Data Analytics Application Deployed in AWS Cloud Computing Platform

Giovanni Pica - 1816394
Antonio Andrea Gargiulo - 1769185
Elios Buzo - 1669351

SAPIENZA
UNIVERSITÀ DI ROMA

Contents

1 The Problem

The project aim is to design, develop, deploy and test a data analytics cloud-based application that allow users to perform technical analysis on the valuation in time of different stock values and to integrate the data coming from other sources such as financial news and users' sentiment of these stocks on social networks.

The application is designed to give a powerful tool to users with a basic understanding of financial information, to support the choices they made with their investments or to help financial analyst to choose the right time to make an investment for their clients.

The final result is a responsive web-app, better suited for wide screen or horizontal mode on smartphones, accessible with an internet connection, from any browser, on any operating system. The application shows a series of pages with statistical plot of majors technical financial indicators and statistical plots of news and sentiment concerning those symbols. The data showed are updated in a continuous way, these data are processed and then stored in a high available and reliable storage solution provided from AWS. Various type of processing are made on the data to obtain relevant information to display to the end-user, and when not possible, a specific area on the application left the possibility to perform some user-made filtering, grouping and pivoting operation on the data, to obtain a better suited and user-tailored analysis.

2 Design of the Solution

To design the application, the solution proposed aim to use the most appropriate services offered by the **AWS** Cloud Computing platform. The design is based on a microservices architecture, that aim to decouple the integration of new function and to better and easily manage the continuous integration of these functions to the application. The workflow of the solution (see Figure 1) is based on two main parts, that can be differentiated by the philosophy of the **Stateless Server**.

2.1 Data Retrieving and Storing

The data used by the application are collected from two public APIs, one is **Alpaca API** (more information can be found here: https://alpaca.markets/docs/api-references/market-data-api/) which returns stock data such as the prices and the news of some related symbols for example AAPL for Apple or AMZN for Amazon; the other one is **Twitter API** (more information here https://developer.twitter.com/en/docs/twitter-api) which returns the tweets related to specific company.

The AWS cloud services used in this part are **AWS Lambda** which is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers, **AWS Step Functions** which is a serverless orchestration service that let you define your application's workflow as a series of event-driven steps and **Amazon EventBridge** which is a serverless event bus that makes it easier to build event-driven applications at scale using events generated from applications, integrated Software-as-a-Service (SaaS) applications and AWS services.

Those services cooperate to retrieve the data and to store them, for high availability and reliability purposes, in an **Amazon S3** bucket which is an object storage service offering industry-leading scalability, data availability, security, and performance.

1. **AWS Lambda** is used to get the data from the REST API endpoint supplied by Alpaca and Twitter, storing them in an AWS S3 bucket. With Lambda, we can have a microservice architecture where each function is decoupled from others, easy manageable to code and maintain and better optimize the need of resources because it could be triggered only when needed.

2. **AWS Step Functions** is used to call more lambda functions in pipeline, to have a single customizable Lambda function that can be used for different purposes with various configuration of the same function, in this way we can reuse the code and generalize the data retrieving process.

3. **Amazon EventBridge** is used to create scheduling events to trigger the different lambda functions to keep the data updated without the need for a human in the loop and in this way the updates can be scheduled with a better alignment with the needs of stakeholders or with a balance between costs and need in performance.

2.2 Data Visualization and Analytics

The second part refine the raw data from the S3 to perform relevant analysis and to have a better understanding of whats going on in these data. With the help of the majors technical analysis indicators, the integration with more and different sources like news and tweets and the web-based visualization platform, a user can easily visualize, process and understand the data and derive conclusions.

1. **Streamlit** turns data processing into shareable web application with low boilerplate code. It manages the creation of the web-application showing data analysis from the information retrieved and processed from the **S3** bucket.

2. **Docker** is a software platform that allows you to build, test, and deploy applications quickly. The container purpose is to deploy the web-application and to create a fault-tolerant and isolated environment between the application and the web server.

3. **AWS Elastic Beanstalk** is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS. This service expose the web server, which allows the users' interaction and communication with the final application.

