SAPIENZA
UNIVERSITÀ DI ROMA

Cloud Computing a.y. 22-23

Project Report - VoiceFork

Alessio Lucciola - lucciola.1823638@studenti.uniroma1.it
Danilo Corsi - corsi.1742375@studenti.uniroma1.it
Domiziano Scarcelli - scarcelli.1872664@studenti.uniroma1.it

June 28, 2023

Contents

SAPIENZA
UNIVERSITÀ DI ROMA

1 Introduction

The system we designed and implemented is called **'Voicefork'**, a **mobile application** that allows users to manage their reservations at the restaurants. Users can register and login to access the functionality of the service. Once logged in, they can quickly enter, cancel or modify reservations at restaurants. In addition, they can search for restaurants by name, or they can see the restaurants nearby their location.

2 Design of the solution

We structured the backend in different microservices, in order for the application to be highly maintainable and scalable. Each microservice has its own purpose, and it's as independent as possible with respect to the others microservices, in order to reduce the overall coupling of the system. Each microservice has its own independent database.

Moreover, we used additional technologies such as MinIO (an Amazon S3-compatible object storage), to store any non-relational data such as images. To invoke the various features offered by the system, we will use APIs for single, well-defined tasks. The APIs have been built using REST technology.

In the local development environment, we defined a `docker-compose.yml` file in order to start all the containers that are needed to get the system up and running.

There are four main microservices:

Users It handles the users' operations, such as user registration and login. It stores users' data inside of a MySQL database.

Reservations Its purpose is the management of reservations. It exposes APIs that allow to add a new reservation for a certain user at a certain restaurant, for a given day and time. The data is stored inside of a MySQL database.

Restaurants Its purpose is to manage everything that concerns a restaurant. It handles the API calls for searching a restaurant with a given name, inside of a given city, nearby a certain set of coordinates and it's capable of applying other filters such as the restaurant's cuisine. Regarding the database, we used PostgreSQL. Differently from the other microservices that use MySQL, we opted for PostgreSQL mainly because the geographic queries support through the *Postgis* plugin, which allows querying restaurants that are in the radius of a certain set of coordinates in an efficient way.

Embeddings The *Embeddings* service provide a fundamental role in the restaurant search operation. In particular, it exposes an API that, given a query, returns the top k restaurants whose name is the most similar from the query. We leveraged the performances of the *Faiss* library in order to achieve extreme efficiency at performing vector L_2 similarity calculations. We built the indexed

collection of restaurant name embeddings (i.e. the *Faiss* index) by pre-computing the embeddings for all the restaurants in the database using the *Universal Sentence Encoder* model (downloaded from *tensorflow hub*).

At inference time, the query embedding is computed on the fly with the same model, and the result is cached using *Redis*, an in-memory key-value storage that allows the computation for future requests with the same query to be even faster.

The final distance value is then averaged with the *Levenshtein* distance between the query and the restaurant name, due to the empirically better results we obtain with respect to using only the embedding L_2 distance.

