SAPIENZA
UNIVERSITÀ DI ROMA

Master's Degree in Computer Science, Data Science, Cyber-Security

Title

Cloud Computing Project Report Group 3

Professors:
Emiliano Casalicchio

Students:
Andrea Ciccotti - ciccotti.1956198@studenti.uniroma1.it,
Andrea Bernini - bernini.2021867@studenti.uniroma1.it,
Donato Francesco Pio Stanco - stanco.2027523@studenti.uniroma1.it

Academic Year 2021/2022

1 Project goals

The project aims to provide a machine learning model and create a web application that communicates with it, allowing users to input data and receive predictions about football matches. The project code can be consulted at the following GitHub repository: cloud-computing-project.

1.1 The scenario

We imagined the following real scenario: a user enters a series of data about a football match, such as the match date, the name of the home team, and the away team. Once these parameters have been entered, they will be processed, and a prediction of the outcome of the match will be provided in the output, i.e. victory of the home team, draw, or away team.

2 Proposed Solution

2.1 Integration of the system

The tools we used for the development are as follows:

- **Flask**, for the creation of the Web app through the use of Python, HTML, and CSS code.
- **Docker**, for the creation of a containerized environment in which to run our application.
- **Kubernetes**, for the management of containerized applications, or to completely manage their life cycle. The reason why we chose to use Kubernetes is that it allows you to balance the load, automatically distributing it among the containers, and resizing, that is, it allows you to automatically add or remove containers when it changes.
- **Google Cloud Platform** (GCP), provides Google Kubernetes Engine, which is an implementation of the open source Kubernetes framework, as well as various cloud computing services. One of the reasons we chose to use GCP is that it allowed a free trial worth $ 300 for 90 days.

2.2 Simulation of the system

To monitor and stress our application, we decided to use Locust, which is an open source load testing tool, and the Google Cloud Console by analyzing a series of data such as user load, CPU usage time, memory usage, and other values.

3 Implementation

3.1 Build the Machine Learning Model

The first step in the development of a model was to collect the data on the Football-Data site, which contains different data-sets, one for each football season and league. For our model we have decided to use the data-sets of the last ten seasons of Serie A, combining them into a single large data-set having as features the date of the match, the name of the Home Team and Away Team, and many other characteristics concerning the match statistics, such as the number of goals, corners, or red and yellow cards.

However, using these features to train the model would be like cheating as we will get a very high Accuracy, as we will encounter an overfitting problem. For this reason, the next phase involves the extrapolation of further features, using the existing ones, such as the number of points of each team before the game, their position in the standings or the number of days elapsed since the last game, or the winner of the previous meeting between the two teams, and many others. At the end of this phase, we will delete all the features that cause overfitting, thus obtaining a data-set of 40 columns and about 4200 rows.

The next phase involves the development of a pipeline for the preprocessing of the features and the training of the model. During the encoding phase of the categorical features, we decided to store the string "associations": integer in a JSON in such a way as to translate the data entered by the user in the same way. In the training phase, we decided to locally test the performance of different models to choose the best one, and from this analysis, the Logistic Regression was the winner, which has a good Accuracy (about 49%) and a good prediction time. After the hyperparameter tuning phase of the Logistic Regression, we saved the model through the Python pickle library. All the work done on the model can be consulted on the following notebook by Google Colab.

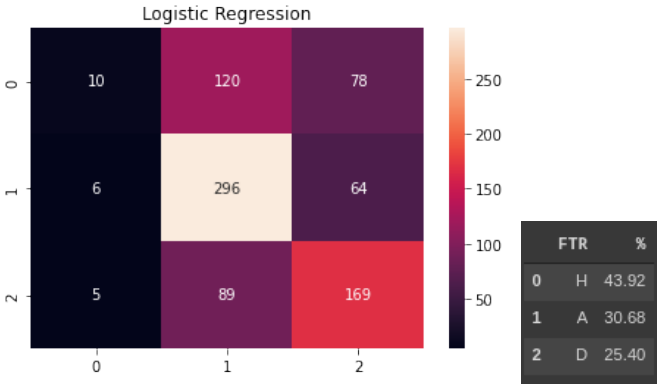


Figure 1: Performance of the ML Model

As can be seen from the Confusion Matrix, the Draw result (0) is very difficult to predict, as there are few examples of this result (about 25%). Instead, it is much

easier to predict the victory of the home team (1), since we have a large number of examples, that is, about 45%.

