

**Phase 10 – Final report**

**Group 2:**

Diogo Lança – fc53495

Meuri Canhanga – fc42559

Miguel Silva – fc58974

Pedro Santos – fc46417

Renato Ribeiro – fc57433

1. **Motivation**

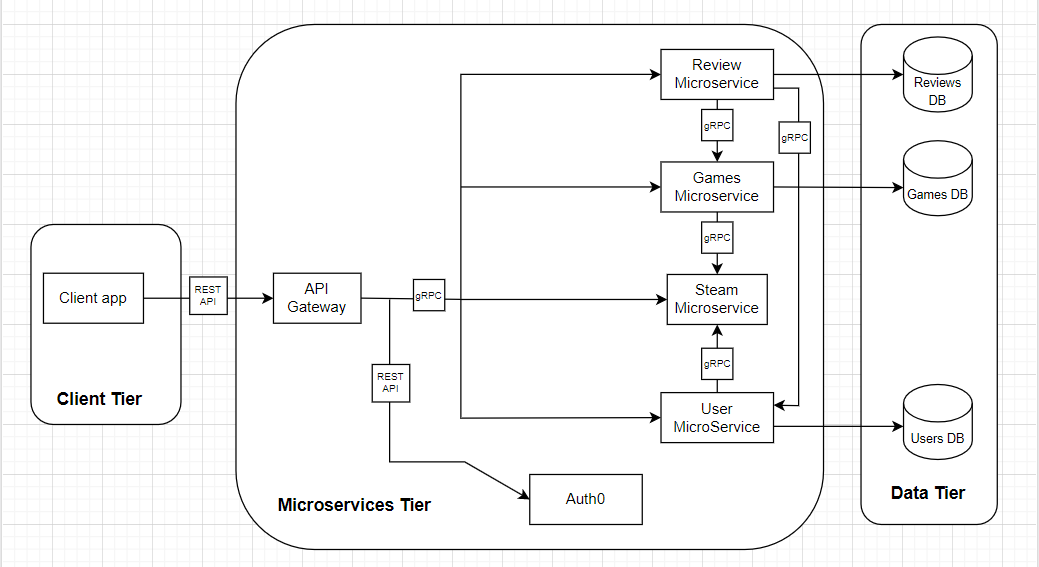
Steam is one of the most important platforms in the world of PC gaming, boasting over 150 million users, and we (project maintainers) are also included.

Due to our use of Steam, being familiar with the game service and having the chance to develop a cloud native application, we thought it would be cool to offer a service like Steam, where it would be possible to review games. These reviews would help other users by recommending games, based on the reviews written by other users.

1. **Dataset characterization**

For the project, we selected a subset of a **dataset** of around 21 million **user reviews** in multiple languages of around 300 different games on Steam (video game digital distribution service). This dataset has **1,07 GB** worth of user reviews, **last updated on 25 January of 2021**. Worth noting that the dataset was downloaded from Kaggle[[1]](#footnote-1) in a **CSV** **format**.

1. **Architecture**



**Figure 1** – API architecture

Figure SEQ Figura \\* ARABIC 1 – API architecture

On our architecture the client will use the operations with the Swagger User Interface (UI).

The client will access our project by a public IP address on a browser – for example ‘http://[35.233.14.122/’, where he needs to include ‘ui/](http://35.233.14.122/ui/)’ on the URL, to access the user interface. The IP is obtained on the Google Cloud Platform (GCP) after deployment. On this moment forward the client can “try out” all our implemented operations, which are restricted by roles through the Auth0.

In total we have six microservices working at the same time, but not all run on the google cloud, like - auth0. The main objective or reasoning is the same as Steam, where everything would be around games. We would have a main microservice, where it would show recommended games to anyone (don’t need to authenticate) and by “clicking” on the game would be possible to see the game reviews. Only after the login more operations would be available. For example, if a user logged in and accessed a game, it will not only show the game reviews, but also be possible for that user to review that game, and “upvote” other user’s reviews – like insinuating that review is useful/helpful.

Since our UI is not the typical webpage like [Steam](https://store.steampowered.com/), we adapted the architecture to make sense in the Swagger UI – where lists all operations in the page. Instead of requesting to a main microservice, the gateway communicates with all the other services, with the proper authentication its possible to utilize the respective implemented operations. The API gateway is responsible for transforming the REST requests from the client to gRPC requests that are used internally between services.

