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

Written by:
Daníel Jóhannsson - 2311176
Vanessa Nguyen - 2313156
Danilo Vaz - 2313099
Sofia Canas - 2313066

October 8, 2023

1 Flask Application Deployment Procedure

In this assignment we were required deploy a flask application on each EC2 instance launched. The flask application's task is to upon receiving a request provide the id of the EC2 instance handling the request. This way we can ensure that the Load Balancer and the clusters work as intended.

In order to deploy the flask applications on the virtual machines we pass along a bash script. This is done by passing along a function in the UserData-field that runs the bash script upon creation of the EC2 instance. This script updates the operative system, installs pip3 and the required python packages. Lastly after all the configuration is done, the script deploys the flask application on port 80. The flask application is of simple nature and uses the EC2 metadata libarary to return the id of the EC2 instance.

2 Cluster setup using Application Load Balancer

The setup starts with the AWS clients being initialized for Elastic Compute Cloud (EC2) and Elastic Load Balancing v2 (ELBv2) services. All the setup was automatized using a Python script and the boto3 library, so for this first step we used the *client()* function with the type of service and *region_name* as parameters.

Then, we create one EC2 security group with the name "1st-Tp-sg3" using the *create_sg()* function, which takes the EC2 client as a parameter and returns the security group ID. We also create a subnet for launching the instances across multiple availability zones with *fetch_subnet()*, wich returns a list of subnets. After that, we sequentially create and configure the load balancer architecture:

2.1 Launch instances

We create nine EC2 instances in total: five in the first cluster of the m4.large type and four in the second cluster of the t2.large type. For doing this, the function *EC2_instances()* takes the parameters:

- avZones: The list of availability zones.
- ec2_client: The EC2 client.
- sgId: The security group ID.

It returns:

- instancesIds: The list of IDs of the instances.
- KPName: The key pair name, wich is set to be '1st-assign-key3' in our code.

After that, the instances will be arranged in two separate clusters by creating target groups.

2.2 Load balancer and target groups creation

To create the load balancer, target groups, listener and routing rules at the same time, the *create_lb_listener()* takes these parameters:

- subnets: The list of subnets.
- ec2_client: The EC2 client.
- elb_client: The ELB client.
- sgId: The security group ID.

Internally, the first thing this function does is fetching the Virtual Private Cloud (VPC), then passing it to *create_lb_tg()*, among with the target group names each time, 'cluster1-1' and 'cluster2-2'. A load balancer is created, using the fetched subnets, the security group and the name 'LoadBalancer'. By default, it is an Application Load Balancer (ALB). A listener is attached to the load balancer and two rules are created for the listener:

- Requests with the path /1 are routed to cluster1-1.
- Requests with the path /2 are routed to cluster2-2.

Finally, this *create_lb_listener()* function returns:

- lbArn: The list of subnets.
- tg1Arn: Amazon Resource Name for the first target group.
- tg2Arn: Amazon Resource Name for the second target group.
- lbDNS: Domain Name System of the load balancer.

2.3 Attach instances to the target groups

The five instances of type m4.large are registered to the first target group cluster1-1. The remaining four EC2 instances of type t2.large are registered to the second target group cluster2-2. The function *attach_instances()* uses:

- tg1: The first target group ARN.
- tg2: The second target group ARN.
- elb_client: The ELB client.
- instancesIds: The list of IDs of the instances.
- port: The port on which each service within the target group is set to receive traffic, we use port 80.

At the end, we can represent the setup with the diagram in Figure 1:

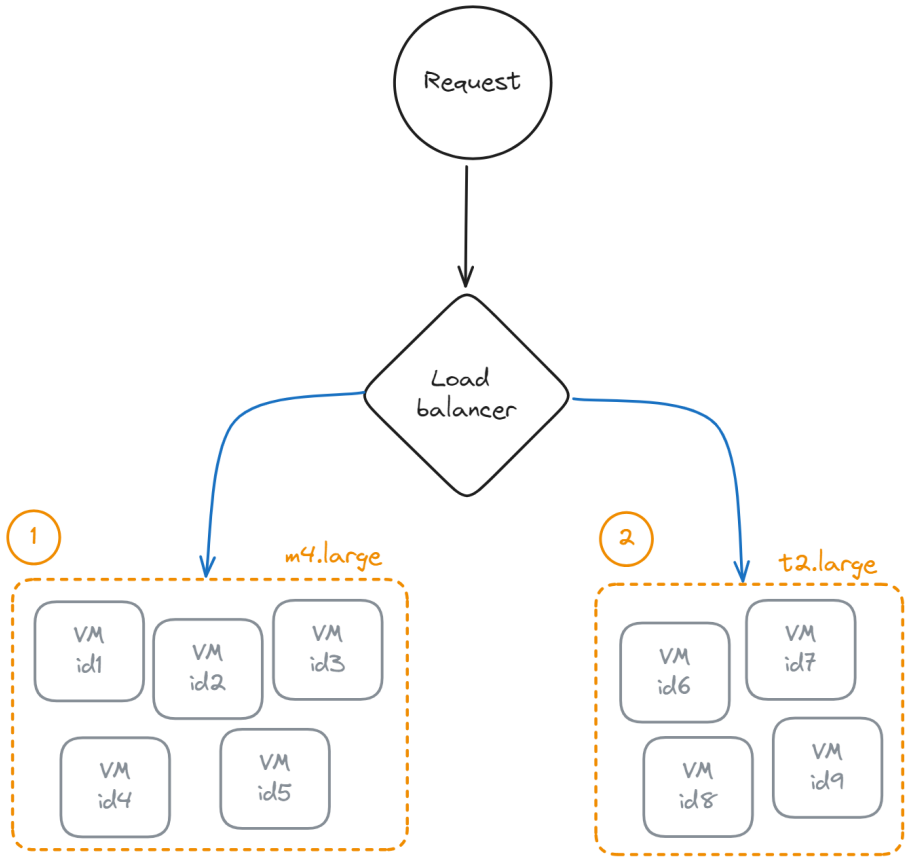


Figure 1: Load Balancer Architecture

3 Results of the benchmark

After finishing the implementation of the clusters and load balancer, we benchmarked each cluster and compared their performance. Also, the performance of the EC2 instances was benchmarked.

The test consisted of two threads running at the same time, sending requests to each cluster. One thread sending 1000 requests and the other sending 500 requests, waiting for one minute and then sending 1000 more requests. This test was done for both clusters and tracked using CloudWatch.

We chose to track the following metrics:

Metric	**Unit**	**Description**
HTTPCode_ELB_4XX_Count, HTTPCode_ELB_5XX_Count	Sum count	The number of HTTP 4XX/5XX client error codes that originate from the load balancer. This count does not include response codes generated by targets [1]
RequestCountPerTarget	Sum	The average number of requests received by each target in a target group [1]
TargetResponseTime	Average seconds	The time elapsed, in seconds, after the request leaves the load balancer until a response from the target is received [1]
HTTPCode_Target_2XX_Count, HTTPCode_Target_4XX_Count	Sum count	The number of HTTP response codes generated by the targets. This does not include any response codes generated by the load balancer [1]
CPUUtilization	Percent	The percentage of physical CPU time that Amazon EC2 uses to run the EC2 instance, which includes time spent to run both the user code and the Amazon EC2 code.[2]

To run the test, the function *requests_main()* takes the DNS of the load balancer and internally runs both threads. After that, *benchmarks_analysis.main()* and *benchmarks_analysis.instances_metrics()* will return dictionaries with the load balancer and instances metrics.

In the subsequent sections, we present the values recorded for each metric, accompanied by visual representations of the data. A close examination reveals that the data is consistent with the graphs.

3.1 HTTPCode_ELB_4XX_Count

This metric and the next one report client error codes that originate from the load balancer, so it makes sense to have an empty answer, the reporting criteria is a nonzero value.

