Project Report

Group 5
Andrea Conti - 1849300
Simone Chieppa - 1846140

October 25, 2022

1 The problem addressed

Nowadays it is really useful and interesting to understand the public opinion about some topics of interest and exploit this knowledge for marketing, political and other purposes. This is the reason behind our choice of implementing a web application called Twitter Analyzer that, given a topic, send an email with the results of the sentiment analysis on tweets about that topic.

We chose to use Twitter as the reference social network due to the large variety of topics and opinions we can find there and because all the most influential people on the planet post on it.

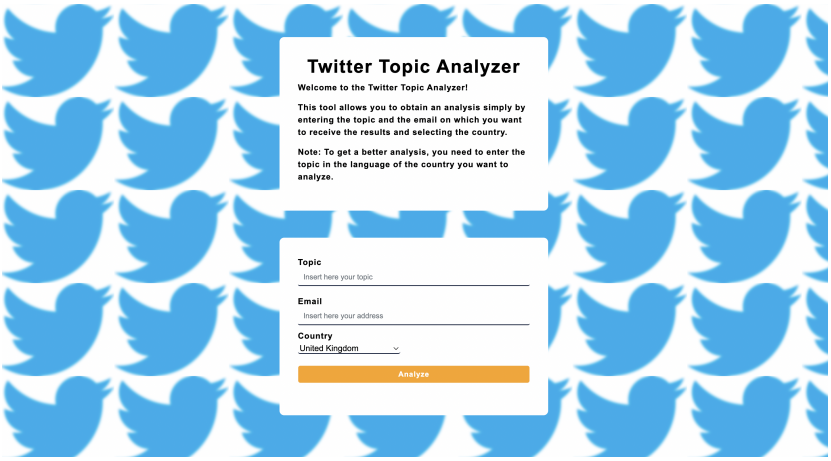

Figure 1.1: Front-end of the application.

2 Design of the solution

Twitter Analyzer is an easy to use web application, the front-end is very intuitive (as you can see in Figure 1.1) and there are only three fields to compile: the email address, the topic you want to analyze and the country where the topic should be analyzed.

We decided to send the sentiment analysis report by email for three reasons: because the analysis can take some time due to the constraints of Twitter (downloading the data from it can take up to 15 minutes although most of the time it is fast); to not force users to keep the web page open until the end, in fact they can also close the window and in any case the report will arrive by email; the most important reason however is that in this way the user is able to track the change in sentiment on the topic of his interest thanks to the older reports he has in his inbox.

The report that the user receives by email has different plots and information:

- The interval of time to which the tweets that has been analyzed belong;
- The sentiment analysis pie-plot;
- A pie-plot that illustrates the languages of the tweets analyzed;
- The tweet that caused more interactions (likes, replies and retweets).

The number of tweets analyzed by the application is 1000 because we found some difficulties with the twitter API when we tried to analyze more tweets. Anyway we think that this is a number big enough to have an interesting and indicative analysis. The report that the user receives is in English, also the tweet with most interaction is reported in English even if the original one is written in another language.

3 Description of the Implementation of the solution

3.1 The code

Our code is divided into two parts: front-end and back-end.

Starting from the former we can say that it is composed of an HTML web page with a form in which you can insert your data, when you submit the form a PHP page is invoked, it takes the arguments passed through POST method and it launches the Python code through a *system* call. There are also a CSS and JavaScript files but their roles are marginal so we will not discuss them here.[1]

Then we have the back-end code implemented in Python, it receives the arguments from the front-end and starting from them it accomplishes our task by following the 5 steps below:

1. Scraping from twitter;
2. Translation of the tweets in English for the ones that are in other languages;
3. Tweets cleaning: in this part we clean the tweets text by removing links and special characters using simple regex statements;
4. Sentiment analysis with all the plots and the other analysis (time interval of the tweets, languages of the original tweets and the tweet with most reactions);
5. Email sender.

