POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

LOG8415

Advanced Concepts of Cloud Computing

Cluster Benchmarking using EC2 Virtual Machines and Elastic Load Balancer

Team 6

Ana Karen López Baltazar 2311301

Ali Hazime 2270891

Lucy Nguyen 2089225

Majdi Saghrouni 1733556

October 7, 2023

Github repo: https://github.com/LuuN2122/LOG8415

Submitted to : Vahid Majdinasab

Flask Application Deployment Procedure

To deploy the Flask web application, we may use one of two methods. In our case, we applied the second method, i.e., the automatic deployment procedure.

1- Manual deployment (not used)

- After the creation of your EC2 instance, copy the bash script to your remote instance.

```
scp -i <pem-key>.pem deploy_flask_app.sh remoteuser@remotehost:
/remote/directory
```

- Connect to your remote machine via ssh

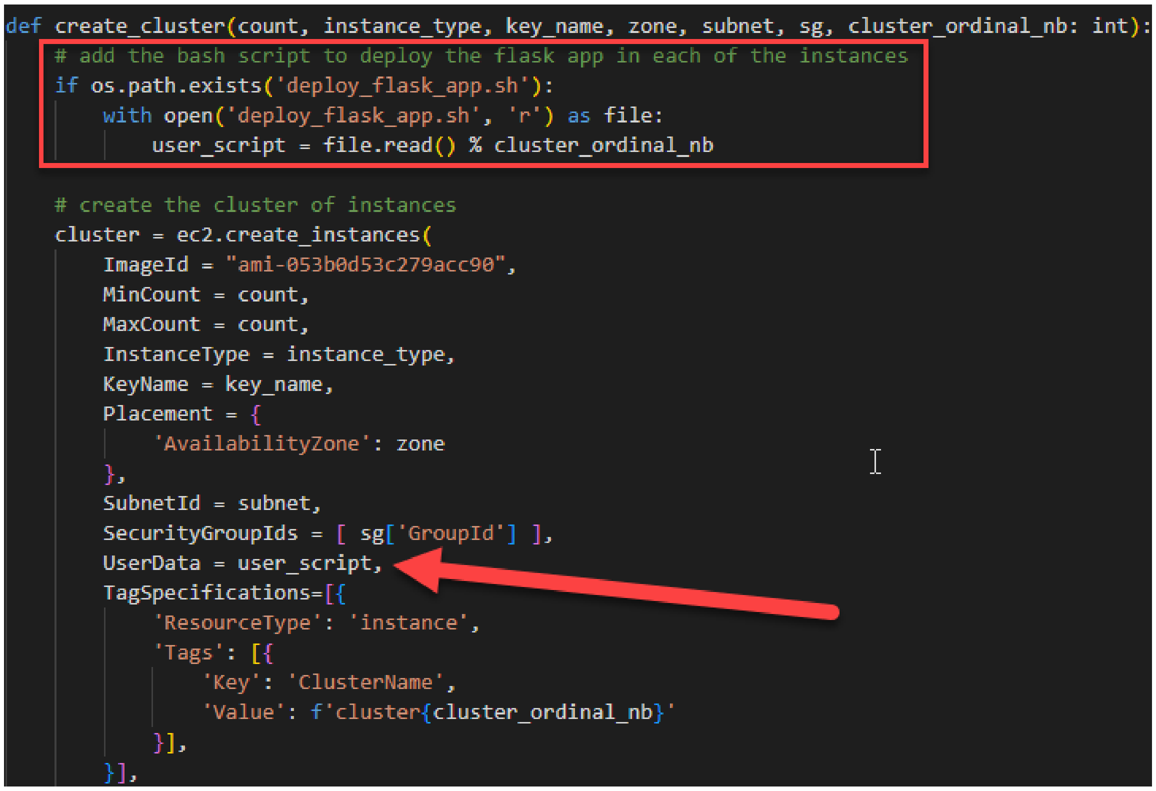
```
ssh -i <pem-key>.pem remoteuser@<public-ip>
```

- Execute the bash script

```
bash deploy_flask_app.sh
```

2- Automatic deployment (used)

Pass the Bash script containing the flask app deployment as a variable when creating the EC2 instances. The *UserData* parameter in the *ec2.create_instances(**kwargs)* method can be employed for this purpose, as demonstrated in the screenshot on the next page.



```python
def create_cluster(count, instance_type, key_name, zone, subnet, sg, cluster_ordinal_nb: int):
    # add the bash script to deploy the flask app in each of the instances
    if os.path.exists('deploy_flask_app.sh'):
        with open('deploy_flask_app.sh', 'r') as file:
            user_script = file.read() % cluster_ordinal_nb

    # create the cluster of instances
    cluster = ec2.create_instances(
        ImageId = "ami-053b0d53c279acc90",
        MinCount = count,
        MaxCount = count,
        InstanceType = instance_type,
        KeyName = key_name,
        Placement = {
            'AvailabilityZone': zone
        },
        SubnetId = subnet,
        SecurityGroupIds = [ sg['GroupId'] ],
        UserData = user_script,
        TagSpecifications=[{
            'ResourceType': 'instance',
            'Tags': [{
                'Key': 'ClusterName',
                'Value': f'cluster{cluster_ordinal_nb}'
            }],
        }],
```