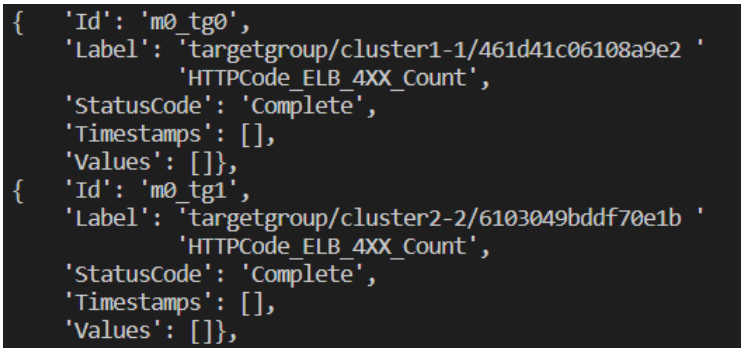


Figure 2: HTTPCode_ELB_4XX_Count values

3.2 HTTPCode_ELB_5XX_Count

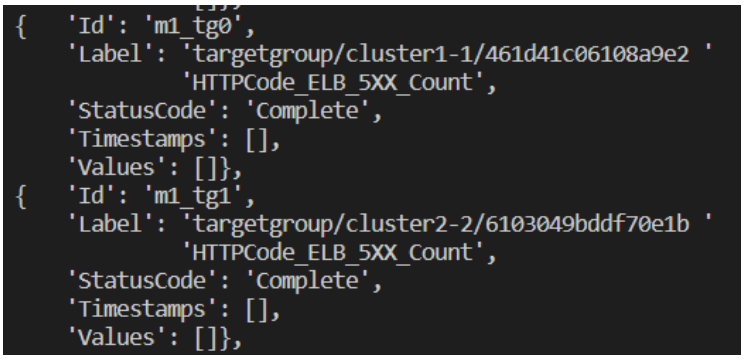


Figure 3: HTTPCode_ELB_5XX_Count values

3.3 RequestCountPerTarget

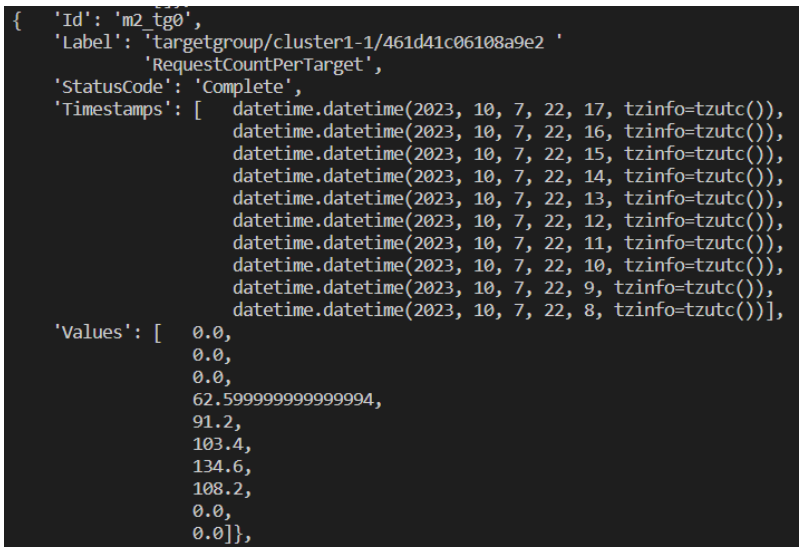


Figure 4: RequestCountPerTarget values for the first cluster

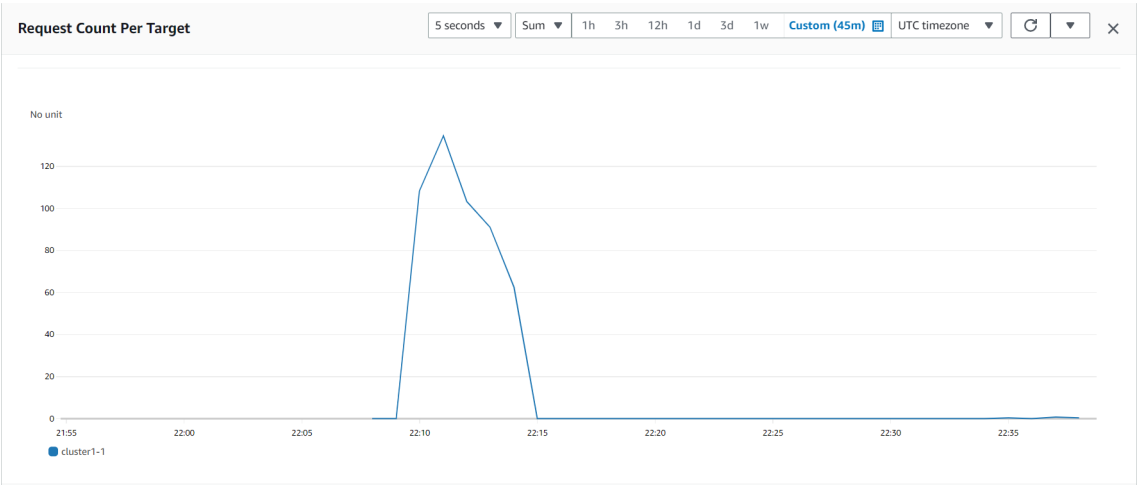


Figure 5: RequestCountPerTarget visualization for the first cluster

```
{   'Id': 'm2_tg1',
    'Label': 'targetgroup/cluster2-2/6103049bddf70e1b '
             'RequestCountPerTarget',
    'StatusCode': 'Complete',
    'Timestamps': [   datetime.datetime(2023, 10, 7, 22, 17, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 16, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 15, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 14, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 13, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 12, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 11, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 10, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 9, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 8, tzinfo=tzutc())],
    'Values': [   0.0,
                  0.0,
                  0.0,
                  78.5,
                  113.75,
                  129.25,
                  168.25,
                  135.25,
                  0.0,
                  0.0]},
```

Figure 6: RequestCountPerTarget values for the second cluster

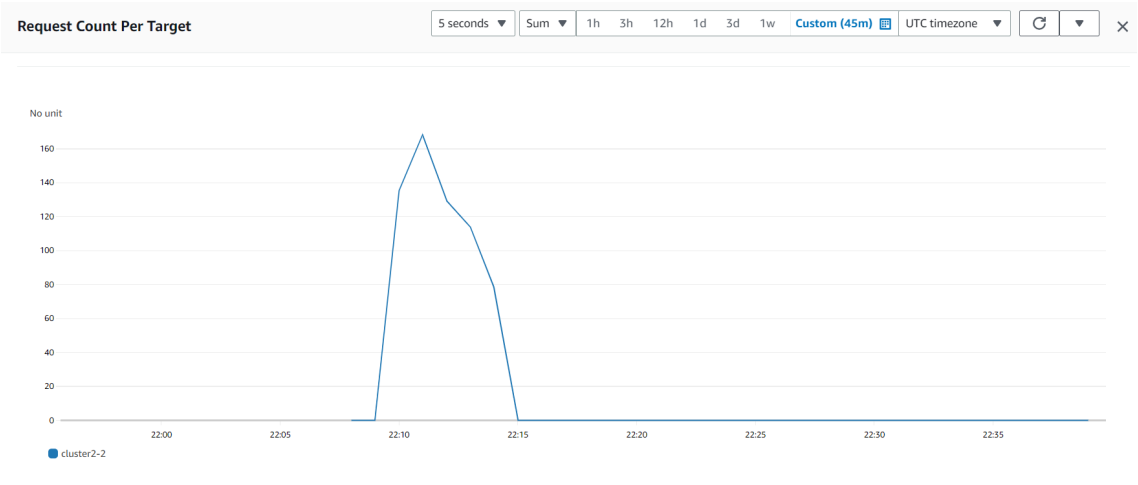


Figure 7: RequestCountPerTarget visualization for the second cluster

We can observe that the request counts per target appear consistent across both clusters, indicating that the load balancer distributed the workload evenly to the target groups.