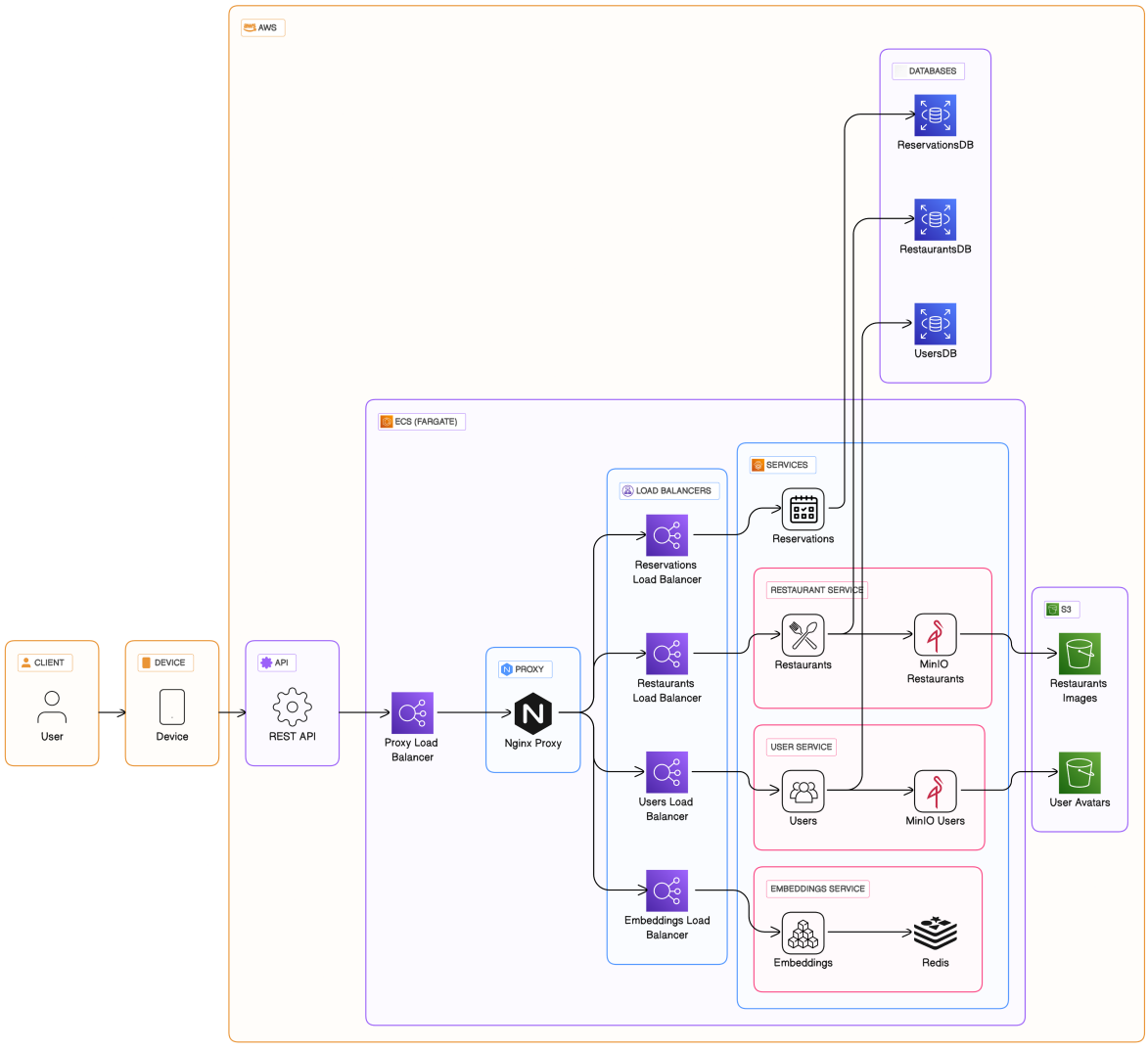

Figure 1: Architecture diagram

Before the just described solution, we attempted to use only the *Levenshtein* distance without using the Embeddings service, but since this metric only depends on the single characters that make up the restaurant name, it wasn't precise enough. We then tried to pairwise match the query embedding with a subset of the nearest restaurants' embedding from the user (which we stored on an S3 bucket). This solution had the same results as the current solution but it was extremely inefficient because of the pairwise embedding matching and the matching had to be done on a subset of restaurants, otherwise the computation and retrieval of the embedding from the S3 bucket were intractable.

Nginx Gateway

In order to connect all the microservices under the same URL, we used an Nginx gateway. This allows to define a single entry point, from which to access all the different microservices, by just appending the microservice's name.

Let's say the public URL is `https://voicefork-api.com/`, we can reach the `restaurant` microservice by making requests to the `https://voicefork-api.com/restaurants/` URL, which will redirect the requests to the `restaurant` microservice endpoints.

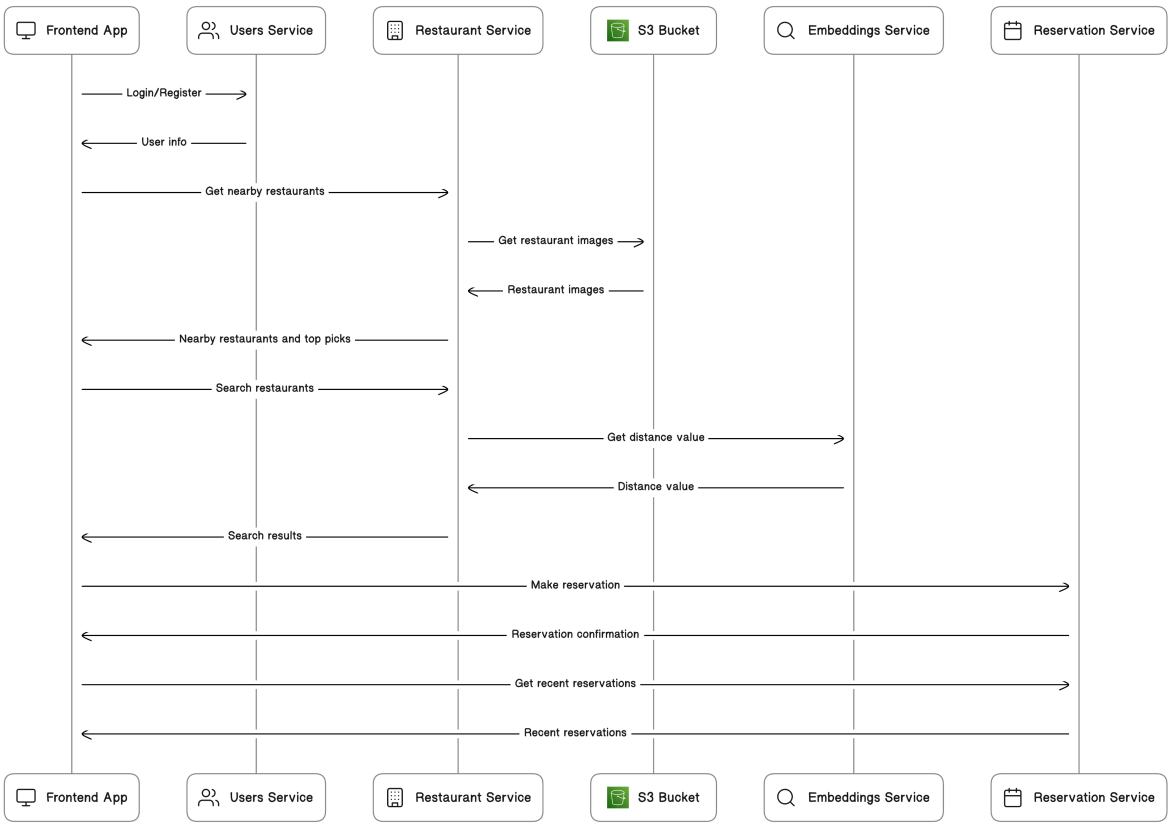

Figure 2: Microservices diagram

3 Frontend Application

The user interacts with the backend using a cross-platform mobile application developed using *React Native*. Let's analyze the API calls that are made when the user browses inside of the application.

3.1 Login and Registration

When the user loads the application, they're asked to login or register (Figure 3 and 4). Both these actions involve the `users` microservice.