3.2 Develop of the Web-App

Once we have developed our machine learning model, we will take care of building a web application that can connect to our trained pipeline to generate predictions on new data points in real-time. Our web app can be divided into two parts:

- Front-end (designed using HTML and CSS), in which we have developed the graphics of our application to allow the user to enter the data on which he wants to obtain a prediction.
- Back-end developed using Flask, a Python framework that allows you to render HTML code (using the `render_template` function) and manage the GET and POST methods through annotations, which refer to specific functions. The main method of our application is `/predict` which allows us to manage POST calls obtaining a prediction of the result.

```
# Method for Locust Testing
@app.route('/predict_test',  methods=['POST'])
def predict_test():
    request_data = request.json

    match_date_str = request_data['match_date']
    home_team = request_data['home_team']
    away_team = request_data['away_team']
    match_date = datetime.strptime(match_date_str,
                                   '%Y-%m-%d').date()

    # Calculate the season
    season = get_season(match_date)
    final = [season, match_date, home_team, away_team]
    data_unseen = pd.DataFrame([final], columns=config.COLS_URL)
    prediction = get_prediction(data_unseen)

    return '/predict_test - season: {}, match_date: {},
    home_team: {}, away_team: {}, prediction: {}\n'
    .format(season, match_date, home_team, away_team, prediction)


```

Listing 1: HTTP Method with Flask

Another important function is `data-set_manipulation` which allows to obtain all the features necessary for the prediction starting exclusively from the data entered by the user.

The final result obtained is the following:

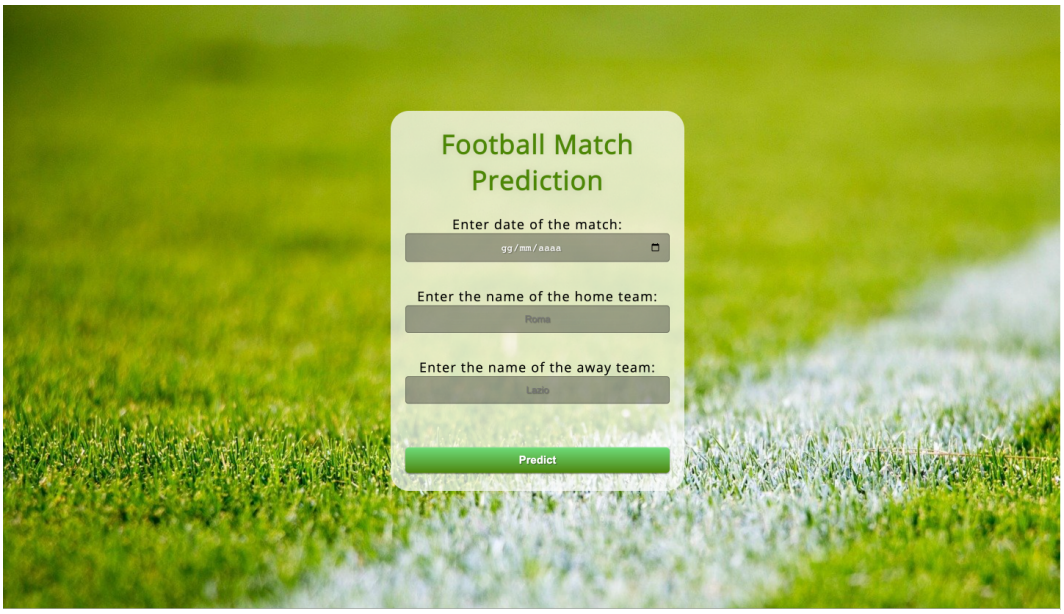


Figure 2: Main Page of the Web App

3.3 Deploy Machine Learning Model on GKE

The final step involves creating a Docker image, uploading it to Google Container Registry, and then implementing the pretrained machine learning pipeline and the Flask app on **Google Kubernetes Engine** (GKE).

The first step involves creating a project within **Google Cloud Console** (GCP) and cloning it into our repository containing the web application code developed in the previous step, through the Google Cloud Shell or the command line tool of GCP.

Then we proceed to the creation of the `football-bet-cluster` **Cluster**, which will contain our Web application, enabling the Monitoring of resources, the Autoscaling by setting the minimum number of replicas to 3 and the maximum number to 7, and enabling the HorizontalPodAutoscaling necessary to allow the cluster to resize pods and use Google Cloud Load Balance.

Once the application Cluster has been created, we proceed by building the **Docker Image** of our application, through the Dockerfile contained in our repository, and then upload the docker image in the **Google Container Registry**. After that, we proceed with the deployment of the image in the Cluster in order to distribute and manage the application; moreover, by default, the containers we run on GKE are not accessible from the Internet because they do not have external IP addresses; for this reason, we must expose our application (we chose port 80).

Therefore, we have distributed our application, which can be consulted at the following address: `http://34.154.93.188:80`.

