Project Report for the Cloud Computing course project

Groups members [Bianca, Francesco; Magnante, Andrea; Manduca, Lorenzo; Prandini, Manuel;]
Project title: [Twitter Analytica]

Deviations from the Project proposal

Compared to our first project idea, which included "the *development/deployment of a cloud application that allows user to do scraping from Twitter (in form of tweets) based on a specific argument"* and specifically *"the data taken from Twitter will be used by Spark in real time, without the use of a storage, due to not overload the architecture of the project and its operations",* we decided not to use the live streaming mode and have made the following changes:

1. We have saved the streaming of the various tweets in Amazon S3 to allow the user to search for the topic in the previous days (up to 7 days).
2. Twitter API does not allow live streaming to multiple instances from the same ip address as it was seen by Twitter security systems as a DDOS attack. To avoid account blocking and for computational reasons, we decided to create a single live stream of all tweets filtered by English language
3. We didn't want to have an architecture that would oblige to create, for each user, a dedicated stream and a dedicated spark cluster, which would be computationally very expensive and not optimized.

Description of the problem addressed

Our project is focused on the development/deployment of a cloud application that allows user to do scraping from Twitter (in the form of tweets) based on a specific argument. Our application has been designed and optimized for users interested in learning statistical information from Twitter in order to use it in the field of marketing/advertising/social media.
The final prototype developed is a responsive web-app, which is accessible from any device with an internet connection, independent of the type of operating system, with User-Friendly graphical user interface. In output it will show to the user a series of plots and statistical data. The data taken from Twitter is constantly updated in near-real-time in our Amazon S3 and processed on our Spark cluster. The application uses many AWS services to process the data stream, to structure the raw data, save it and process it by making it available to the end user that uses the system. For example possible functionalities are: given a specific argument and scraped related tweets, the application analyses them and returns, as output, information about the most used #hashtag referenced to the chosen argument, or again a study on the ones with a larger amount of retweets.

Design of the solution

For our application we used the following architecture:

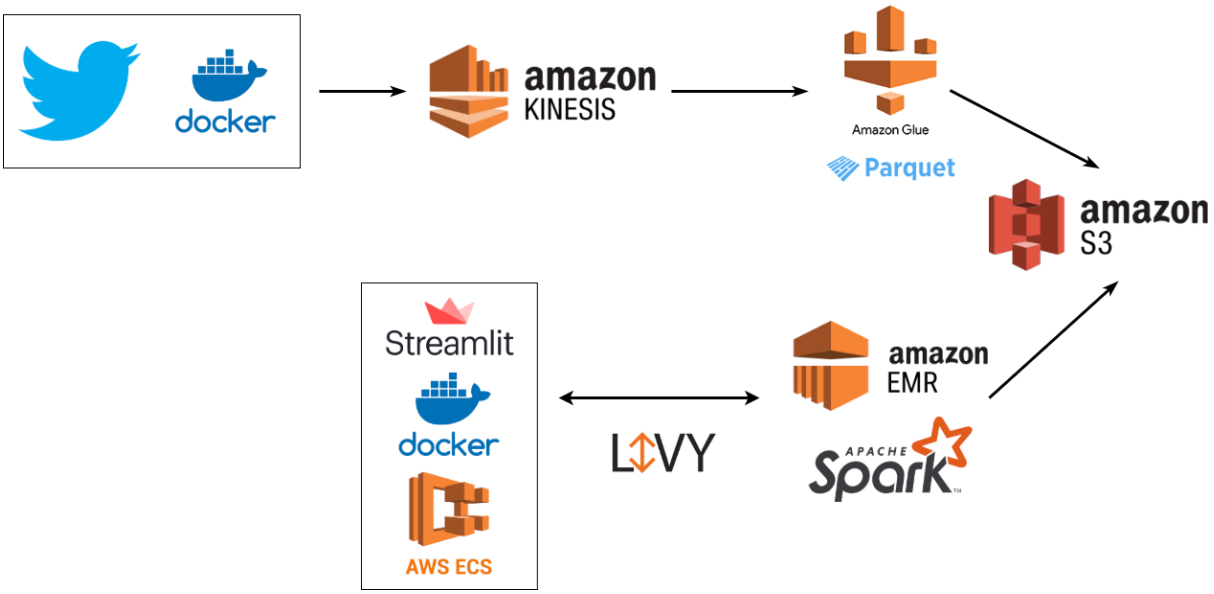


Figure 1: System Architecture

For the realization of our project we made use of different technologies that led us to achieve our goal. Technologies used in our project are:

- **Twitter REST API:** Twitter is simply a Web-based way to tell certain people know what you are currently doing in 140 characters or less. Twitter is not only a useful tool within the social media space, it also offers developers a comprehensive array of services to enable automation of Twitter functionality. One of those services (and perhaps the most popular) is the REST API. REST is an acronym for Representational State Transfer. REST enables developers to access information and resources using a simple HTTP invocation. Twitter REST API allows you to retreive tweets and related information from Twitter.

- **Docker:** Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into standardized units called containers that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale applications into any environment and know your code will run. Running Docker on AWS provides developers and admins a highly reliable, low-cost way to build, ship, and run distributed applications at any scale. AWS supports both Docker licensing models: open source Docker Community Edition (CE) and subscription-based Docker Enterprise Edition (EE).

- **Amazon Kinesis Firehose:** Amazon Kinesis Data Firehose is the easiest way to reliably load streaming data into data lakes, data stores and analytics tools. It can capture, transform, and load streaming data into Amazon S3, Amazon Redshift, Amazon Elasticsearch Service, and Splunk, enabling near real-time analytics with existing business intelligence tools and dashboards you're already using today. It is a fully managed service that automatically scales to match the throughput of your data and requires no ongoing administration. It can also batch, compress, transform, and encrypt the data before loading it, minimizing the amount of storage used at the destination and increasing security.

- **Amazon Glue:** AWS Glue is a fully managed extract, transform, and load (ETL) service that makes it easy for customers to prepare and load their data for analytics.

- **Apache Parquet:** Apache Parquet is a free and open-source column-oriented data storage format of the Apache Hadoop ecosystem. It is similar to the other columnar-storage file formats available in Hadoop namely RCFile and ORC. It is compatible with most of the data processing frameworks in the Hadoop environment. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.

- **Amazon S3:** Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. This means customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides easy-to-use management features so you can organize your data and configure finely-tuned access controls to meet your specific business, organizational, and compliance requirements.

- **Amazon EMR:** Amazon EMR is the industry-leading cloud big data platform for processing vast amounts of data using open source tools such as Apache Spark, Apache Hive, Apache HBase, Apache Flink, Apache Hudi, and Presto. With EMR you can run Petabyte-scale analysis at less than half of the cost of traditional on-premises solutions and over 3x faster than standard Apache Spark. For short-running jobs, you can spin up and spin down clusters and pay per second for the instances used. For long-running workloads, you can create highly available clusters that automatically scale to meet demand.

- **Apache Spark (PySpark):** Apache Spark is a project started in 2009 and much more recent than MapReduce. It is based on the representation of the program as a DAG.

MapReduce has limitations and weak points like programming model problems, no native support for iterations, efficiency problem and it doesn't fit well with the real-time data process. Spark works in-memory and this makes it 10x faster than Hadoop. Spark introduces an abstraction called resilient distributed datasets (RDDs). PySpark is the Python API for Spark.

- **Apache LIVY:** Apache Livy is a service that enables easy interaction with a Spark cluster over a REST interface. It enables easy submission of Spark jobs or snippets of Spark code, synchronous or asynchronous result retrieval, as well as Spark Context management, all via a simple REST interface or an RPC client library. Apache Livy also simplifies the interaction between Spark and application servers, thus enabling the use of Spark for interactive web/mobile applications.

- **Streamlit:** Streamlit lets you create apps for your machine learning projects with deceptively simple Python scripts. It supports hot-reloading, so your app updates live as you edit and save your file.

- **Amazon AWS Cloud Formation**: AWS CloudFormation provides a common language for you to model and provision AWS and third party application resources in your cloud environment. AWS CloudFormation allows you to use programming languages or a simple text file to model and provision, in an automated and secure manner, all the resources needed for your applications across all regions and accounts. This gives you a single source of truth for your AWS and third party resources.

- **Amazon ECR:** Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. Amazon ECR is integrated with Amazon Elastic Container Service (ECS), simplifying your development to production workflow. Amazon ECR eliminates the need to operate your own container repositories or worry about scaling the underlying infrastructure. Amazon ECR hosts your images in a highly available and scalable architecture, allowing you to reliably deploy containers for your applications.

- **Amazon ECS:** Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service. ECS supports Fargate to provide serverless compute for containers. Fargate removes the need to provision and manage servers, lets you specify and pay for resources per application, and improves security through application isolation by design.

- **Apache JMeter:** may be used to test performance both on static and dynamic resources, Web dynamic applications. It can be used to simulate a heavy load on a

server, group of servers, network or object to test its strength or to analyze overall performance under different load types.

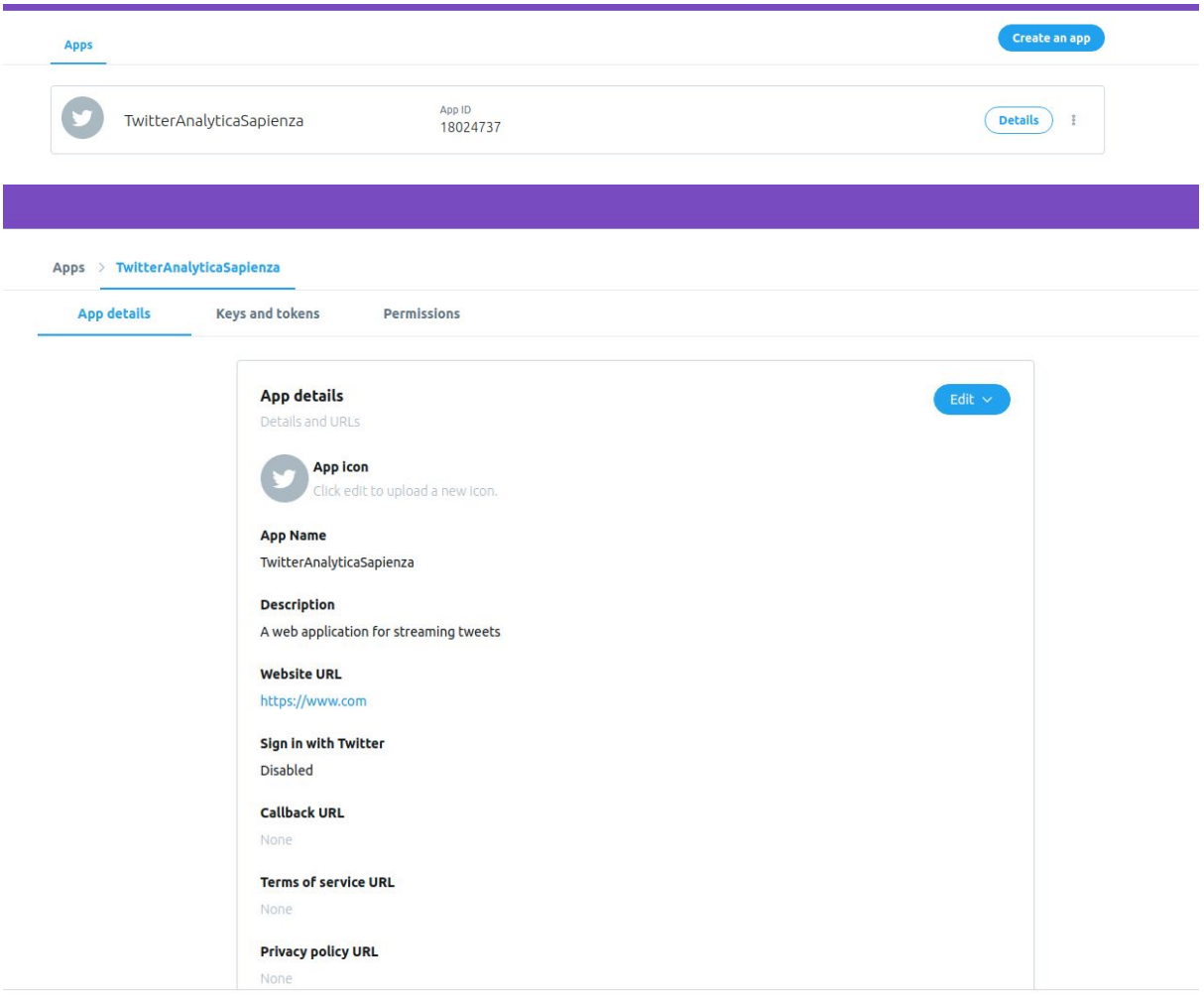
- **Apache Bench**: Apache Bench (ab) is a load testing and benchmarking tool for Hypertext Transfer Protocol (HTTP) server. It can be run from command line and it is very simple to use.

- **Amazon AWS Cloudwatch**: Amazon CloudWatch is a monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch provides you with data and actionable insights to monitor your applications, respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. CloudWatch collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications, and services that run on AWS and on-premises servers. You can use CloudWatch to detect anomalous behavior in your environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep your applications running smoothly.

Starting from the description of the addressed problem, we have developed the architecture of the system proposed in figure 1. As a first operation the real-time stream of the various tweets is retrieved using the "Twitter REST API" deployed in a docker container that is continuously running on an EC2 instance.
Subsequently, this stream is passed as input to the Amazon Kinesis service that performs some preliminary operations on the input data and thanks to Amazon Glue these data are structured and stored in an Amazon S3 bucket.
As for the client side, our system is usable by the user thanks to a web-app, developed thanks to the framework called "Streamlit". The web-app has also been dockerized and deployed on a cluster in order to ensure its scalability.
Through it, the user has the possibility to search for a specific topic by selecting a specific time interval.
All these requests are essentially operations of analysis of large amounts of data that are processed thanks to Apache Spark (PySpark in our case). It processes these data in a special cluster defined by us through the Amazon EMR service. In order to allow the communication between the interface and the EMR cluster the Apache LIVY service and in particular its REST API is used.
In the following sections we will analyze in detail why the choice of these components and how they work.

Description of the Implementation of the solution

Starting with the design phase, we will go in more detail by showing the implementation of the various components.
As we have already seen before, the first step to do in our system is to retrieve the real-time stream of the various tweets of the users.
In order to use the Twitter API, the first step is to register on the Twitter Developer platform (https://developer.twitter.com/en) and create a new App in order to obtain valid credentials to develop our application.



