**Report**

**Introduction**

**Project title**: CovidAWS

**Groups members**: Ciammaruconi Daniele; Caparvi Giorgio

**Description**

Our project’s purpose is to create a cloud application that keeps track about people affected by covid virus.

We provide a website where a user can see data about people infected and healed by covid virus. A user can also label himself sick or cured with two buttons on the site.

The application relies on the use of Amazon AWS and the following services:

* PostgreSQL
* S3 bucket
* Elastic Beanstalk

**Deviations from Proposal**

Elastic Beanstalk

Instead of EKS we used this technology, which is like Fargate, because we decided to put a greater effort on code development and minor on the infrastructure. It enables automatic scalability of the Microservices and associated database.   
Because we used a Lab Environment, we couldn’t use some features just like Load Balancing and instances scaling.   
The great advantage of Elastic Beanstalk is that you only pay for the resources that are used by the application, with no additional cost for using the technology itself, unlike Fargate. It also allows scaling configuration based on cpu load.

**Implementation (Development)**

Analysing the service to be implemented, we built the application with following components:

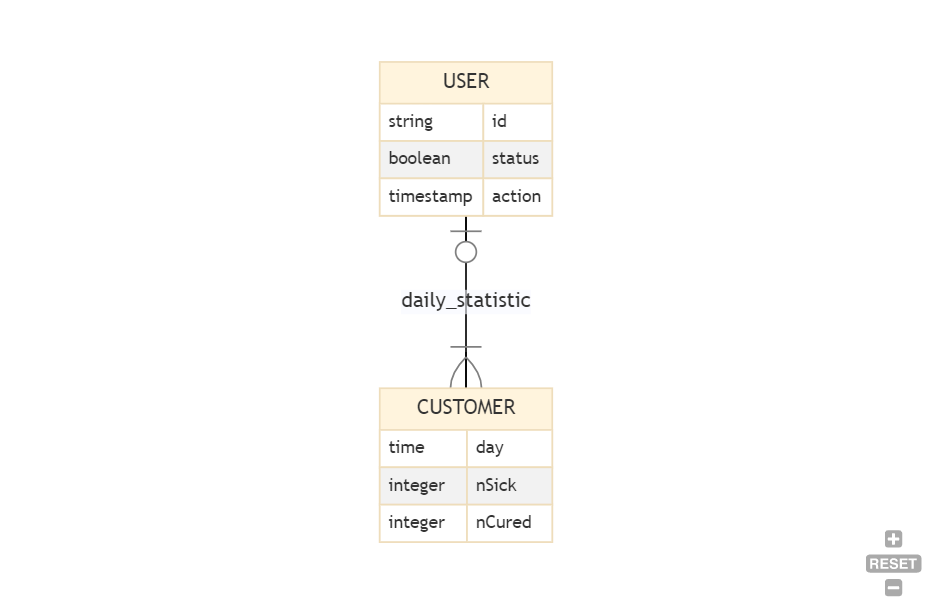
* PostgreSQL (Contanerized)
* Counters microservice
* Eureka
* Gateway
* React

DB

During the development, a containerized image of PostgreSQL and PGAdmin were used. We decided to use PostgreSQL because It has a very efficient built-in replication system that can be utilized to provide good scalability.

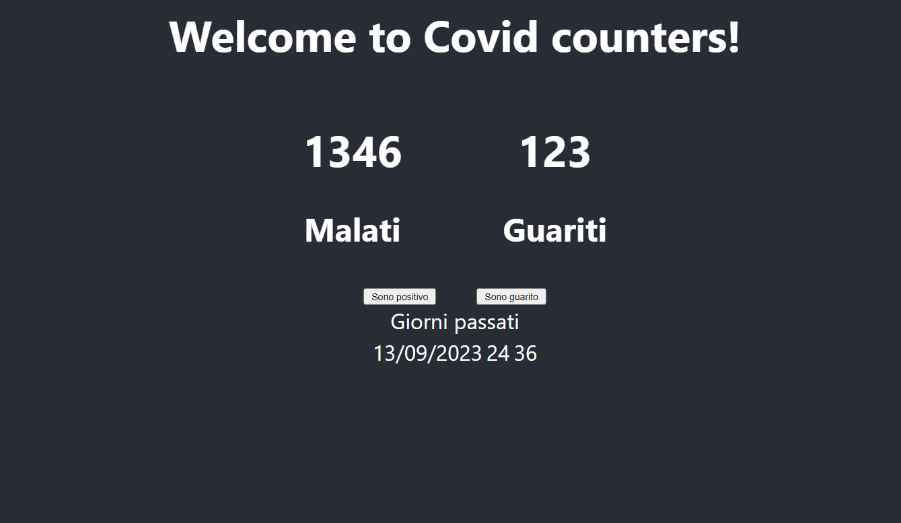
As for the db, the following entities were analyzed: **users** and **counters**.