Finally, we will let GKE know that we want our pods to remain at around 80% load, so if the load exceeds this threshold, it can create more pods, up to a maximum of 10, and if the load is less than 80%, then it can delete some pods.

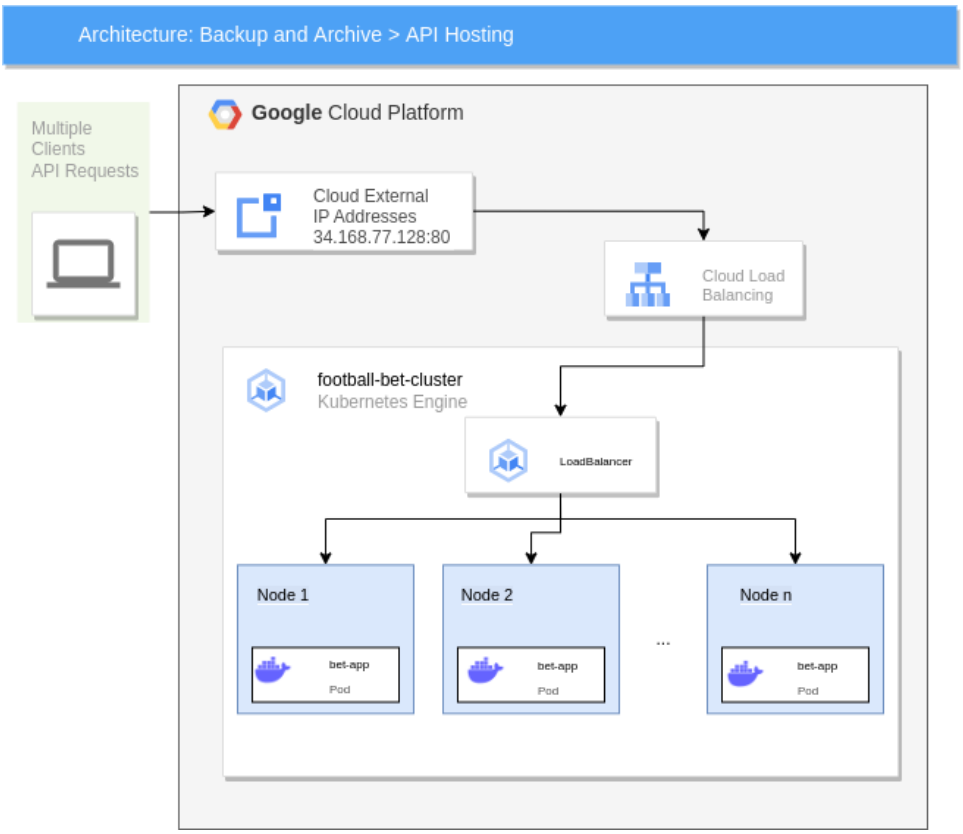


Figure 3: Web App Architecture

3.4 Simulation with Locust

After creating the cluster and once the application is running, we need to perform load tests; to do this, we will use Locust.io.

Locust.io is an open-source tool that allows you to perform performance tests against applications. One of the benefits of Locust is that it gives you the ability to spread your performance tests across multiple machines so that you can generate an even greater load on your application.

```
class MetricsTaskSet(TaskSet):
    _match_date = None
    _home_team = None
    _away_team = None
    wait_time = between(1, 5)

    def on_start(self):
        # Create a dataframe from the HTML form

```

```
        # Get tomorrow's date
        self._match_date = date.today() + timedelta(days=1)
        self._match_date = self._match_date.strftime("%Y-%m-%d")
        all_season = pd.read_csv(DATASET, low_memory=False)
        # Check Teams
        r = random.randint(0, all_season.HomeTeam.nunique() - 1)
        self._home_team = all_season.HomeTeam.unique()[r]
        all_at = all_season.AwayTeam.unique()
        all_at = all_at[all_at != self._home_team]
        r = random.randint(0, len(all_at) - 1)
        self._away_team = all_at[r]

    @task
    def predict_test(self):
        myheaders = {'Content-Type': 'application/json',
                     'Accept': 'application/json'}
        self.client.post( '/predict_test',
            json={ "match_date": self._match_date,
                   "home_team": self._home_team,
                   "away_team": self._away_team
                }, headers=myheaders)

class MetricsLocust(FastHttpUser):
    tasks = {MetricsTaskSet}
```

Listing 2: HTTP Method with Locust

A `FastHttpUser` class represents a user, and Locust will generate an instance of the `FastHttpUser` class for each simulated user. Within the `MetricsTaskSet` class we define all the tasks that a user must undertake and a `wait_time` that will make the simulated users wait between 1 and 5 seconds for the execution of each activity. The `on_start` method that is called for each simulated user at startup, inside which random data is generated with which to POST thanks to the `predict_test` method that calls the endpoint `/predict_test`.