Cluster setup using Application Load Balancer

First, we created five instances of type ‘t2.large’ in the availability zone 'us-east-1a' and four instances of type ‘m4.large’ in the availability zone ‘us-east-1b’. At the moment of creation, we gave all nine instances a customized key pair and security group to prevent unauthorized access.

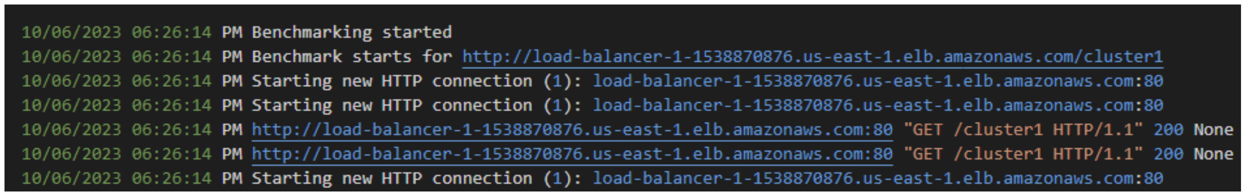
Then, we configured an application load balancer with a listener on Port 80 using the HTTP protocol. We organized our instances into two target groups, each representing a cluster. The load balancer directs traffic to these target groups based on the path patterns: the first target group is for paths having the pattern “/cluster1” while the second one is for paths having the pattern “/cluster2”. The default action of the listener, which is the ‘Bad Request’ message having status code 400, is triggered when the path sent to the listener does not include any of those patterns.

For your reference, the source code for the clusters and load balancer setup can be found in the python script *aws_setup.py*.

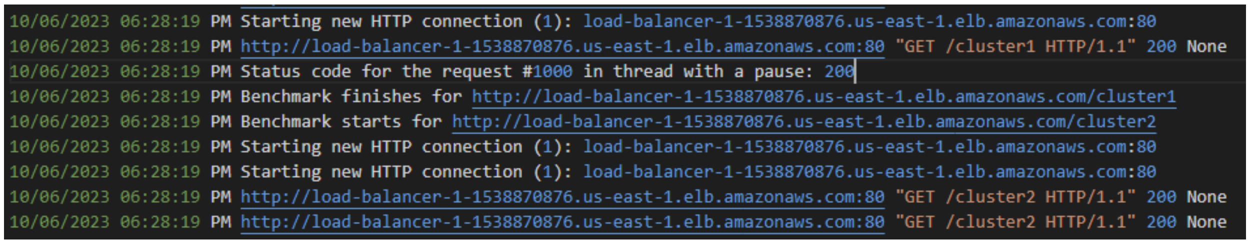
Benchmark Results

Load balancer metrics:

In the graph of RequestCount, it was observed that the load balancer started receiving requests for the first cluster (/cluster1) at 18:26, which is coherent with the time the first request was sent, as indicated in the benchmark.log. At 18:28, we can see that there is a second spike, indicating the start of the benchmark for the second cluster (/cluster2). After comparing the benchmark.log with the graph, we observe that the timestamps in the benchmark.log are matching with those in the graph. Similarly, the graph HTTPCode_Target_2XX_Count, representing the count of HTTP responses with a status of 200 shows a synchronized pattern with the timestamps from the benchmark.log. Moreover, the graph NewConnectionCount, illustrating the total number of new TCP connections established from clients to the load balancer and from the load balancer to targets, follows the same pattern. As anticipated, all graphs display a similar shape, showing the load balancer's immediate response when receiving a request.