In the users query, all users who were sick or healed are recorded, along with the times when the operations were performed. To optimize the database, the **id** of each user is unique, so that a user is not allowed to add himself multiple times to the sick counter. In the counters query, on the other hand, the total daily number of sick and healed users is collected at 6 p.m. each day, and to optimize the service, once the counter is created, we remove all healed users from the query, as the system does not benefit from keeping records of healed users after defining that they are no longer sick. Below we can see the ER schema.



### Front-end

For the front-end part we decided to use **React** to make a simple interface: it allows you to see the counters and through the buttons below you can notify the service if you are positive or healed; to simplify input operations, the call that starts from the buttons contains only a random id. Each time the user refreshes the page, the services will be called to update the counts. Finally at the bottom you can see the counters of the past days.



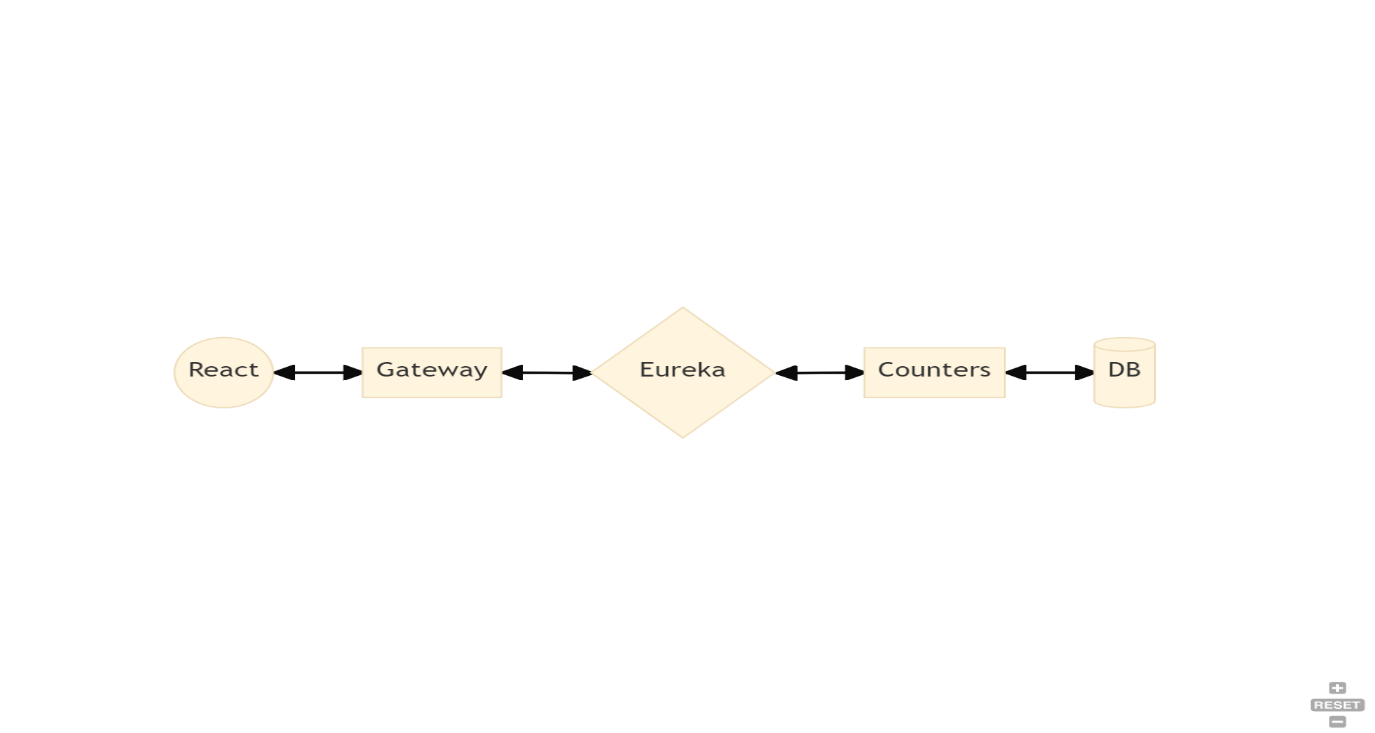
### Microservices

We implemented the microservices with Java Spring Boot.

