LOG8415E - TP1

Nathan Rullier, Mathieu Marchand, Philippe Rivest and Charles-Étienne Joseph

October 10th, 2021

Abstract

In this assignment, we created two clusters of virtual machines and performed extensive benchmarks on several instance types in our clusters. This report aims to answer some questions and to explain our work. Firstly, we'll explain the deployment procedure of a Flask application in the virtual machines. Secondly, we'll explain the setup of our clusters using the Application load balancer. Thirdly, we'll present and analyze the results of our benchmarks. Last but not least, we'll present the instructions needed to run our code, since all the process to launch the virtual machines and benchmark them has been automated for efficient usage.

1 Questions

1.1 Flask application Deployment Procedure

There are several viable methods to deploy Flask applications to AWS. While it is possible to setup EC2 instances to run our own Python code, we preferred Elastic Beanstalk (EB) to manage our deployments.

EB provides an abstracted interface to easily deploy web applications without sacrificing configurability or automation. An EB application is executed on a specific platform. In our case: `Python 3.8 running on 64bit Amazon Linux 2/3.3.6`. The platform acts as an EC2 instance template. That way, the spawned instances have Python preinstalled and are already configured to serve HTTP(S) traffic.

To automate our deployments and express them as code, additional EB configuration is bundled with our source code. The EB command-line tool (`eb`) reads the contents of `.config` files located in the `.ebextensions` subdirectory. We added a `python.config` to configure `gunicorn` WSGI path, process and thread count.

To replicate the desired dual-cluster setup, we created two environments to our application: cluster1-env and cluster2-env. The application is deployed on environment creation as described in section 1.2. To update the code on an already existing env, the following commands can be used:

```
eb deploy cluster-1
eb deploy cluster-2
```

1.2 Cluster setup using Application load balancer

Setting up the two clusters requires many steps, but we managed to automate the process in a script named deploy.sh. AWS CLI and EB CLI must be installed and configured properly before beginning (i.e. setting the default profile in `/.aws/credentials`. This section will present every step that is needed to deploy the clusters.

The first step is to create a security group in which HTTP ingress is allowed. Without it, associated load balancers won't respond to inbound traffic.

The next step is to create a load balancer (elb) with two availability zones using our custom security group. We used us-east-1a and us-east-1b. Then, we need to add an HTTP listener. The two clusters we are about to create will share this elb.

Before deploying our clusters, we need to initialize our local Flask application code with `eb init`. Then, we can generate the shared-elb configuration file from a template that is under the `app/.ebextensions` directory. Since the ARN and DNS of the created elb is random, our templating approach is required. The resulting configuration describes the rules to append to our shared elb to route traffic using `/cluster1` and `/cluster2` routes.

Once this is done, we are ready to create our first cluster of m4.large instances. For the second cluster we again create a shared elb configuration file from the template to use the `/cluster2` route. Then, we create our cluster with t2.xlarge instances. As mentioned earlier, our Python code is automatically bundled and deployed to our new instances on env/cluster creation.

Once all those steps are completed, our two clusters will be online and we will be able to query them using our elb. It's now time to benchmark them.

1.3 Results of your benchmark

In order to have a better idea on how our EC2 Virtual Machines and Elastic load balancer were going, we used metrics from CloudWatch in order to further our understanding. We used the metrics : RequestCount, RequestCountPerTarget and TargetResponseTime because they gave us the most insight. We judged the rest of the metrics to be futile to the understating of this lab. Each graph will represent metrics we were able to extract for a single cluster and we extracted those data for both clusters. The data we fetched can give us an impression of how the load balancing was done as well as how a cluster, as a whole, handled the requests.

Two different clusters were used to receive the requests. We had four m4.large in the first cluster and four t2.xlarge. The t2 are more powerful than the m4 (having double the number of vCPUs and memory). But the m4 have a support for enhanced networking and 450 Mbps of dedicated EBS bandwidth.

In this section we will present images of graphs that were extracted directly from AWS Cloudwatch. The images are under the benchmarking/img folder.