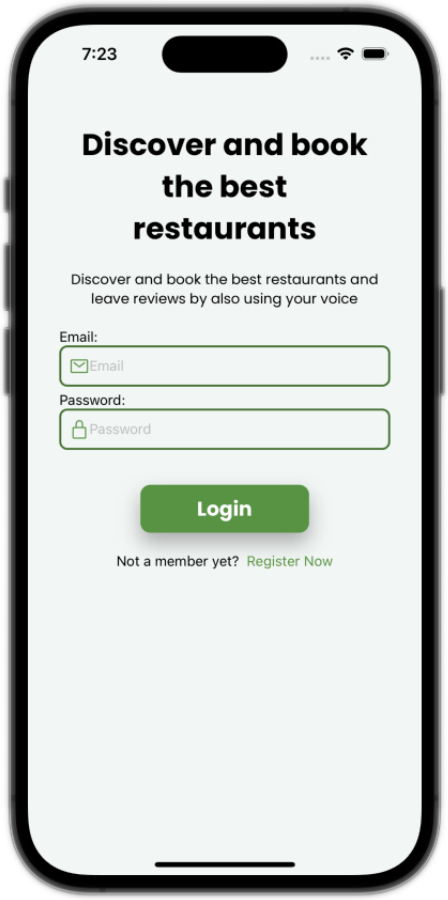

Figure 3: Login Page

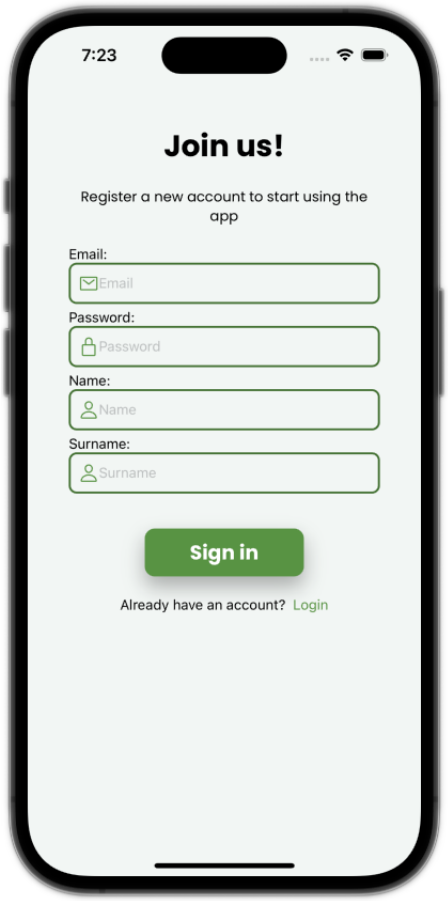

Figure 4: Sign Up Page

3.2 Menu and Homepage

Once the user logs in, the home page is loaded (Figure 5). Here the user is presented with two lists, one displaying the restaurant nearby their coordinates, and another one that displays the "top picks restaurants", consisting of the restaurants nearby that have the best overall review rating. In order

to retrieve this list, the `restaurant` microservice is contacted. During the requests, the `restaurant` microservice also contacts the S3 bucket where the restaurant images are stored. By clicking on the upper left-hand a menu will open (Figure 6) where the user can easily access its reservations, return to the home page or log out.

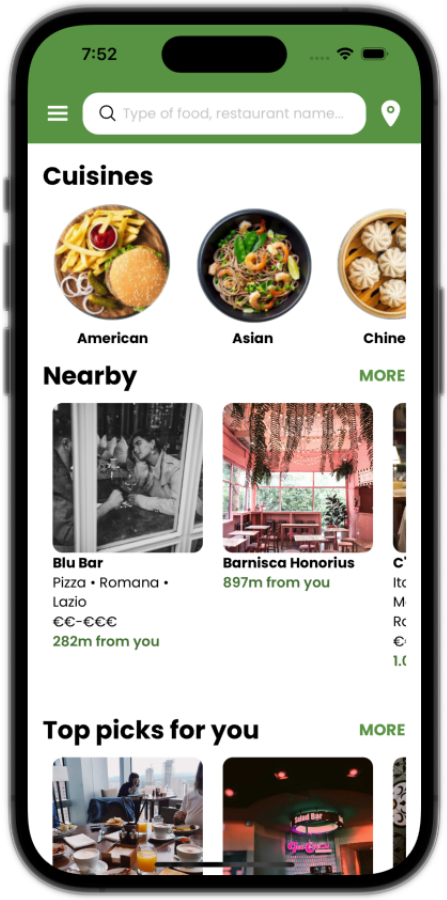

Figure 5: Homepage

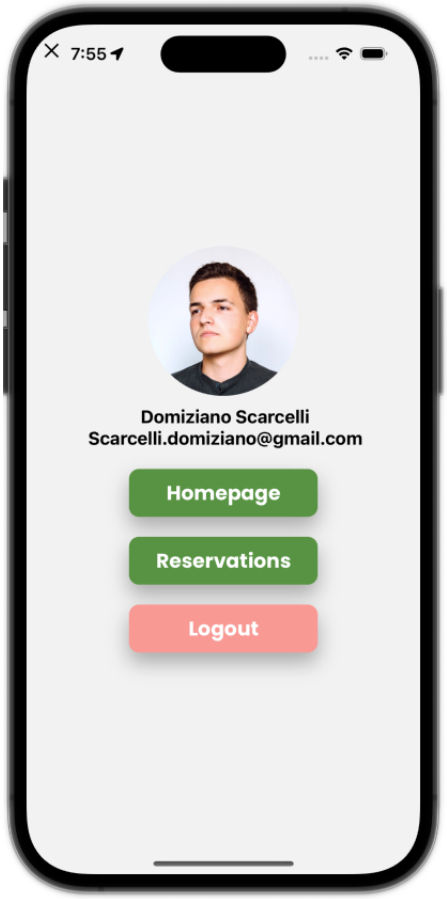

Figure 6: Menu

3.3 Restaurant search and details

The user can search for a particular restaurant by inserting their query inside the top search bar. This triggers another `restaurant` API that will search for the nearby restaurants whose name matches the query. In order to get a value between 0 and 1 that indicates the distance from the restaurant name and the query, the `restaurant` service contacts the `embeddings` service, which will retrieve the list of the k most similar restaurants from the query (Figure 7). Then the user can tap on the restaurant tile in order to visualize the restaurant details (Figure 8). From there the user can click a button to initialize the reservation process. During this process, the user will be asked for all the necessary information in order to complete the reservation (such as date, hour, and number of

people). Once those are provided, the reservation service is contacted in order to register the reservation inside of the database.