For the scraping part, after having some troubles with other modules like Tweepy and Twint we used a module called *snscrape* that is more flexible than the previous ones.

3.2 The cloud architecture

For the cloud platform we decided to use AWS because it is the one that we have studied in class during the lab lectures.

The web application is hosted on an EC2 machine that hosts also the Python code to make the analysis, in the project proposal we said that we wanted to use Lambda functions to download tweets but we noticed that this is not the best solution, in the first place because of the time limit of the Lambda (15 minutes, that is dangerously equal to the limit of Twitter API)[2][3], but also because it is better to have both the front-end and the back-end together.

With respect to the project proposal we added also one service: we decided to connect to our web-application a S3 bucket in order to save for a small amount of time (1 day) the results of the analysis, in this way if two users want a sentiment analysis about the same topic in the same day the system doesn't have to recompute all the analysis but it can use the results which are already in memory, in other words: with respect to our project proposal we have added a cache system to improve the responsiveness of the system and to allow the users to have a better experience.

We created also an AWS EC2 Auto Scaling group[4] connected to a load balancer (through AWS Elastic Load Balancing) in order to make the application scalable and we monitored the CPU usage to allow scale-out and scale-in actions with Amazon CloudWatch.

We will explain in details the implementation of the cloud architecture below.

Amazon EC2

Now let's see in detail the implementation of the EC2 on top of which we ran our code.

First of all we chose the Amazon Machine Image (AMI) to use for our project, we went for the Amazon Linux 2 AMI because it provides a security-focused, stable, and high-performance execution environment to develop and run cloud applications.

[1] You can find all these files in the archive attached to this report

[2] This addresses your note 1 to project proposal: "verify if for long running functions, like the one you would like to implement, lambda is the more appropriate choice".

[3] This also addresses your note 3 to project proposal: "It will be interesting to monitor how many lambda functions (that process tweets) run concurrently and what is their execution time". We don't have Lambda anymore but we will do something similar with executions on EC2.

[4] This address your note 2 to project proposal: "The front-end should be scalable but is not the big problem in that case (is a static page collecting inputs). So good to put it an autoscaling group"

Then we decided which type of instance to use, for our case we found that the best solution for us was a r5.large instance type. According to the AWS documentation R5 instances are optimized for memory-intensive applications and fit well for applications that perform real-time processing of unstructured big data or data mining and analysis that is exactly our case.

After choosing the instance type we configured the network settings: we created a new security group that allowed the SSH traffic from everywhere (to allow our computer with dynamic IP addresses to connect to the machine and install the needed software) and the HTTP/HTTPs traffic from the Internet.

Finally we chose an Elastic Block Store (EBS) volume of 8 GB for the storage and we modified the advanced settings of the EC2: we selected the default Identity and Access Management (IAM) role because it was not possible to create a new one with our limited license, then we chose 'stop' as shut down behaviour and we enabled the termination protection to avoid to terminate the instance by mistake and to loose the data on the EBS volume after shutting down. For the EBS volume we decided to enable the encryption to ensure the security of both data-at-rest and data-in-transit between the instance and its attached EBS storage.

We obviously enable the Detailed CloudWatch monitoring in order to monitor, collect, and analyze metrics about the instance.

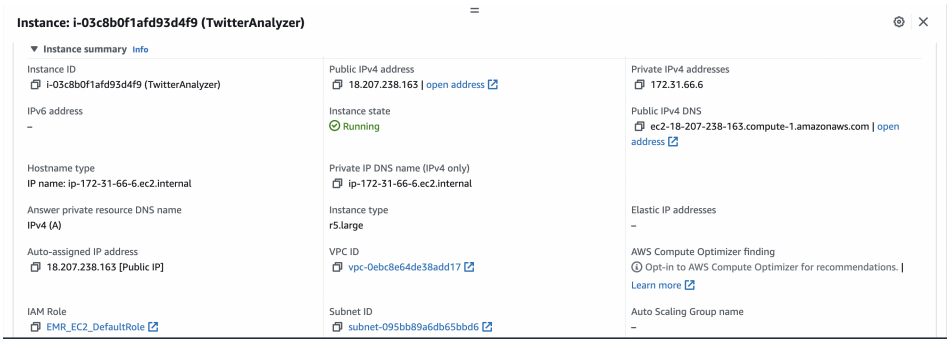

Figure 1.2: Summary of the EC2 Instance.