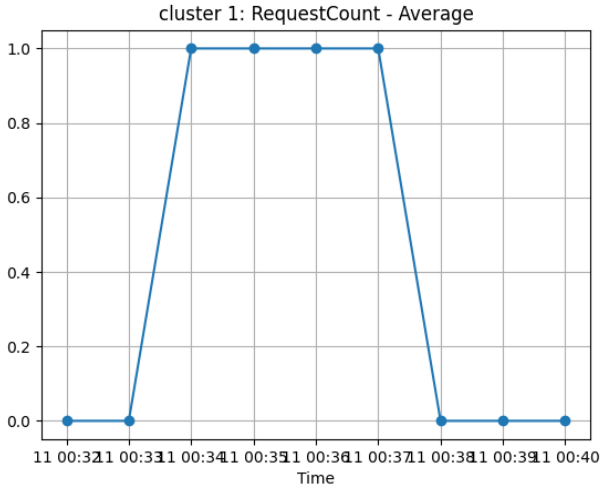

Figure 1: RequestCount average in function of time for cluster 1

The graph above shows us the time that it took the cluster 1 to receive and process all the requests it received. We are able to see that the requests were received between 00:33 and 00:38. It took 5 minutes to respond to all requests.

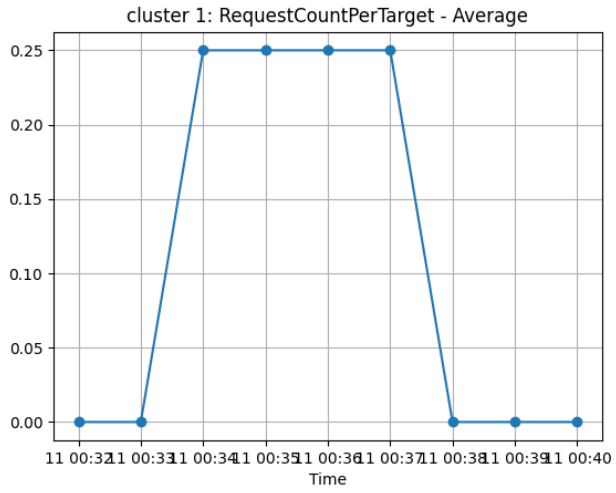

Figure 2: RequestCountPerTarget average in function of time for cluster 1

The second graph shown above looks very much alike the first graph. The only difference is that the average request count per target is a fourth of the request count. This is logical because we have 4 target instances and 1 elastic load balancer and that the requests using a round robin.

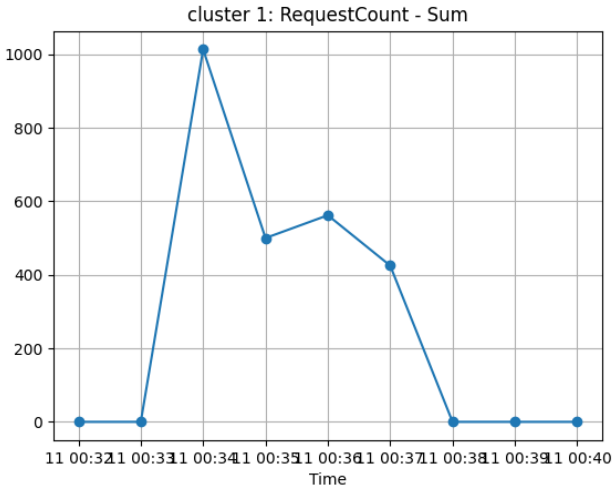

Figure 3: RequestCount sum in function of time for cluster 1

This first thing we can see in this graph is that the total of requests that was received is 2500. This is the number of request that was supposed to be received. The first requests were received during the first minute. We can attribute the decrease in request to the one minute pause in the script. The cluster then processed the remaining incoming requests.

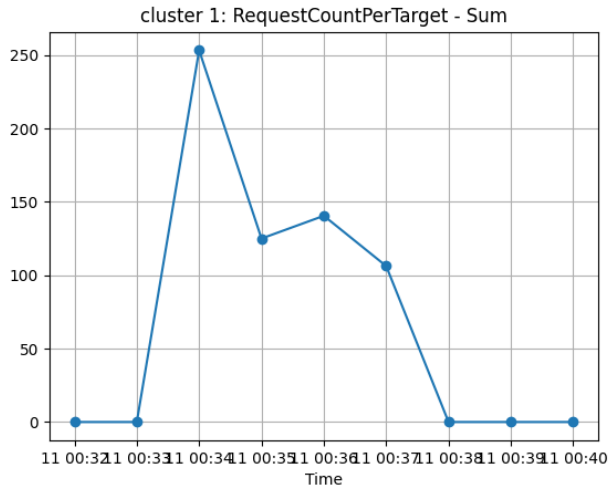

Figure 4: RequestCountPerTarget sum in function of time for cluster 1

The fourth graph shows us practically the same thing as the third graph. The difference being the number of requests processed by each of the four instances. Thus the results are divided by four.