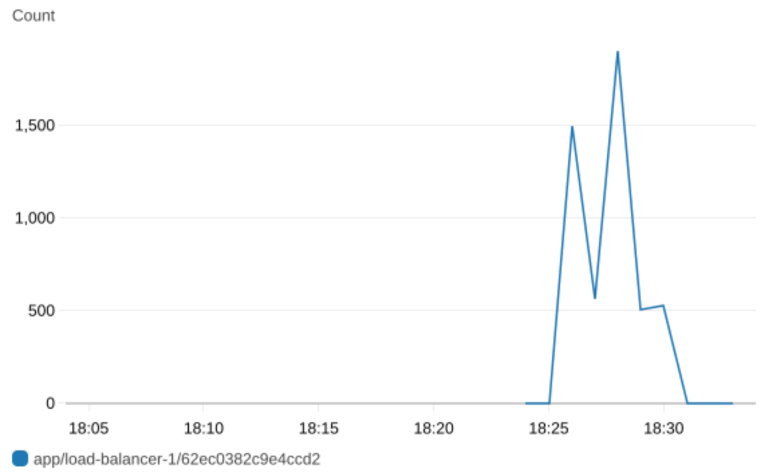


```
10/06/2023 06:26:14 PM Benchmarking started
10/06/2023 06:26:14 PM Benchmark starts for http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com/cluster1
10/06/2023 06:26:14 PM Starting new HTTP connection (1): load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80
10/06/2023 06:26:14 PM Starting new HTTP connection (1): load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80
10/06/2023 06:26:14 PM http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80 "GET /cluster1 HTTP/1.1" 200 None
10/06/2023 06:26:14 PM http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80 "GET /cluster1 HTTP/1.1" 200 None
10/06/2023 06:26:14 PM Starting new HTTP connection (1): load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80
```

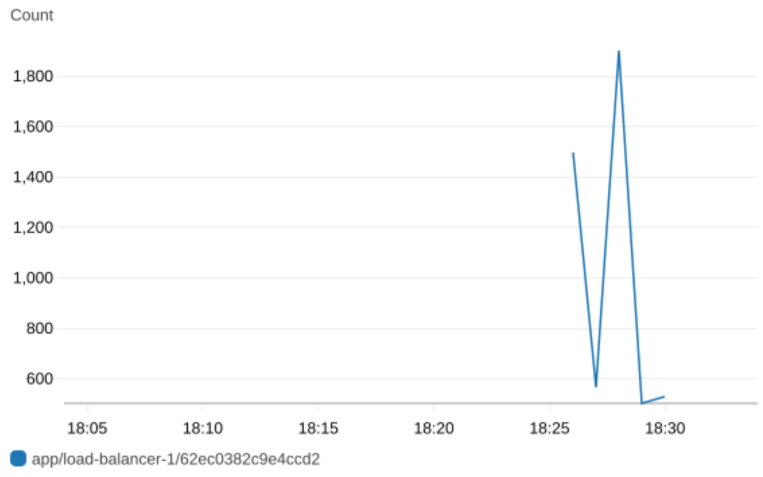


```
10/06/2023 06:28:19 PM Starting new HTTP connection (1): load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80
10/06/2023 06:28:19 PM http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80 "GET /cluster1 HTTP/1.1" 200 None
10/06/2023 06:28:19 PM Status code for the request #1000 in thread with a pause: 200
10/06/2023 06:28:19 PM Benchmark finishes for http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com/cluster1
10/06/2023 06:28:19 PM Benchmark starts for http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com/cluster2
10/06/2023 06:28:19 PM Starting new HTTP connection (1): load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80
10/06/2023 06:28:19 PM Starting new HTTP connection (1): load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80
10/06/2023 06:28:19 PM http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80 "GET /cluster2 HTTP/1.1" 200 None
10/06/2023 06:28:19 PM http://load-balancer-1-1538870876.us-east-1.elb.amazonaws.com:80 "GET /cluster2 HTTP/1.1" 200 None
```