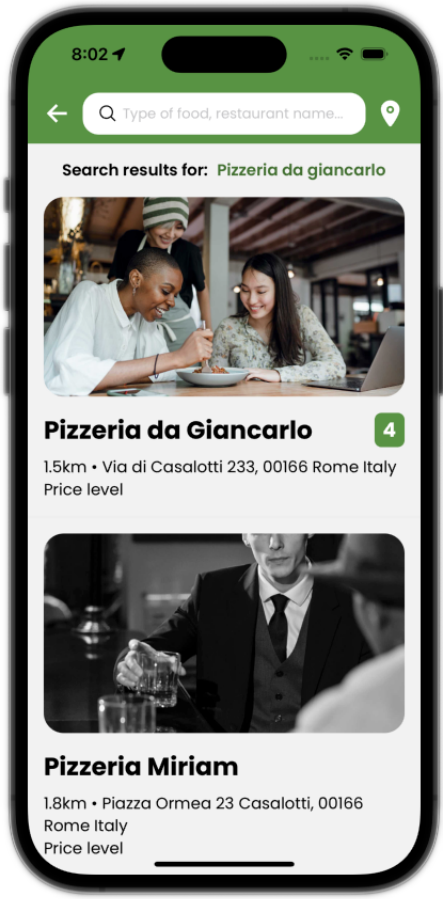

Figure 7: Search

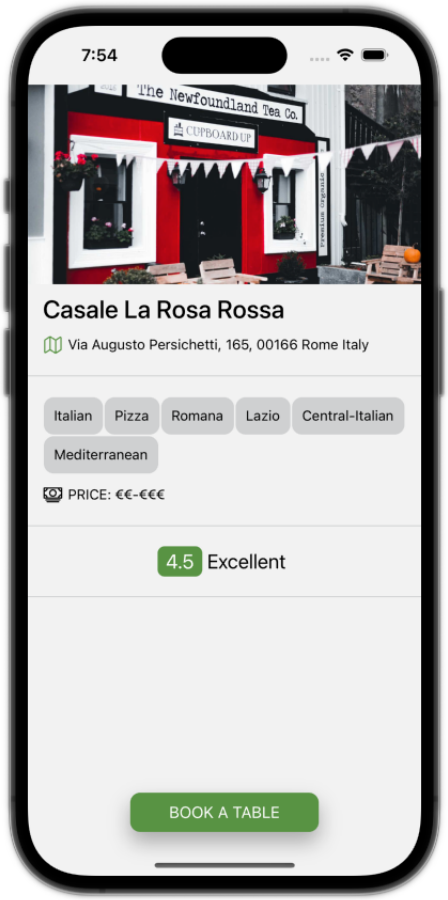

Figure 8: Restaurant Details

3.4 Recent reservations

The user can view their list of reservation inside of an application section (Figure 9), which will trigger the retrieval of the user's reservations from the `reservation` service. From here they can click on a reservation to see its details (Figure 10).

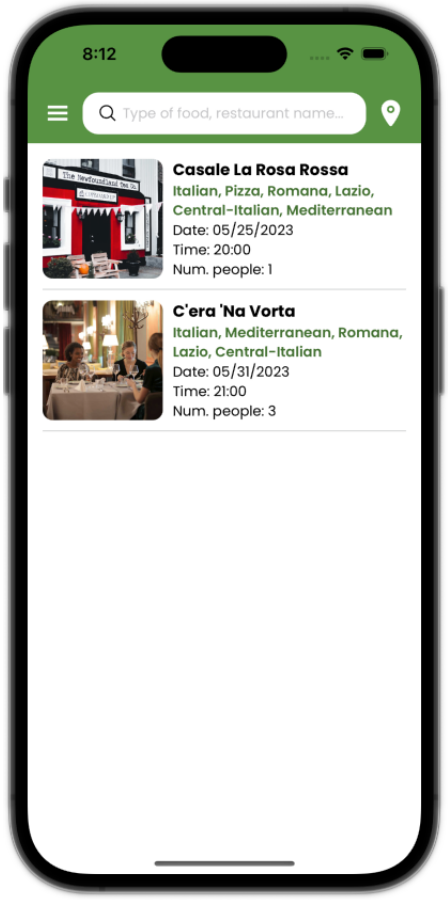

Figure 9: Reservation List

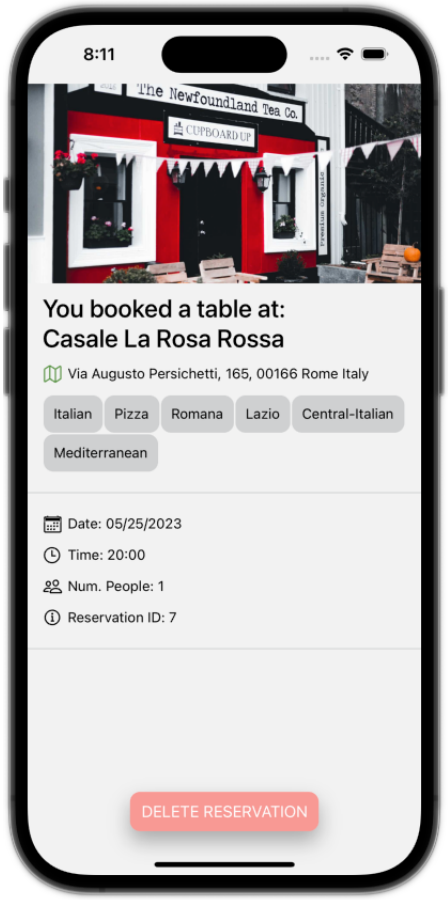

Figure 10: Reservation Details

4 Data

In order to make the application operate with a dataset close to reality, we decided to use the "Tripadvisor European Restaurants" [1], limiting only to the Italian restaurants. The original dataset has 1M+ instances. After the filtering, we are left with a dataset of over 200K instances.

5 Deployment

The usage of containerized microservice led us to the decision of using **AWS Elastic Container Service** to deploy the different microservices. In particular, we used **AWS Fargate** for running the containers.

For each service, we defined a *task definition* file that replicates the information inside the `docker-compose.yml` file. The task definition tells AWS what is the image to pull, how to start the different containers, which ports to map, which VPC to use, which volumes to mount, and how to manage autoscaling and load balancing.

As said before, regarding the unstructured data such as images and embedding vectors, we used the **AWS S3** service, accessed by MinIO client, the latter helped us to conciliate the operations between the production and development environment.