3.4 TargetResponseTime

```
{   'Id': 'm3_tg0',
    'Label': 'targetgroup/cluster1-1/461d41c06108a9e2 TargetResponseTime',
    'StatusCode': 'Complete',
    'Timestamps': [   datetime.datetime(2023, 10, 7, 22, 14, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 13, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 12, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 11, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 10, tzinfo=tzutc())],
    'Values': [   0.0018639041533546326,
                  0.0019082807017543862,
                  0.0019551237911025147,
                  0.001956392273402675,
                  0.0020953068391866914]},
{   'Id': 'm3_tg1',
    'Label': 'targetgroup/cluster2-2/6103049bddf70e1b TargetResponseTime',
    'StatusCode': 'Complete',
    'Timestamps': [   datetime.datetime(2023, 10, 7, 22, 14, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 13, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 12, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 11, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 10, tzinfo=tzutc())],
    'Values': [   0.002266493630573248,
                  0.002359923076923077,
                  0.0026941972920696325,
                  0.0024884323922734026,
                  0.0026660924214417744]},
```

Figure 8: TargetResponseTime values

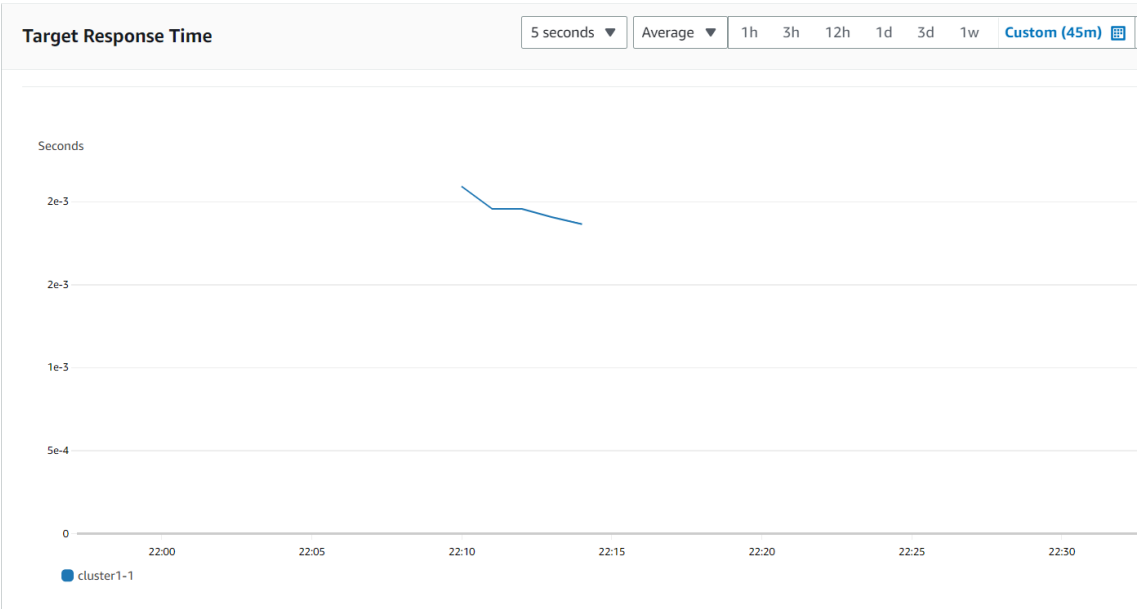


Figure 9: TargetResponseTime values for the first cluster

According to Figure 9 and 10, the second cluster has a slightly higher target response time. There is a peak in the middle of this cluster graph also, while the first cluster shows a descending behavior.