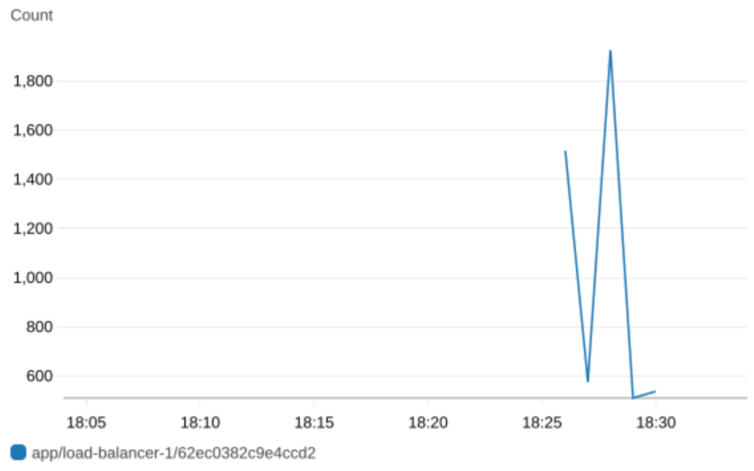
Graph of RequestCount



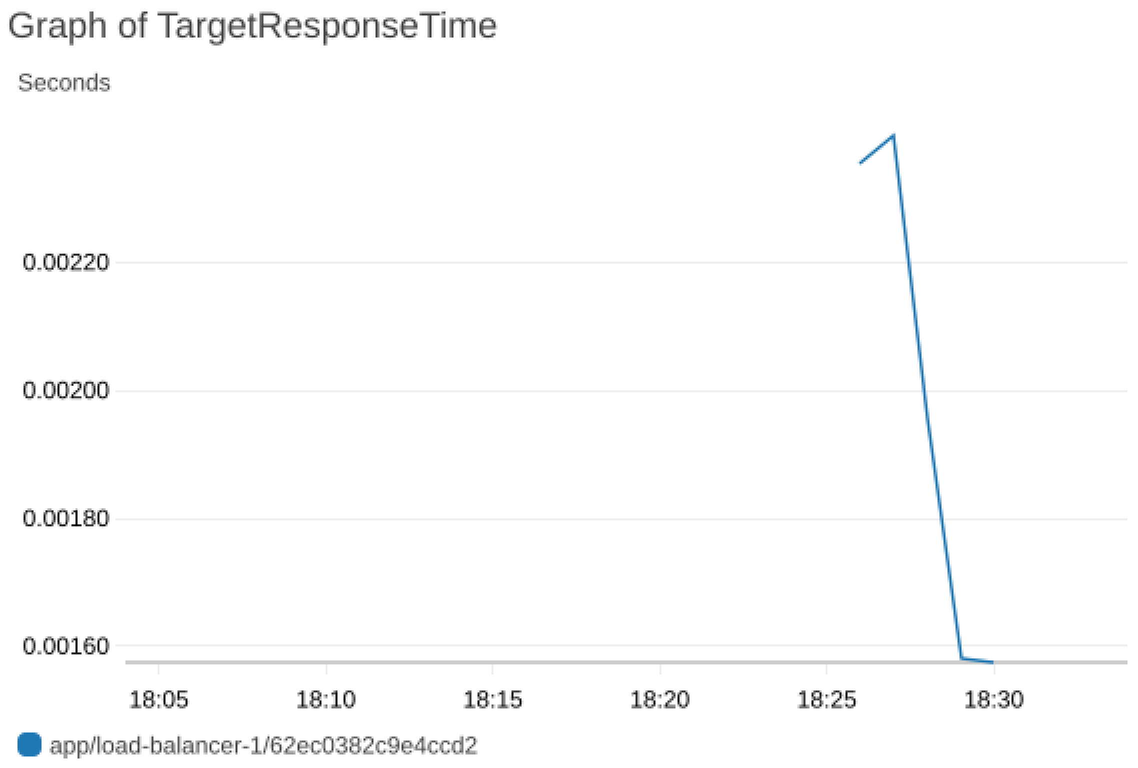
Graph of HTTPCode_Target_2XX_Count



Graph of NewConnectionCount

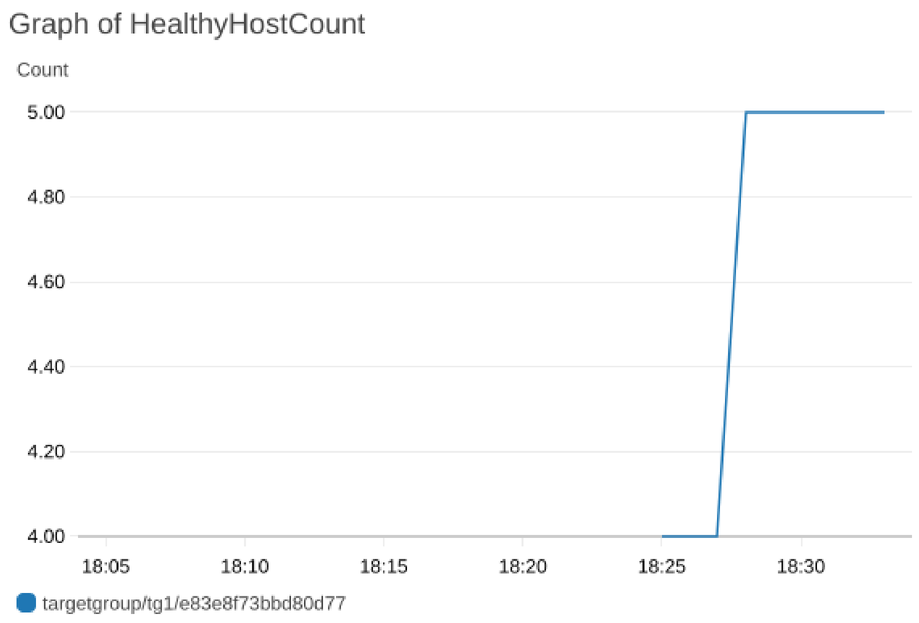


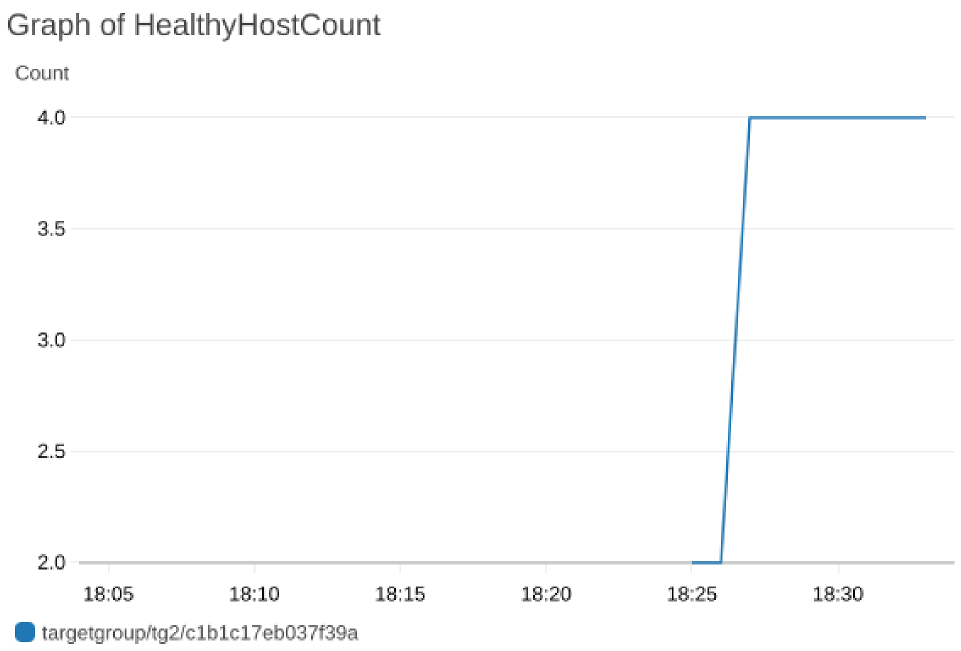
Now, this graph illustrates the time elapsed, in seconds, after the request leaves the load balancer until a response from the target is received. Upon examination, we observe that the average response time ranges from 0.0016 to 0.0024 seconds, roughly