

LOG8415
Advanced Concepts of Cloud Computing

TP1: Cluster Benchmarking using EC2 Virtual Machines and Elastic Load Balancer

October 18, 2022

Petr Šmejkal
Felipe Manoel
Koen van Oijen
Lucia Čahojová



Contents

1	Abstract	2
2	Introduction	2
3	Flask Application Deployment Procedure	2
4	Cluster Setup Using Application Load Balancer	3
5	Results of the Benchmark	3
5.1	Metrics choice	3
5.2	CPU utilization	3
5.3	Network In	5
5.4	Network Out	7
6	Instructions to Run the Code	8
7	Conclusion	9

1 Abstract

The aim of this paper is to learn about Cloud provider computing service and evaluation of the performance of used services. We launched virtual machines, created two clusters with different instance types, used load balancer to distribute workloads and analysed their performance.

2 Introduction

For this paper, the authors get introduced to the AWS (Amazon Web Services), especially the elastic compute cloud (EC2). This program lets developers use virtual machines (VMs) of Amazon and automatically scale up or down the resources when needed. The paper describes our process of experimenting and using this Amazon service.

Firstly, in an automated solution, two clusters using EC2 and elastic load balancers are created. On these services a flask application is deployed. Then the clusters performances are compared and results are visualised into graphs with Amazon Cloud Watch Metrics. Those graphs are analysed and concluded afterwards.

3 Flask Application Deployment Procedure

We used the lightweight Python web framework Flask, to deploy a simple application written in a single Python file. At first, we wrote a shell script, which executes the following actions:

- installs python3-pip
- installs python3-venv
- binds the Flask application to port 80
- creates directory for the Flask application [3]
- creates and activates new Python environment
- installs Flask
- creates Python file with Flask application code
- runs the Flask application

After that, we passed this shell script to each instance as *UserData* parameter. To the script itself, we passed the name of the target group, to make the application route to the specific URL.

The Flask application prints a simple message to inform the user which instance is currently in use, in the following format:

[VM RESPONSE] Flask app running on VM with ID \$instanceId is responding.

4 Cluster Setup Using Application Load Balancer

To distribute and scale the incoming workload across multiple targets, we created the elastic load balancer. We chose the *Application Load Balancer* and specified this in the *Type* parameter when creating the elastic load balancer.

Application Load Balancer serves as the single point of contact for the clients. It makes routing decisions at the application layer (HTTP/HTTPS), supports path-based routing and can route requests to one or more ports on each container instance in the cluster [2]. To increase the availability of the application we needed to distribute incoming workloads across two target groups and multiple instances assigned to them.

After creating the load balancer, we created *two Target Groups*, as proposed. First one was **cluster1**, for which we firstly created **four t2.large instances** (2 vCPU, 8 GiB of RAM, low to moderate network performance), waited until they were all running and then assigned them to the target group cluster1. After that, we repeated the process with **cluster2** and assigned **five m4.large instances** (2 vCPU, 8 GiB of RAM, moderate network performance) to it.

Once the target groups were created and all the instances were running, we created one *Listener* to check for connection requests from the clients, using **HTTP protocol** and **port 80**.

After that, we created one *rule* for each target group with **path-pattern** condition. Based on these listener rules, requests were forwarded to registered target groups.

5 Results of the Benchmark

5.1 Metrics choice

After performing the benchmark test described in the previous sections, the results are analyzed with Amazon Cloudwatch. Cloudwatch receives all the data from the instances and is able to show this to the developer [1]. It is able to gather all sorts of data, but for this paper, only some of them are relevant and will be discussed. The used metrics are shown and explained in the table below.

Metric	Explanation
CPU Utilization	the percent of CPU is being used for every instance
Network In	number of bytes received from the load balancers
Network Out	number of bytes sent by the instances

The first metric is about the instance. It gives the percentage of the CPU per instance that is being used by the processes. It indicates if the instances will get overloaded. The second metric shows the number of bytes that it received and the third will represent the amount of bytes it sends away.

These metrics will benchmark the performance of the two different clusters. The m4.large cluster and t2.large cluster. For the m4.large cluster, we ran 5 instances and therefore for every metric 5 different graphs will be shown. As for the t2.large cluster only 4 instances had ran.

5.2 CPU utilization

Below you can find all 9 plots of the CPU utilization over time. As you can see the task started on 20:21. 8 of the 9 follow a hockey stick pattern. Starting with their maximum utilization and then

decrease to a 0-10 percent.