Amazon S3

The big change with respect to the project proposal is the introduction in our cloud architecture of an Amazon S3 Bucket.

We decided to implement this solution because we noticed that in this way our web application could have had better performances. In fact in this way we created a cache for our web application in order to have fast responses to user that request the same analysis in the same day. We decide to make this storage temporary because we though that two analysis made in different days have different results because they are computed on different tweets, however for analysis that are made in the same day this solution speed up a lot the response of our application that, in this way, just send the saved files by email.

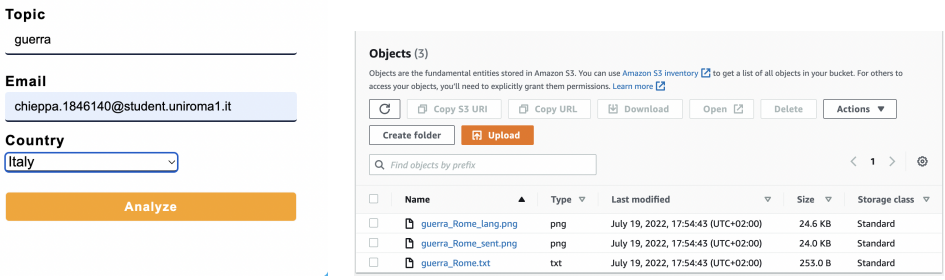

Figure 1.3: Analysis storage

In the figure above (Figure 1.3) we can see the S3 bucket that saves the two plots of the analysis as PNG images and the other information of the analysis as text file.

For setting up the S3 Bucket we have chosen to enable server-side encryption to increase security and to make the objects saved in it temporary we set a *Lifecycle* rule to apply to all the object in the bucket: after one day from the creation the objects expire as we can see in the Figure 1.4.

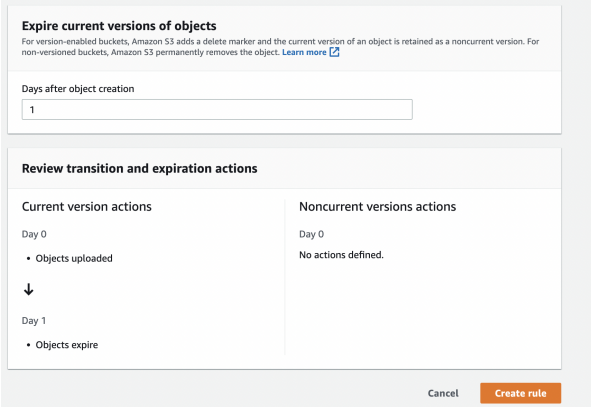

Figure 1.4: settings of the Lifecycle rule

Elastic Load Balancing and Amazon EC2 Auto Scaling

To load balance and scale our infrastructure we used Elastic Load Balancing that automatically distributes incoming application traffic across multiple Amazon EC2 instances and EC2 Auto Scaling that maintains application availability and allows you to scale your Amazon EC2 capacity out or in.

First of all we created an AMI from the EC2 instance that we created before, in this way new instances can be launched with identical content. Then we created a Target Group that define where to send traffic that comes into the Load Balancer, for the Load Balancer we chose an application Load Balancer because it only works at layer 7 (HTTP) and it is typically used for web applications.

After the choice of the Load Balancer we set the launch configuration for the auto scaling group, we chose the AMI that we had created before and obviously we selected the same type of instance of the already existing EC2 (r5.large). Then we created from these configurations the auto scaling group but before we enabled EC2 instance detailed monitoring within CloudWatch.