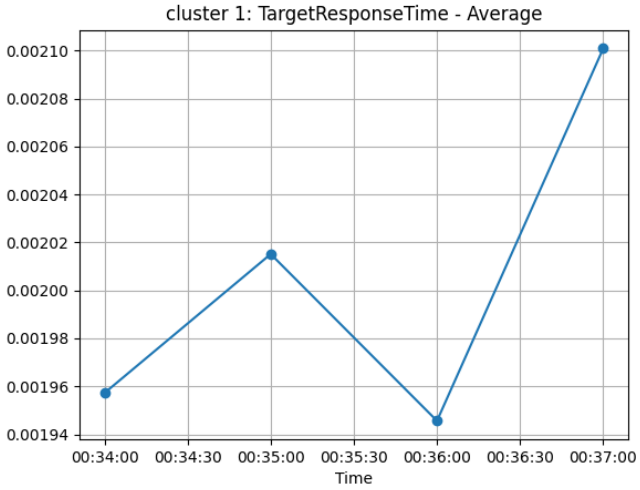

Figure 5: TargetResponseTime average in function of time for cluster 1

In the graph above, it's possible to see that the target response time has been really stable during the execution of our script. The response time only varied from 0.00194 to 0.00210 minutes. The two lower points could be associated with a lack of requests sent. The first lower point would be the start of the script and the second dip to the pause of one minutes in our script.

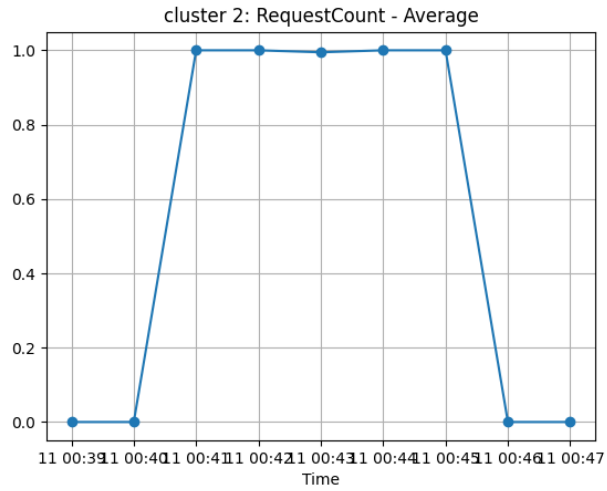

Figure 6: RequestCount average in function of time for cluster 2

The graph above shows us the time that it took the cluster 2 to receive and process all the requests it received. We are able to see that the requests were received between 00:40 and 00:46. It took 6 minutes to respond to all requests.

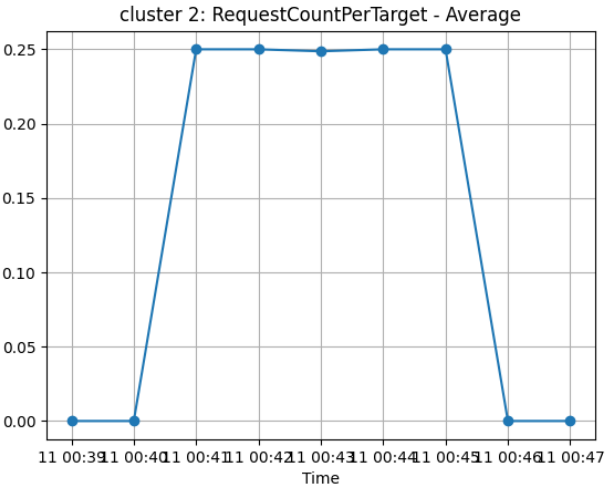

Figure 7: RequestCountPerTarget average in function of time for cluster 2

This graph shown above looks very much alike the graph number 6. The only difference is that the average request count per target is a fourth of the request count. This is logical because we have 4 target instances and 1 elastic load balancer and that the requests using a round robin.

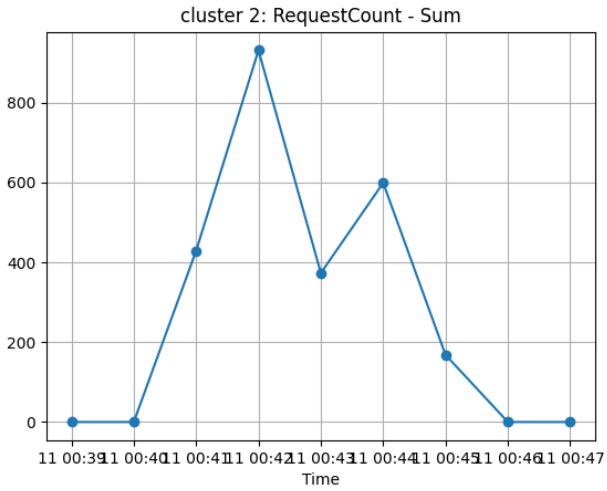

Figure 8: RequestCount sum in function of time for cluster 2

We can see in this graph as well that all the requests have been processed. This is the number of requests that was supposed to be received. The first requests were received during the first minute. We can attribute the decrease in request to the one minute pause in the script. The cluster then processed the remaining incoming requests.