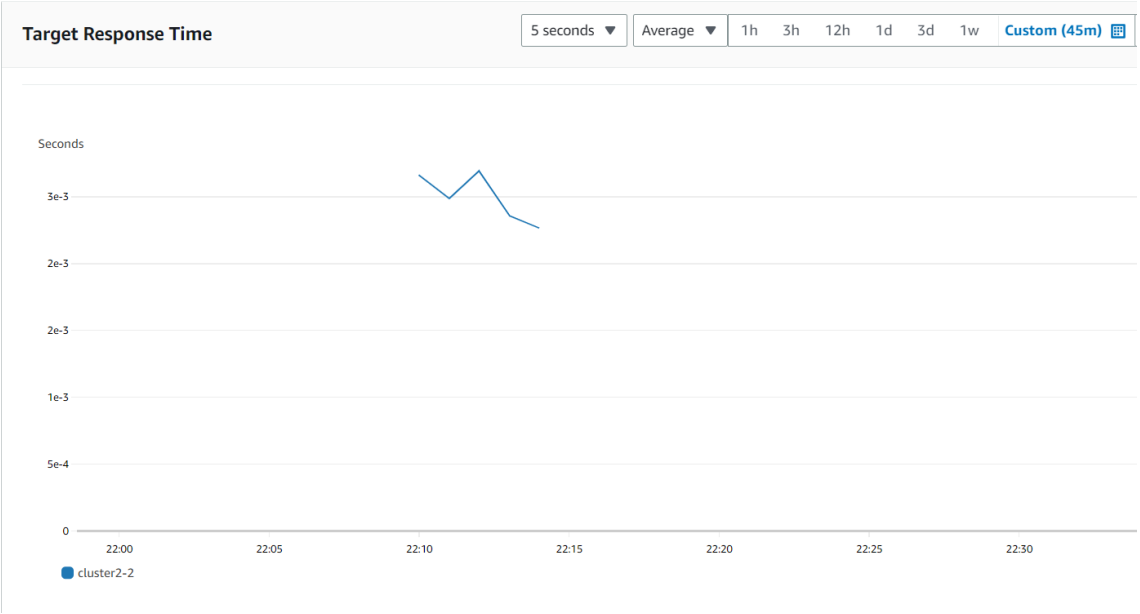


Figure 10: TargetResponseTime values for the second cluster

3.5 HTTPCode_Target_2XX Count

```
{   'Id': 'm4_tg0',
    'Label': 'targetgroup/cluster1-1/461d41c06108a9e2 '
             'HTTPCode_Target_2XX_Count',
    'StatusCode': 'Complete',
    'Timestamps': [   datetime.datetime(2023, 10, 7, 22, 14, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 13, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 12, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 11, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 10, tzinfo=tzutc())],
    'Values': [313.0, 456.0, 517.0, 673.0, 541.0]},
{   'Id': 'm4_tg1',
    'Label': 'targetgroup/cluster2-2/6103049bddf70e1b '
             'HTTPCode_Target_2XX_Count',
    'StatusCode': 'Complete',
    'Timestamps': [   datetime.datetime(2023, 10, 7, 22, 14, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 13, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 12, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 11, tzinfo=tzutc()),
                      datetime.datetime(2023, 10, 7, 22, 10, tzinfo=tzutc())],
    'Values': [314.0, 455.0, 517.0, 673.0, 541.0]},
```