When we created the auto scaling group we attached it to our Load Balancer and we set the scaling policies and we decided the group size.

For the group size we set:

- Desired capacity = 1.
- Minimum capacity = 1.
- Maximum capacity = 9.

We decided to set the minimum capacity to 1 because a single EC2 can handle all the workload when there are few requests, the maximum capacity instead has been set at 9 because of the limits imposed by AWS (not more than 32 CPUs at time and not more than 9 active instances at time).

For the Target tracking scaling policy we decided to monitor the average usage of the CPU and we set the target value to 50 because after a few attempts with other values we noticed that this was the best solution that combines the capacity to promptly add more computational resources and to avoid useless scaling.

We decided to use the average CPU utilization as a metric because our project is an application that does a lot of computations per request so with the increase of the users requests it increases the computation and so the CPU utilization, network parameters were useless because we use the network just for taking request and to send email so they didn't fit well[5].

[5]Even if the network is actually used to download tweets this is just an effect produced by the computation itself

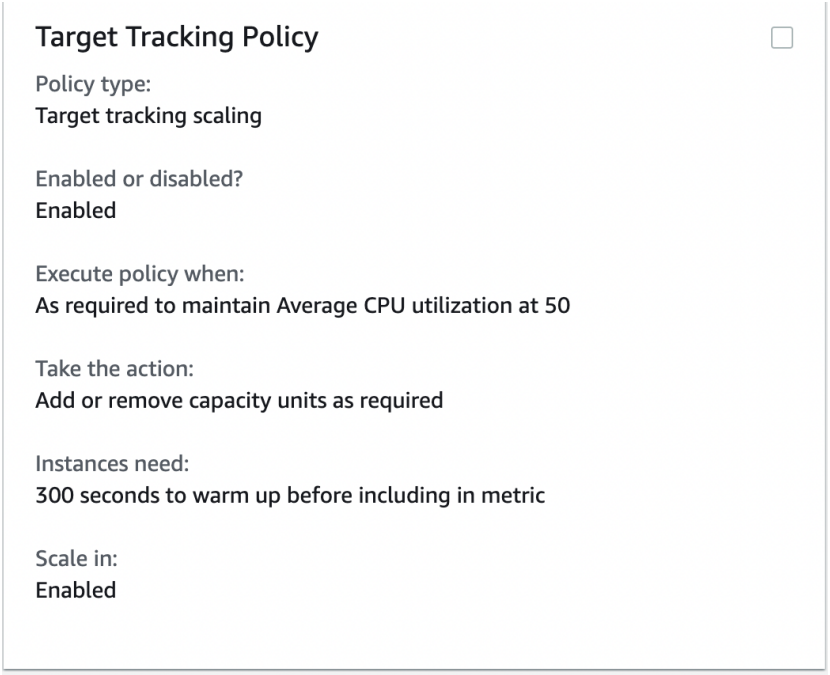

Figure 1.5: Target tracking policies.

4 Test design

To test the scalability and performance of our application, we first thought about the situation in which our web application should scale that is when it has to do a lot of computations so when more users request analysis at the same time.

Our initial idea, since the increasing of the users requests increases the CPU utilization, was to use a CPU stress tool but then we realized that this wasn't the best idea because in this way the full load would have been only on one instance (the one in which the CPU stress tool is active).

So to make this test we decided to write a code that sends a lot of requests concurrently[6], more specifically the number of requests that we decided to send to the application are: 8, 16, 32, 64, 128, 256, 512, 1024, 2048 [7], note that the number increases exponentially, it will be useful in the next section. After the implementation of the code we had to decide which measure to take.

We ended up deciding to register the amount of time that the application took to fulfill the requests, then we averaged these times and starting from these results we computed the average amount of time for a single request.

During the tests we also monitored all the metrics of Amazon Cloud Watch to check the average utilization of the CPU, the number of in-service instances and the network utilization (in and out bytes).

Since we wanted to make our conclusions the least biased possible we decided to make the tests 10 times and to take the average value.