This architecture involves two main components:

- The Locust Docker container image, which contains the Locust software, is the Python file described above.
- The container orchestration and management mechanism.

To deploy load testing activities, we will deploy a load testing master and load testing worker group, with which a significant amount of traffic can be created for testing purposes. The architecture is defined by the following diagram, in which the master Pod serves the web interface used to operate and monitor the load tests. Work pods generate REST request traffic for the application being tested and send metrics to the master.

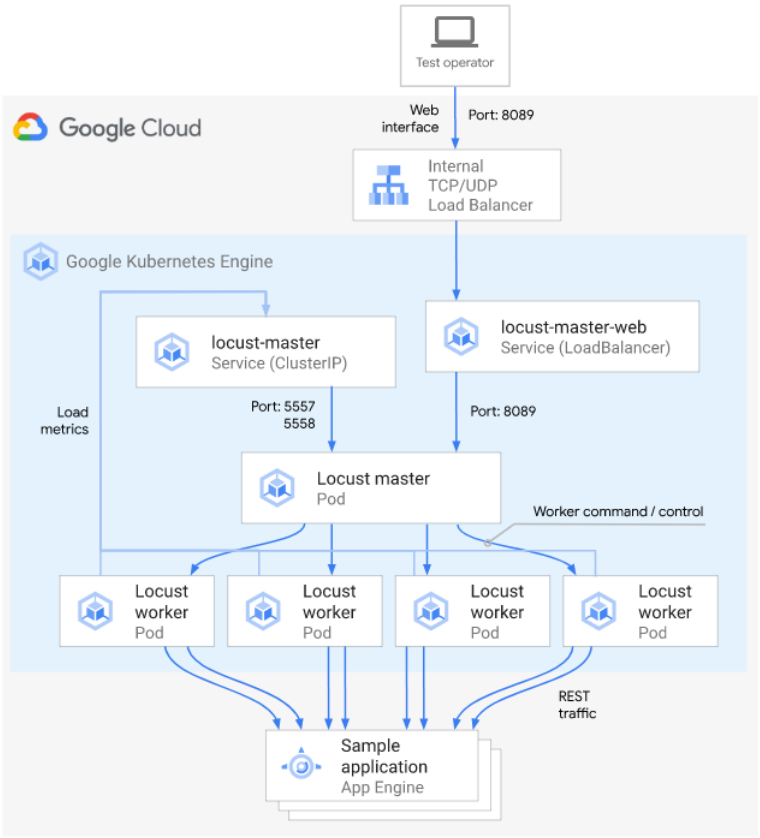


Figure 4: Locust Architecture from Distributed load testing using gke

So, first, we will create a separate cluster that will be responsible for managing the load test to ensure that the load test does not run out of resources, using a Master and Worker configuration.

We will now create a file called `loadtest-deployment.yaml` which contains the three elements needed for our load test, allowing you to deploy the Master distribution, the Worker distribution and a service.

Then we can access the Locust interface for testing at the following address: `http://34.154.197.48:8089`

4 Test

In this section, we will analyze the various load tests that have been carried out on our application by examining a series of parameters such as CPU and memory usage. We have decided to apply **horizontal scaling** or even called **scaling out** of our resources, adding more "machines" to our resource pool. The goal of the load/performance test is therefore not to verify the correctness of the code or data, as they would require integration or acceptance tests. Furthermore, the responsiveness of the different page elements was not tested in the load/performance test. The path we tested for our application is `/predict_test` (managed back-end with Flask) by developing the Locust Python code as described in the previous section.

As a case study, we have analyzed a load of `100 users` and these are entered into the system with a spawn rate of `5 users/second`.

4.1 CPU Usage Time Metric

The parameter we analyzed in the tests was CPU utilization. We did a series of tests to understand the optimal number of pods to obtain a lower response time during requests and also not have a large number of failures. Initially, we found that each pod used an amount of CPU equal to `250m`, where m stands for millicores, also in our cluster we have 4 nodes for a total of `3760m`. For this reason, in the YAML file that the Google Cloud Console creates at runtime, we have specified a limit and a request for the CPU in the resources parameter. The limit is calculated using the following formula:

```
total / number of pods
```

and we performed tests with 5, 7, 10 and 12 pods, each with a duration of about 20 minutes.

4.1.1 Test with 5 pods

The first test performed was with 5 pods, in order to define the limit in the YAML file we applied the formula defined above, so in this case, it was set to 750m. The results below are those provided by Locust and the Google Cloud Console, where we used **Cloud Monitoring tool**.