We used several technologies from Spring framework to develop the application in a local environment:

* Eureka: It’s a Load balancer and discovery client. It allows communication across all the instances of all the microservices.
* Gateway: Part of Spring Cloud technology, it gather all the request coming from front end. It is implemented with a CORS Filter, which define which API Calls are allowed from a specific source.
* Counters: the developed microservice. We have two get calls (to fetch the number of sick and healed users), two post calls (to send statistics) and a scheduled function to generate the daily counter.  
  Thanks to the timestamp in the **user query**, we were able to correctly implement the function, thus fetching (and subsequently manipulating) all records that were correctly registered by 18:00

All configuration file were written in **.yml** extension, which is compatible with container technology.



**Deployment**

## Several technologies were removed because they are already implemented in AWS technologies.

## We use a S3 bucket to host our static web application. The deployment was straightforward: after we built our react application, we just uploaded all files and allowed public access. Then we wrote a simple configuration of Properties and CORS, for the bucket, and the app was ready to go.

## Immagine che contiene testo Descrizione generata automaticamente

## Talking about Elastic Beanstalk we were able to remove Eureka, because load balancing is already implemented in the technology. Unfortunately, due to environment limitation, we couldn’t correctly configure the Gateway, we are not allowed to create IAM Role or Security Groups. So, we implemented the CORS Filter inside Counter Microservice, and Elastic Beanstalk oversees all the traffic across all the possible instances.