Figure 11: HTTPCode_Target_2XX Count values

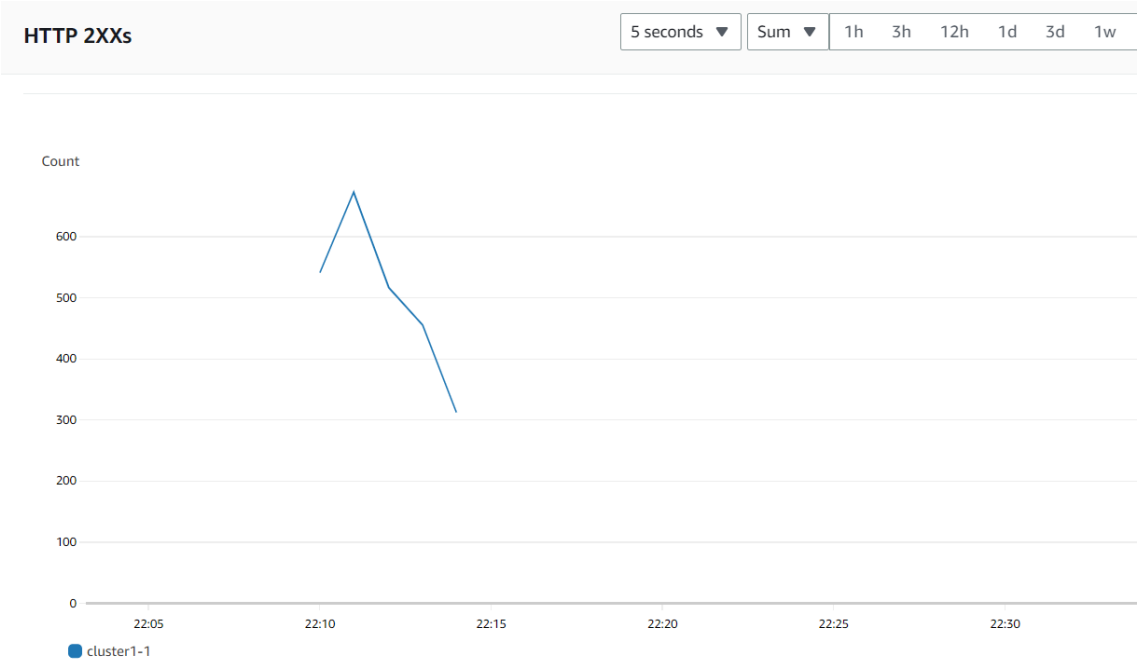


Figure 12: HTTPCode_Target_2XX Count visualization for the first cluster

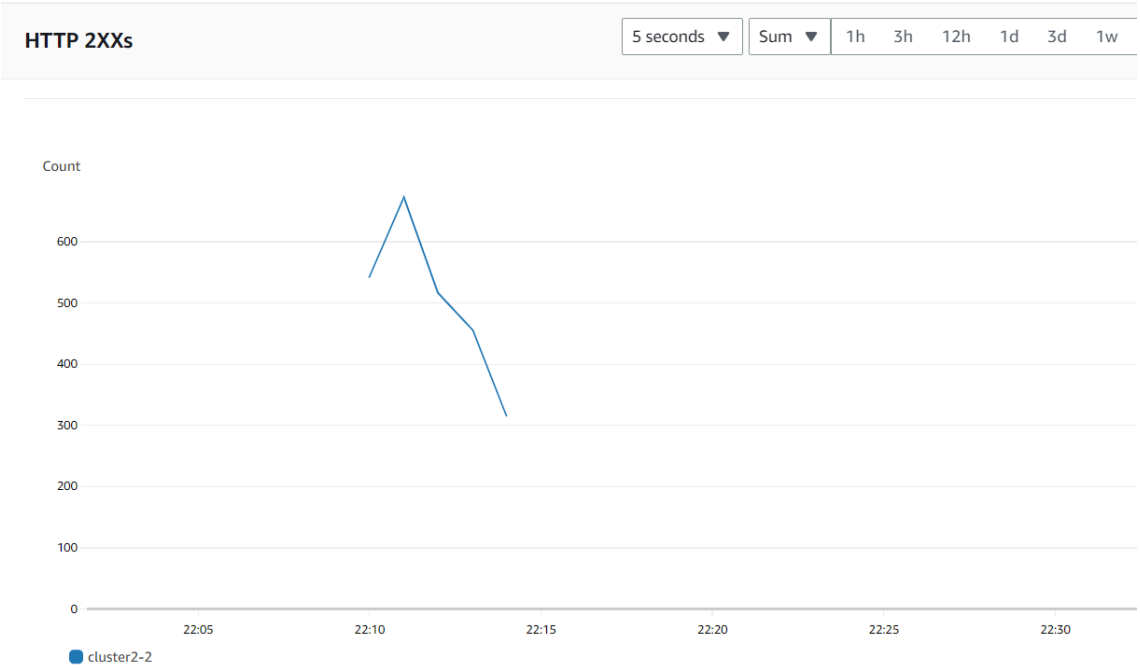


Figure 13: HTTPCode_Target_2XX Count visualization for the second cluster

The number of HTTP 2XX response codes generated by the targets. In this case, a 200 response is a success message, so the data is similar to RequestCountPerTarget but times the number of instances.

3.6 HTTPCode_Target_4XX Count

The results are empty as expected.

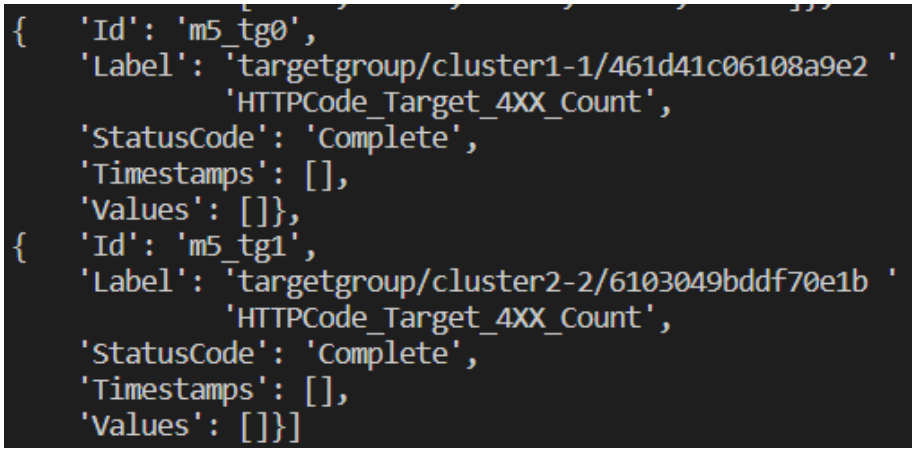
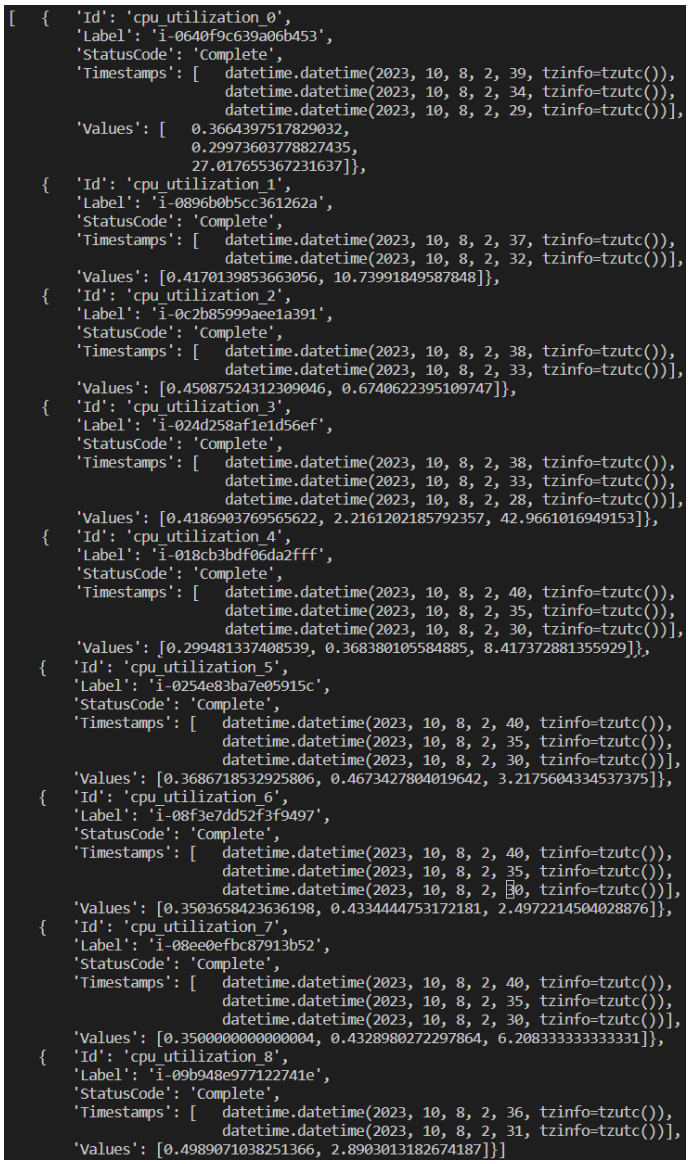