For what concerns the databases, we opt for **Amazon RDS**, which allows us to have a fast, reliable and scalable database.

To simplify the deployment process we used **Terraform**, an *Infrastructure as Code* tool that enables the declaration of the entire cloud infrastructure through a set of files. This approach allowed the replication of the infrastructure across various AWS accounts using a single command. Furthermore, it provided the capability to include dynamic parameters (such as `.env` variables) that rely on values from other services. Without this tool, such configuration tasks would have required a lot of manual effort.

6 Design of experiments

To test the application performance under different load levels, we used the **k6 load testing tool**, which allowed us to create and run load tests to simulate various scenarios and analyze application behavior under different load levels. In addition, k6 provides real-time metrics and reports to help us analyze application performance during testing, including response time, error rates, and throughput. We created a scenario in which the user has a real interaction with the application in order to simulate the behavior of the actual system. In this scenario the user can create a new profile or login the application (this will trigger the users microservice and the avatar bucket in which user pictures

are stores), get the list of restaurants in the homepage or search for a specific restaurant (this will trigger the restaurant microservice as well as the restaurant bucket that is used to retrieve the embedding and the images) and reserve a table or see the list of restaurants (this will trigger the reservation microservice). Everything is randomized so the user can perform any action every time only noticing that some of them are constrained by other actions (e.g. to reserve a table the user must first visit the homepage or look for a restaurant).

7 Results of experiments

7.1 Scenario tests

The test was done 10 times and each of them gave us some results. We retrieved the frontend metrics (response time, latency, throughput, etc..) using **Grafana** but since we were interested in the cloud side metrics, we got them directly from **AWS CloudWatch**. What we did was download the csv files showing the results of each test and then averaging the results taking into consideration the following metrics: percentage of cpu and memory usage, percentage of cpu and memory reserved and the number of tasks in the running state. Each test was conducted for **30 minutes** where each user, after registering or logging in, randomly performs one or more actions between searching for restaurants, loading nearby restaurants, loading the top rated restaurants, making a reservation and displaying them (so that each microservice is involved in the tests). The tests were composed of several phases:

- During the first 60 seconds, 20 users are simulated.
- During the next 120 seconds, 100 users are simulated.
- During the next 480 seconds, 300 users are simulated.
- During the next 60 seconds, 30 users are simulated.
- During the next 600 seconds, 400 users are simulated.
- During the next 30 seconds, 300 users are simulated.
- During the next 90 seconds, 100 users are simulated.
- During the next 60 seconds, 10 users are simulated.
- During the next 180 seconds, 150 users are simulated.
- During the next 100 seconds, 20 users are simulated.
- During the next 100 seconds, 5 users are simulated.

We decided to apply a set of Warm-up (WU), Rump-up (RU), Steady (S) and Rump-down (RD) periods in order to test the scaling ability of each microservice and to see their behaviors. Notice that each microservice was given a cpu of 512 units (0.5 vCPU) and a memory 1024MB (except for that the embeddings service that was given respectively 1 vCPU and 4096MB). We equipped a **load balancer** for each microservice in order to distribute the traffic to each running task (if more than one are in the running state). The **autoscaling target** had a minimum capacity of 1 and a maximum capacity of 10. Finally, the **autoscaling policy** was given a target value of 60 regarding the average percentage of CPU utilization with a scale in cooldown of 180 seconds and a scale out cooldown of 120 seconds. These values were given after having done some smaller tests to see how the autoscaling behaved. Let's now analyze how the various microservices performed.