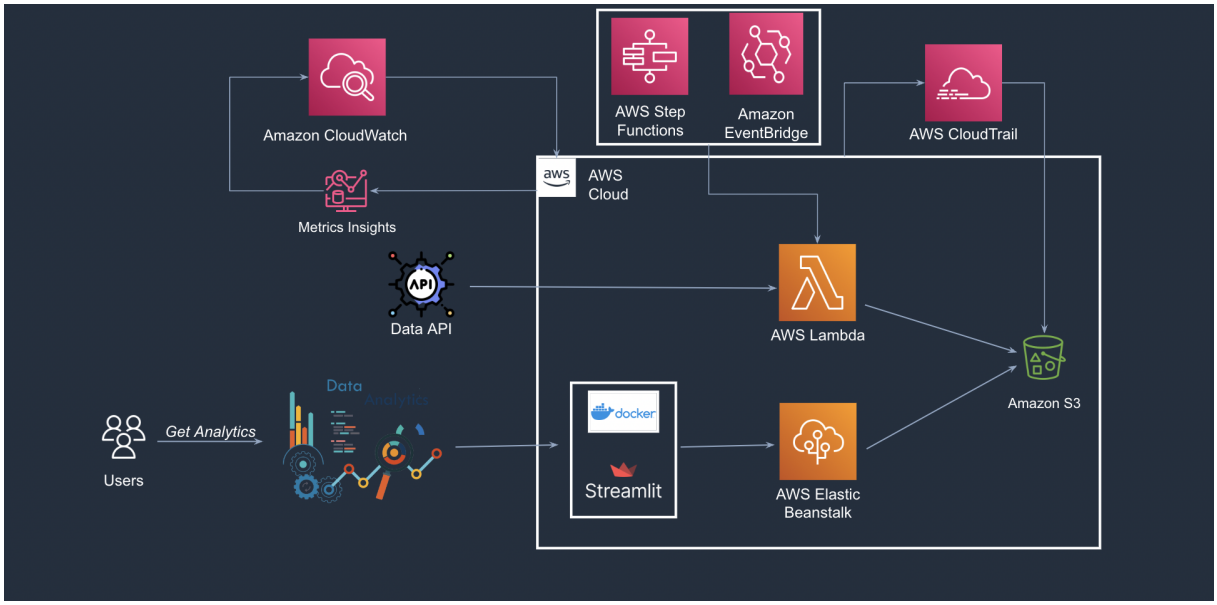

Figure 1: Workflow of the solution

3 Implementation

For the implementation phase we have chosen **Python** as the programming language. Because of its wide adoption and great community support as well as its ease of use we decided that it was the best solution. In order to make ad easy communication with AWS services we used the official AWS Python SDK, or Boto3.

3.1 Data Retrieving and Storing

The first step of the implementation is the development of the module that retrieve the data from the external source and store them internally. As we said in the previous section we two data source for our analysis, Alpaca and Twitter. The **Alpaca API** allows choosing the price data of which company you want to see by specifying the stock market symbols (for example AAPL for Apple) as well as the date from which we want to analyze the data, both as a parameter of the HTTP request. In this case we decided to analyze the data from Apple, Tesla, Meta, Netflix and Amazon in 2022. On the other side the **Twitter API** is more restrictive and returns a number of tweet taken from the last seven days.

In order to make the code reusable we decided to design a general **Lambda** function for each of the data source that make a generic request and find a way to call it multiple times, one for each company. The solution we found expect the use of **Step Function**. With this service we've been able to define a state machine where each step is a call to one of the two lambda functions we have implemented with the possibility of defining different input configuration, in this case the company we want to analyze.

Finally with the **Eventbridge** service we scheduled an event that starts the execution of this state machine. In order to reach a compromise between the incurred and the need of keeping the data up to date we decided to schedule it every 7 days.(Figure 2)

Lastly, the news API, provided by Alpaca, follows a different approach because we can retrieve all of the data that we want with a single request, but with the limit of only 50 units in response.