equivalent to 415 to 625 requests per second. This average value is consistent with the data presented in the HTTPCode_Target_2XX_Count graph on the previous page.



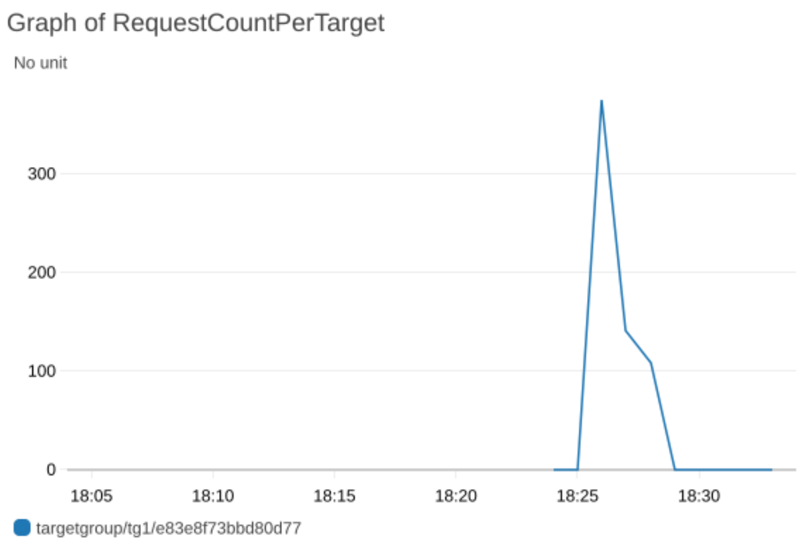
Target groups metrics:

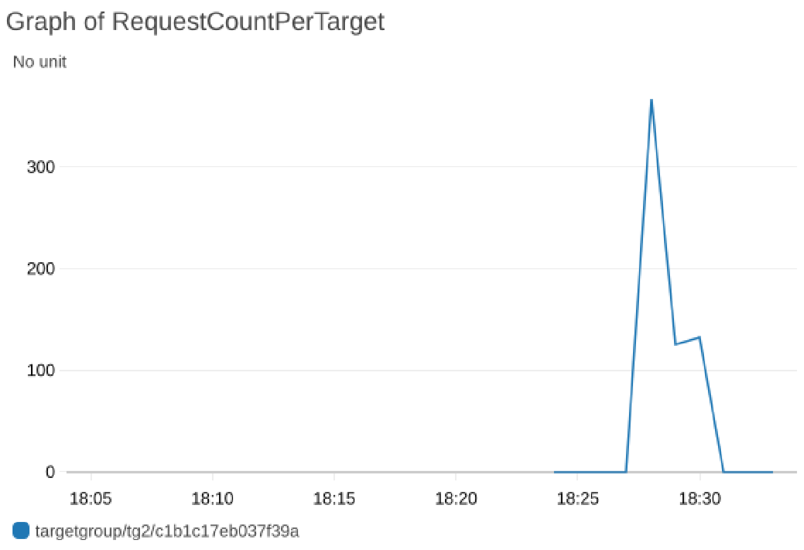
The following graphs illustrate that the first target group consists of five healthy instances, whereas the second target group is composed of four healthy instances. These observations are consistent with the expected number of instances.