5 Experimental results

For our tests, as we explained in the paragraph above we decided to increase exponentially the number of concurrent requests and to see the scaling actions of our application and monitor the time that it takes to fulfill a request. Now lets see what happened.

The Figure 1.6 represents the time (in seconds) to fulfill the concurrent requests, we can notice that the time increases linearly in contrast to the fact that the number of requests increases exponentially.

[6] You can see it in the archive

[7] As can be deduced this encompasses your note 0: "not sure you can make concurrent requests of downloading twitter data with the same account. Evaluate is eventually is better to use a dataset (for testing purposes)" because our API allows concurrent requests

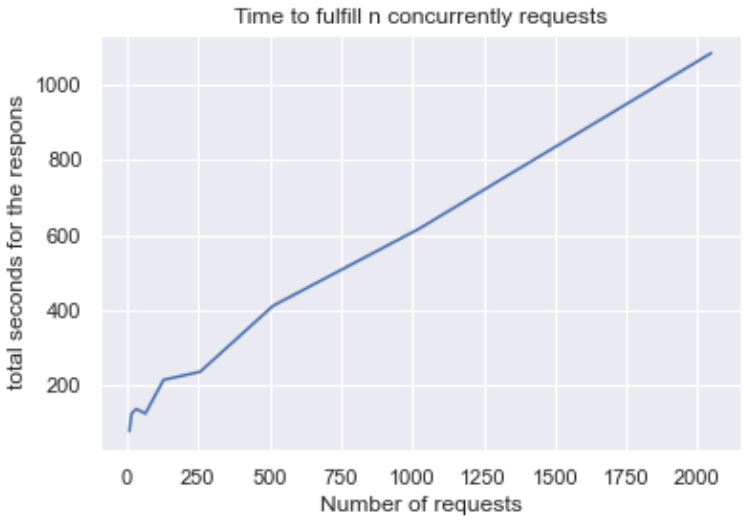

Figure 1.6: Plots of the time in seconds to responds to n requests.

This behaviour is a consequence of the scalability of our application, in fact, the increase of the requests cause an increase of the CPU utilization and for this reason also the number of EC2 instances increases.

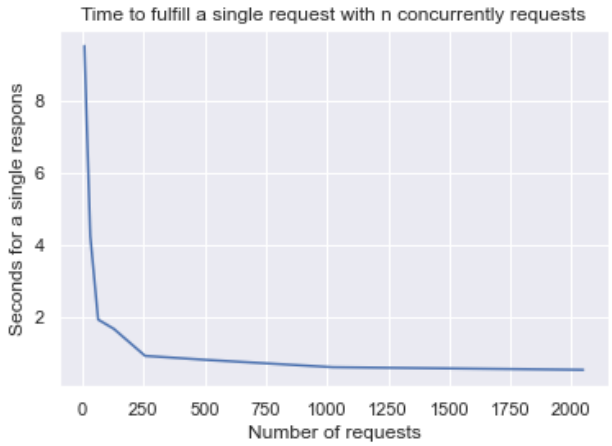

Figure 1.7: Plots of the time in seconds to responds to a single request.

What we have just said is supported by the figure 1.7 that represents the time that occurs to fulfill a single request when there are n concurrent requests, you can easily note that although the number of requests increases the per-request time decreases, this is due to the fact that there are more EC2 instances that work on the requests.

To understand the behaviour of our application during the test we can see and analyze all the information that Amazon Cloud Watch gives us.

First of all as soon as the CPU utilization of the running instance exceeds the 50% threshold for 3 consecutive data points in 300 seconds Cloud Watch rises an alarm and the system performs the scaling action. From the Figure 1.8 we can notice the change in the amount of running instances with the increase and decrease of the work load.