Figure 14: HTTPCode_Target_4XX Count values

All the above are load balancer metrics, we selected a metric also for the instances, and is presented in the next section.

3.7 CPUUtilization



```
[   {   'Id': 'cpu_utilization_0',
        'Label': 'i-0640f9c639a06b453',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 39, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 34, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 29, tzinfo=tzutc())],
        'Values': [   0.3664397517829032,
                      0.2997360377882743 5,
                      27.017655367231637]},
    {   'Id': 'cpu_utilization_1',
        'Label': 'i-0896b0b5cc361262a',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 37, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 32, tzinfo=tzutc())],
        'Values': [0.4170139853663056, 10.73991849587848]},
    {   'Id': 'cpu_utilization_2',
        'Label': 'i-0c2b85999aee1a391',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 38, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 33, tzinfo=tzutc())],
        'Values': [0.45087524312309046, 0.674062239510974 7]},
    {   'Id': 'cpu_utilization_3',
        'Label': 'i-024d258af1e1d56ef',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 38, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 33, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 28, tzinfo=tzutc())],
        'Values': [0.4186903769565622, 2.2161202185792357, 42.9661016949153]},
    {   'Id': 'cpu_utilization_4',
        'Label': 'i-018cb3bdf06da2fff',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 40, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 35, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 30, tzinfo=tzutc())],
        'Values': [0.299481337408539, 0.368380105584885, 8.417372881355929]},
    {   'Id': 'cpu_utilization_5',
        'Label': 'i-0254e83ba7e05915c',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 40, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 35, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 30, tzinfo=tzutc())],
        'Values': [0.3686718532925806, 0.4673427804019642, 3.2175604334537375]},
    {   'Id': 'cpu_utilization_6',
        'Label': 'i-08f3e7dd52f3f9497',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 40, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 35, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 30, tzinfo=tzutc())],
        'Values': [0.3503658423636198, 0.4334444753172181, 2.4972214504028876]},
    {   'Id': 'cpu_utilization_7',
        'Label': 'i-08ee0efbc87913b52',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 40, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 35, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 30, tzinfo=tzutc())],
        'Values': [0.3500000000000004, 0.4328980272297864, 6.208333333333331]},
    {   'Id': 'cpu_utilization_8',
        'Label': 'i-09b948e977122741e',
        'StatusCode': 'Complete',
        'Timestamps': [   datetime.datetime(2023, 10, 8, 2, 36, tzinfo=tzutc()),
                          datetime.datetime(2023, 10, 8, 2, 31, tzinfo=tzutc())],
        'Values': [0.4989071038251366, 2.8903013182674187]}]
```


Figure 15: CPUUtilization values for the instances

We present a visualization of the data for an instance in each cluster in Figure 16 and 17. We observe that the instance in the first cluster has a much higher CPU utilization. This could be representative of the cluster or not, it requires a more exhaustive analysis, but in the data we can see very diverse values for all the instances and target groups.