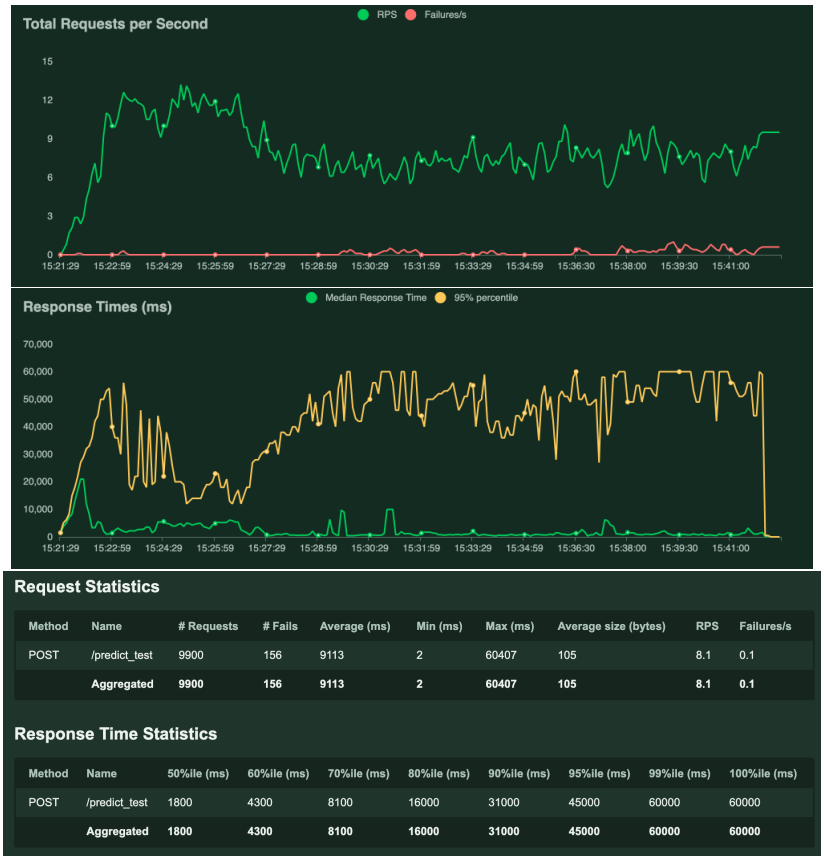


Figure 5: Locust - 5 pods

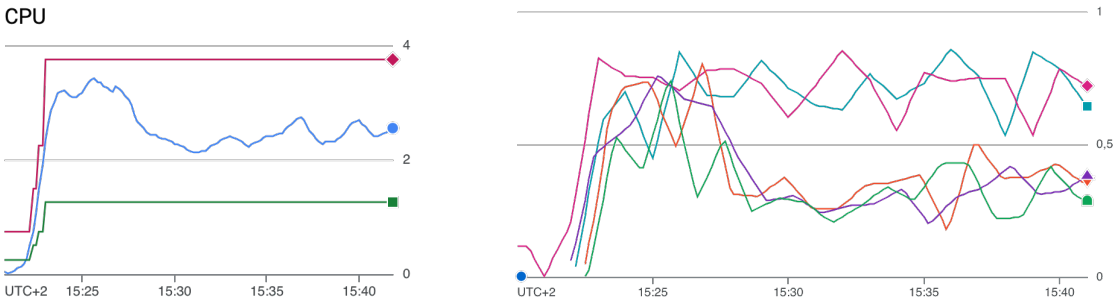


Figure 6: Metrics Explorer - CPU and Pods usage time with 5 pods

4.1.2 Test with 7 pods

The second test performed was with 7 pods, in order to define the limit in the YAML file we applied the formula defined above, so in this case, it was set to 537m.