## 

## Other two important features of Elastic Beanstalk are Logging and Monitoring:

## Logging: it allows to download, in any moment, a log file with all, or the last 100, lines.

## Monitoring: there is a specific sections which report several metrics, such as: health, cpu load, network load.

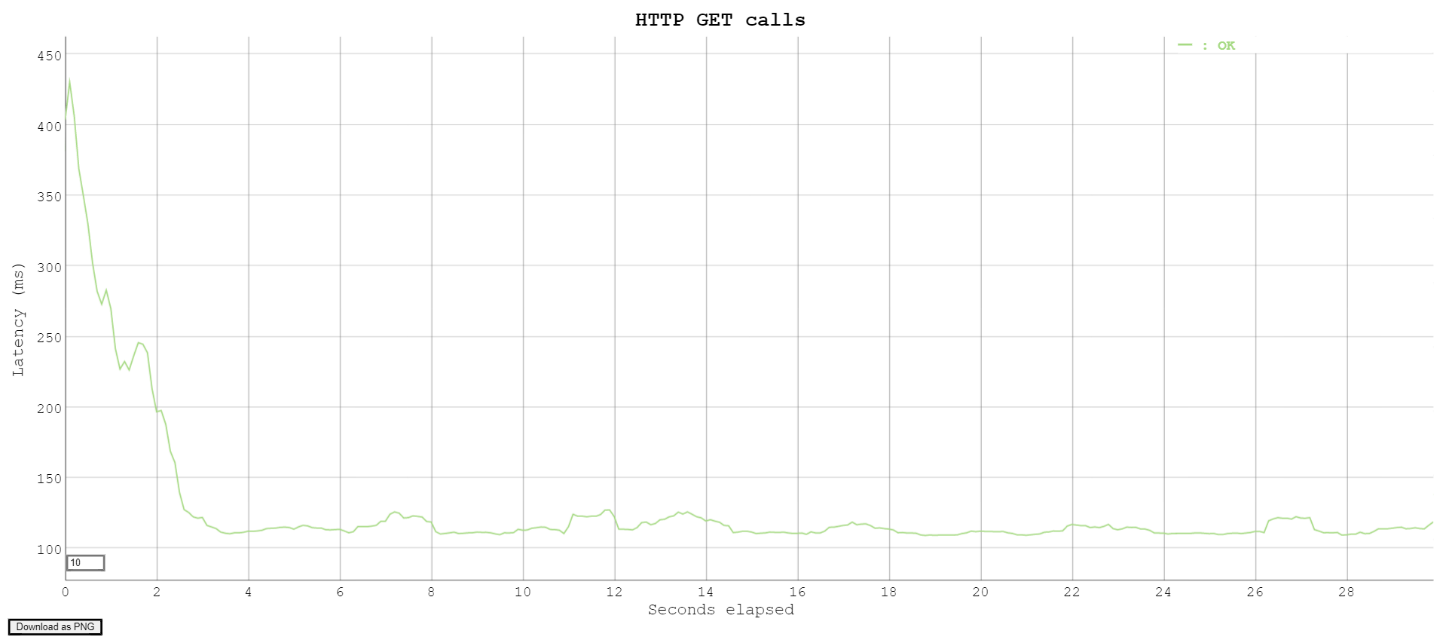
## 

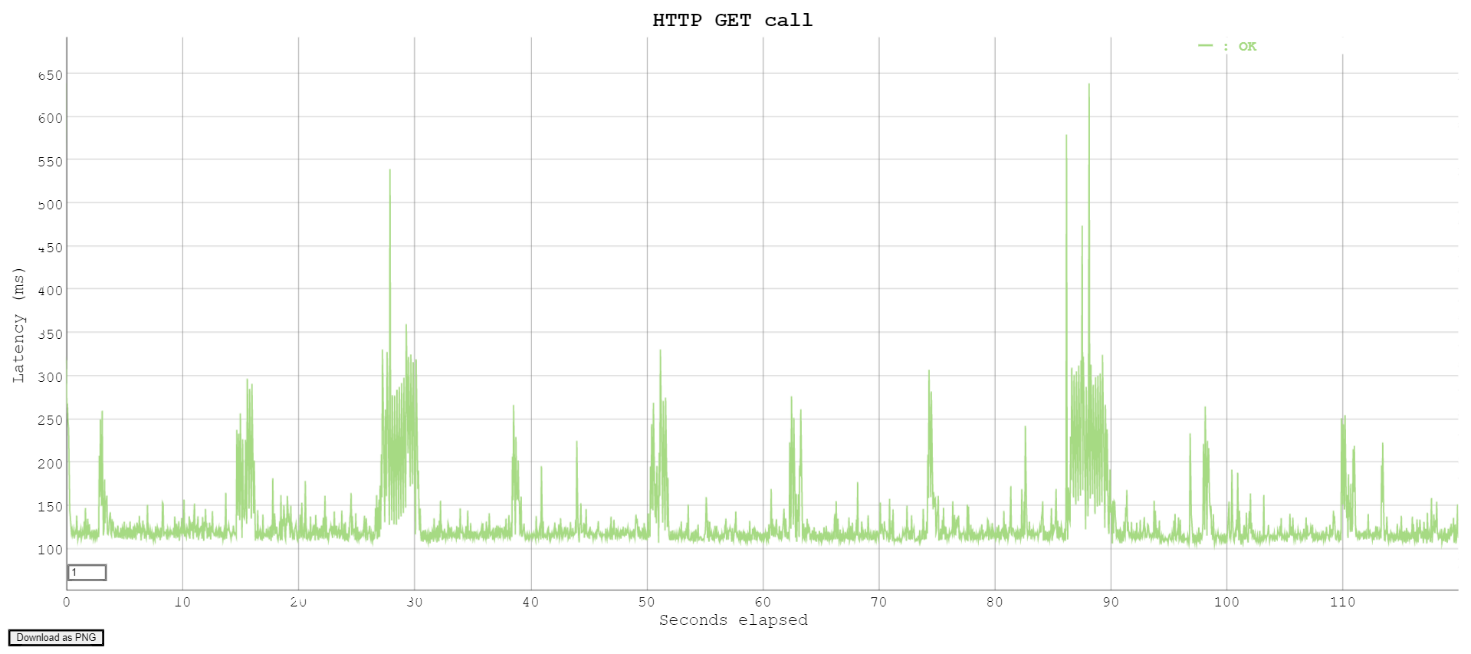
## To conclude, the management of the Database, which is automatically allowed to communicate with the microservice, has been delegated to Elastic Beanstalk. Thanks to the concept of environments, which is intrinsic in Elastic Beanstalk, we weren’t in charge to create, or configure all the AWS technologies needed, and we were able to manage the connection between microservices and the DB without hardcoding all the needed parameters and, we used environment variables for an easier deployment.