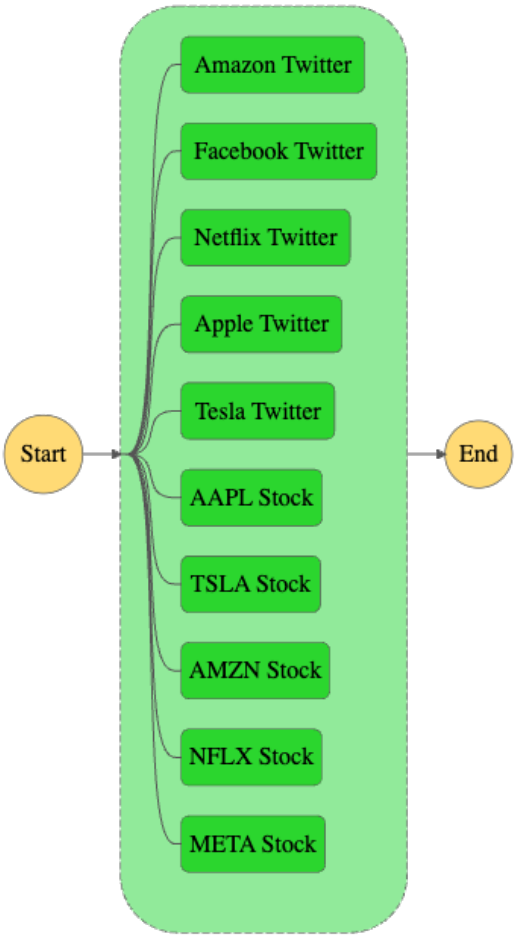

Figure 2: AWS Step Functions State Machine

The *boto3*, AWS SDK library for python, is used to store the data (in JSON format) on the S3 bucket. Moreover, Amazon EventBridge is used to create events in an interval of scheduling time in which the Stock Data is updated, for example in our implementation every week the Lambda functions are triggered and they store in the S3 the newly updated data. For the News, we decide to not use this approach because they are in a beta version of the Alpaca API and it returns only a maximum of 50 news.

3.2 Data Visualization and Analytics

As was mentioned in the previous chapter the library used for visualize the data is Streamlit, in particular the library used for plotting the data is **Plotly**. First of all we have the reading of the data with *boto3* and after that for convenience the data in JSON format is transformed into **Pandas** DataFrame. The pages of the web-app are:

- **Stock Price**: a plot that has in the x-axis the DateTime and in the y-axis the Close value of the stock. The user can choose which stock symbol he wants to see, in which period of time and also the days of the exponential moving average (by default 12 and 26 for short period but can be changed to 50 and 200 for longer ones). On the bottom of this page the user can see some relevant information such as the max and minimum price and the days where it happened.

- **Candlestick**: a plot showing the variation of opening, closing, high and low prices with a *candlestick* graph or an *ohlc* and also here the user can choose the symbol, period of time and the days for the exponential moving average. On the bottom part of the plot there are other 3 parts: the first is a *MACD* indicator with its signal line and a area chart showing the difference between the two; the second is an *RSI* indicator with his line of overbought and oversold; the third is a plot of *trades volumes*. At the bottom of the same page there is another plot showing a *K-D stochastic* indicators.

- **Tables**: a page that shows the table of the chosen symbol and also it can apply some filters to the values, for user-specific interrogation and analysis.

- **Comparison**: a page showing a comparison plot of all the symbols mentioned before.

- **News**: a page that shows 3 plots, one for the top authors, the most cited symbols and the most frequent words by using the **WordCloud** library that is a word cloud generator in Python.

- **Twitter**: a page that shows the trend of the total number of tweet and the sentiment analysis of a sample of 300 tweets about a specific argument, both in the time window of the last 7 days. This analysis has been made with the library **nltk** that is a platform for building Python programs to work with human language data.

3.3 Deployment

The Lambda functions are deployed with a simple python script and for the interaction with them we have used the AWS Dashboard Console. The Streamlit web-app is deployed on a Docker container with the usage of a DockerFile and after that, the application is pushed in a repository contained in the **DockerHub** which is a container image library. When we have to deploy the dockerized web application in the AWS Elastic Beanstalk environment we load in it a file called "Dockerrun.aws.json" which specifies the Image name (the repository path) and the port number (8501 for streamlit) to expose the service.

4 Test & Validation

To give the possibility to the application to manage more and more users, we have chosen to use an **Auto Scaling Group**, which contrary to what the name implies does not provide automatic scaling capabilities. We choose to scale by the max CPU utilization of all the instances, we set a scaling policy monitoring the CPU utilization of all running instances because from a testing phase of the application we have seen a drop in performances if this metric go over a 50% **threshold**, so we add another instance at a upper bound of 30% because we also take into the account the time to deploy the other instance. On the contrary if there is a lower CPU utilization (an utilization of 18% for all the running instances) for a range of 1 minute, the scale-down drop an instance, because it means that the number of users and consequently the number of requests are diminishing. The request to the application are intercepted by an **application load balancer** (ALB) that balance the amount of work routed to a single instance and thanks to the **CloudFront** AWS service, that is a Content Delivery Network that caches responses to HTTP requests, if there are common contents inside the request, it is directly managed by the low-latency CDN proxy servers and the introduction of caching decreases the load on all underlying parts of the architecture.

For the testing of our application we decide to use **Locust** (more details here http://locust.io/) that is a load testing tool in Python and consists in "swarm" the system in a fixed number of simultaneous users. The monitoring part of the application has been used **Amazon CloudWatch** which is a monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), IT managers, and product owners and the metrics chosen are:

- **TargetResponseTime**: indicates the time elapsed, in seconds, after the request leaves the load balancer until a response from the target is received.
- **SumRequests**: the CPU utilizations processed over IPv4 and IPv6. This metric is only incremented for requests where the load balancer node was able to choose a target. Requests that are rejected before a target is chosen are not reflected in this metric.
- **Environment Health**: is represented by one of seven statuses. In the CloudWatch console, these statuses map to the following values:
  - 0 OK
  - 1 Info
  - 5 Unknown
  - 10 No data
  - 15 Warning
  - 20 Degraded
  - 25 Severe
- **CPU Utilization**: The percentage of allocated EC2 compute units that are currently in use on the instance.
- **Network In**: the number of bytes received by the instance on all network interfaces.
- **Network Out**: the number of bytes sent out by the instance on all network interfaces.