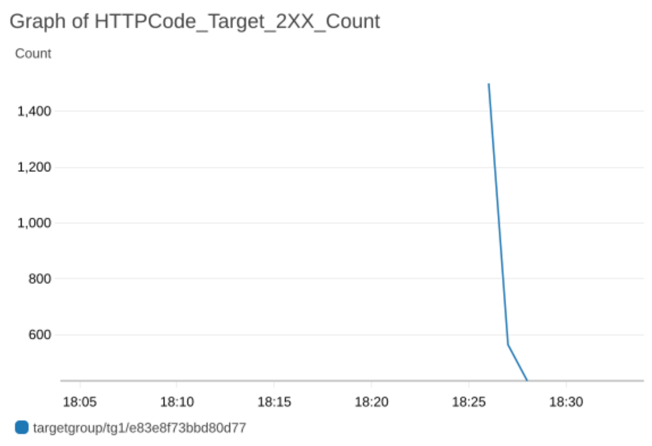


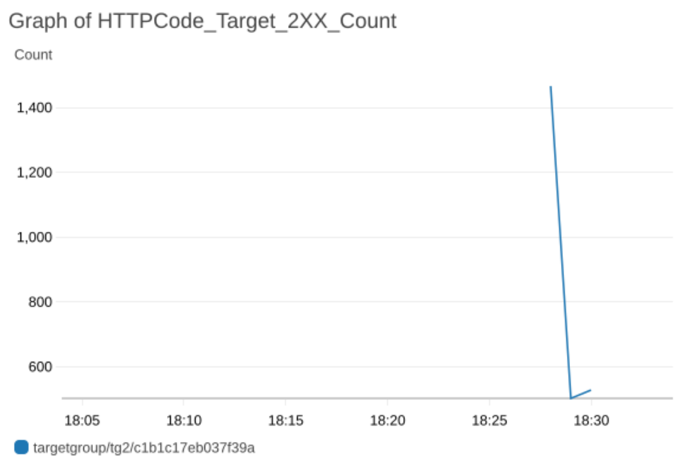
As shown in the benchmark.log, the requests for the first cluster started at 18:26 and concluded at 18:28. The first graph below is only displaying one spike, in contrast to the corresponding load balancer graph, because it is only considering the first cluster (/cluster1). The same pattern can be observed in the second graph for the second cluster (/cluster2). These findings are consistent with our request order, since we started sending requests having path pattern '/cluster1' before those having path pattern '/cluster2'. Interestingly, these two graphs combined represent the two bumps observed in the load balancer results earlier.

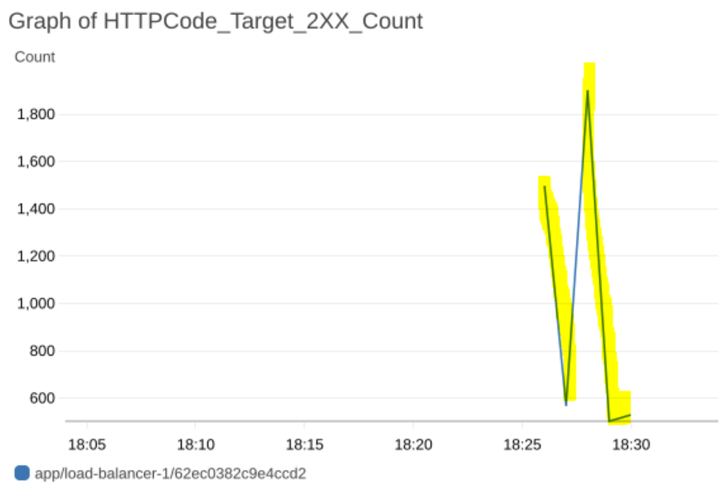




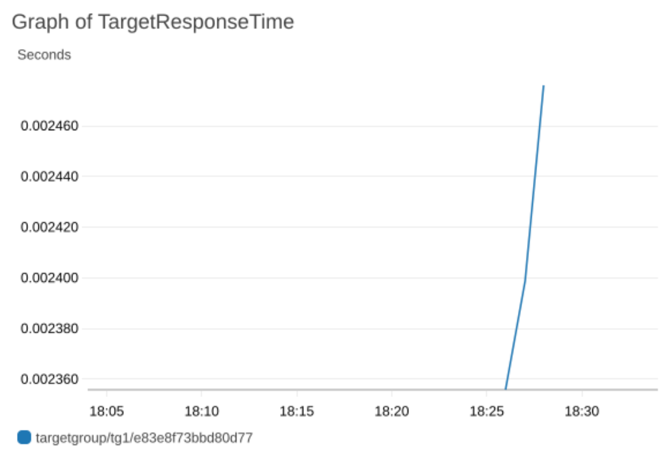
After careful examination of the first two graphs below (for target group 1 and 2), it becomes evident that their combination makes the third graph representing the load balancer's behavior.