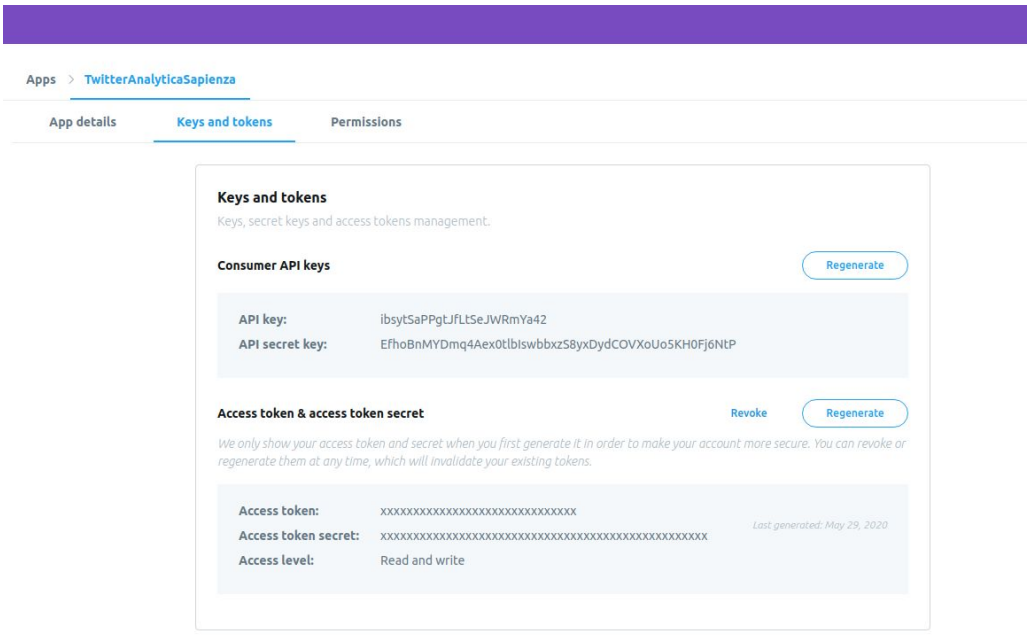


Figure 2: Twitter developer account configuration

Once registered and obtained the credentials, we used a python script (as we can see in figure 3) to make calls to the Twitter REST API through the use of a library called Tweepy. Tweepy is a library for Python that provides us with a Wrapper for the Twitter API which makes the communication process with this really quick. In our project we are interested in listening to the stream of tweets. Once the API credentials have been set, we start the stream on all the real-time tweets filtered by English:

```python
stream_name = 'Sapienza_F' # fill the name of Kinesis data stream you created

kinesisClient = boto3.client('firehose',region_name='us-east-1')

def deEmojify(text):
        regrex_pattern = re.compile(pattern = "["
            u"\U0001F600-\U0001F64F"  # emoticons
            u"\U0001F300-\U0001F5FF"  # symbols & pictographs
            u"\U0001F680-\U0001F6FF"  # transport & map symbols
            u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                               "]+", flags = re.UNICODE)
        return regrex_pattern.sub(r'',text)

class TweetStreamListener(StreamListener):
    # on success
    def on_data(self, data):
        # decode json
        tweet = json.loads(data)
        if "text" in tweet.keys():
            payload = {'tweet': deEmojify(str(tweet['text']))},
            print(payload)

            try:
                put_response = kinesisClient.put_record(
                    DeliveryStreamName=stream_name,
                    Record={
                        'Data': json.dumps(payload)
                    })
            except (AttributeError, Exception) as e:
                print (e)
                pass
        return True

    # on failure
    def on_error(self, status):
        print(status)

if __name__ == '__main__':
    # create instance of the tweepy tweet stream listener
    listener = TweetStreamListener()
    # set twitter keys/tokens
    auth = OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_token_secret)
    # create instance of the tweepy stream
    stream = Stream(auth, listener)
    stream.sample(languages=["en"]) # Filtro solo i tweets in inglese
```

Figure 3: python script used to communicate with Twitter REST API

The Twitter API provides results in JSON format from which you can collect a lot of information such as time, user, geographic coordinates and so on.
We are only interested in the text of the tweet therefore we create a new json with the following format:

```
payload = {'tweet': deEmojify(str(tweet['text']))},
```

where the `deEmojify` function takes care of doing a first cleaning of the text by removing some unrecognized characters. In order to be analyzed on AWS, data must pass through an object called Kinesis which is the easiest way to reliably load streaming data into data lakes, data stores and analytics tools. Once this script was completed we decided to dockerize it, so that we can distribute it anywhere and make it available through a simple command. We then created a container locally on our machine and distributed it on the docker hub.

As mentioned previously, the data to be examined must pass through Amazon Kinesis. So through our Python script we send them to our firehose stream called `'Sapienza-F'`:

```
put_response = kinesisClient.put_record(
    DeliveryStreamName=stream_name,
    Record={
        'Data': json.dumps(payload)
    })
```

The input source of the Kinesis Firehose is `'Direct PUT or other sources'`; the data is sent by code through the client created through the AWS SDK for Python. Different functions can be applied on a kinesis firehose: for example, every time new data arrives, an on demand function called Amazon Lambda can start which performs some operations on the input or the data can be structured using AWS Glue which changes its output format (JSON, Parquet etc). In order to have more structured data and to be able to save them correctly in the S3 buckets we have chosen to use a parquet-type encoding.
Kinesis Data Firehose buffers incoming data before delivering it to Amazon S3 choosing a buffer size or buffer interval. The condition that is satisfied first triggers data delivery to Amazon S3. If you enable data transformation, the buffer interval applies from the time transformed data is received by Kinesis Data Firehose to the data delivery to Amazon S3. If data delivery to the destination falls behind data writing to the delivery stream, Kinesis Data Firehose raises the buffer size dynamically to catch up. This action helps ensure that all data is delivered to the destination.

After defining the kinesis firehose, we need to structure our data before storing it in Amazon S3 bucket. To do this we use Amazon Glue.
The format of the table created by Glue is as follows: On each line there is the text of a certain tweet (as we can see under `FieldSchema` in figure X)

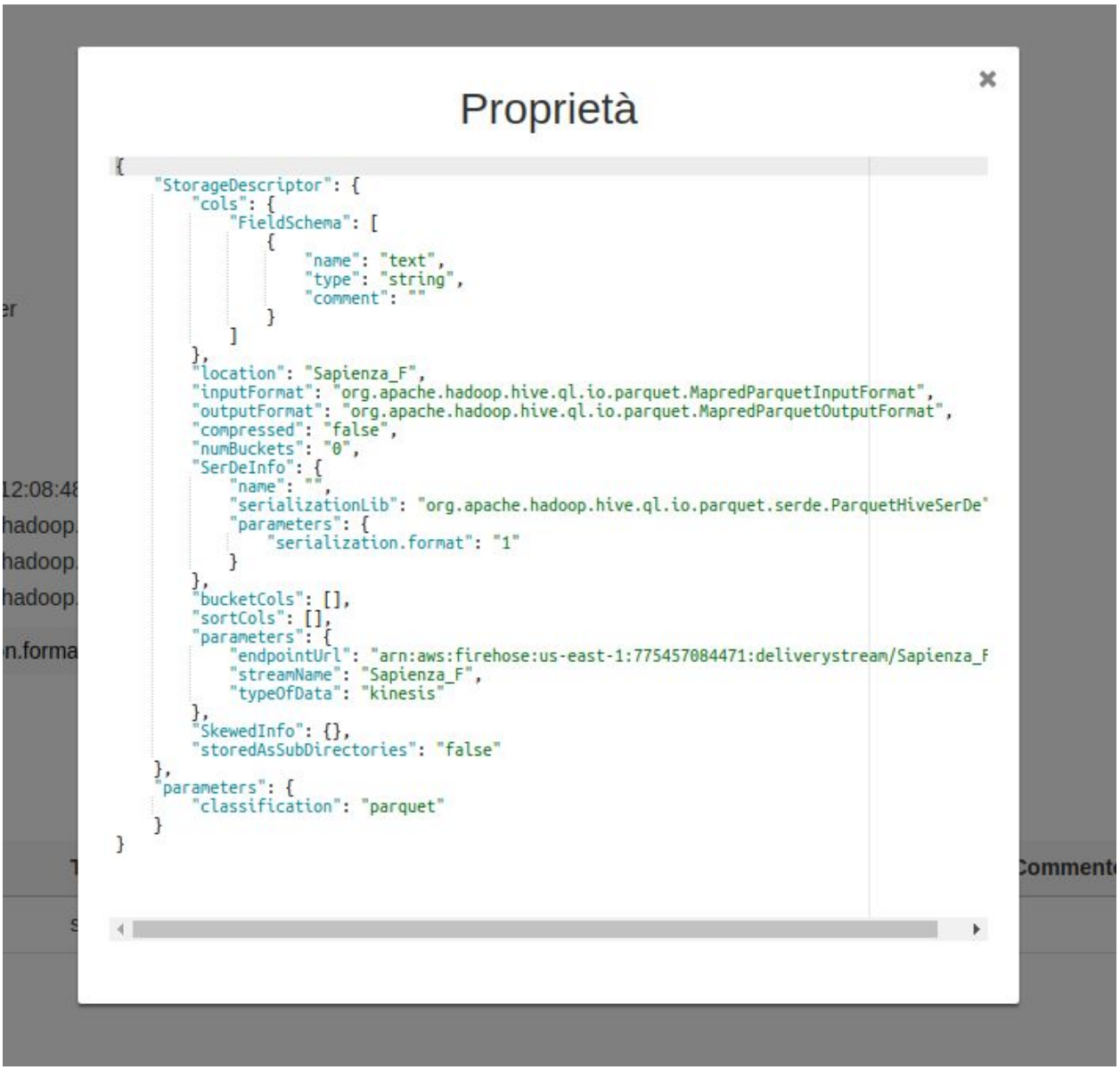


Figure 4: Amazon Glue table configuration

You could easily expand the table by adding the other information that the Twitter API can provide, but this is not necessary for our application.
Glue's output format is Apache Parquet which is a free and open-source column-oriented data storage format of the Apache Hadoop ecosystem.
It is similar to the other columnar-storage file formats available in Hadoop namely RCFile and ORC.

When the buffer conditions condition is triggered on Kinesis, the data is sent to S3 which is the main storage service of AWS. So a bucket called `'sapienza-b'` has been created; it is organized into directories divided by year / month / day / hour. Then the output parquet file created by kinesis is sent to the right folder. You could easily establish a policy to delete for example periodically the files created by at least 7 days to keep the bucket size limited.

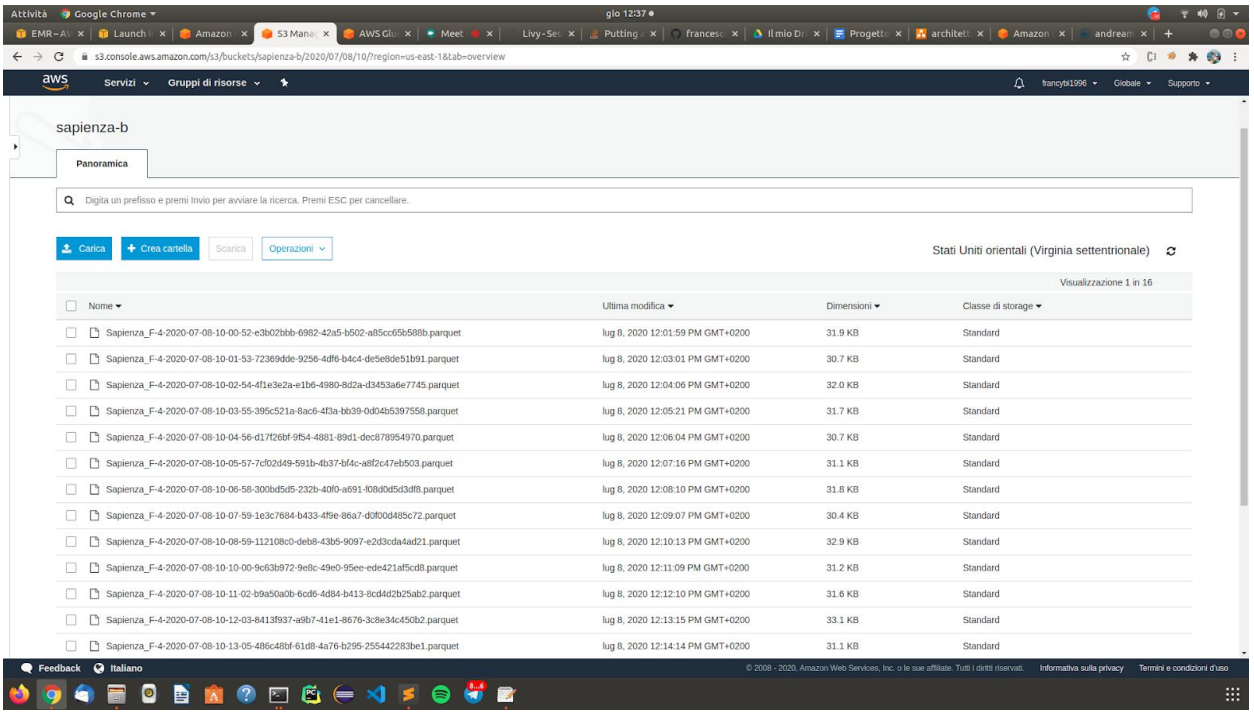


Figure 5: Screenshot of sapienza-b bucket

Instead, on the client side, the user interfaces with the system using the web-app. It was developed using the Streamlit python framework used for datascience applications. The user makes requests that are sent through the Apache Livy rest API to the EMR cluster which performs the various tasks and returns the result to the user.

```
if (button_clicked):
    # Qui faccio partire una funzione che mi ricalcola il nuovo df con le statistiche spark
    tweets = spark_twitter.spark_search(selected, interval)
    del tweets[0]

    if len(tweets) == 0:
        t = "<div><span class='highlight red'> <span class='bold'>Nessun tweet collegato all'argomento</span></span></div>"
        st.markdown(t, unsafe_allow_html=True)
        #st.write("Nessun tweet collegato all'argomento")

    else:
        for x in tweets:
            x = clean_text(x)
            st.write(x)

        # Altre 2 funzioni di statistiche
        retweets = spark_twitter.count_retweet(tweets)
        tags = spark_twitter.count_tags(tweets)

        if retweets != 0:
            t = "<div><span class='highlight blue'><span class='bold'>Numero retweets relativi all'argomento: "+ str(retweets) +" </span></span></div>"
            st.markdown(t, unsafe_allow_html=True)

        if len(tags) > 0:
            df_tags = pd.DataFrame(tags, columns =['Tags','occurences'])
            df_tags = df_tags.rename(columns={'Tags':'index'}).set_index('index') # Setto la colonna hashtag come indice per visualizzarla su xaxis

            df_tags_limit = df_tags.head(20)
            st.dataframe(df_tags_limit)

            chat_data_tags = pd.DataFrame(df_tags_limit, columns=['Tags', 'occurences'])
            st.bar_chart(chat_data_tags)

        dict_result = spark_twitter.word_count(tweets)
```

Figure 6: snippet of Streamlit implementation

The interface is responsive and accessible from any device as shown in the following figures.