Three of the six microservices are responsible for the database connections (all three hosted by [MongoDB](https://www.mongodb.com/)). Since each have its own purpose, the databases are independent of each other. The three microservices translate the requests they receive into inserts, deletes, updates or queries to the respective database and capable of translating responses from the database into responses so that the other microservices can understand.

**Note**: Since leaving a machine deployed uses resources on the GCP, it’s possible that the machine is down, and the IP address won’t work. So, if by any chance the client has access to our git repository and doesn’t have a GCP account with money to utilize google's resources, it’s possible to access it locally – on the bash 🡪 go to the script’s directory 🡪 and run the build\_dockers.sh file (not forgetting to have the docker running). Then on a browser the client can access the project with ‘localhost:5000/ui/’ and “try out” the operations.

1. **Use Cases**

We defined 3 different roles for the access to our application: Admin is an authenticated user that has all the user permissions plus some extra ones, User is the normal authenticated user, and then Anyone is for the users non-authenticated that try to access the app, this last ones have very few permissions since they are not verified.

**Table 1 -** Use Cases of the application for each role

|  |  |  |
| --- | --- | --- |
| **Services** | **Role** | **Functionalities** |
| **Normal** | **Anyone** | User sign in |
| User login |
| Suggest games on the main page based on recommendations |
| Search for a game |
| **User** | Review a game |
| Edit a review |
| Upvote a review of another user |
| Filter Reviews |
| See all Reviews |
| Set a game has recommended/not recommended |
| See the total number of games in user possession |
| See total playtime (since account creation) |
| User logout |
| **Admin** | Add games |
| Remove games |
| Delete Review |
| Create Users |
| Remove Reviews (case of wrong/ inappropriate content) |
| Remove users (case they don’t comply with the laws reserved to the service) |

1. **REST API**

**Table 2** *-* API Specification, all operations possible in the application

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Micro**  **Services** | **Roles** | **Call** | **URL** | **HTTP Body** | **Result** |
| **Steam** | Any | GET | /steam/recommendedGames | empty | Get recommended Games |
| User | GET | /steam/activeusers | empty | Get most active users |
| **Users** | Admin | POST | /users | JSON string | Creates or updates a user |
| DELETE | /users/{user\_id} | empty | Delete account |
| User | GET | /users/{user\_id} | empty | Get information about a specific user |
| POST | /users/{user\_id}/games/  {app\_id}/reviews | JSON string | Create a new Review |
| **Reviews** | Any | GET | /reviews/{id} | empty | Get review by id |
| Admin | DELETE | /reviews/{id} | empty | Delete a review by it’s id |
| User | PUT | /reviews/{id} | JSON string | Updates an existing review |
| PUT | /reviews/helpful/{id} | empty | Set review as helpful |
| GET | /reviews/games/{id}/  {language} | empty | Get game reviews by language |
| GET | /reviews/filter/users/  {user\_id} | empty | Get all reviews from a specific user |
| GET | /reviews/filter/helpful | empty | Get all the helpful reviews |
| **Games** | Any | GET | /game/{name} | empty | Search for a game by name |
| GET | /games/{id}/reviews | empty | Get all reviews of a game |
| Admin | POST | /game | JSON string | Create a new game on the list |
| DELETE | /game/{name} | empty | Delete a game by its name |

1. **Implementation**

We have implemented two different deployments, one that runs locally and the other that runs in the cloud.

**6.1. Microservices**

In total, we have implemented six microservices – Review Microservice, Games Microservice, User Microservice, Steam Microservice, API Gateway and Auth0.

**6.1.1.** **Review**

With this microservice the user can request reviews based on its id, set a review as helpful, get reviews of a specific game in a particular language and many other capabilities.

**6.1.2. Games**

This microservice is responsible for operations like, searching for a game by its name, get all the reviews of a specific game by giving its id and it is even possible to add new games on the list.

**6.1.3. User**

This microservice is responsible for creating and deleting users, getting information about a specific user and creating reviews about a specific game.

**6.1.4. Steam**

Its function is to simulate steam’s own homepage, where anyone (doesn’t need to be authenticated) can get the top ten most recommended games by the users as well as the most active users.