Then it depended on the situation for all m4.large instances, the utilization dropped even further and the t2.large increased one more time. All patterns happened around the same time. One more thing that must be noted is that the utilization share of the m4.large instances is much higher (all around 50 percent), compared to the t2.large (around 25 percent).

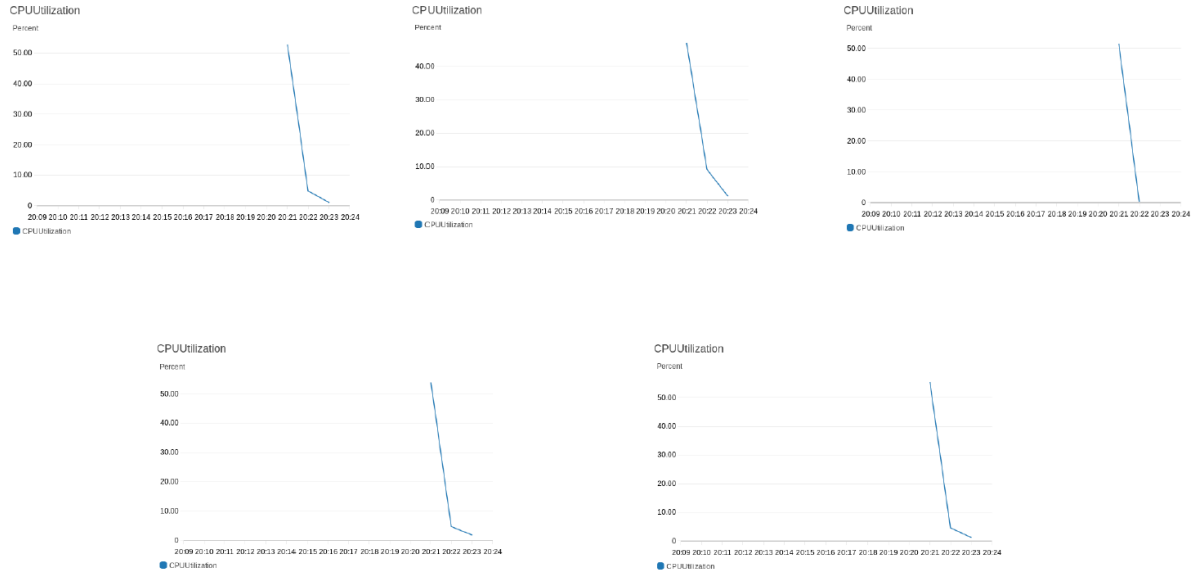


Figure 1: CPU utilization of m4.Large cluster



Figure 2: CPU utilization of t2.Large cluster

5.3 Network In

Below you can find another comparison of the two clusters. Again, the hockey stick shape can be found as before. This is not strange since CPU utilization and Network In are closely related to each other. From the numbers, we see that the t2.Large cluster is using only 800k bytes per instance and the m4.Large cluster are all getting around 80M bytes in. Which is 100 times more than the t2.large cluster receives.