7.1.1 Users

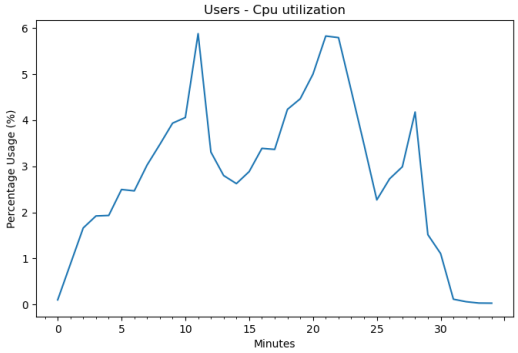

Figure 11: Users - Cpu utilization

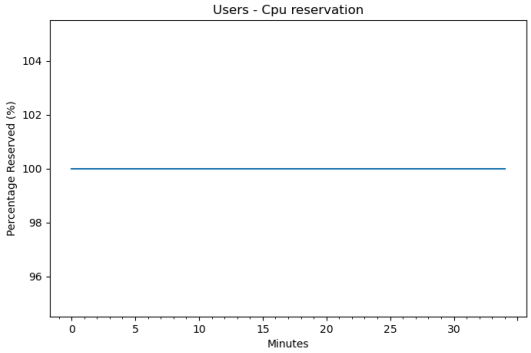

Figure 12: Users - Cpu reservation

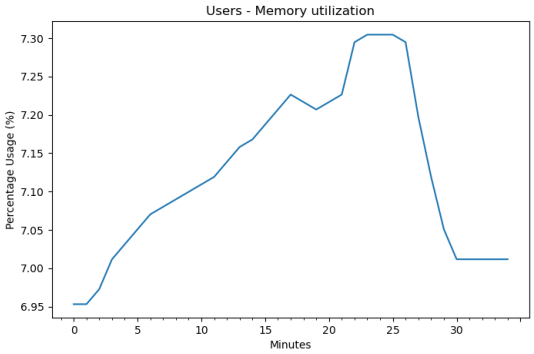

Figure 13: Users - Memory utilization

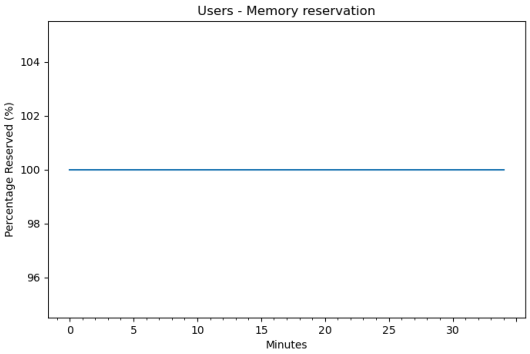

Figure 14: Users - Memory reservation

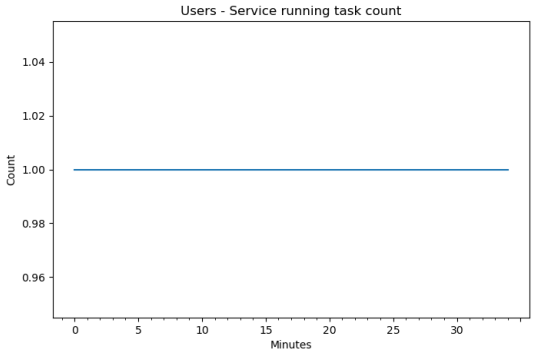

Figure 15: Users - Service running task count

The graphs show that the microservice did not perform any scaling operation, as the number of running tasks, such as memory and reserved cpu, remained unchanged. However, it shows efficient CPU utilization during peak periods and an increase in memory utilization that is not very high as the workload increases. Notice that CPU and memory reservation are always at 100%. This is due to the fact that the microservice never scales (the number of running tasks is always one) so the allocated CPU is always 0.5 vCPU as well as the memory that is 1024 MB.

7.1.2 Reservations

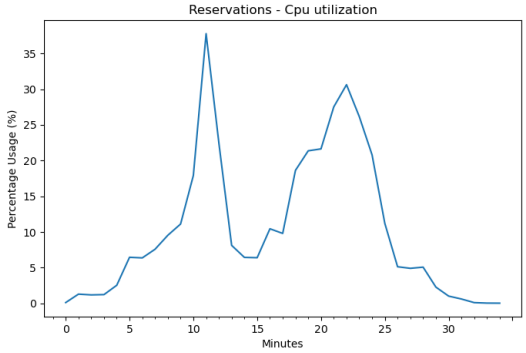

Figure 16: Reservations - Cpu utilization

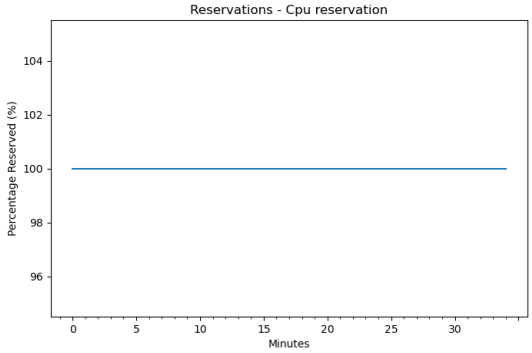

Figure 17: Reservations - Cpu reservation

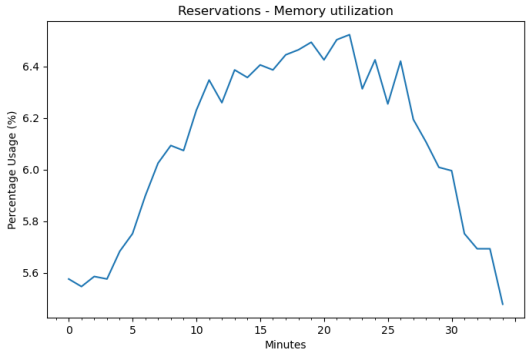

Figure 18: Reservations - Memory utilization

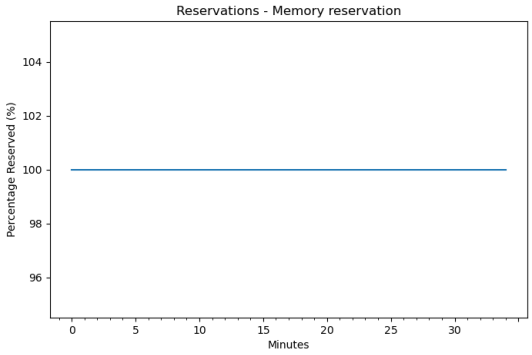

Figure 19: Reservations - Memory reservation

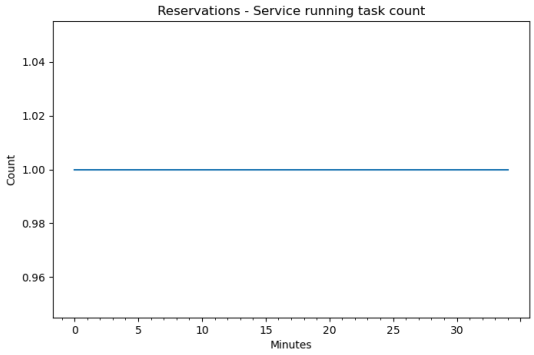

Figure 20: Reservations - Service running task count

From these graphs we see that this microservice also did not perform any scaling operation. Here the cpu utilization was more intense, especially during the two peaks where there was more activity. While memory utilization continued to increase but did not exceed critical levels throughout the duration of the test.