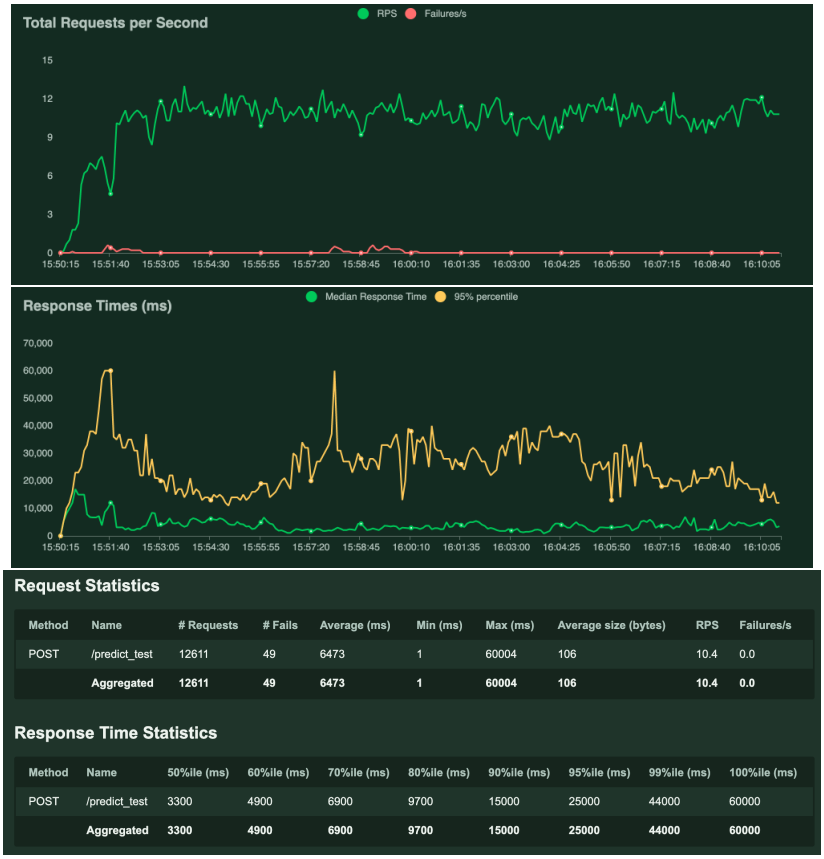


Figure 7: Locust - 7 pods

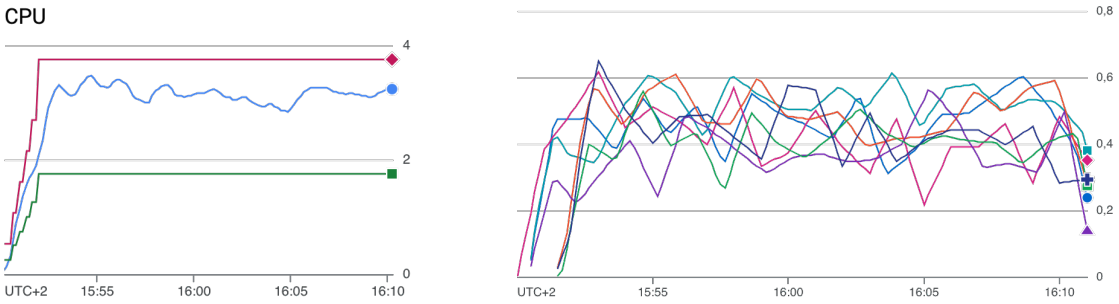


Figure 8: Metrics Explorer - CPU and Pods usage time with 7 pods

4.1.3 Test with 10 pods

The third test performed was with 10 pods; in order to define the limit in the YAML file, we applied the formula defined above, so in this case it was set at 376 m.