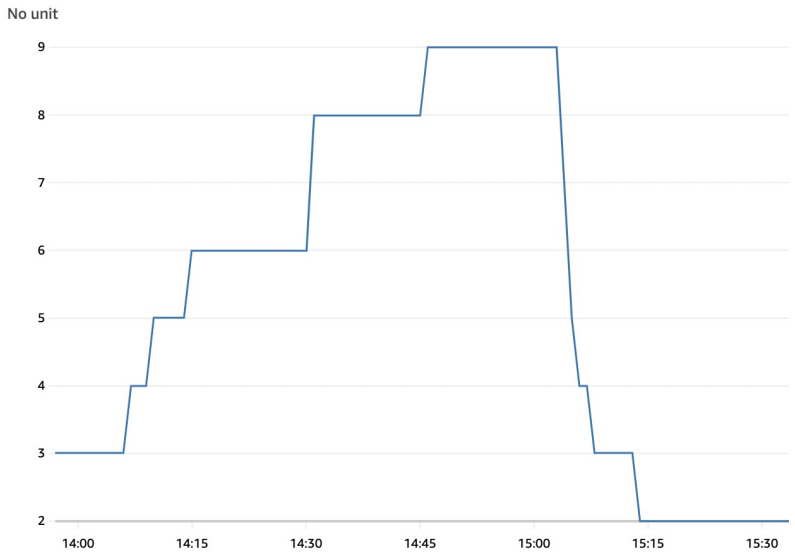

Figure 1.8: scaling in/out

We can notice that by increasing the work load the in-service instances scale out and then when the test finishes scale in massively. We can also see that the first steps in the graph are very rapid and this is due to the exponential increase of requests, then when a sufficient number of instances is up&running the scaling actions are slower because those machines are able to handle the work load pretty well.

Another interesting thing to observe is the average usage of the CPU, as we can see in Figure 1.9 the usage first goes up to more than 90%, then with the activation of the other instances it has a strong decrease and it converges around 45% and this is fine because it avoids to launch new instances.

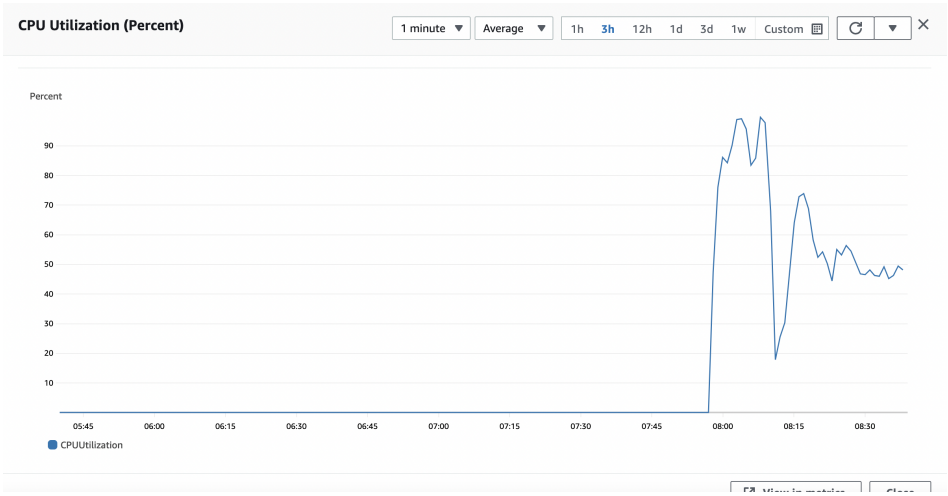

Figure 1.9: Average CPU utilization.

In the Figure 1.10 we can see the behaviour of the entire autoscaling group that adds machines when needed, it is interesting to see how this graph supports the previous one with the CPUs utilization that end up around the same percentage as before.

If we want to have an insight on the variation of CPU utilization when a new instance is started we can see the Figure 1.11 that shows how the workload on the first machine decreases as soon as the new instance is started, you might think that is a bad thing because the workload goes entirely on the second machine but as you can notice from the previous graph in the long period the load is distributed among all the active machines.