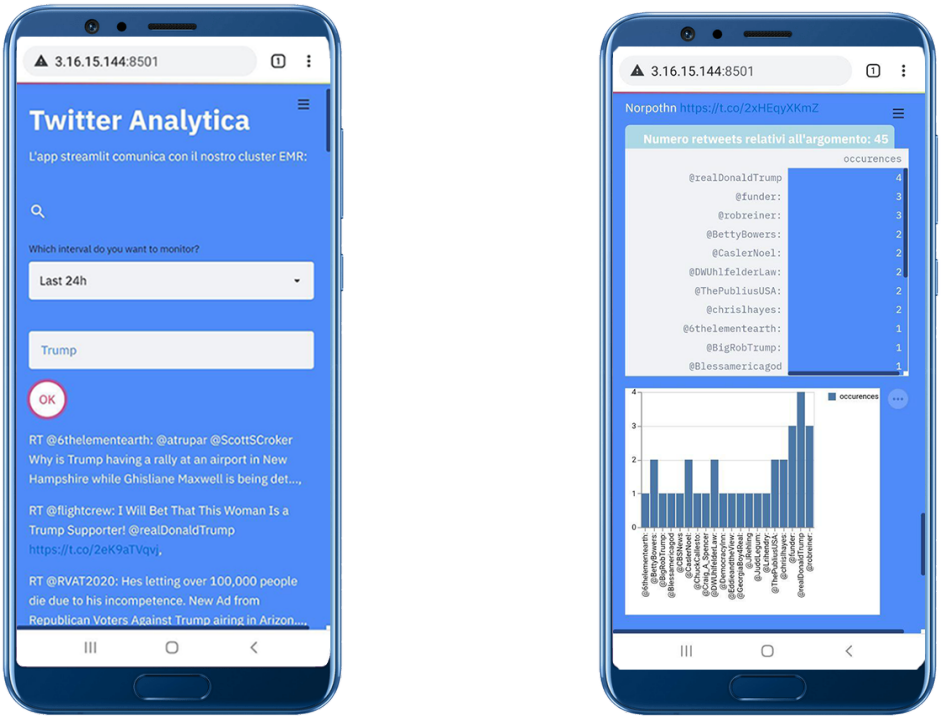
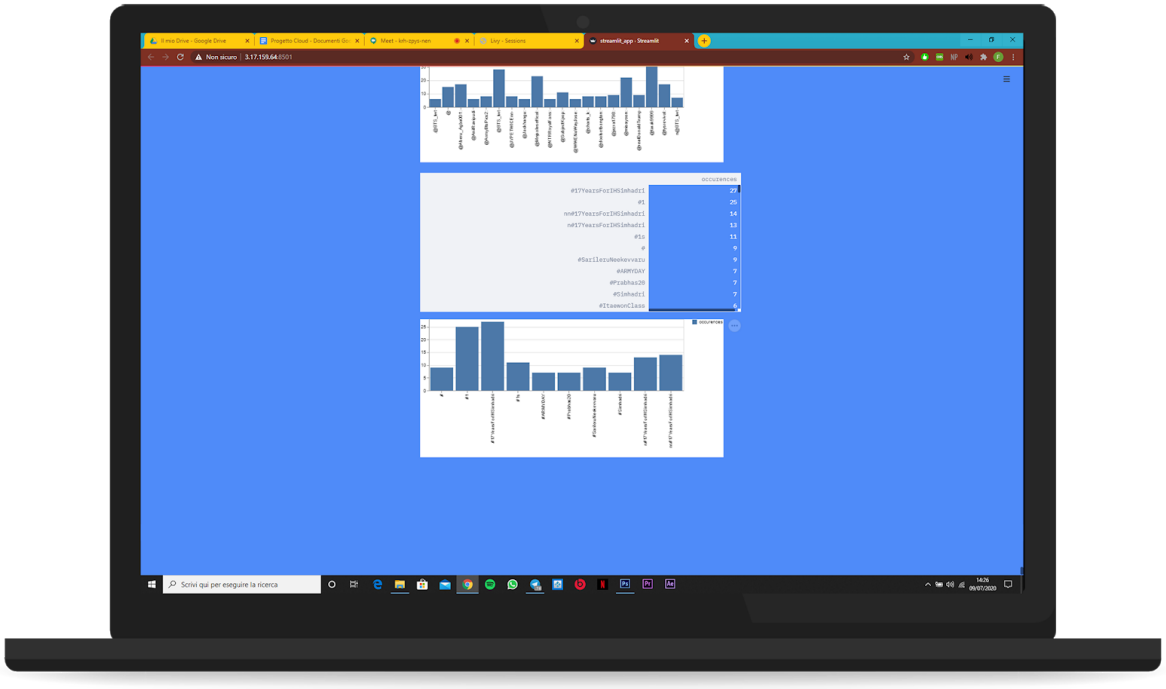


Figure 7: Mobile design of Twitter Analytica



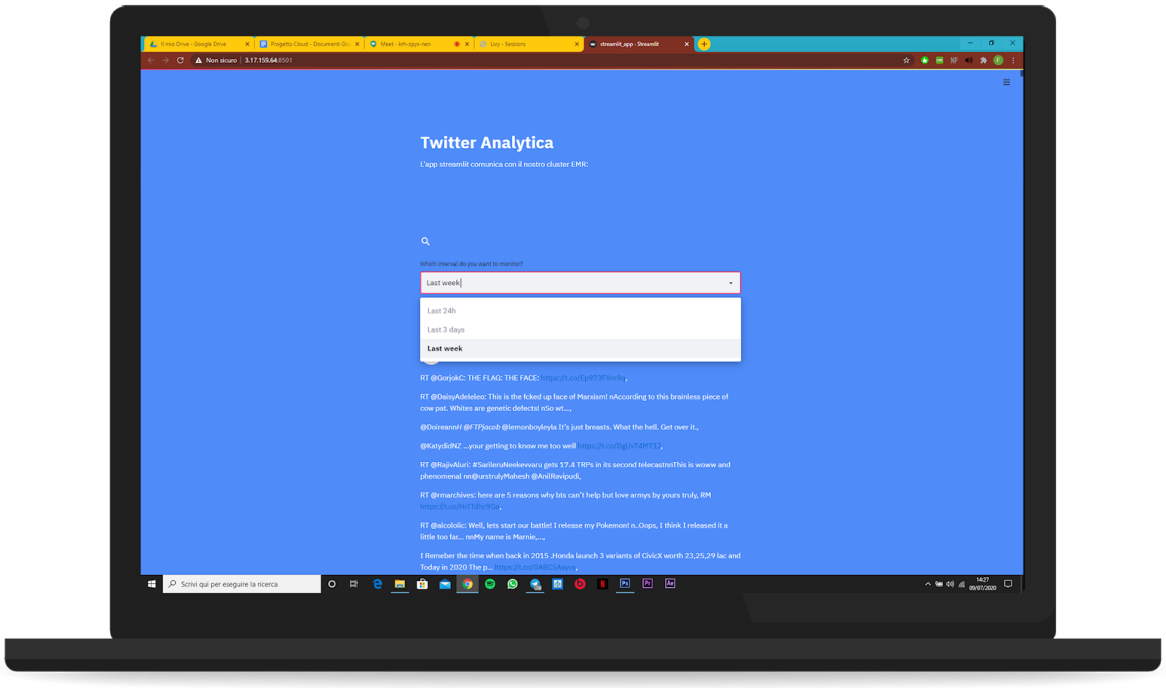


Figure 8: Desktop design of Twitter Analytica

To scale the web application we decided to use docker to create a container and distribute it on the Amazon Elastic Container Service.

The various statistics and calculation operations on the stream of the various tweets have been realized through the Apache Spark framework, running on an Amazon EMR cluster.

Since it is not possible to directly make http calls to the EMR cluster it is necessary the use of another tool called Apache Livy.
It enables programmatic, fault-tolerant, multi-tenant submission of Spark jobs from web/mobile apps (no Spark client needed). So, multiple users can interact with your Spark cluster concurrently and reliably.

The EMR cluster provides an Apache Livy management interface at the following address: `http://master-public-dns-name:8998/`.

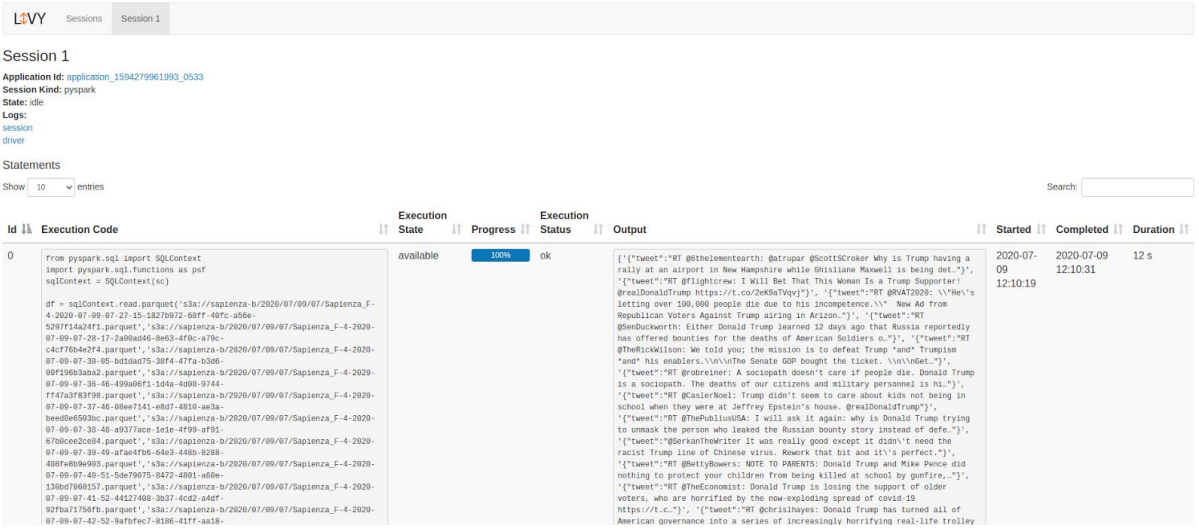


Figure 9: Screenshot of Apache Livy management interface

To work with Apache LIVY you need to create a session. Running an interactive session with Livy is similar to using Spark shell or PySpark, but the shell does not run locally. Instead, it runs in a remote cluster, transferring data back and forth through a network. The Livy REST API supports GET, POST, and DELETE calls for interactive sessions.

Session management was developed entirely by us using Python code in the `init_session()` method:

```
def init_session():
    global statements_url
    r = requests.get(session_url, headers=headers)

    sessions_json = r.json()
    session_id = 0
    session_found = False

    for s in sessions_json['sessions']:
        if s['state'] == 'idle' or s['state'] == 'busy':
            session_id = s['id']
            session_found = True
            statements_url = session_url + str(session_id) + '/statements'
            break


    if not session_found:
        r = requests.post(host + '/sessions', data=json.dumps(data), headers=headers)
        new_session_id = r.json()['id']
        condition_start = False
        while not condition_start:
            r = requests.get(host + '/sessions/'+str(new_session_id), headers=headers)
            if r.json()['state'] == 'idle':
                statements_url = session_url + str(new_session_id) + '/statements'
                condition_start = True
```

Figure 10: Snippet of init_session() method

When a user use the web-application he tries to connect to an active session of apache livy; otherwise a new session is created. Then just define spark code (pyspark in our case) and submit it to the cluster as an http call payload.

```
data = {
  'code': textwrap.dedent("""
    textFile = sc.parallelize(['""" +text+"""'])
    counts = textFile.flatMap(lambda line: line.split(" ")).filter(lambda w: '@' in w).map(lambda word: (word, 1)).reduceByKey(lambda v1,v2: v1 + v2)
    counts.collect()
    """)
}
r = requests.post(statements_url, data=json.dumps(data), headers=headers)
```

Figure 11: sample of Apache Livy HTTP request to EMR Cluster

The answer is a json that is parsed by us to take the result of the Spark operation.

```
while not condition:
    r = requests.get(statements_url, headers=headers)
    response = r.json()

    for x in response['statements']:

        if (x['id'] == id_request):
            current_state = x['state']
            print(current_state)
            if current_state == 'available':
                query_result = x
                condition = True
            else:
                time.sleep(2)
                break

query_result = query_result['output']['data']

query_result = list(query_result.values())[0]

dict_result = ast.literal_eval(query_result)

dict_result = sorted(dict_result)
dict_result = sorted(dict_result, key=lambda x: x[1], reverse=True)
```

Figure 12: Apache Livy response parsing

Once the data is continuously saved cleanly in the bucket, the main important task of our project begins, namely the analysis of this data on demand at the request of a user through a special web application. The data analysis is carried out through Apache Spark which runs on an AWS EMR cluster. Amazon EMR is the industry-leading cloud big data platform for processing vast amounts of data using open source tools such as Apache Spark, Apache Hive, Apache HBase, Apache Flink, Apache Hudi, and Presto. Regarding the cluster configuration and scaling policies will be discussed in the next chapter, while we will now focus on the development of the various functionalities in PySpark.