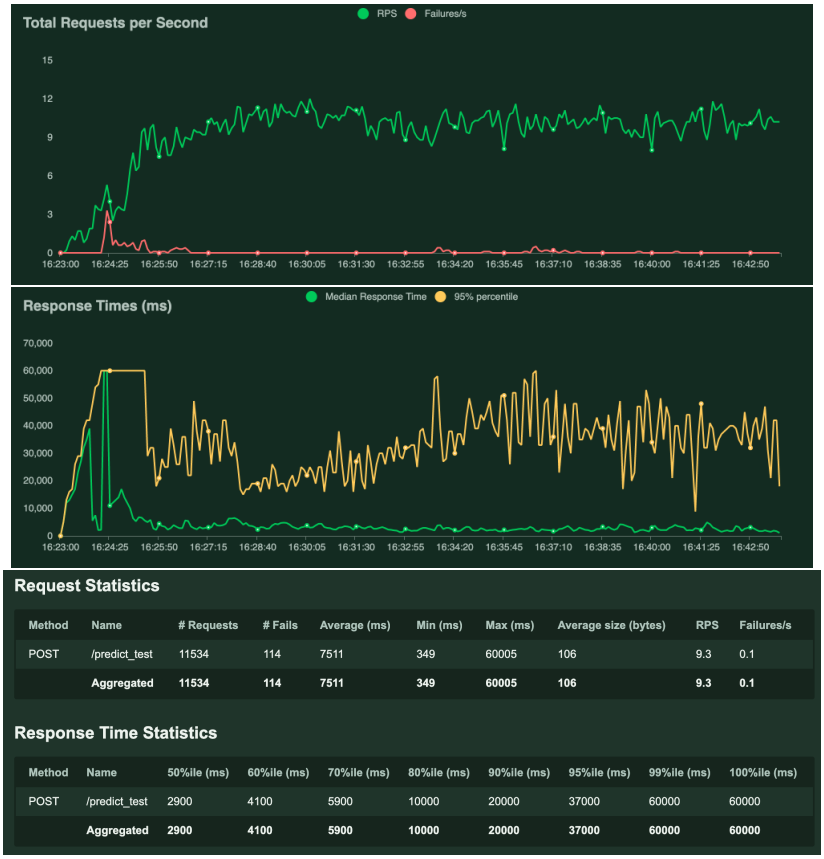


Figure 9: Locust - 10 pods

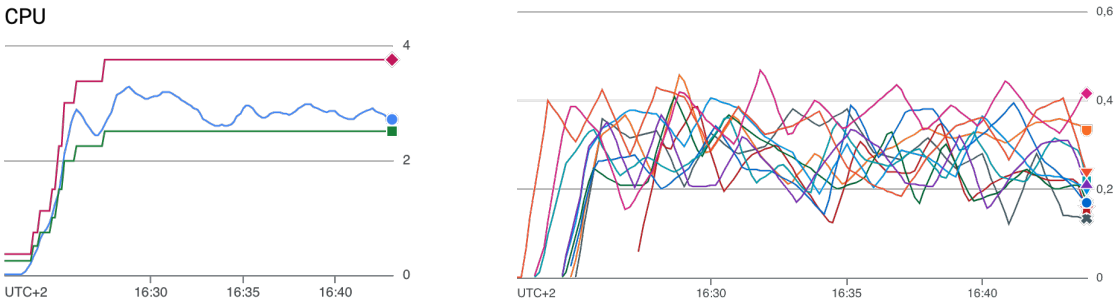
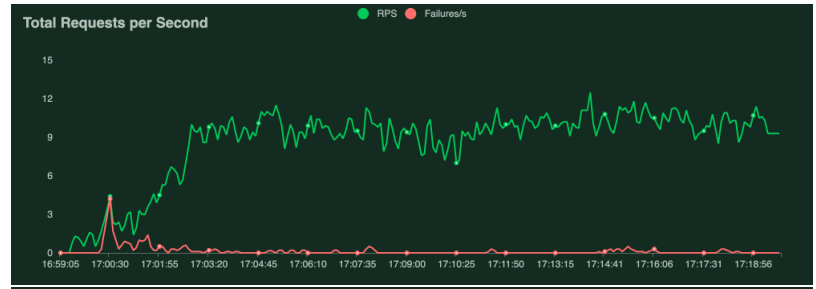
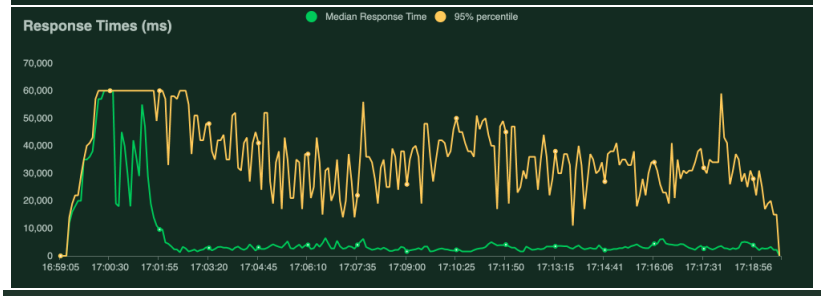


Figure 10: Metrics Explorer - CPU and Pods usage time with 10 pods

4.1.4 Test with 12 pods

The fourth test performed was with 12 pods, in order to define the limit in the YAML file we applied the formula defined above, so in this case it was set to 313m.





Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/predict_test	10500	181	8408	441	60077	105	8.6	0.1
	Aggregated	**10500**	**181**	**8408**	**441**	**60077**	**105**	**8.6**	**0.1**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/predict_test	3100	4500	6500	11000	25000	41000	60000	60000
	Aggregated	**3100**	**4500**	**6500**	**11000**	**25000**	**41000**	**60000**	**60000**

Figure 11: Locust - 12 pods

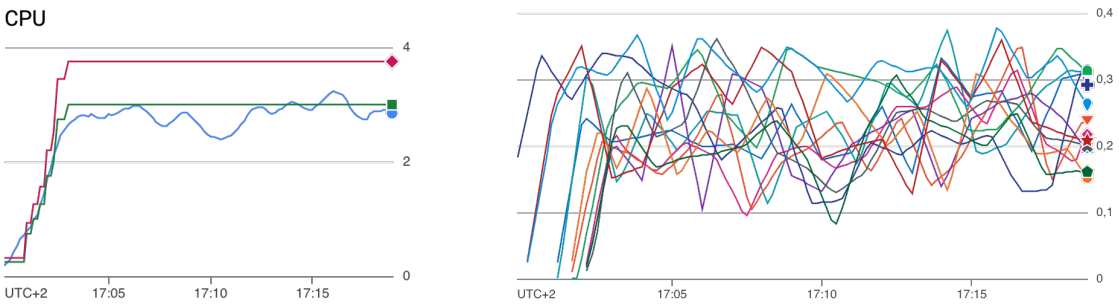


Figure 12: Metrics Explorer - CPU and Pods usage time with 12 pods

4.2 Considerations

Locust charts can be analyzed based on response and volume metrics. As for the Response Metrics we have:

- The **Average Response Time**, which measures the average amount of time that passes between a client's initial request and a server's response (including the delivery of all HTML, CSS, JavaScript, and image assets). It is the most accurate standard measurement of the actual user experience.
- The **Peak Response Time**, which measures the round trip of a request / response cycle (RTT), but focuses on the longest cycle rather than the average. They are useful for identifying problematic anomalies.
- **Error rates**, which measure the percentage of problematic requests out of total requests. It is not uncommon to have some errors with a high load; however, you need to minimize them to obtain a good user experience.

Instead for the Volume Metrics:

- **Concurrent users**, which measures how many virtual users are active at any given time.
- **Requests per second**, which measures the number of requests sent to the server every second (including all HTML, CSS, JavaScript, and images).
- **Throughput**, which measures the amount of bandwidth, in kilobytes per second, consumed during the test. Low throughput may suggest the need to compress resources.

In the Total Required chart, the green line shows successful requests and the red line shows unsuccessful requests. Instead, in the response time graph, the green line shows the median response time, and the yellow line shows the 90th percentile. As you can see from the latter, we have the best results using 7 pods; in fact, we have an Average Response Time of about 6400ms, that is, just over 6 seconds, and an Error Rate of 2.57%, instead, testing the application with a low number of users and therefore not being under stress, we have an Average Response Time of about 300ms which is very close to the 200ms recommended by Google in the Benchmark of the average desktop Time to First Byte (TTFB) speed (in which it was measured that the average is about 1.3 seconds for websites).

5 Estimated Bill

To analyze the costs of our project, we used the **Google Cloud Price Calculator** service, a platform that allows you to create a quote for Google Cloud services. As evidenced by the quote below, the cost of our project with the listed services is **$225.31** per month for a total of **$2703.72** for one year.

Analyzing the costs of the various services offered, we realized that prices vary according to the region in which the various resources are allocated. For example, in the europe-west8 region which corresponds to Milan in Italy, costs are much higher than in the us-west1 region which corresponds to Oregon in the United States. For this reason, we decided to choose us-west1 as the area to have considerably reduced costs, about **50% less**, since with the europe-west8 region the total amounted to about $444 per month and consequently of $5328 for one year.

Searching the Web, we saw that it was appropriate to use a second cluster, separate from the application one, to carry out the load tests, so that the latter does not exhaust the resources available to the first, but increases costs. However, the cost of the Locust cluster does not have to be charged all year, as load tests may only be performed in the initial application implementation period. In addition, a single cluster can still be used to perform all operations. Another aspect that raises costs, related to the number of clusters, is related to the startup disk of each node of the cluster, whose capacity of 100GB is assigned by default to the creation of the cluster, which turns out to be excessive from our tests, in when we go to occupy a maximum of about 30GB. So a solution to solve this problem is to reduce the disk size by half when the cluster is created or after its creation by going to the Disks page of the Google Cloud Console.

Your Estimated Bill *

Estimated Monthly Cost: USD 255.31

CloudBuild	Cloud Build	1	USD 0.00
	Inbound data processed by load balancer	3.01361083984375e-06 GiB	USD 0.00
	Outbound data processed by load balancer	8.058547973632812e-06 GiB	USD 0.00
4 x FootballBet cluster	e2-medium	2920 total hours per month	USD 61.64
4 x boot disk	Persistent Disk - GKE Standard	100 GiB	USD 40.00
3 x LoadTesting cluster	e2-medium	2190 total hours per month	USD 46.23
3 x boot disk	Persistent Disk - GKE Standard	100 GiB	USD 30.00
GKE Clusters Fee	Zonal Clusters	1460 hours	USD 71.60
	Inbound data processed by load balancer	3.01361083984375e-06 GiB	USD 0.00
	Outbound data processed by load balancer	8.058547973632812e-06 GiB	USD 0.00
Used on standard VM instances IP addresses	Assigned and used in standard VM IPs	1460	USD 5.84
Total Estimated Monthly Cost			**USD 255.31**

Figure 13: Cost per month of the application