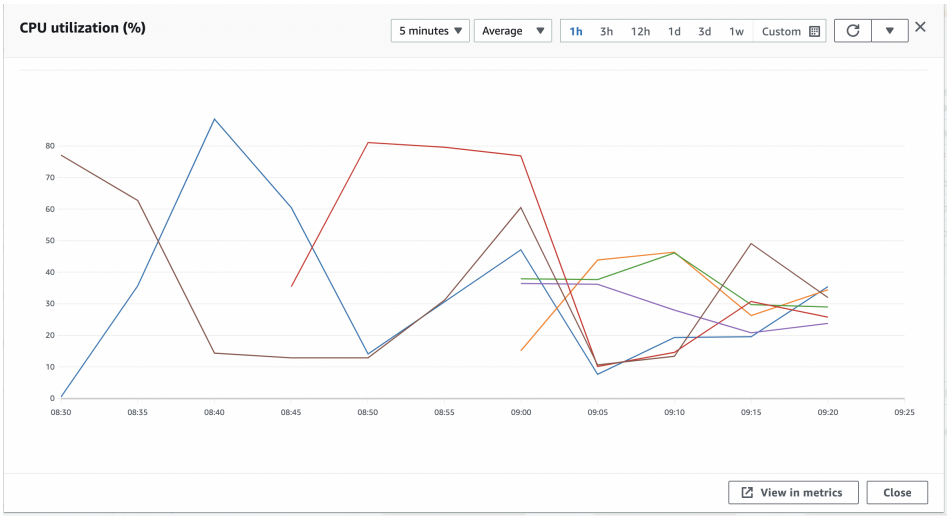

Figure 1.10: CPU utilization of the auto scaling group.

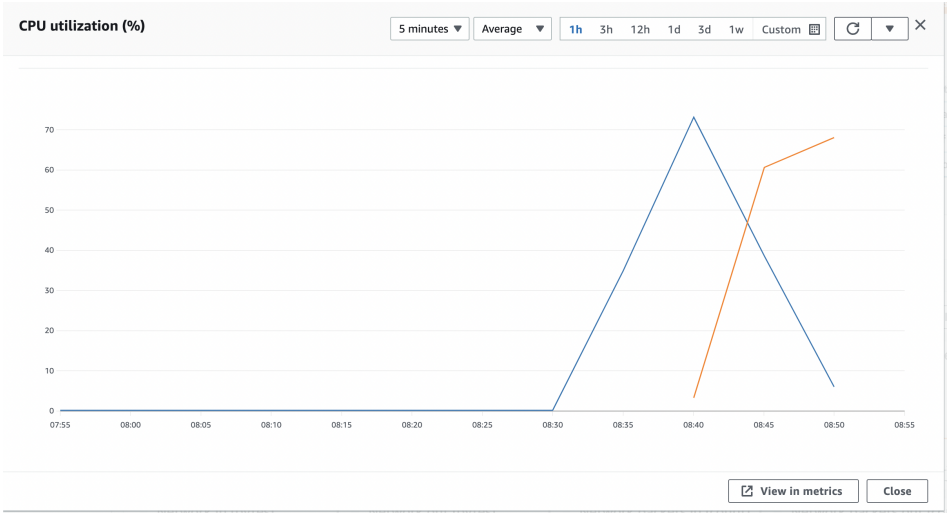

Figure 1.11: CPU utilization of two instances.

6 Conclusions

Our Twitter Analyzer application has been able to satisfy all the 4 requirements imposed by the project because our app runs on AWS using different services and -after the project proposal- we followed the remaining three phases demanded by the requirement 3, here we want to highlight that, with respect to the project proposal, this report shows some differences such as the removal of the Lambda functions and its replacement with a normal EC2, the introduction of S3 bucket and the possibility of caching related to it and some other little change needed to best accommodate this project.

The latter part of our project is about tests and they confirm our idea that is, the infrastructure resists to a high workload. This is primarily thanks to the cloud infrastructure that makes Twitter Analyzer capable to run in a very flexible and available environment.