While with regard to the metrics of Locust, it has the Total Requests per Second, Response times (ms) and Number of users. To help finding bugs to the application and also to add visibility and auditability we used **AWS CloudTrail** which monitors and records account activity across our AWS infrastructure, giving control over storage, analysis, and remediation actions, recording this activity in the form of audit logs and helping identifying possible malicious behavior and surfacing parts of our infrastructure that might not be configured properly.

5 Experimental Results

In this section we document the results obtained from the phase of testing focusing on the availability to the end-user.

As we can see from Figure 3 we have performed various load testing with a variable number of simulated users, HTTP requests and spawn rate of users.

The first test is executed with 1000 interactive users that performs requests on various page of the application in a variable timeframe of 1 to 5 seconds to simulate a normal behaviour of a user utilizing the application. The application in this first test seems to perform very well, there are

no fault on the requests performed by this number of users and the utilization of the instance is slightly below to scale to multiple instances. After, we performed a second experiment to test the scalability performances of the application increasing the number of concurrent users utilizing the application to 5000, spawning 100 users per second, in this test the application performances seems to suffer a lot from the new huge number of request arriving in a small amount of time, as we see from the red line in the first plot of Figure 3 there is a 5% of failures in respond to the request from users for an interval of 2/3 minute, but after a second and a third instance are correctly loaded, deployed and correctly balanced, the failure reach again zero.

Then, we execute another stress test to understand the capability of our application in up-scaling and add another 5000 users to reach 10000 concurrent active users performing request to the application and we doesn't see any degradation in performances, telling us that the application can be scaled to serve also more and more users, with other instances. After this, we decide to drop some users to see if the scale-down works correctly and the threshold is appropriate and holds. We drop the 10000 users to 5000 and the instances are reduced from 4 to 3, after that to 2500 users, where the scaling policy don't drop any instances, because the CPU utilization doesn't go down the lower threshold. Finally, to 1000 users, where the instances are reduced to 2 and then 1 and as we can see from Figure 3 the application is capable of managing the load without any failure.

Figure 3: Locust request per second, Response time and Number of Users.

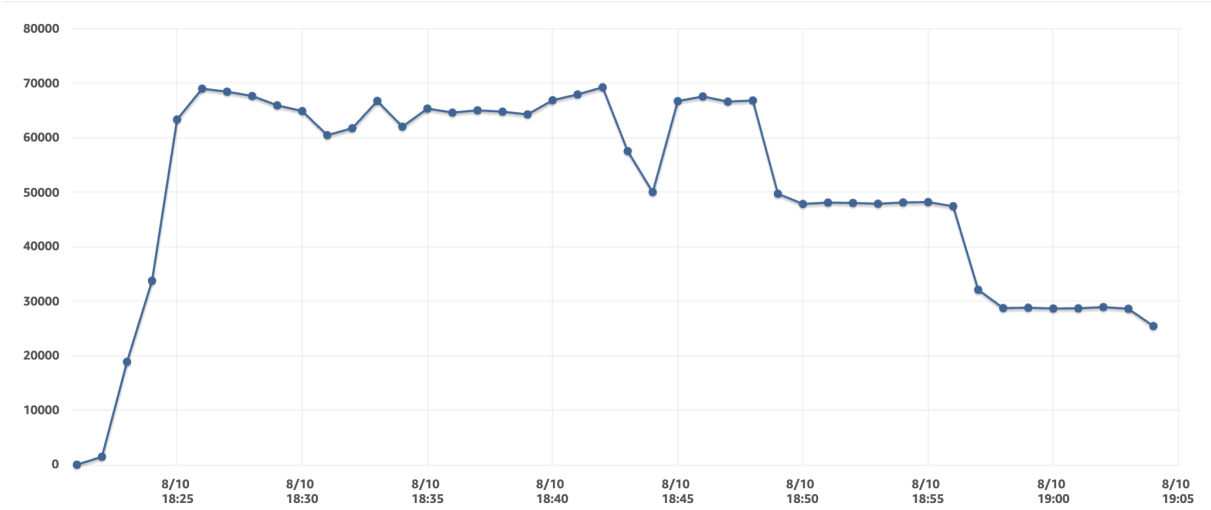

Figure 4: Users' requests to the application during the load testing phase.