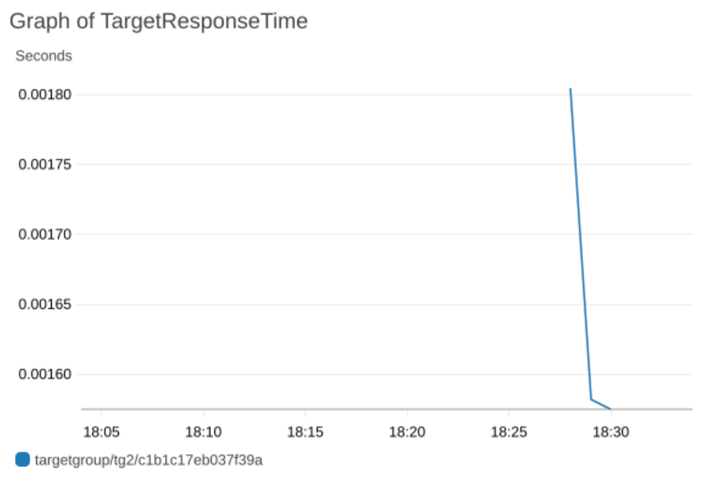


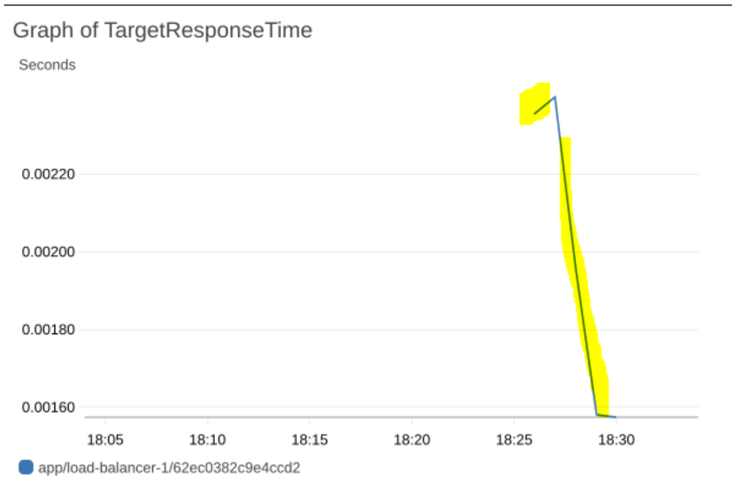




Similar to the previous analysis, the combination of the first two graphs also makes the third one, which is the graph of the load balancer. This is the expected result.