**6.1.5. API Gateway**

This microservice is very important because it serves as an intermediary between the client and the other microservices. Is responsible for redirecting the client’s requests to the microservice that is responsible for that request, as well as showing the response given by the respective microservice. This microservice is also responsible for verifying the user authorization.

**6.1.6. Auth0**

This microservice is used to block some operations, this way we have functionalities that can only be accessed by authenticated users, for example, create a new review, others for admins like create and delete games from the database, and others that the user doesn’t need to be authenticated like getting the top ten most recommended games, thus ending up having three types of users.

**6.2. Protobuf**

We have created four protobufs files (games.proto, reviews.proto, steam.proto and users.proto) and they are the basis for our services to communicate between each other. In each protobuf file, we define the format of the request, the content of the responses and a service with the functions that this microservice supports.

In order to generate Python files from the protobufs that contain all the functions defined to interact with our API, we used a bash script to compile them and copy each one to the microservices that use them.

**6.3. Docker**

We’ve created a shell script that builds the Docker Network, the Docker images and the containers as well in order to automate the process, so we don’t need to run every single command.

**6.3.1. Docker images**

We created five docker images, one for each service, except for Auth0. The images are created from each Dockerfile – each file creates a new folder called *‘service’* and copies the current folder to the new one. After that they run the installation of the requirements – *‘RUN pip install -r requirements.txt’* as well the code to generate Python code from the protobufs.

Finally, the port to access the service from outside is exposed and the entry point is defined to specify the executable that should run when the container starts.

**6.3.2. Docker compose**

To make the microservices deployment and start all the containers, we declared our microservices in a YAML file called *‘docker-compose.yaml’* which defines each microservice container info. In this file, we define the name of the services and under each service is defined its environment, its image and container names, which we set the same, the name of the network that is the same for every service and finally the port that is assigned for each service.

**6.4. Database**

Because some of us already had some experience with MongoDB, we decided to use it as our database to store the data.

**6.4.1. MongoDB**

As said before, there are three microservices that are responsible for the database connections (games, reviews and users) and since they are all independent of each other, the initial goal was for each one to have its own cluster. We quickly realized that just one cluster for each microservice would not be enough since we are using the free subscription of MongoDB so we can only have up to 512 Mb in each cluster we create. To resolve this problem, we’ve created five different clusters – 1 for the *‘games’* microservice, 1 for the *‘users’* microservice and 3 for the *‘reviews’* microservice where the third one is used to insert new reviews.

In order to upload our datasets, we used a Python script for each cluster. Each script connects to each database, opens the CSV file and every 100,000 lines that it reads, with the exception of the *‘reviews’* scripts that reads every 50,000 lines, inserts into the database doing the same process in a loop until it reaches the defined threshold.

To solve the problem of having duplicate users or games we used a Python script to remove them.

**6.5. CI/CD Pipeline**

We created a github action “Python application” that is described in the file *‘.github/workflows/python-app.yml’*.

This action creates a workflow for running a job that builds the application, starts the microservices and then runs the unit tests. The purpose is to run that workflow when a pull request is created for the main branch.

1. **Deployment**

**7.1. Databases**

The solution we implemented assumes that there are already databases running with the information we need. Since we didn’t implement the in the GCP but rather made use of the MongoDB Compass system, there’s only a need to upload the data to this service once. Everything we run on the GCP afterwards fetches that data from this service. As such, things like automatic deployment do not take into consideration the action of populating the databases.

**7.2. Docker Hub**

We also made use of the DockerHub to push container images we could then pull to our GCP cluster. We made a script to push them there, and we only need to update them when we make alterations to the code. This script can be found on the GitHub repository (next section), under CN22\_Group2/scripts/dockerhub\_push\_imgs.sh.

Regarding the deployment of a stable version though, it’s safe to say the project assumes these images are there and so that any automatic deployment does not push any images to Docker Hub. Instead, we make use of yaml files that fetch these images from the Docker Hub.

**7.3. Git Repository**

Another tool we used to facilitate the deployment of our project was a repository on GitHub. We would make changes to the code locally, upload it to GitHub and then update the repository on our GCP machine. The deployment thus requires the cloning of the following Git Repository to our GCP cluster:

<https://github.com/meuriC/CN22_Group2>

This repository includes all the scripts we did to populate the databases, push images to dockerhub, etc. But it also includes things such as the yaml files that fetch the images from Docker Hub.