As we can see from Figure 5, when the upper threshold alarm (30%) is activated, the scaling policy starts to scale up with a new EC2 instance. Moreover, in the same figure we can see that when the lower threshold alarm (18%) is activated, the EC2 instances are scaled down, so the scaling policy correctly remove an instance and the application load is balanced between the remaining instances. In the Figure 6 we can see the average CPU utilization in the entire test between all the active EC2 instances.

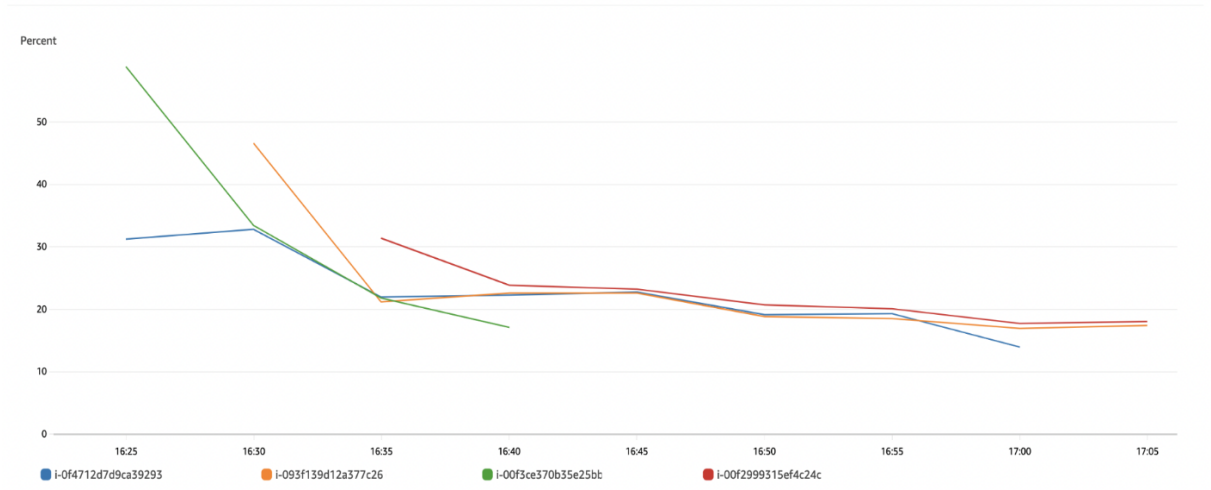

Figure 5: CPU Utilization of the single instances in AWS Elastic Beanstalk to understand the behavior of the Application Load Balancer.

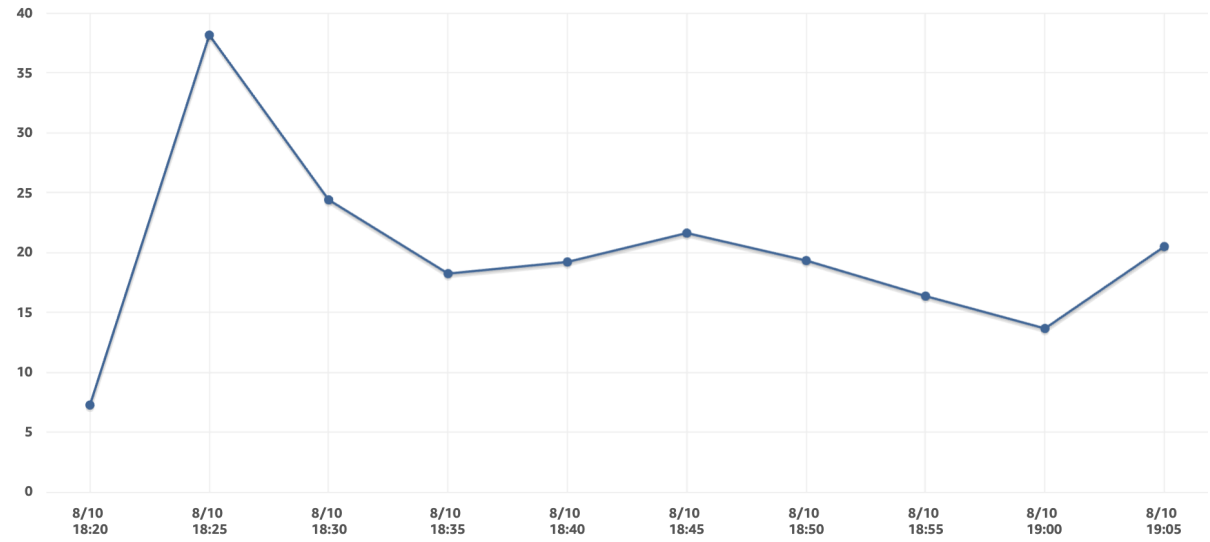

Figure 6: Average CPU utilization of the EC2 instances in AWS Elastic Beanstalk.