Since our aim is to call functionality from a web app we have chosen to use Apache Livy to interface with the cluster. At this point we have defined an API in Python to send requests from the web app to the cluster. Each method of this API performs a series of tasks defined in PySpark.

```
all_today_files = []

s3 = boto3.resource('s3')
my_bucket = s3.Bucket('sapienza-b')

tod = datetime.datetime.now()

if interval == "Last 24h":
    date_today = tod.strftime('%Y/%m/%d')
    print(date_today)
    for object_summary in my_bucket.objects.filter(Prefix =str(date_today)+'/'):
        all_today_files.append('\'s3a://sapienza-b/' + object_summary.key + '\'')
elif interval == "Last 3 days":
    for i in range(3):
        d = datetime.timedelta(days = i)
        a = tod - d
        current_day = a.strftime('%Y/%m/%d')
        for object_summary in my_bucket.objects.filter(Prefix =str(current_day)+'/'):
            all_today_files.append('\'s3a://sapienza-b/' + object_summary.key + '\'')

else:
    for i in range(7):
        d = datetime.timedelta(days = i)
        a = tod - d
        current_day = a.strftime('%Y/%m/%d')
        for object_summary in my_bucket.objects.filter(Prefix =str(current_day)+'/'):
            all_today_files.append('\'s3a://sapienza-b/' + object_summary.key + '\'')

parquets_path = ','.join(all_today_files)

data = {
  'code': textwrap.dedent("""
    from pyspark.sql import SQLContext
    import pyspark.sql.functions as psf
    sqlContext = SQLContext(sc)

    df = sqlContext.read.parquet("""+parquets_path+""")
    #print(df.count())
    words_list = ['"""+keyword+"""']
    result = df.filter(psf.col('text').rlike('(^|\s)(' + '|'.join(words_list) + ')(\s|$)'))
    result_list = result.select("text").rdd.flatMap(lambda x: x).collect()
    print(result_list)
    """)
}

print(statements_url)

r = requests.post(statements_url, data=json.dumps(data), headers=headers)
print(r)
```

```
def word_count(tweets):
    text = clean_text(tweets)

    data = {
      'code': textwrap.dedent("""
        textFile = sc.parallelize(['""" +text+"""'])
        counts = textFile.flatMap(lambda line: line.split(" ")).filter(lambda w: '#' in w).map(lambda word: (word, 1)).reduceByKey(lambda v1,v2: v1 + v2)
        counts.collect()
        """)
    }
    r = requests.post(statements_url, data=json.dumps(data), headers=headers)

    k = r.json()
    id_request = k['id']
    state_request = k['state']

    condition = False
    query_result = {}

    while not condition:
        r = requests.get(statements_url, headers=headers)
        response = r.json()

        for x in response['statements']:

            if (x['id'] == id_request):
                current_state = x['state']
                print(current_state)
                if current_state == 'available':
                    query_result = x
                    condition = True
                else:
                    time.sleep(2)
                    break

    query_result = query_result['output']['data']

    query_result = list(query_result.values())[0]

    dict_result = ast.literal_eval(query_result)
```

Figure 13: API to interface with the SPARK cluster

When the user starts searching for a topic from the interface, the `def spark_search(keyword, interval):` method is called. Based on the keyword and the time interval, a Spark dataframe is created from the parquet files in the S3 bucket. The dataframe is filtered to search for tweets related to the topic and the final RDD of the tweets is created.

```
result = df.filter(psf.col('text').rlike('(^|\s)(' +
'|'.join(words_list) + ')(\s|$)'))
result_list = result.select("text").rdd.flatMap(lambda x:
x).collect()
```

The other API methods are a series of operations on this RDD. For example, let's see the `word_count` method that searches for the most used hashtags related to the topic. Let's see how it works:

```
textFile = sc.parallelize(['""" +text+"""'])
counts = textFile.flatMap(lambda line: line.split("
")).filter(lambda w: '#' in w).map(lambda word: (word,
1)).reduceByKey(lambda v1,v2: v1 + v2)
counts.collect()
```

The RDD containing the tweets to be examined is created via `sc.parallelize`. The `flatMap` splits the text; subsequently this text is filtered to search for hashtags (through the

`filter` function); the `map` creates the various pairs (hashtag, 1) and finally `reduceByKey` aggregates the pairs with the same key. The `collect` method returns the contents of the RDD.

Description of the deployment of your solution

Once all the various services are implemented, it is necessary to configure them in the best possible way to have a highly scalable and functioning architecture in the cloud environment.

Concerning the collection of the tweets stream is concerned, the most appropriate choice was to dockerize the python API previously made in order to make the image available in the hub docker so that it can be easily installed in any machine and anywhere in the world with a simple command.

```
FROM python:3.7

WORKDIR /app

COPY requirements.txt ./requirements.txt

RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python", "./tweepy-kinesis.py" ]
```

Figure 14: DockerFile of Tweepy-Kinesis.py

Once we wrote the docker file (Figure X) and created the docker container on our machine, we published it in the docker hub.

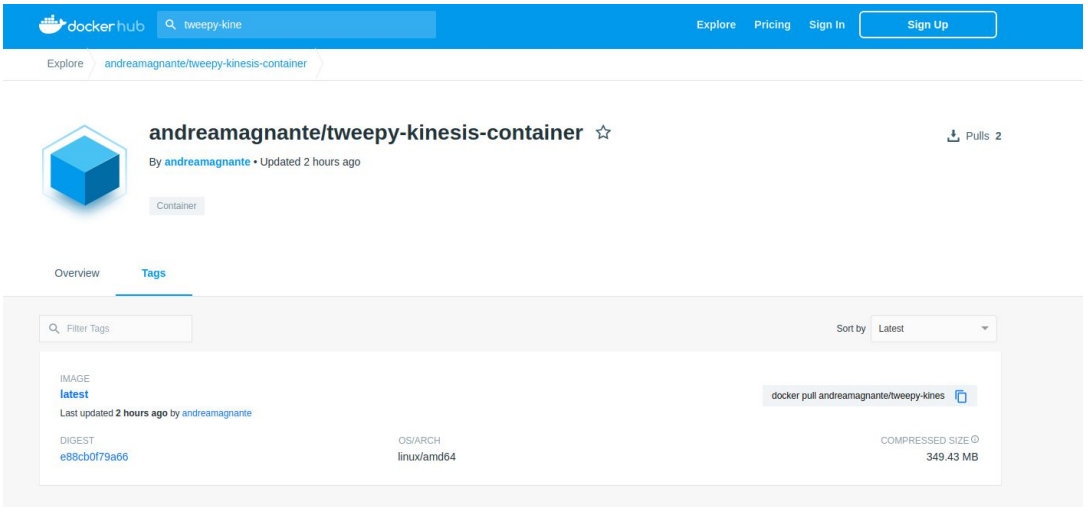


Figure 15: docker image in docker hub

Once published, this container is sent running on an Amazon EC2 machine to collect tweets in real time and store them in the S3 bucket.

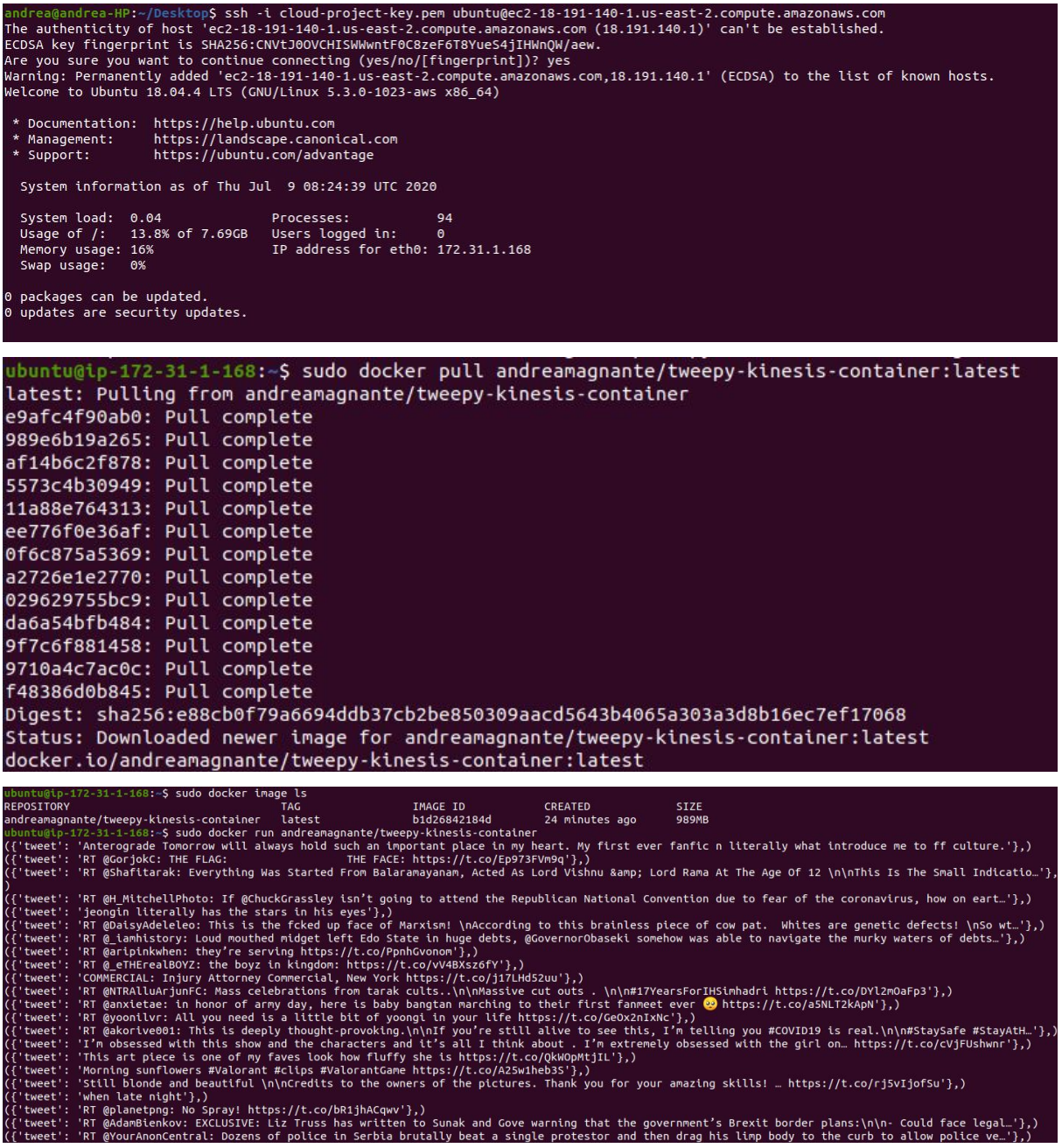


Figure 16: screenshot of docker image launched on EC2 machine

In this case, considering the simplicity of the script to launch and the constant workload to maintain, we did not consider necessary to define scaling policies.

Regarding the Amazon Kinesis configuration, we used the following setting:

- buffer conditions: 64 MiB or 60 seconds.

Also in kinesis firehose there is a section called monitoring which keeps track of a series of interesting statistics such as: Incoming bytes per second, Incoming put requests per second, Records successfully converted using AWS Glue (Sum), Delivery to Amazon S3 success etc.

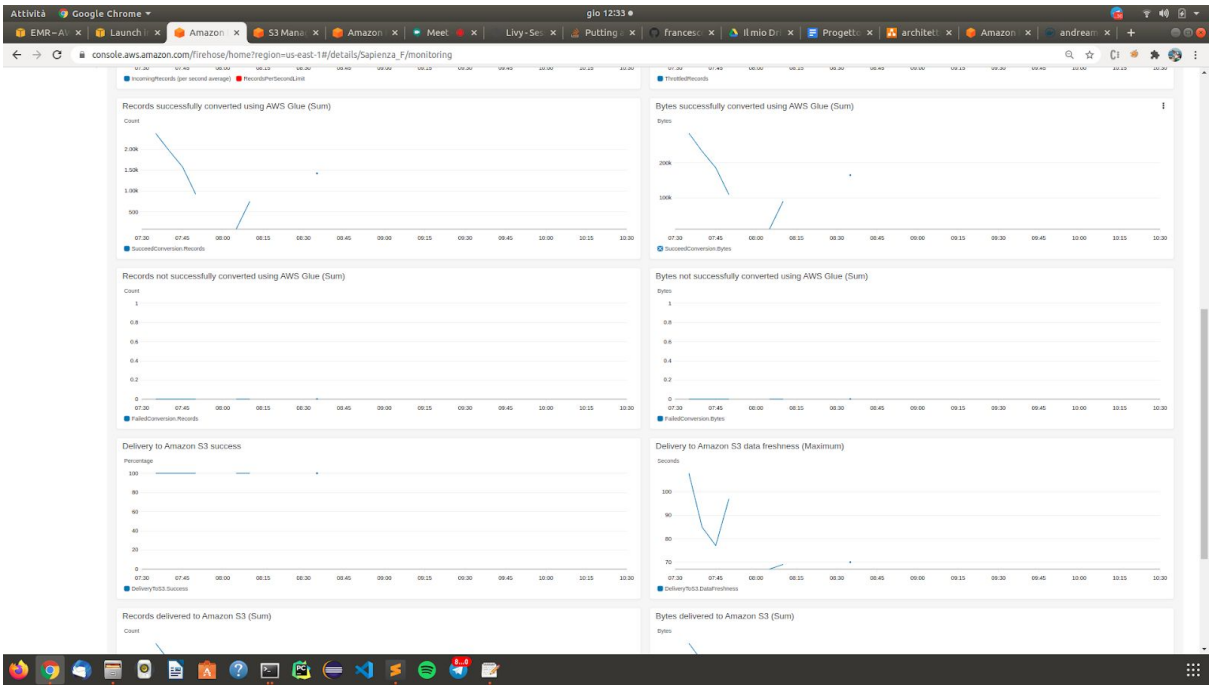


Figure 17: monitoring section of Kinesis Firehose

The main parts of the deployment phase were to configure the cluster to process data with Apache Spark and the cluster on which the web application runs.
Regarding Apache Spark we used the Amazon service called EMR (Elastic Map Reduce).

The cluster was created with the properties shown in the figure:

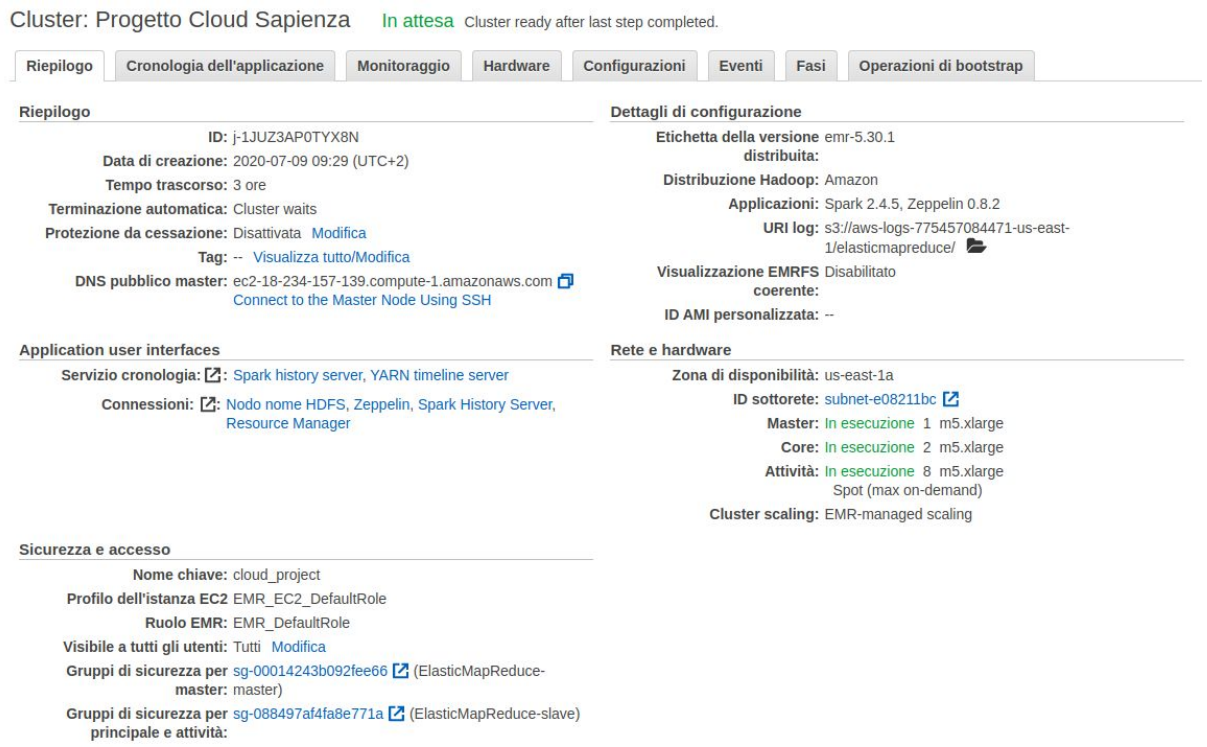


Figure 18: screenshot of EMR Cluster configuration

EMR has an automatically managed scaling to automatically increase or decrease the number of instances or units in the cluster based on the workload. EMR continuously evaluates cluster metrics to make scaling decisions that optimize clusters for cost and speed. This feature is available for clusters made up of instance groups or instance fleets.