**7.4. Deployment Process**

Assuming all of the pre-requisites mentioned above are up and running, the deployment process starts by creating a cluster on the GCP and installing the Prometheus+Grafana addon from the GCP marketplace. It’s important to note that the cluster version must be 1.21 gke or below, otherwise the Prometheus+Grafana service is not supported. Afterwards we clone the Git repository into the GCP machine and run the following scripts:

deploy-kubernetes-cluster.sh

prometheus\_grafana.sh

1. **Test and Evaluation techniques and results**

**8.1. Tests**

To confirm that the microservices were acting in accordance with the system requirements defined, we use the pytest module to develop unit tests for each. component of the "games", "users", "reviews" and "steam" microservices. We try to cover not only the positive cases, but also the cases in which invalid inputs were passed to the components. In total were defined 24 unit tests, and the result was:Uma imagem com texto

Descrição gerada automaticamente

**Figure 2** – Unit tests

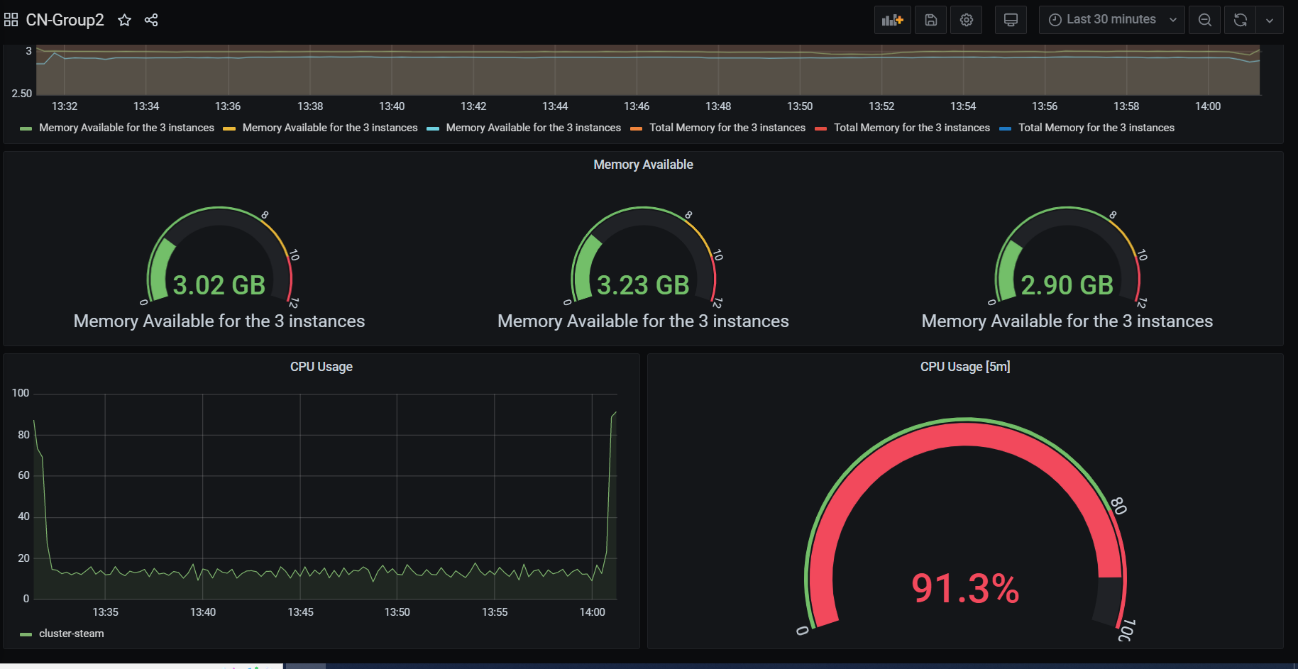
Based on test results, we note that the unit tests defined for the games Microservice aren't enough, since the code coverage obtained was less than 70% (which we considered an acceptable level).

**8.2. Evaluation**

In order to evaluate our system through stress tests, we use:

* Prometheus & Grafana - for system monitoring, which allowed us to get the data directly from the cluster during the execution of orders and display it in Grafana in an intuitive way;
* Locust - scriptable and scalable performance testing tool;

We run the Locust to stress testing from a local machine so as not to influence the data retrieved by Prometheus.