5.1 Application Screens

As we can see from Figure 7 that is the main page of our application, the user has a page showing the technical analysis of a chosen stock price with some sliders to change the EMA analysis and a sidebar to switch to others analysis pages.

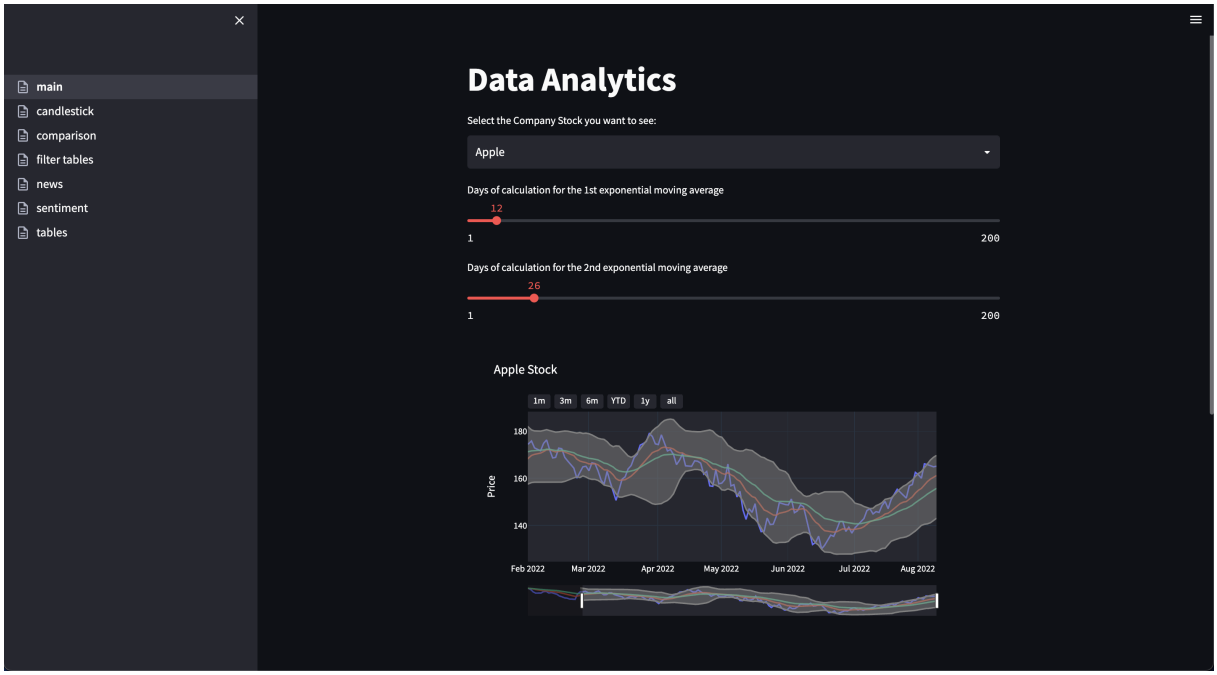

Figure 7: Main page of the web application.

In Figure 8 we can see the candlestick graph showing the most used technical analysis indicators such as: Bollinger bands, EMA, MACD, RSI, Volume of trades, that supports the users on the choice made for their stock analysis purposes.