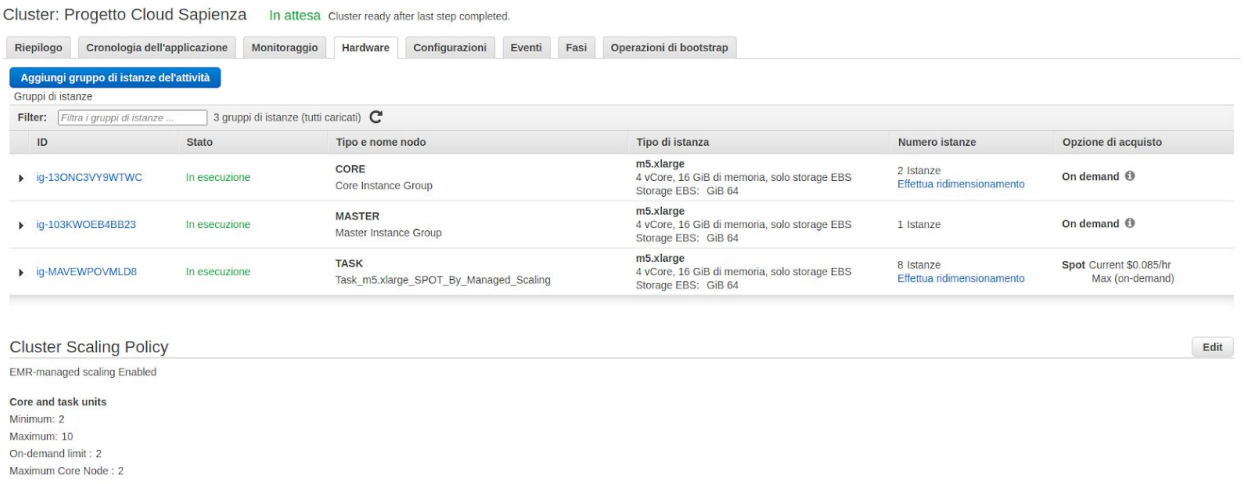


Figure 19: EMR Cluster scaling policy

We have set some parameters regarding the configuration of autoscaling:

- **Minimum** (MinimumCapacityUnits): **2.** The lower boundary of allowed EC2 capacity in a cluster. It is measured through virtual central processing unit (vCPU) cores or instances for instance groups. It is measured through units for instance fleets.
- **Maximum** (MaximumCapacityUnits): **10**. The upper boundary of allowed EC2 capacity in a cluster. It is measured through virtual central processing unit (vCPU) cores or instances for instance groups. It is measured through units for instance fleets.
- **On-Demand limit** (MaximumOnDemandCapacityUnits): **2**. The upper boundary of allowed EC2 capacity for On-Demand market type in a cluster. If this parameter is not specified, it defaults to the value of MaximumCapacityUnits.
- **Maximum core nodes** (MaximumCoreCapacityUnits): **2** The upper boundary of allowed EC2 capacity for core node type in a cluster. If this parameter is not specified, it defaults to the value of MaximumCapacityUnits.

  This parameter is used to split capacity allocation between core and task nodes. For example, if you set the minimum parameter as 2 instances, the maximum as 100 instances, the maximum core node as 17 instances, then EMR managed scaling scales up to 17 core nodes and allocates the remaining 83 instances to task nodes.
  In defining this policy we also had to take into account the high costs of the machines used in the cluster: the m5.xlarge

  m5.xlarge: vCPU: 4 **Memory(GiB):** 16 **Instance Storage(GiB):** EBS only
  Network Bandwidth(Gbps): Up to 10 **EBS Bandwidth(Mbps):** Up to 4,750

The cluster is configured to work with Apache Spark. Once the cluster is in action, there are various ways to communicate:
- You can connect via ssh and launch your spark programs directly from the command line.
- You can write a jupyter notebook and launch it on the cluster.
- You can connect through the Rest Apache Livy API.

Once the EMR cluster has started, a series of interfaces are available which are used to monitor correct operation.
For example, at `http://master-public-dns-name:18080/` we find the Spark History Server.

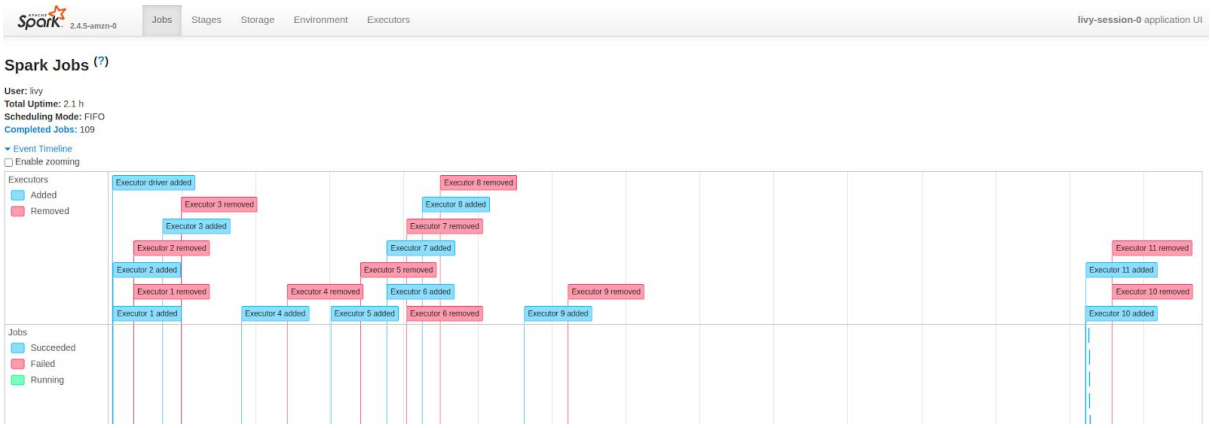


Figure 20: Spark history server interface

The Spark history server is a monitoring tool that displays information about completed Spark applications. This information is pulled from the data that applications by default write to a directory on Hadoop Distributed File System (HDFS).

The last part of the project was to deploy the web-app on a cluster configured in the best way to make the application scalable and available in different data centers.

The first step was to dockerize the web-app and proceed to configuration by defining a docker file.

```
FROM python:3.7

EXPOSE 8501

WORKDIR /app

COPY requirements.txt ./requirements.txt

RUN pip3 install -r requirements.txt

COPY . .

CMD streamlit run streamlit_app.py
```

Figure 21: Docker file configuration of the web-app

As shown in figure X, port 8501 is displayed, that is the port where the Streamlit framework runs; the various requirements (streamlit, pandas, matplotlib, boto3, etc) have been installed; moreover, the workdir is defined and finally, as the last command, `streamlit run` is used to launch the application on the machine.

Unlike the container created previously, in this case the image is distributed on Amazon ECR (Elastic Container Registry) which can be considered the equivalent of the docker hub in Amazon.

Figure 22: Docker image of Twitter-analytica on Amazon ECR

The best approach to distribute containers on AWS is to use Amazon ECS (Elastic Container Service) which is a fully managed container orchestration service.

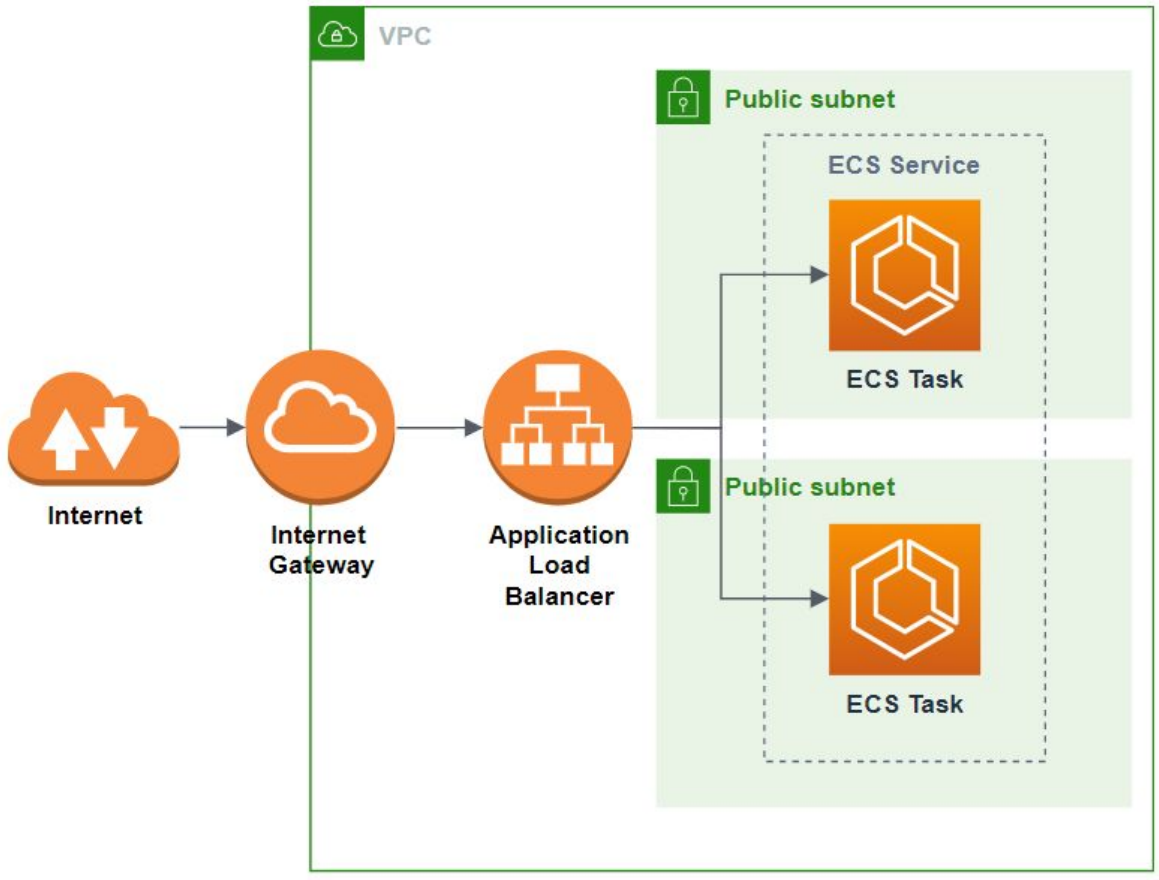


Figure 23: sample of Load Balancer orchestration

In order to correctly configure ECS together with the application load balancer, we used the Amazon CloudFormation service.
It allows us to describe the resources we want (networks, load balancers, EC2 instances, …) in a JSON or YAML template and provides commands within the AWS CLI to configure those resources.

We'll create two CloudFormation stacks:

- A network stack that creates a VPC (virtual private cloud) with three public subnets (each in a different availability zone for high availability), an internet gateway, and a load balancer that balances traffic between those networks.

- A service stack that places a Docker container with the application we want to run into each of the public networks. For this, we take advantage of ECS (Elastic Container Service) and Fargate, which together abstract away some of the gritty details and make it easier to run a Docker container.

```
VPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: '10.0.0.0/16'

PublicSubnetOne:
  Type: AWS::EC2::Subnet
  Properties:
    AvailabilityZone:
      Fn::Select:
        - 0
        - Fn::GetAZs: {Ref: 'AWS::Region'}
    VpcId: !Ref 'VPC'
    CidrBlock: '10.0.1.0/24'
    MapPublicIpOnLaunch: true

PublicSubnetTwo:
  Type: AWS::EC2::Subnet
  Properties:
    AvailabilityZone:
      Fn::Select:
        - 1
        - Fn::GetAZs: {Ref: 'AWS::Region'}
    VpcId: !Ref 'VPC'
    CidrBlock: '10.0.2.0/24'
    MapPublicIpOnLaunch: true

PublicSubnetThree:
  Type: AWS::EC2::Subnet
  Properties:
    AvailabilityZone:
      Fn::Select:
        - 2
        - Fn::GetAZs: {Ref: 'AWS::Region'}
    VpcId: !Ref 'VPC'
    CidrBlock: '10.0.3.0/24'
    MapPublicIpOnLaunch: true
```

```
ECSCluster:
  Type: AWS::ECS::Cluster

ECSSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: Access to the ECS containers
    VpcId: !Ref 'VPC'

ECSSecurityGroupIngressFromPublicALB:
  Type: AWS::EC2::SecurityGroupIngress
  Properties:
    Description: Ingress from the public ALB
    GroupId: !Ref 'ECSSecurityGroup'
    IpProtocol: -1
    SourceSecurityGroupId: !Ref 'PublicLoadBalancerSecurityGroup'

ECSSecurityGroupIngressFromSelf:
  Type: AWS::EC2::SecurityGroupIngress
  Properties:
    Description: Ingress from other containers in the same security group
    GroupId: !Ref 'ECSSecurityGroup'
    IpProtocol: -1
    SourceSecurityGroupId: !Ref 'ECSSecurityGroup'
```

Figure 24: Snippet of network.yml

The following instructions have been defined in the file network.yml:

- VPC
- Public Subnet
- Internet Gateway
- Routing
- Load Balancer
- ECS Cluster
- Roles

This allowed us to configure the three availability zones to which it was possible to direct traffic even in situations where one of them was not available; the load balancer manages the incoming traffic balancing it among the various subnets created providing a public DNS reference; the ECS Cluster is responsible for the management of the container docker

```
andrea@andrea-HP:~/Desktop/ecs-in-two-public-subnets$ aws cloudformation create-
stack \
>    --stack-name twitter-analytica-network \
>    --template-body file://network.yml \
>    --capabilities CAPABILITY_IAM
{
    "StackId": "arn:aws:cloudformation:us-east-2:963989550427:stack/twitter-anal
ytica-network/ec60c8f0-c28b-11ea-b161-02356f1d181e"
}
```

Figure 25: Creation of Network stack

```
Service:
  Type: AWS::ECS::Service
  DependsOn: LoadBalancerRule
  Properties:
    ServiceName: !Ref 'ServiceName'
    Cluster:
      Fn::ImportValue:
        !Join [':', [!Ref 'StackName', 'ClusterName']]
    LaunchType: FARGATE
    DeploymentConfiguration:
      MaximumPercent: 200
      MinimumHealthyPercent: 50
    DesiredCount: !Ref 'DesiredCount'
    NetworkConfiguration:
      AwsvpcConfiguration:
        AssignPublicIp: ENABLED
        SecurityGroups:
          - Fn::ImportValue:
              !Join [':', [!Ref 'StackName', 'ECSSecurityGroup']]
        Subnets:
          - Fn::ImportValue:
              !Join [':', [!Ref 'StackName', 'PublicSubnetOne']]
          - Fn::ImportValue:
              !Join [':', [!Ref 'StackName', 'PublicSubnetTwo']]
    TaskDefinition: !Ref 'TaskDefinition'
    LoadBalancers:
      - ContainerName: !Ref 'ServiceName'
        ContainerPort: !Ref 'ContainerPort'
        TargetGroupArn: !Ref 'TargetGroup'
```

Figure 26: Snippet of service.yml

The following instructions have been defined for the service.yml file:

- Paramters
- Re-route the Load Balancer
- Task definition
- ECS Service

```
andrea@andrea-HP:~/Desktop/ecs-in-two-public-subnets$ aws cloudformation create-
stack \
>    --stack-name twitter-analytica-service \
>    --template-body file://service.yml \
>    --parameters \
>        ParameterKey=StackName,ParameterValue=twitter-analytica-network \
>        ParameterKey=ServiceName,ParameterValue=twitter-analytica \
>        ParameterKey=ImageUrl,ParameterValue=963989550427.dkr.ecr.us-east-2.amaz
onaws.com/twitter-analytica-container:latest \
>        ParameterKey=ContainerPort,ParameterValue=8501 \
>        ParameterKey=HealthCheckPath,ParameterValue=/ \
>        ParameterKey=HealthCheckIntervalSeconds,ParameterValue=90
{
    "StackId": "arn:aws:cloudformation:us-east-2:963989550427:stack/twitter-anal
ytica-service/852b5140-c28c-11ea-8a8e-02402ec64522"
}
```

Figure 27: command to create cloudformation stack

Using the AWS CLI to create the service stack we have defined the following parameters: the name of the service, the address at the container image published on Amazon ECR, the container port 8501 and its path.