7.1.3 Restaurants

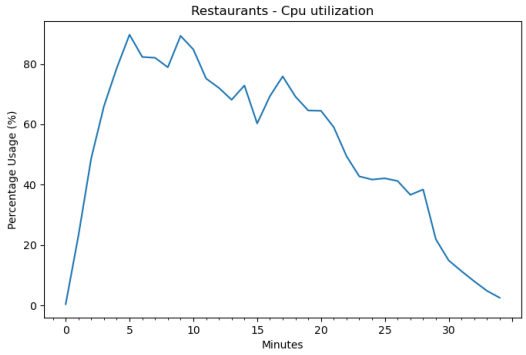

Figure 21: Restaurants - Cpu utilization

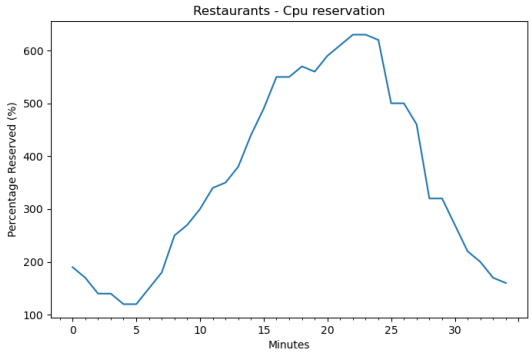

Figure 22: Restaurants - Cpu reservation

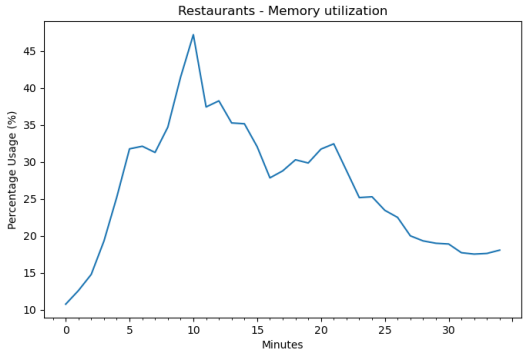

Figure 23: Restaurants - Memory utilization

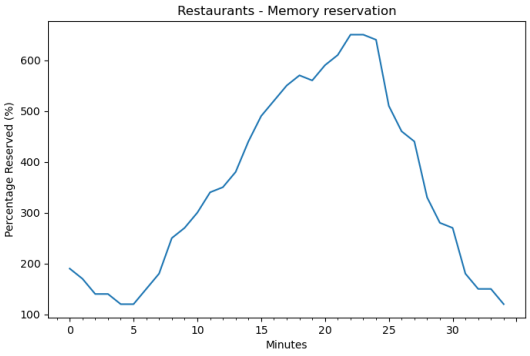

Figure 24: Restaurants - Memory reservation

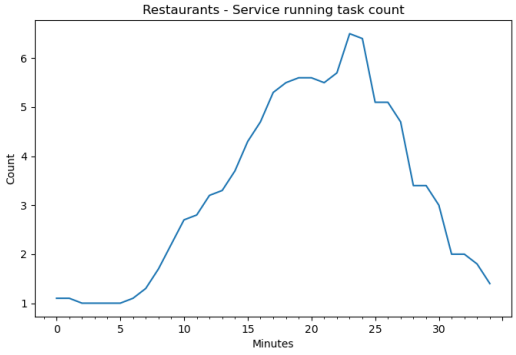

Figure 25: Restaurants - Service running task count

In this case, we see that the microservice scales effectively as cpu utilization increases to over 60% for about 180 seconds (threshold at which autoscaling is activated) and then scales out as its use decreases. In fact, CPU utilization undergoes a noticeable increase as requests grow and the same thing happens to memory utilization where we have two peaks, suggesting an initial demand for memory and an increase in utilization toward the end of the tests. Concerning the CPU and memory reservation, it is possible to notice how they have the same behavior or the running tasks. In fact, the CPU and memory reservation in the container increase and decrease according to how many tasks are running in a certain instant. During the peak, there is a 600% CPU/memory reservation. Given that one task allocates 0.5 vCPU and 1024 MB of memory, during the peak there is an average of 6 tasks so the total CPU and memory reservation in the container is about 3 vCPU and 6144 MB of memory that is the 600% of the normal CPU and memory reservation. In summary, we compute the % by multiplying the memory reservation of a single task (1024 MB for the restaurant microservice) for the number of running tasks in a certain instance of time (during the peak we have 6*1024 MB = $\sim$ 6144 MB) and then dividing the result by the memory reserved by a single task multiplied by 100 (6144 MB/1024 MB = 6*100 = 600%). The same applies to CPU reservation but using the number of vCPU reserved.

7.1.4 Embeddings

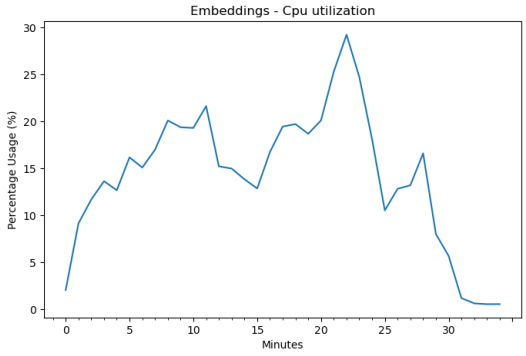

Figure 26: Embeddings - Cpu utilization

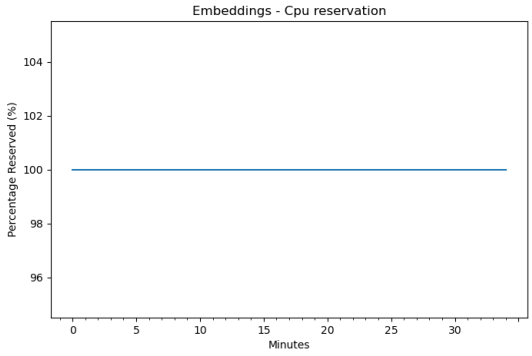

Figure 27: Embeddings - Cpu reservation

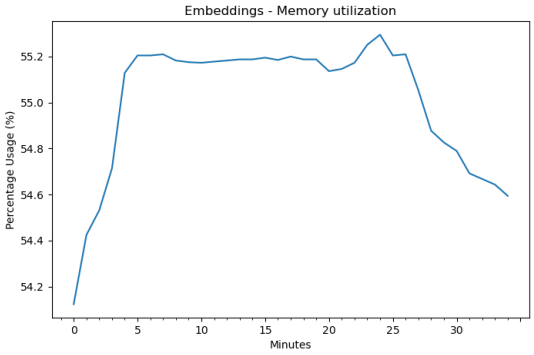

Figure 28: Embeddings - Memory utilization

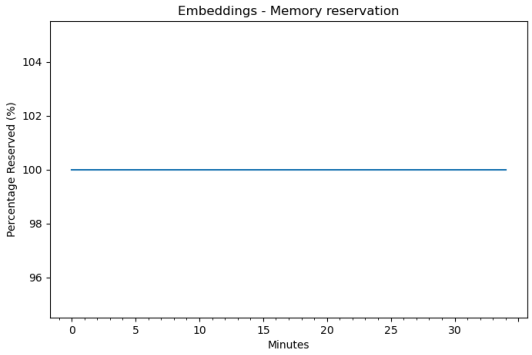

Figure 29: Embeddings - Memory reservation

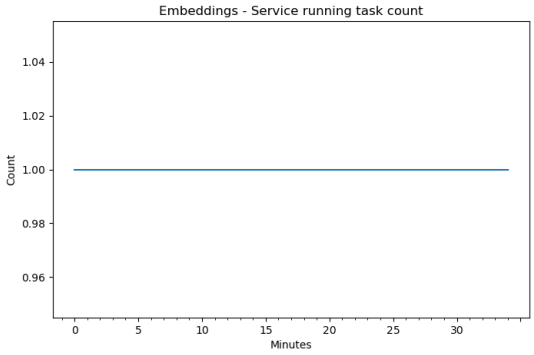

Figure 30: Embeddings - Service running task count

This microservice also did not perform any scaling operation. CPU utilization is not very high, while memory utilization increases rapidly from the beginning of the tests and remains constant throughout, indicating a stable allocation of memory resources.