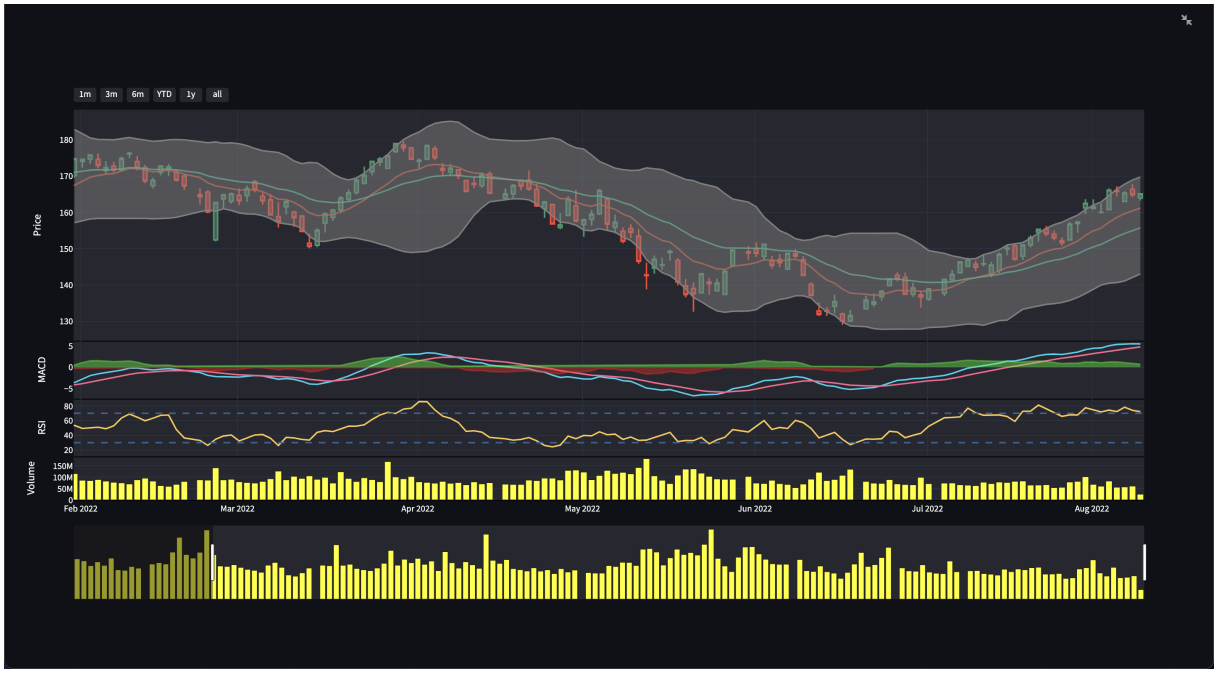

Figure 8: Candlestick graph.

The Figure 9 refers to a page of the application which primary role is to help the user to have a comparison between the chosen symbols that he has selected and a matrix of correlation of those symbols.

Figure 9: Stock comparison page.

The Figures 10 & 11 are in the news page of the application that has a wordcloud graph and two barplots showing most cited author and comparison of cited symbols.

Figure 10: Wordcloud for the news.

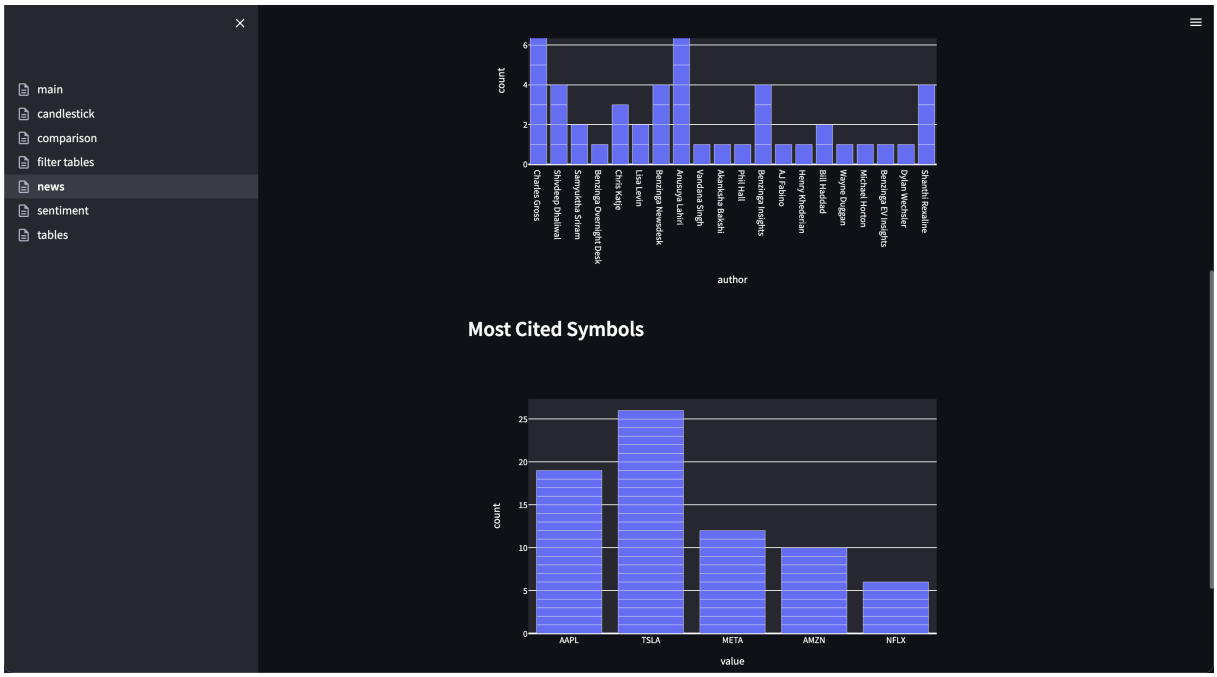

Figure 11: News page.

Figure 12 refers to the sentiment analysis page of the selected symbol's tweets, showing the outlook of the symbol from a series of tweets.