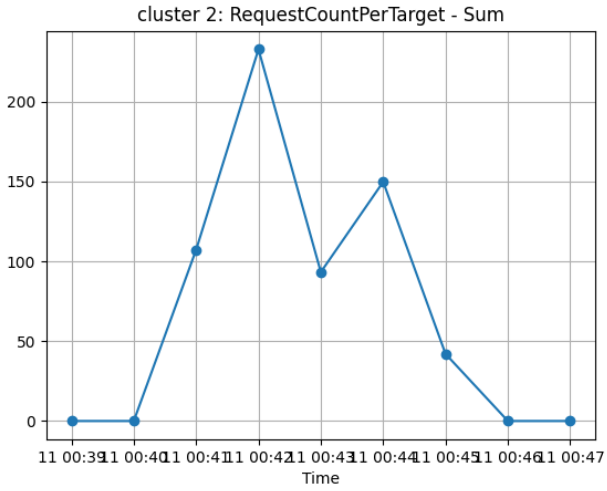

Figure 9: RequestCountPerTarget sum in function of time for cluster 2

The fourth graph shows us practically the same thing as the third graph. The difference being the number of request processed by each of the four instances. Thus the results are divided by four.

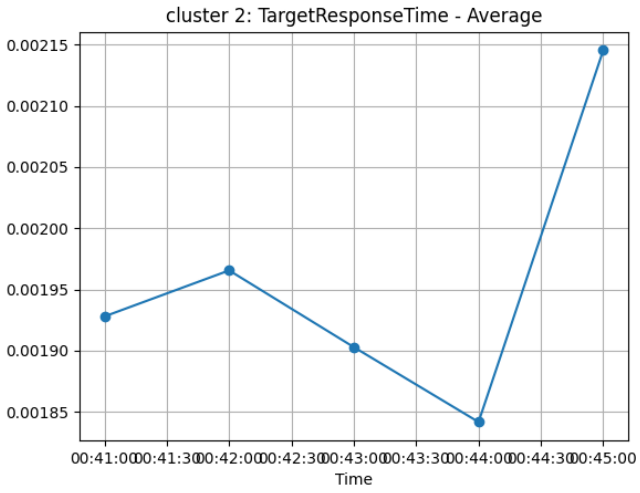

Figure 10: TargetResponseTime average in function of time for cluster 2

This graph shows the response time has been pretty stable during the benchmarking, even if it went higher at the end. At its highest it took in average 0.00215 minutes to answer a request and at its lowest, it took less than 0.00185. The lower points could be associated with a lack of requests.

Analyzing the two clusters

The first thing to note when comparing our two clusters is that our second cluster took around 1 more minute to process all the requests it received. It's also possible to see that the cluster 1 was able to process a bit over 1000 requests in a minute. The most request that our second cluster was able to process in the same amount of time was a little over 950. We think that these differences can be justified by the additional network resources that the m4 instances were given. Even when comparing our target response time the cluster was slower at his peak. Since our requests weren't compute heavy or memory hungry, it's normal that the instances with the best networking had the best performances.

1.4 Instructions to run your code

The instructions to run our code can all be found in the README.md at the root of our project.

As mentioned earlier, you have to start by installing AWS CLI and EB CLI. They are essential to run the project.

The project is divided in three parts. First there is the Flask application, second there is the automated deployment of our clusters with our shared load balancer and then last but not least there is the benchmarking app.

To start the Flask application you only need to install the libraries required, which can be found in the requirements.txt file under the app folder. To execute it locally, run the following commands in the `app` directory:

```
pip install -r requirements.txt
gunicorn --bind 0.0.0.0:8000 app:app
```

Steps to setup our clusters using the load balancer have already been explained in section 1.2. As it was said earlier, everything has been automated in a script named deploy.sh. Simply run the follow-

ing command lines to deploy the clusters.

```
cd app/
./deploy.sh
```

Finally, out benchmarking app is located in the benchmarking folder. First, you need to have Docker and Docker Compose installed. Then, simply execute the following command lines.

```
cd benchmarking/
docker-compose up --build
```