7.1.5 Proxy

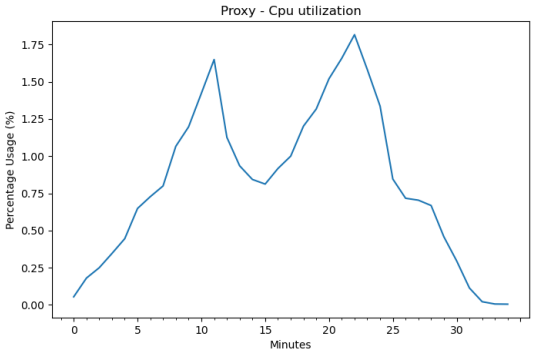

Figure 31: Proxy - Cpu utilization

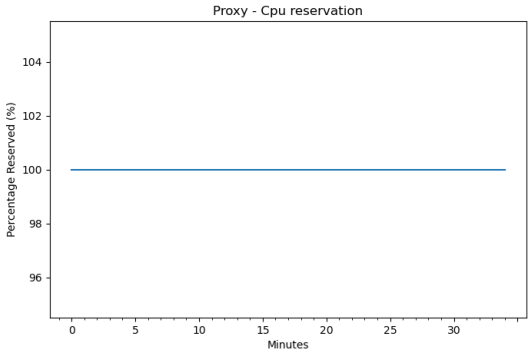

Figure 32: Proxy - Cpu reservation

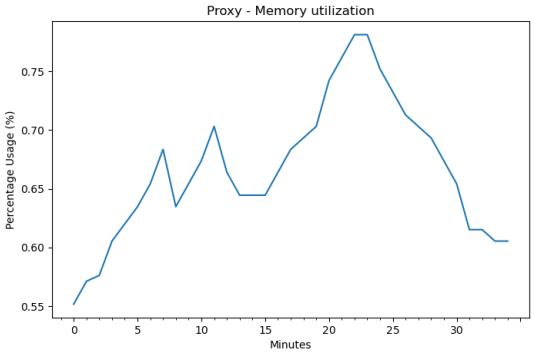

Figure 33: Proxy - Memory utilization

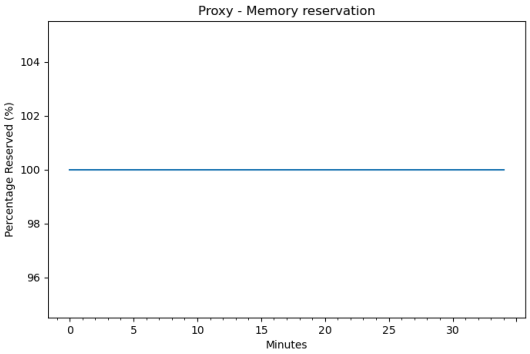

Figure 34: Proxy - Memory reservation

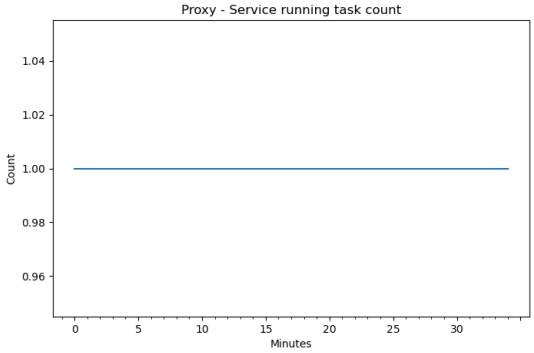

Figure 35: Proxy - Service running task count

This microservice also did not perform any scaling operation, where we have very low cpu and memory utilization, a symbol that the load of requests did not slow down its execution.

7.2 Availability tests

In addition, we did an in-depth test to check the availability of each microservice, specifically we tried to interrupt them one by one to see if the system continued to be available and which microservices would be affected by this. Tests showed that when we killed the microservice:

- **Users**: If the user had already logged in, as long as the session persisted, he could perform all possible actions without interruption, and all other microservices continued to run smoothly. But this did not allow users who were not registered or logged in to use the application.
- **Reservations**: Also in this case all the microservices continued to run normally, the only actions that could not be performed were the actual reservation of the restaurants and the visualization of the reservations made.
- **Restaurants**: Here we encounter the first problem when we tried to access the home page as we were not able to view any restaurant, for this reason it was also impossible to make any reservations, but we could see the ones which were already done.
- **Embeddings**: In this case the only operation that we couldn't perform was the restaurant search by using the appropriate search bar.
- **Proxy**: This is the main microservice that connects all the others, without this it is impossible to access all the application features.

8 Conclusions

We built the application using the microservices architecture, and this allowed us to take advantage of various benefits of this architecture such as the fact that it is independently deployable (we started by building the restaurant service and then built the other services alongside), loosely coupled (each microservice is completely independent) and organized by a small team (in our case each member of the group focused on a specific service). Each service was containerized using Docker Compose and this allowed a fast deployment on Amazon ECS using Terraform to do the job. Several tests performed on the deployed application showed us the potential of the cloud in managing the workload very well by adding and removing resources automatically based on the previously chosen policy. This allowed us to manage the infrastructure quickly so we didn't have to worry about buying new physical resources.

References

[1] Tripadvisor European Restaurants Dataset