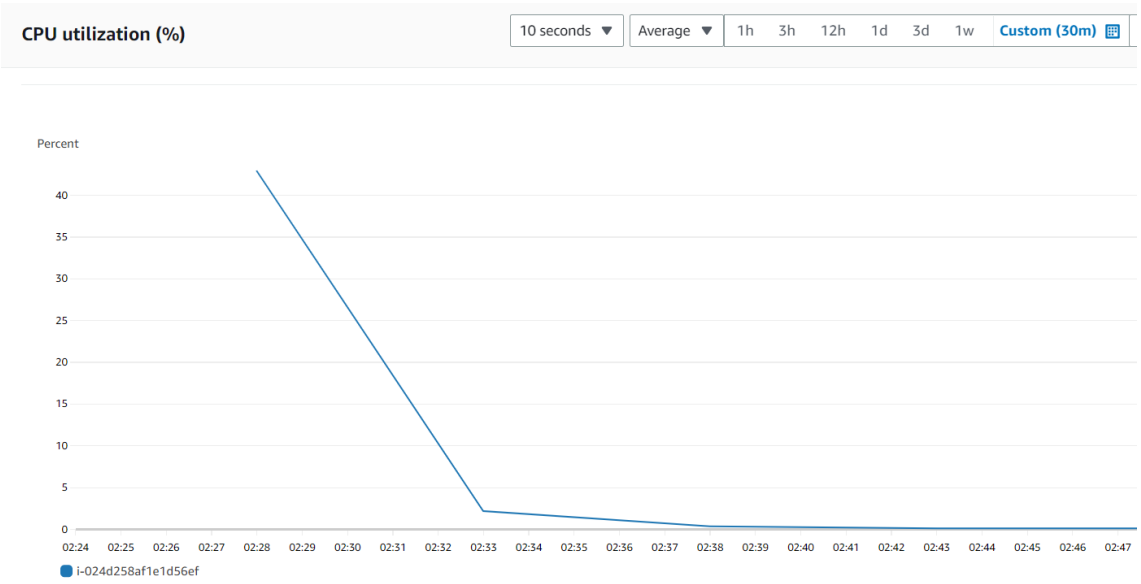


Figure 16: CPUUtilization visualization for an instance of type m4.large in the first cluster (id i-024d258af1e1d56ef)

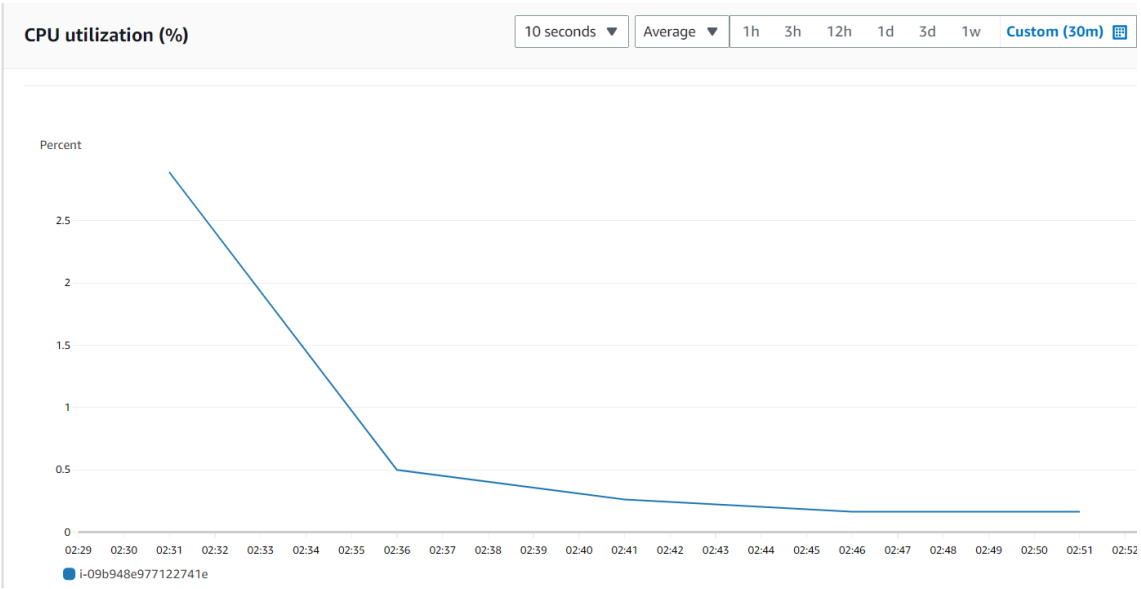


Figure 17: CPUUtilization visualization for an instance of type t2.large in the second cluster (id i-09b948e977122741e)

4 Instructions to run the code

- A linux operative system is **REQUIRED**
- Clone repo from Github
- Download Docker
- The following values need to be stored in ∼/.aws/credentials:
  - AWS_ACCESS_KEY_ID
  - AWS_SECRET_ACCESS_KEY
  - AWS_SESSION_TOKEN

Once all that is done we locate the file named "scripts.sh", then run the following commands.

```
#!/bin/bash

# Building Docker
sudo docker build -t 1st-assign-img .

# Running Docker passing the credential use your's credential file path
sudo docker run -v $HOME/.aws/credentials:/root/.aws/credentials:ro 1st-assign-img
```

Listing 1: Bash script to run the application

This builds and runs our docker image which in turn copies over the files and installs the dependencies, as well as running the python Main. Now the clusters are being setup and should be up and running in a few minutes.

References

[1] Cloudwatch metrics for your application load balancer. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-cloudwatch-metrics.html. Accessed: 2023-10-06.

[2] List the available cloudwatch metrics for your instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html. Accessed: 2023-10-06.