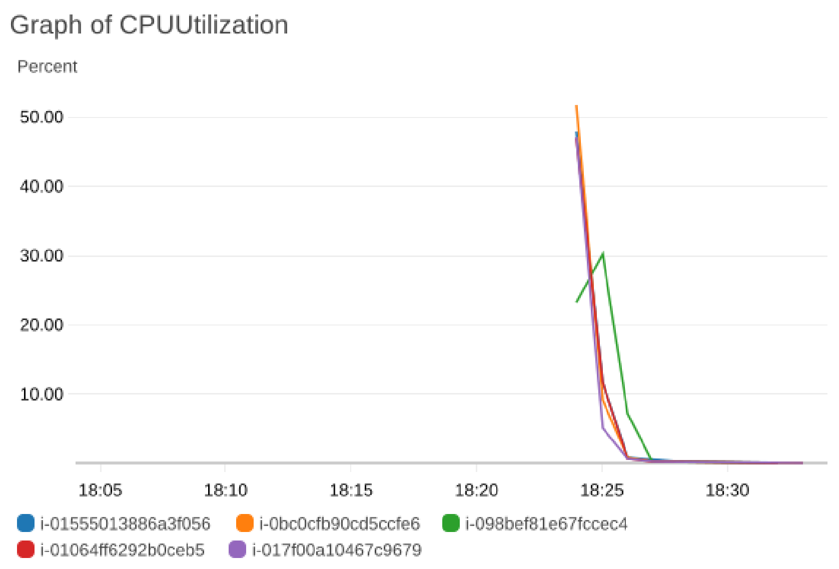


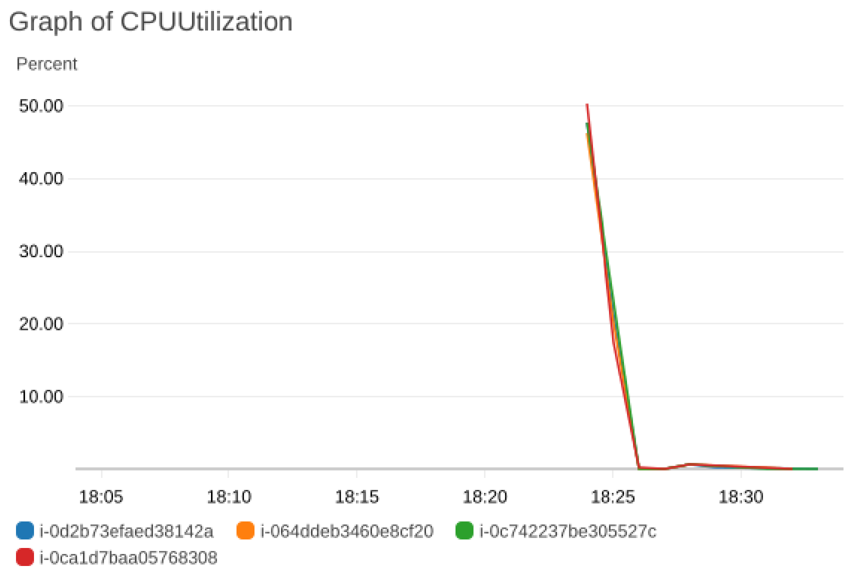




Instances metrics:

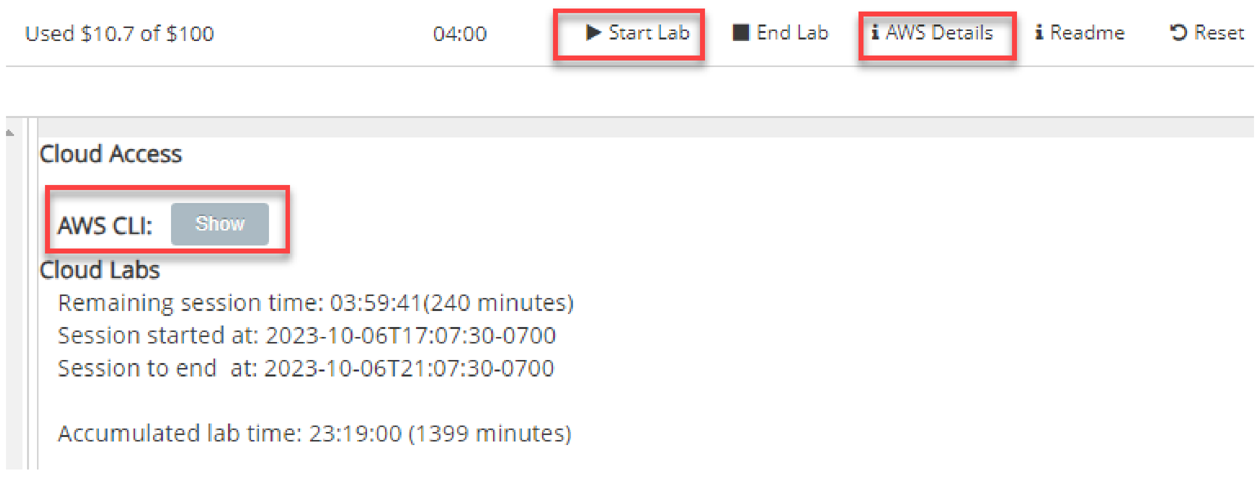
These graphs from both the first and second clusters clearly indicate that there are 5 instances in the first cluster (/cluster1) and 4 in the second one (/cluster2), and that all of them are running properly. Also, we notice that all the instances inside the clusters use CPU resources at some point, demonstrating that the load balancer is indeed distributing requests uniformly among instances and optimizing performance, as opposed to targeting the same instance.





Code Running Instructions

1. Click on 'Start Lab' on the AWS website
2. Once the AWS status turns green, click on 'AWS Details'
3. Click on 'Show' next to AWS CLI, as presented below



```
Remaining session time: 03:59:41(240 minutes)
Session started at: 2023-10-06T17:07:30-0700
Session to end  at: 2023-10-06T21:07:30-0700

Accumulated lab time: 23:19:00 (1399 minutes)
```

4. Copy all the new content that is displayed after the click and paste them in a file called *credentials* under the folder C:\Users\%USERPROFILE%\.aws
5. In the same folder, create another file called *config* and copy these 2 lines inside :

```
[default]
region = us-east-1
```

6. Navigate to the directory containing our source code and open a terminal
7. Execute the following command in the terminal: bash scripts.sh
8. Navigate to C:\Users\%USERPROFILE%\Downloads to see the output folder containing the CloudWatch metrics results and benchmark logs