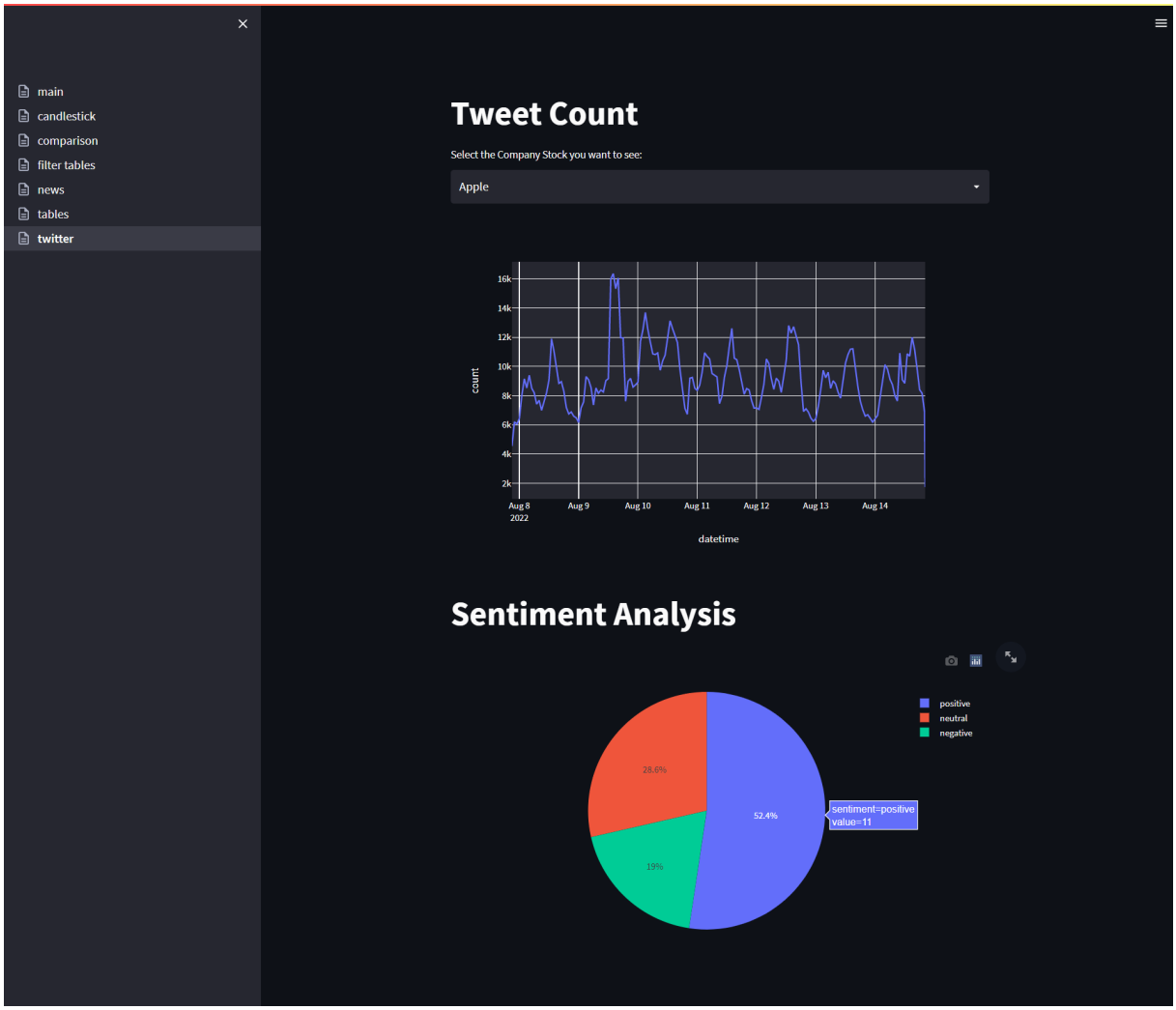

Figure 12: Sentiment Analysis page.

6 Pricing

The Table 1 shows a final bill for this data analytics application serving an estimate of 50000 users, all the prices are in USD dollars and the total estimate for 1 year is 1101.20 $. All the services used not listed in the table falls in the free-plan billing tier of AWS.

Service	Upfront Cost	Monthly Billing	Total of first 12 month
Amazon EC2	44.68 $	40.53 $	531.02 $
Application Load Balancer	0 $	39.79 $	477.48 $
Amazon CloudWatch	0 $	3.73 $	44.70 $
AWS CloudTrail	0 $	4.00 $	48.00 $

Table 1: Pricing table with a scheduling of 1 to 8 EC2 t4g.micro instances for 3 years billing.