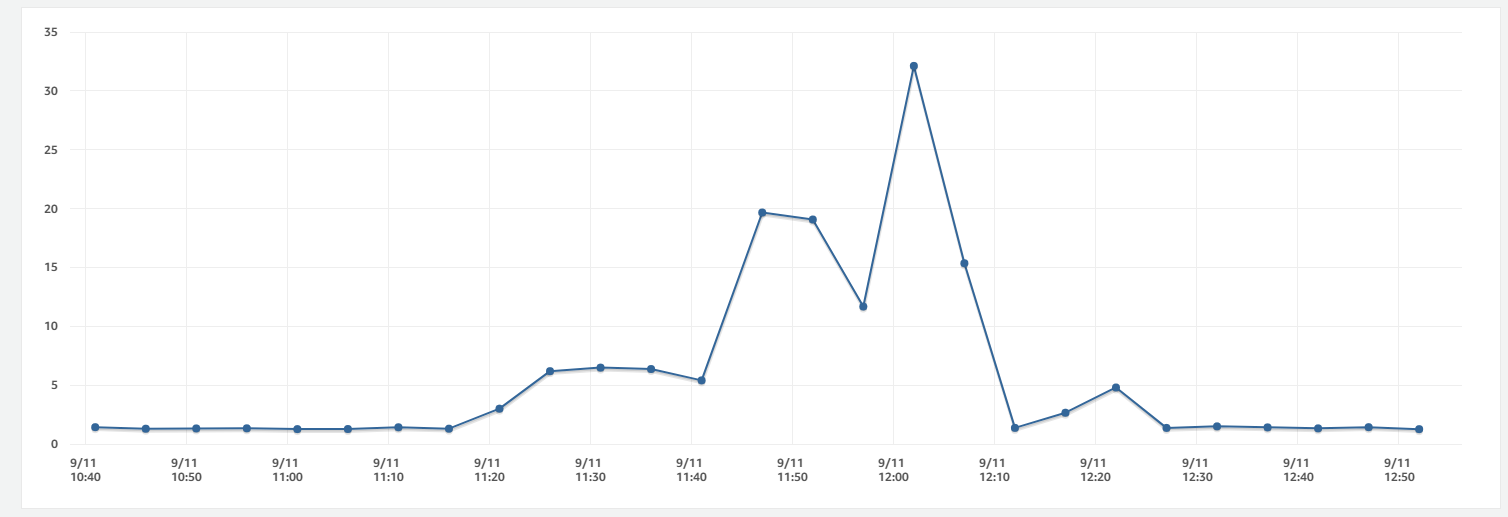
## 

## Test & Validation

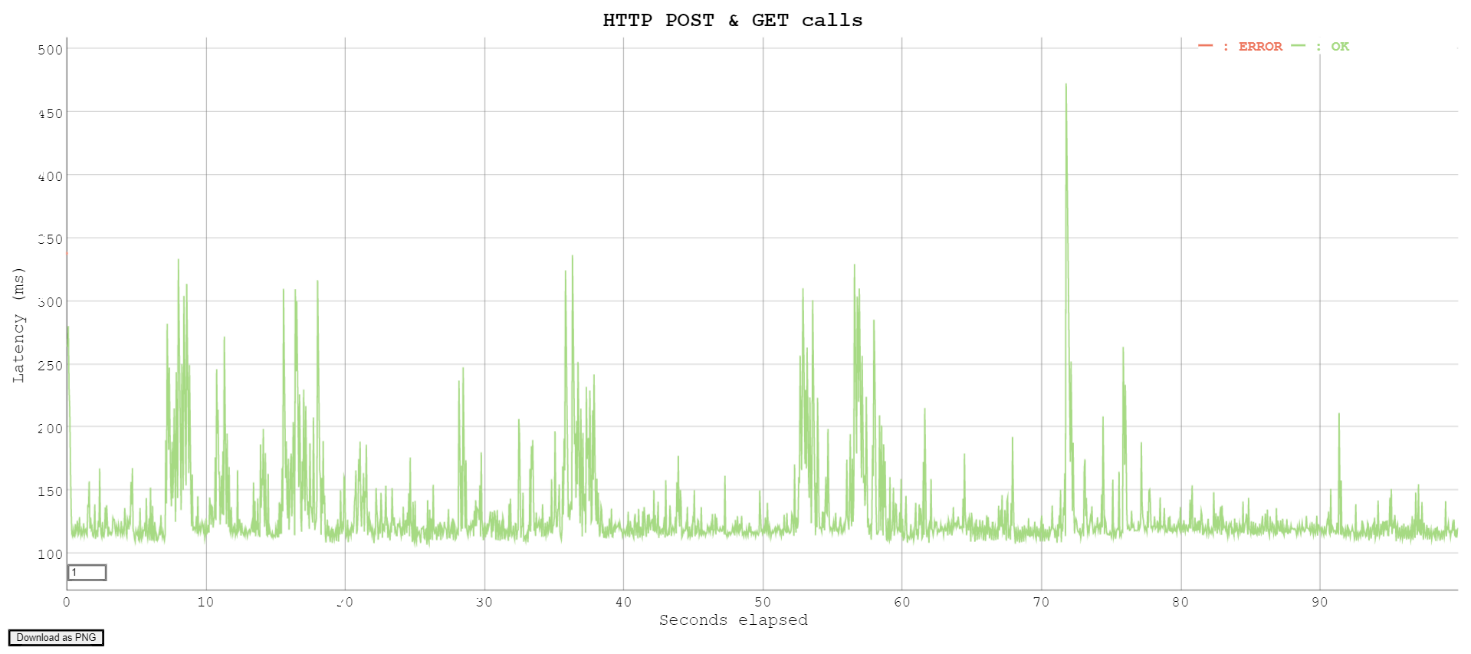
We wanted to test and verify the efficiency of our application with 3 types of tests (multiple reading of data, multiple writing of data, hybrid read/write) with different amounts of requests. These were the results:

In this first test we executed 400 GET requests in 10 seconds, so the machine received about 40 requests per second. As we can see in the graph, they were all executed successfully, with a very promising latency.





In this second test we wanted to test an extremely higher load of requests, trying to execute about 120 GET requests per second. This test was also successful, although the machine had a spike in requests and the workload was significantly higher than in the previous test, all requests were fulfilled but the average latency, compared to the first test, increased.



### In the third test, we tested a different load of requests, in fact we randomly alternated GET and POST requests, always keeping on a workload of about 10 requests per second. This test also succeeded correctly, the machine had a peak of requests, the workload was similar compared to the previous test, and all requests were fulfilled. The average latency this time remained like that of the second test.

### Finally, we did one last test divided into two subtests: in the first one we performed about ten GET and POST requests for half an hour, while in the second one we performed a large number of requests to see how the machine would react. The results are below:

### Immagine che contiene tavolo Descrizione generata automaticamente

### 

### 

### In the first picture shows the first subtest, and the second picture shows one of the peak requests made during the first subtest. As can be seen, the requests in the long run did not have any problems, except when we ran the tests with high peaks of requests: in particular, the first peak of errors was recorded when we ran together another 200 requests per second for 10 seconds, and the other two peaks, however, were recorded when we ran 90 requests per second for 10 and subsequently 15 seconds.

### Analyzing further (thanks to the first two photos) we can see how at first the service responded correctly to all the requests that were made, but after a few seconds the requests start ed to be executed again.

### As we can see in the aws dashboard, the errors are caused by Elastic Beanstalk, which by seeing all these requests from the same machine secures the service from possible malicious attacks, initially by not returning the requested data and later by not giving a response to the service. So, also thanks to this last test, we can say that the performance tests, although carried out on a server in America, were passed successfully and the performance is what we expected to achieve.

### Conclusion and future implementations

The advantages of using cloud technologies such as Docker are obvious. Especially after having developed and tested all our services locally on our computers, first as simple programs, then as Docker images. We have seen how it is easier for a developer to work without worrying in detail about the network or scalability in the software logic.

In addition, thanks to Elastic Beanstalk and scaling policies, we gained benefits such as self-healing and scalability. because any crashed pod is automatically replaced or replicated. We achieve high data availability thanks to the AWS platform and RDS service that hosts our database in the cloud. In the end, our cloud application works well enough, scales well enough not to drop requests and demonstrates how a simple application can serve thousands of requests simply by using cloud technologies.

In the future, we hypothesized that we could improve it by making it a true monitoring service that can handle requests from the population of a nation or country, improving performance based on the requests that will be made and implementing a better interface, thus allowing us to extrapolate anonymous statistical data that will allow us in the future to combat the diseases we encounter in a more targeted way.