We test some api operations and when the system reached to up 800 users, it was observed that it was getting around 91% CPU usage, however the memory indexes do not change much, as can be seen in the image of grafana dashboard:

**Figure 3** – Stress testing (results on grafana)

1. **Discussion**

Based on the previous section it became obvious that our machine was not powerful enough to handle more than 800 users, so it’s missing some scalability where in our services would be one of the main requirements since a lot of people use the gaming platform.

* 1. **Reliability**

To measure the reliability of our implementation we used **SonarQube**. The scanning was locally, where we called the sonar-scanner to our project folder (cloned-repo where all pulled files go to from our repository) and the scanner checked all folders and files, measuring all the metrics specified within SonarQube.

On the SonarQube the primary indication of reliability is the number of bug issues. The difficulty of individual issues, their number, statuses, types, and severities are used to determine reliability rating and reliability remediation effort.

The **reliability remediation effort** is the effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.

**Table 3** – SonarQube rating

|  |  |
| --- | --- |
| A | 0 Bugs |
| B | at least 1 minor bug |
| C | at least 1 major bug |
| D | at least 1 critical bug |
| E | at least 1 blocker bug |

Uma imagem com texto

Descrição gerada automaticamente

**Figure 4** – SonarQube Ratings

**9.2. Cost Evaluation**

**Table 4** - Expected costs after 1 month

| **Services** | **Cost** |
| --- | --- |
| GKE Standard Node Pool (Kubernetes Engine) | 25.30€ |
| GKE Cluster Management Fee | 0.00€ |
| Anthos | 15.16€ |
| Cloud Build | 0.00€ |
| **Total** | **40.47€** |

These are the expected costs after one month of running our application in the cloud. To obtain the cost of each service, we used Google Cloud Pricing Calculator, obtaining a total value of 40.47€, which is possible to cover in its entirety by the coupons given by the professor.

The Kubernetes engine turned out to be the most expensive where it was defined that our node was running 24 hours a day having a total of 730 hours per month.

GKE Cluster is a zonal cluster that has a total of 730 hours per month where one zonal cluster is free per billing account.

We used Anthos for Prometheus and Grafana, where we need 2 vCPUs running 24 hours a day.

Although Prometheus and Grafana are not yet implemented for this phase, it is something we intend to have implemented in the final delivery and so we decided to include its value in the costs.

Finally, the Cloud Build has a cost of 0.00€, due to the fact that the first 120 minutes of building per day are free.

Worth noting that since we are using an external database service where these provider gives us resources, we didn’t use the GCP storage, and that was the key factor to have a low-cost calculation on our resources.

1. **Conclusion**

The API manly uses Google Cloud Platform (GCP) resources like the kubernetes engine services and MongoDB. At the end, we can see that we made a functional API.

This API fulfills all the capabilities described on the user cases, however comparing with the original reference (Steam) has a long way to go but gave us the insights to understand microservices and all the costs that should be thought first before the implementation since, that will be the main obstacle for cloud projects.

Nonetheless due to the use of external services we reduced the google costs, since this provider gives us resources, we didn’t use the GCP storage, and that was the key factor to have a low-cost calculation on our resources.

* 1. **Future Works**

A CI/CD pipeline with Jenkins will be implemented and also a DNS record for the service to be available on a domain name and not only through the public IP.

* 1. **Contributions**

Diogo Lança – fc53495:

* Microservices implementation and containerization
* API specifications
* Populate and configuration of the database
* Auth0 integration
* Report

Meuri Canhanga – fc42559:

* System architecture design
* Microservices implementation and containerization
* Cloud deployment
* Github actions
* Stress testing
* Report

Miguel Silva – fc58974:

* Microservices implementation and containerization
* API specifications
* Populate and configuration of the database
* Auth0 integration
* Report

Pedro Santos – fc46417:

* Project automatization
* API specifications
* Deployment of metrics server
* Report

Renato Ribeiro – fc57433:

* System architecture
* Microservices implementation and containerization
* API specifications
* Deployment of metrics server
* Reliability analysis with sonarqube
* Report

1. Dataset source: <https://www.kaggle.com/najzeko/steam-reviews-2021> [↑](#footnote-ref-1)