Figure 3: Network in of m4.Large cluster

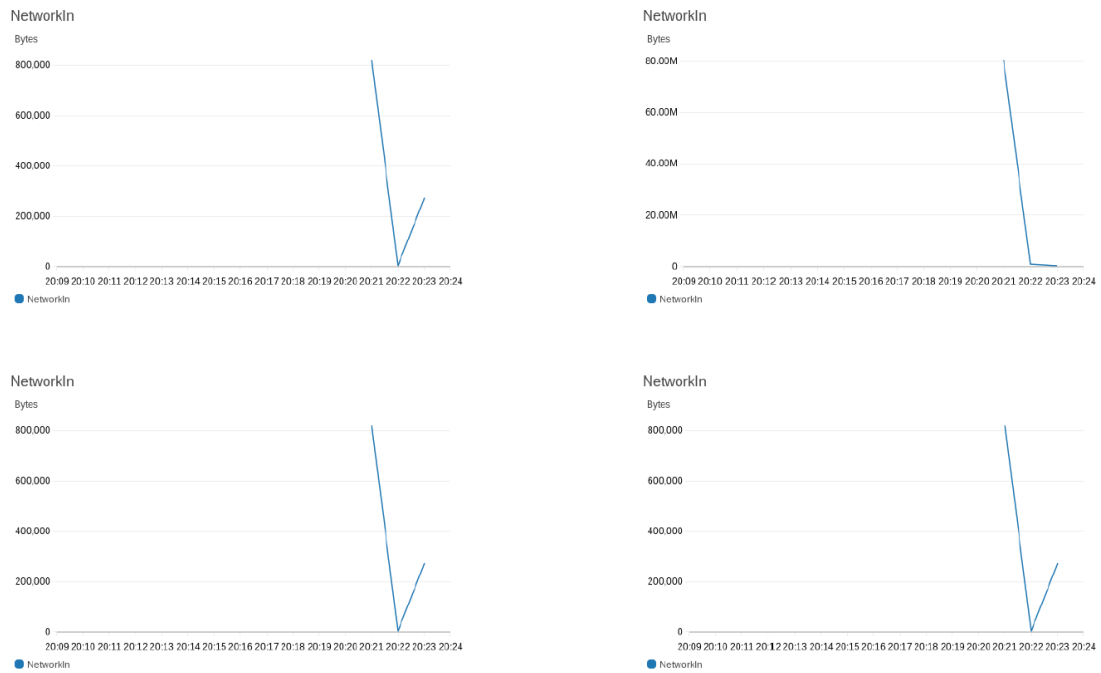


Figure 4: Network in of t2.Large cluster

5.4 Network Out

This is the first metric where the pattern change between the two cluster. With previous cluster only the numbers changed, but as you can see in the table below, the shape changed for the t2.large. The shape suddenly changed to a reverted hockey stick. So first it sends around 25000 bytes, then even less, but eventually increases to 200000 bytes per second.

However, the m4.large starts with their maximum around 500000 bytes per second rate and then drastically decreases to ± 25000 bytes and eventually goes up to 200000 bytes.



Figure 5: Network in of m4.Large cluster

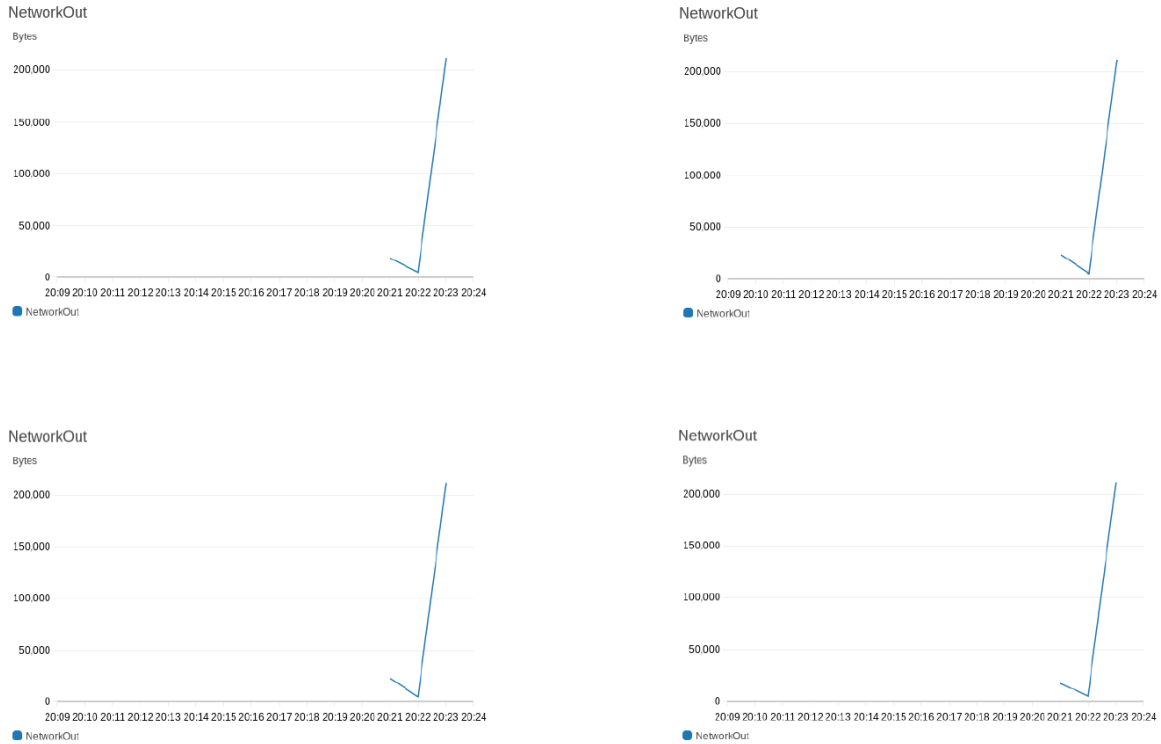


Figure 6: Network in of t2.Large cluster

6 Instructions to Run the Code

In this section, we provide the steps with commands to run our automated procedure:

