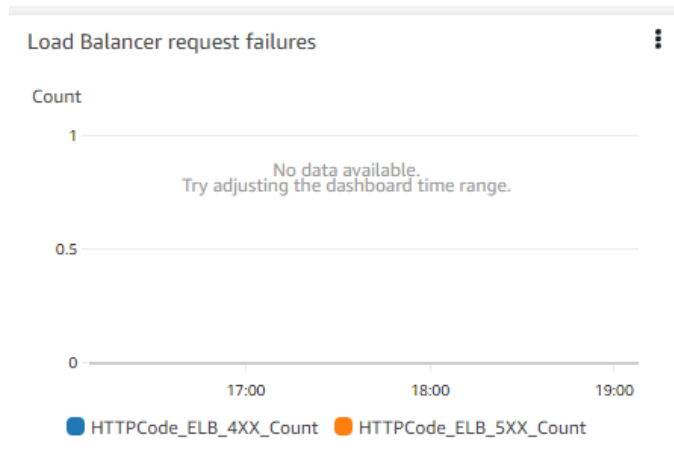


fig 8 : Load Balancer Request Failures

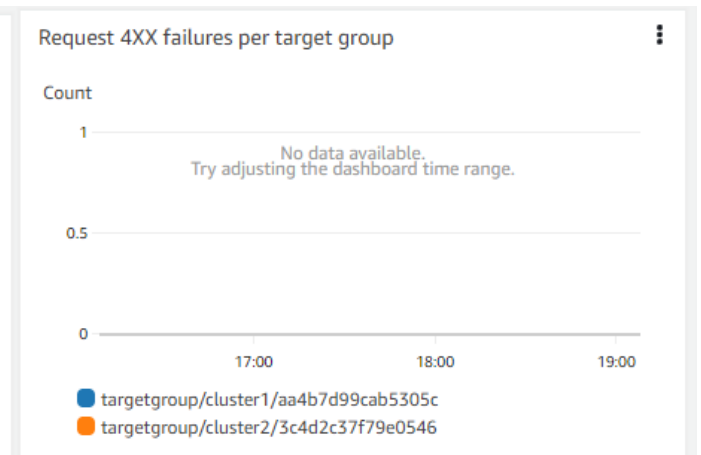


fig 9 : Request 4XX Failures Per Target Group



fig 10 : Request Count Per Target Group

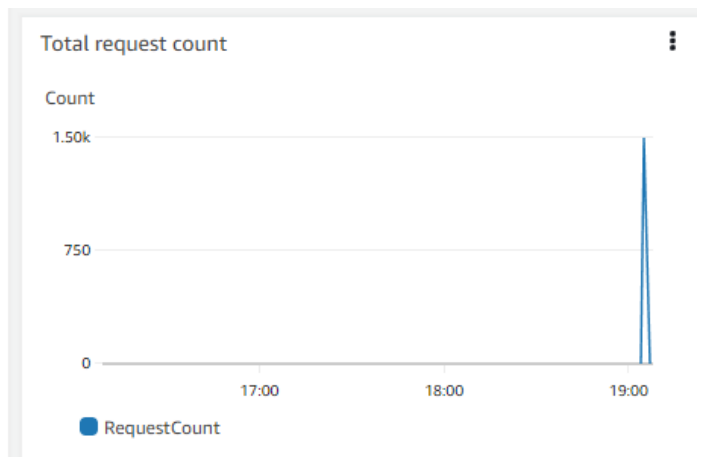


fig 11 : Total Request Count