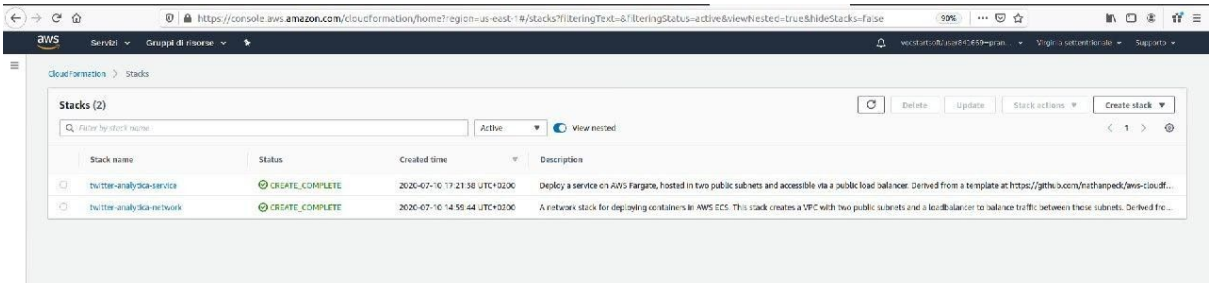


Figure 28: the two stacks created active on AWS

Once the process was completed via cloudformation, an ECS Cluster composed of one service and 3 active tasks was automatically created.

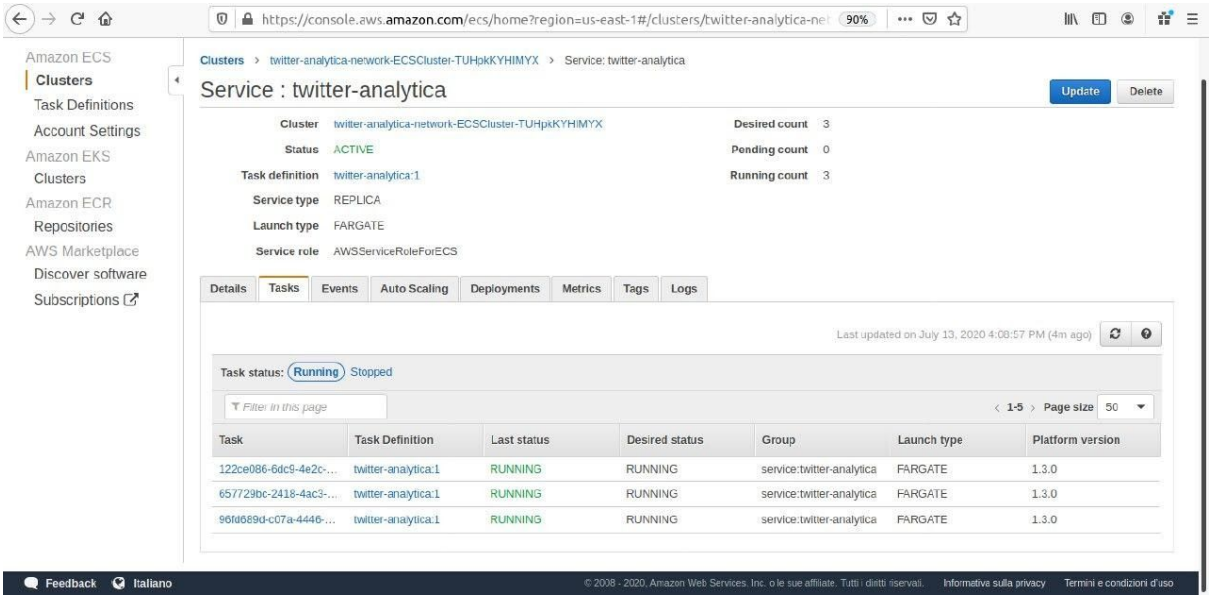


Figure 29: ECS cluster with the default configuration (3 tasks)

In this way it is possible to obtain a public DNS useful to connect to the web-app; moreover, the workload balancement is assigned to the load balancer (shown in the figure X) that redirect the network traffic among the available tasks.

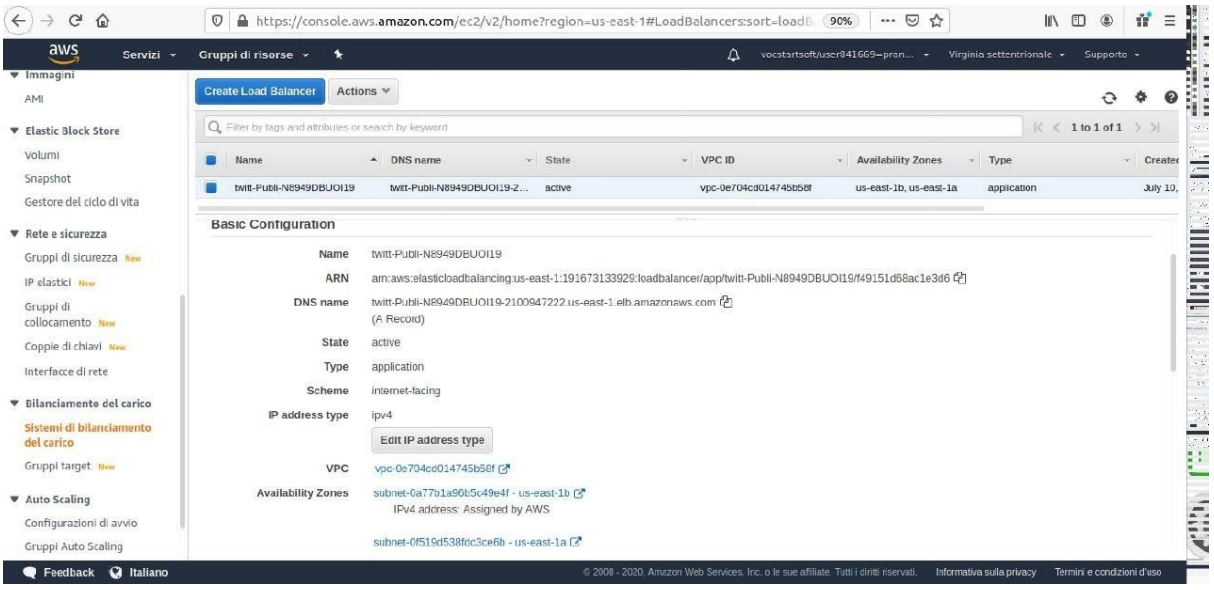


Figure 30: Load Balancer definition

After the service configuration, we have defined the service autoscaling (Automatic scaling is the ability to increase or decrease the desired count of tasks in your Amazon ECS service automatically) with the following parameters:

- **Minimum number of tasks**: 3; the lower limit of the number of tasks for Service Auto Scaling to use. Your service's desired count is not automatically adjusted below this amount.

- **Desired number of tasks**: 5; this field is pre-populated with the value that you entered earlier. You can change your service's desired count at this time, but this value must be between the minimum and maximum number of tasks specified on this page.

- **Maximum number of tasks:** 10; the upper limit of the number of tasks for Service Auto Scaling to use. Your service's desired count is not automatically adjusted above this amount.

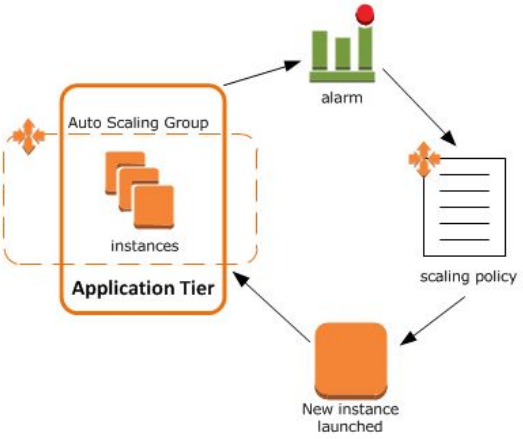
- **IAM role for Service Auto Scaling**: choose the ecsAutoscaleRole. If this role does not exist, choose Create new role to have the console create it for you.

We also added the scaling policy to automate the addition of tasks to the service:

- **Service Autoscaling Policy**: Step scaling policy; Increase or decrease the number of tasks that your service runs based on a set of scaling adjustments, known as step adjustments, that vary based on the size of the alarm breach.

As we can see in the picture, the policy has been set to monitor:
ECSServiceAverageCPUUtilization - Average CPU utilization of the service

To activate the scaling policy the CPU average utilization must be >= of 80% for 60 sec.



Because we have a cooldown period in place, the Auto Scaling group launches an instance and then blocks scaling activities due to simple scaling policies until the specified time elapses (in our case is 300 seconds.) This gives newly launched instances time to start handling application traffic. After the cooldown period expires, any scaling activities that are triggered after the cooldown period can resume. At the end an instance is attached.

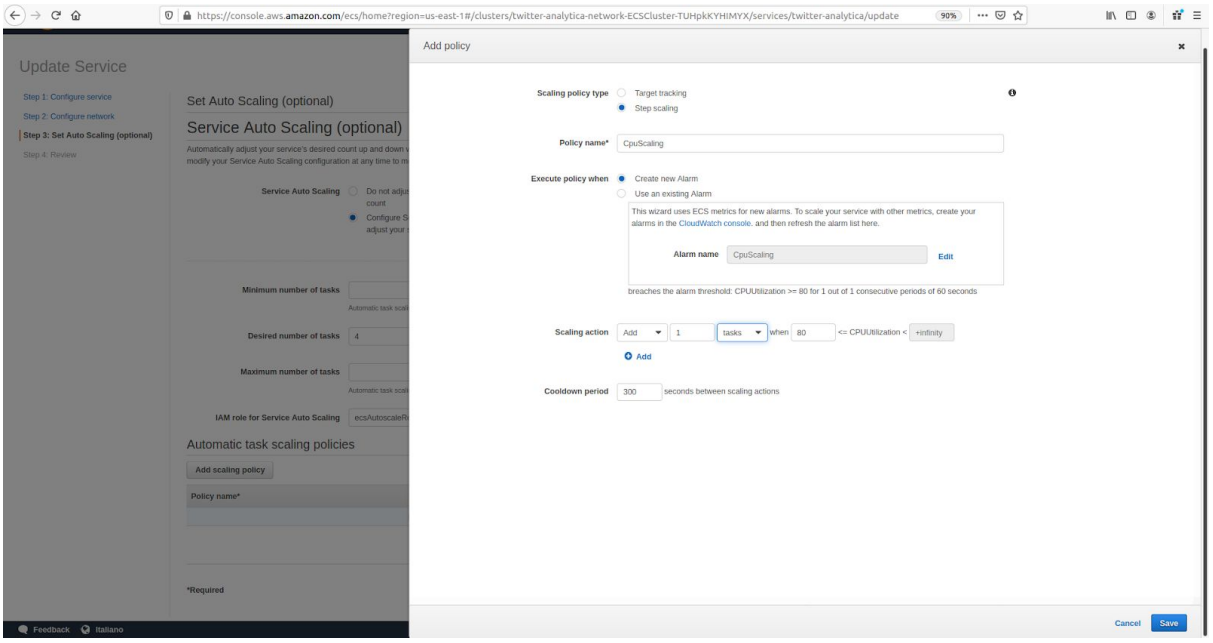


Figure 31: Configuration of step policy scaling

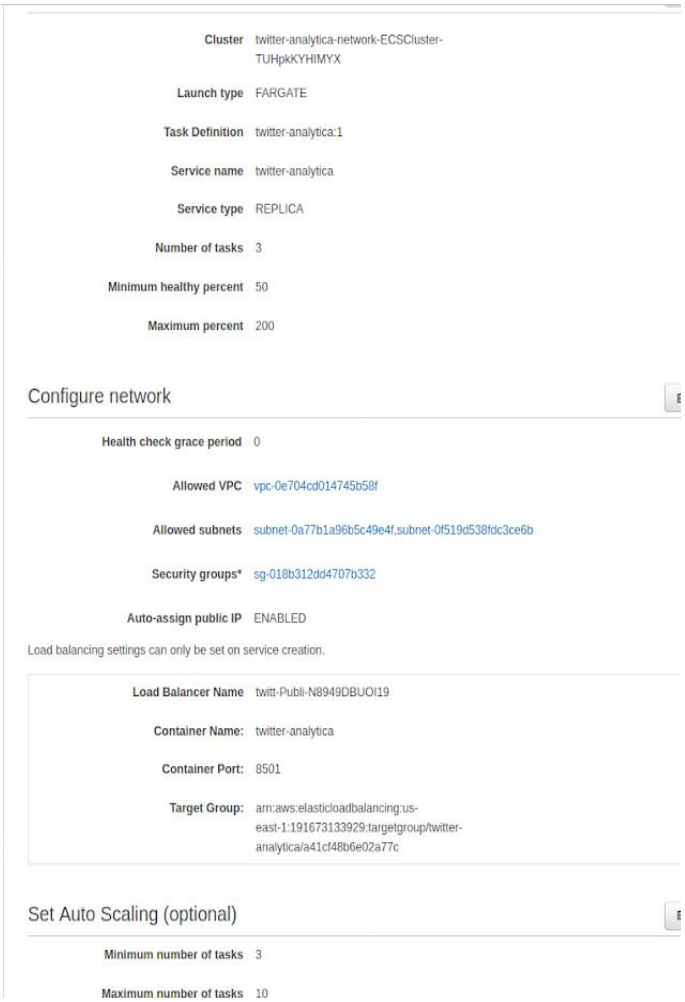


Figure 32: Review Service Configuration

Test/validation design

For the testing phase we focused on stress testing the clusters we configured in order to monitor the performance and proper functioning of the load balancer and scaling policy.
Let's divide this phase in two:

- **Testing of the ECS cluster on which the web-app container has been deployed**
  For the testing of the web app we used more software called Apache JMeter and Apache Bench that allows you to simulate users who simultaneously use the web application by making http requests. Regarding the tests performed with JMeter we saw the behavior of the system when some parameters change, such as the thread group (users who use the application), ramp-up (The ramp-up period tells JMeter how long to take to "ramp-up" to the full number of threads chosen. If 10 threads are used, and the ramp-up period is 100 seconds, then JMeter will take 100 seconds to get all 10 threads up and running. Each thread will start 10 (100/10) seconds after the previous thread was begun. If there are 30 threads and a ramp-up period of 120 seconds, then each successive thread will be delayed by 4 seconds). Instead, concerning Apache Bench, we have carried out several tests by setting the number of total requests to be executed and the number of competing users
  (for example `ab - s 9999 -n 100000 -c 10000 url_site`)

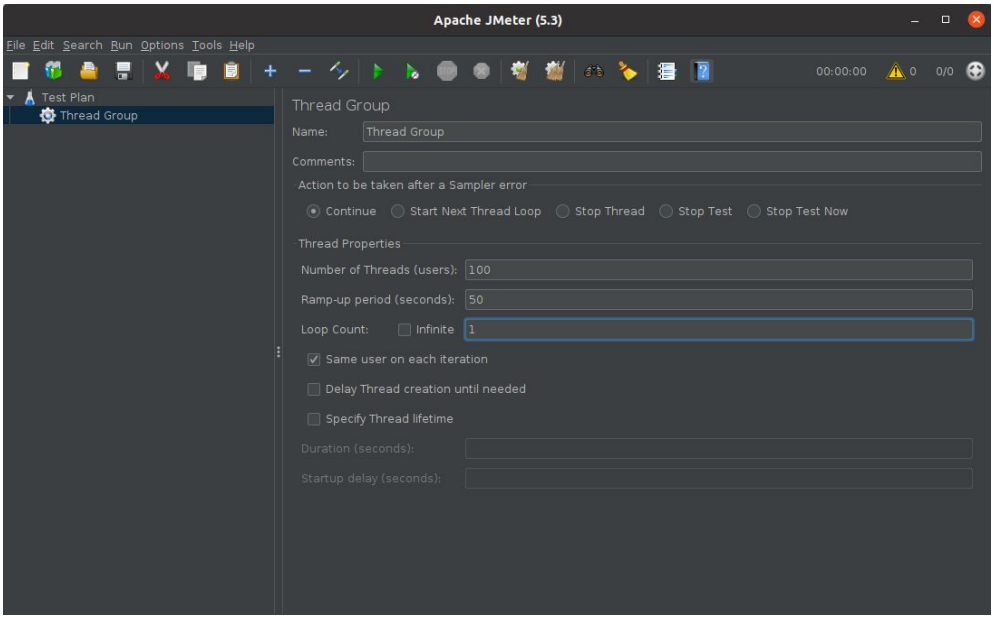


Figure 33: Apache JMeter test configuration.

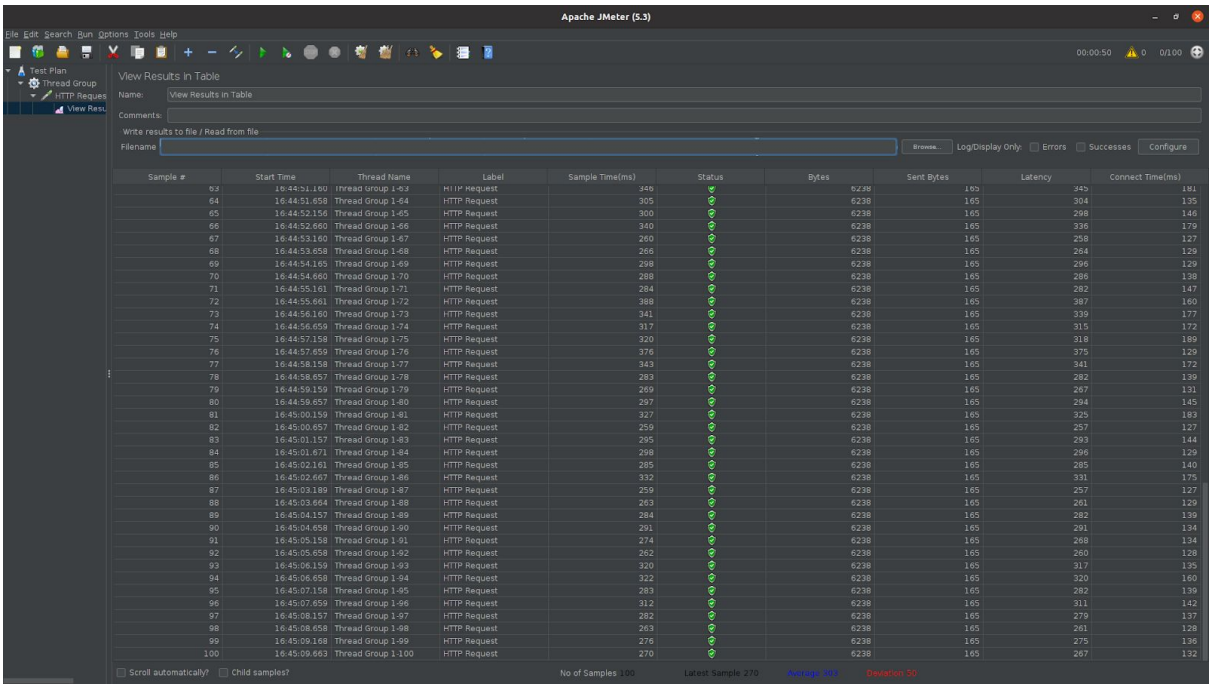


Figure 34: Sample of JMeter test.

The results of the tests will be examined in the next chapter.

- Testing of the EMR cluster on which the calculation operations are performed by Apache Spark

The statistics operations that the user requests through the web-app are not examined on the ECS cluster but are processed by Apache Spark on a dedicated EMR cluster (as specified in the previous implementation and deployment sections).
Another difference with the ECS cluster is that EMR has a default automatic scaling policy. In particular we have set a maximum number of EC2 machines equal to 10.
To test the scaling policy, we also created a script to put workload on the spark cluster using the Apache Livy REST API.

```
xargs -I % -P 8 curl
http://ec2-34-204-8-29.compute-1.amazonaws.com:8998/sessions/0/sta
tements -X POST -H 'Content-Type: application/json' -d '{"code":
"textFile = sc.parallelize(['text.txt'])
          counts = textFile.flatMap(lambda line: line.split("
")).filter(lambda w: '#' in w).map(lambda word: (word,
1)).reduceByKey(lambda v1,v2: v1 + v2)
          counts.collect()"}' \
< <(printf '%s\n' {1..400})
```

This will run give curl command 400 times with max 8 jobs in parallel.

Experimental results

In this section we will report the results of the tests defined in the previous chapter focusing on cluster performance.

- Testing of the ECS cluster on which the web-app container has been deployed

As a first attempt, to better understand how JMeter works and to verify a first reaction of the application subjected to multiple http requests, we launched a session in JMeter with the following parameters :

- **thread-groups**: 500
- **ramp-up**: 25

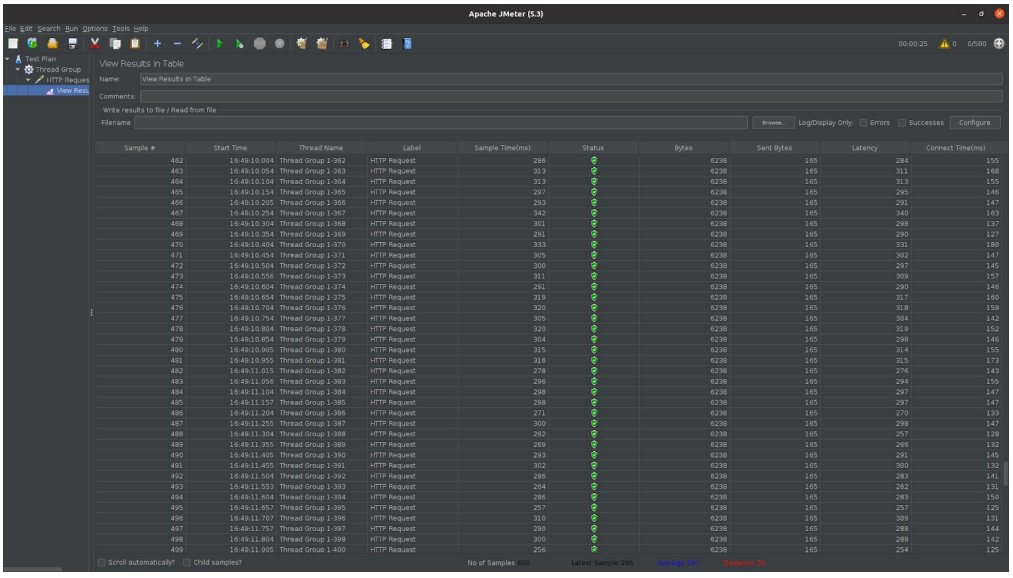


Figure 35: Sample of JMeter test with 500 thread-groups.

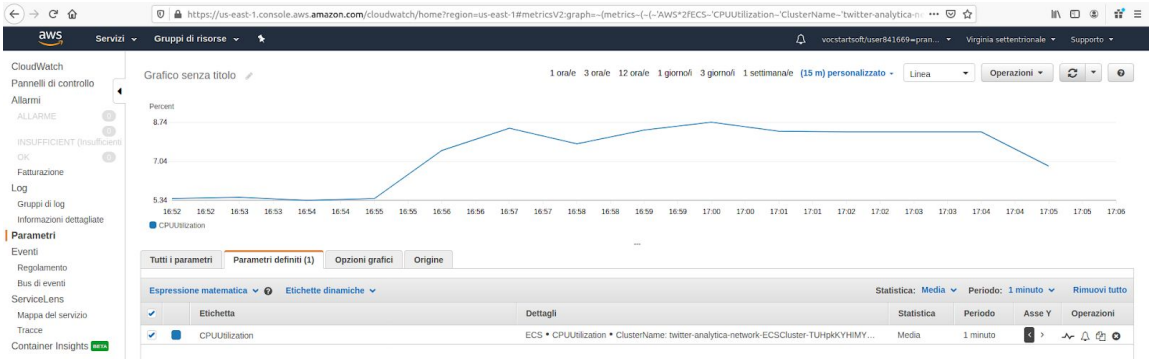


Figure 36: Cpu utilization of this stress test.

We have repeated the following test several times, for a total of 10 minutes of execution, to observe the trend of the average CPU.
The test involved, every 25 seconds, sending 500 HTTP requests to our web-app (via the DNS of the load balancer).

As we can observe from the graph, with the 3 default tasks active, the average CPU utilization examined during the 1 minute time span, reaches a peak of about 8.74% maintaining therefore very low values during the whole duration of the test.
Given these very low values, we decided to increase the workload on the cluster.
The same test, repeated by setting the thread-groups to 2000 and keeping the same ramp-up value, brings the average CPU usage around 14%.

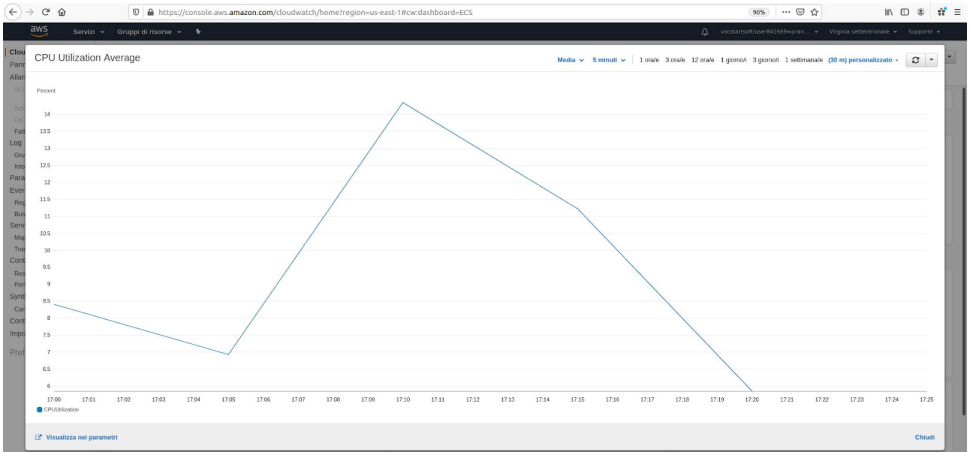
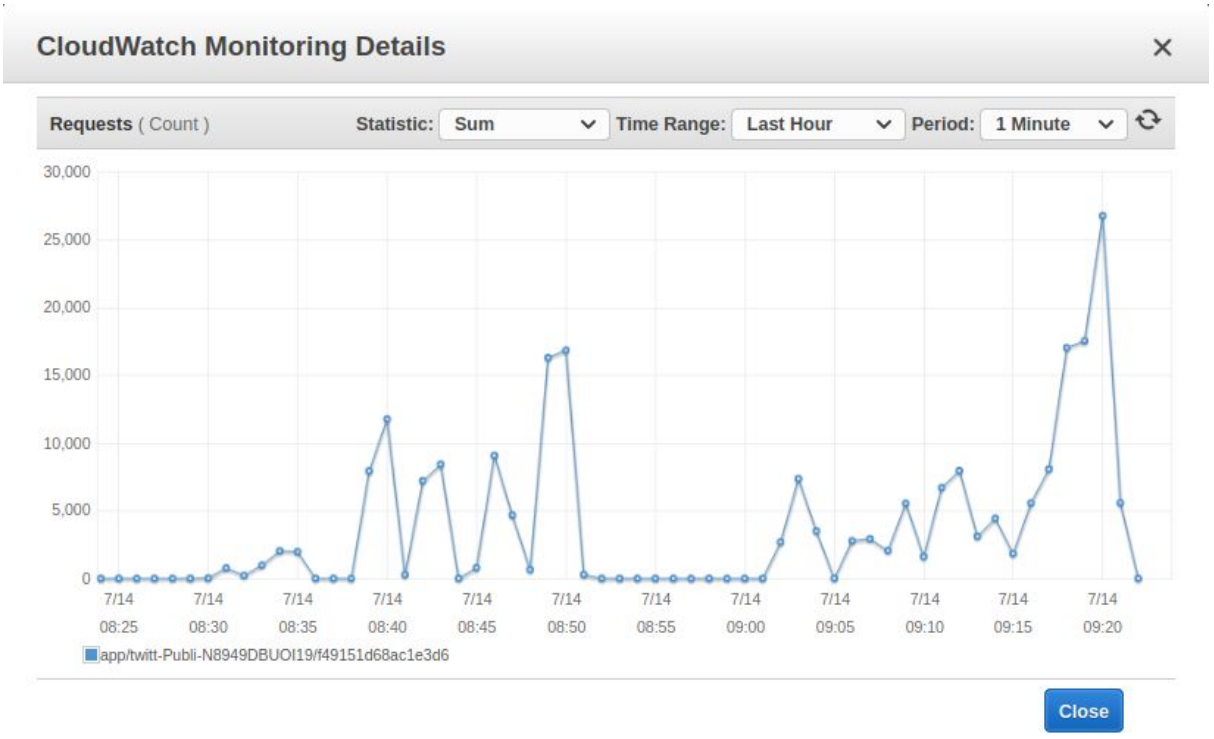