1. Configure AWS Credentials on your computer:
 - (a) Open the file with AWS credentials:
`code ~/.aws/credentials`
 - (b) Navigate to your AWS Academy account, then to AWS Details and copy your AWS CLI information.
 - (c) Paste the AWS CLI information to the opened credentials file and save and close the file.
2. Clone the GitHub repository to your desired location:
`git clone https://github.com/smejkalpetr/cc-tp1.git`
3. Proceed to the project's directory:
`cd cc-tp1`
4. Run the automated application:
`./setup.sh auto`
5. See the results in the `./out` directory.

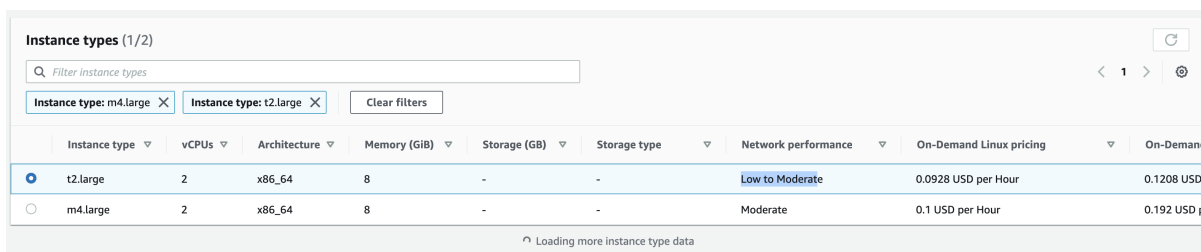
7 Conclusion

In this paper, we explained how we created a load balancer and instances on the AWS. We successfully benchmarked 2 different clusters with two different types of instances: the m4.large and t2.large.

We compared them over their CPU utilization, network traffic inbound and outbound. From this, we could see that the m4.large cluster handled way more traffic than the t2.large cluster and also had twice as much CPU utilization. According to AWS documentation, they both have the same memory size and CPU size.

However, they differ mostly in network performance, which is better for the m4.large instance type. Hence, the load balancer chooses quicker for this cluster. In addition to this, the cluster with m4.large instances was bigger (5 instances instead of 4) enabling this cluster to handle more traffic.

It also needs to be noted that the m4.large is 0.0072 USD per hour more expensive, which is a small difference in price. Yet, the inbound and outbound traffic is much better than the t2.large.



The screenshot shows the AWS Instance Types comparison interface. At the top, there's a search bar and filters for 'Instance type: m4.large' and 'Instance type: t2.large'. Below the filters is a table with columns: Instance type, vCPUs, Architecture, Memory (GiB), Storage (GB), Storage type, Network performance, On-Demand Linux pricing, and On-Demand pricing. The table lists two instance types: t2.large and m4.large. The t2.large instance has 2 vCPUs, x86_64 architecture, 8 GiB memory, and 'Low to Moderate' network performance. The m4.large instance has 2 vCPUs, x86_64 architecture, 8 GiB memory, and 'Moderate' network performance. The On-Demand Linux pricing for t2.large is 0.0928 USD per Hour, and for m4.large it is 0.1 USD per Hour. The On-Demand pricing for t2.large is 0.1208 USD per Hour, and for m4.large it is 0.192 USD per Hour.

Instance type	vCPUs	Architecture	Memory (GiB)	Storage (GB)	Storage type	Network performance	On-Demand Linux pricing	On-Demand pricing
t2.large	2	x86_64	8	-	-	Low to Moderate	0.0928 USD per Hour	0.1208 USD per Hour
m4.large	2	x86_64	8	-	-	Moderate	0.1 USD per Hour	0.192 USD per Hour

Figure 7: t2.large and m4.large comparison according AWS

References

- [1] <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-cloudwatch-metrics.html>
- [2] <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>
- [3] <https://gist.github.com/justinmklam/f13bb53be9bb15ec182b4877c9e9958d>