Figure 37: Cpu utilization of stress test with thread groups 2000

In order to perform a more accurate stress test, we decided to launch stress tests via Apache Bench using EC2 machines that have a higher bandwidth than our machines.
The first attempt was to make 30,000 requests using 5000 concurrent users. To do this we launched the following command:

```
ab -s 9999 -n 30000 -c 5000 url_path
```



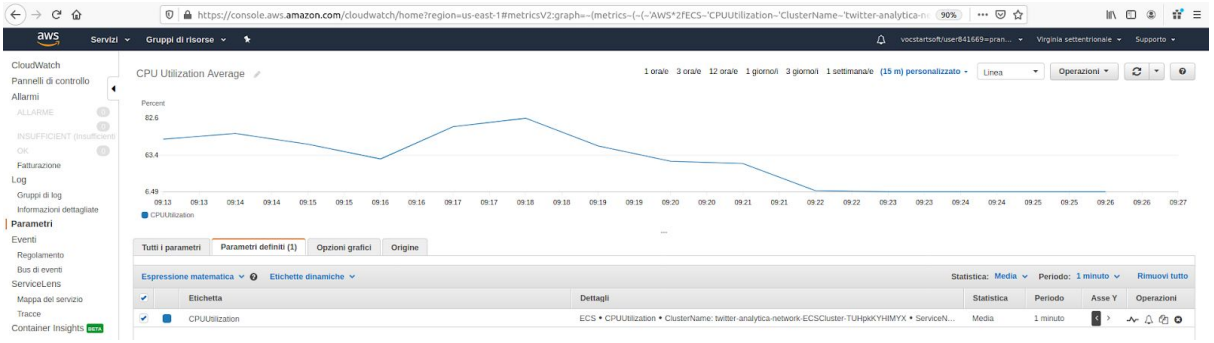


Figure 38: Cpu utilization of stress test with 30.000 requests from 5000 concurrent user

Looking at the graph of the CPU Utilization Average present in Amazon Cloud Watch, we notice that there is a time interval in which the threshold of the average CPU is greater than 80% (at about 9.18 am). This, thanks also to the step scaling policy defined above, triggers an alarm that leads to the creation of a new task. As we can also see from the image, our ECS cluster sees the addition of a new task, bringing the total number of running tasks to 4.
After the activation of this new task, the average CPU usage drops.

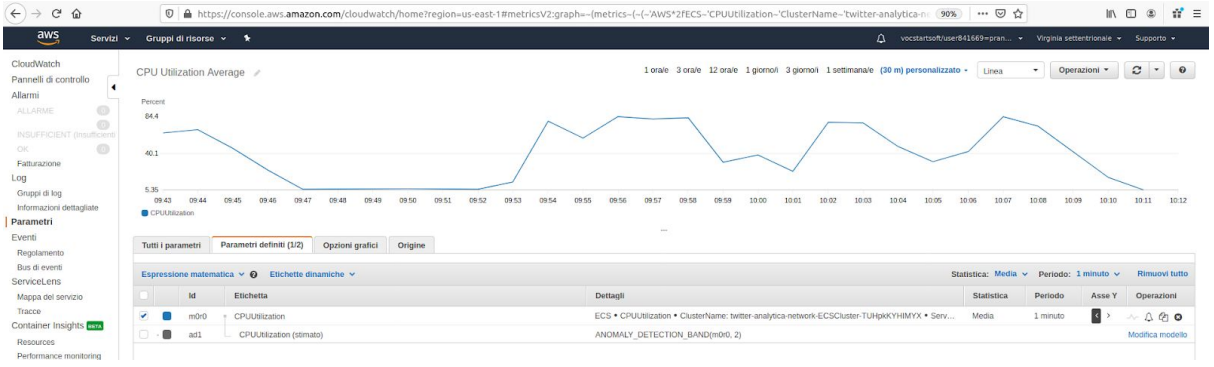
Figure 39: Running tasks after first scaling action

The last test we did to actually see how the cluster behaves was to launch 100,000 requests with 10,000 users simultaneously.

```
Completed 50000 requests
Completed 100000 requests
^C

Server Software:        TornadoServer/6.0.4
Server Hostname:        twitt-publi-n8949dbuoi19-2100947222.us-east-1.elb.amazonaws.com
Server Port:            80

Document Path:          /
Document Length:        5924 bytes

Concurrency Level:      10000
Time taken for tests:   90.928 seconds
Complete requests:      100911
Failed requests:        163
   (Connect: 0, Receive: 0, Length: 163, Exceptions: 0)
Total transferred:      631894012 bytes
HTML transferred:       600371686 bytes
Requests per second:    1109.79 [#/sec] (mean)
Time per request:       9010.732 [ms] (mean)
Time per request:       0.901 [ms] (mean, across all concurrent requests)
Transfer rate:          6786.50 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      108 1666 4814.8    645   65767
Processing:   119 3218 5573.2   1572   89770
Waiting:        0 2252 3991.3   1167   81749
Total:        274 4884 7608.3   2637   90566

Percentage of the requests served within a certain time (ms)
  50%   2637
  66%   4016
  75%   5238
  80%   6128
  90%   9960
  95%  15977
  98%  30184
  99%  40747
 100%  90566 (longest request)
```

Figure 40: Apache Bench output

Also in this case, the workload this time running on 4 tasks, leads to trigger the alarm and increase the number of active tasks to 5 as in the previous case



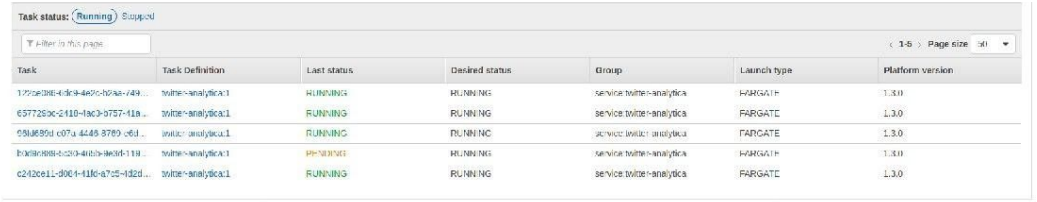
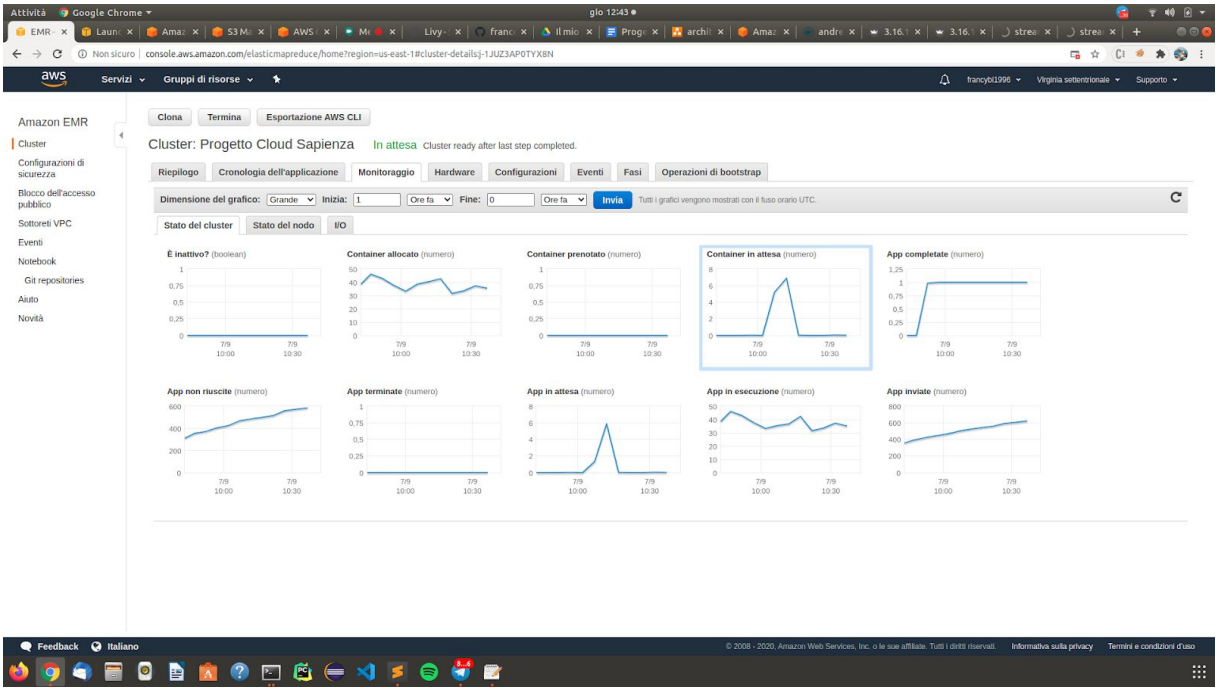


Figure 41: Running tasks after second scaling action

Having examined the proper functioning of the scaling policy, we decided not to carry out any further tests, as we considered this last test to be decisive.
However, our ECS cluster has been configured to increase the maximum number of tasks to 10 and this allows the use of Analytical Twitter to a large number of users even at the same time.

- Testing of the EMR cluster on which the calculation operations are performed by Apache Spark

As seen in the previous section, the workload on the Spark cluster was created using a script that would run pySpark code sent as payload through the Apache Livy REST API.
Thanks to EMR's default autoscaling policy, the cluster responded correctly to the increased workload. We also used m5.xlarge EC2 instances that are extremely high performance and this facilitated the cluster's computing power.



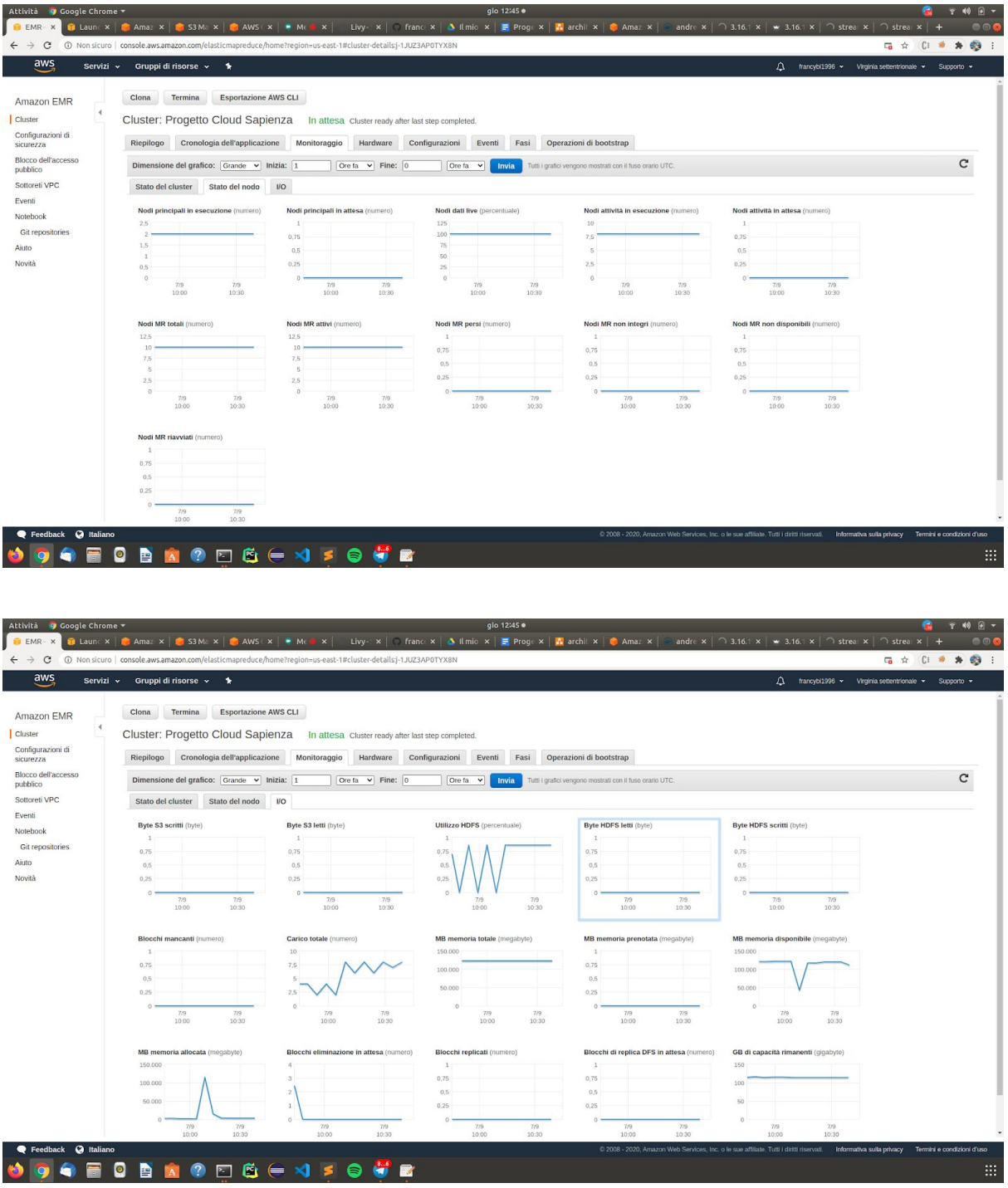


Figure 42: Monitoring of EMR Cluster

Conclusion

In conclusion we can say that the project has been very useful and stimulating for us to put into practice many of the topics covered in the lessons.
The topics covered in TwitterAnalitica allowed us to range from virtualization (docker, EC2, ECS, Fargate) to mechanisms (Cloud Watch), from the use of REST APIs (TwitterAPI and Apache Livy) to data storage (Amazon S3) and data analysis (Amazon EMR, Spark).
Despite an initial difficulty, especially for novice users, AWS has proved to be an excellent solution and probably the best cloud provider; the services provided have proved to be complete and excellent for the development and deployment of our application.

Future developments

As a future development, more information could be gathered from the Json returned by TwitterAPI to make a more detailed analysis and different studies (e.g. peak times of Tweets traffic, geographical areas where most tweets come from, etc.).
It would also be interesting to see how the application could react to the real use of users, without having to simulate the various requests through stress tests.