Cloud Computing

Phase 10 - Final Report

Group 7

André Grilo nº 57461
Catarina Moita nº57502
Martim Mourão nº57482
Thomas Marques nº57460
Tomás Dias nº57582

1. Motivation

This project started with our shared interest in video games. Since it was a theme everyone liked it was very easy for us to think in which ways an application could complement it. How to keep track of the games I play? What games do I want to play in the future based on my taste?

2. Dataset Characterization

We used 2 dataset from the platform *Kaggle*, we cleaned some of the data and added filters to reduce the size of the dataset.

2.1 Steam Reviews Dataset 2021

Dataset of 21 million users from more than 300 games. All data is from the Steam Platform. It was obtained from real data on the SteamWorks API. The size of the dataset was 8.17 Gb, but we filtered by only choosing the english reviews and choosing only reviews from the last 2 years. We ended up with 1 Gb of data.

This dataset is characterized by the following columns: Steam app id, App name, Review id, Language of review, Review text, Review creation timestamp, Review latest update timestamp, Whether review recommends the app, Number of "helpful" votes for review, Number of "funny" votes for review, Score based on number of helpful votes, Number of comments for review, Whether review author purchased the app on Steam, Whether review author received the app for free, Number of games review author owns, Author playtime of reviewed app at time of review, Author time last played reviewed app.

2.2 Steam Games complete dataset

We decided to complement the Steam Reviews dataset by adding a second dataset. This dataset allows us to get more information about the reviewed games. The dataset contains over 40,000 games from the Steam Platform. We used the entire dataset with the final size of 81 Mb.

This dataset is characterized by the following columns: Steam app id, Type of package (Ex: app, sub or bundle), Name of a game, Short description of a game, Recent reviews, All reviews, Release date, Developer of a game, Publisher or publishers of a

game, Popular Tags, Details of a game, Supported languages, Number of achievements, Genre(s) of a game, Game description, Description of mature content in a game, Minimum specs for a game, Recommended specs for a game, Price without discount, Price with discount.

3. Use cases

Admin/User	Use Cases
User	User register
	User log in/log out
	Add/Remove review or game from user library
	Check user reviews/games
	Add/Remove games from user wishlist
	Search games/reviews by filters
	Suggest TOP 10 games according to user preferences
	Show the best reviews of a game based on filters
Admin	Add/Remove/Edit a game
	Remove users

4. REST API

4.1. Microservice - User Operation

HTTP Method	URL	Description
POST	/user	User Register
PUT	/user	User edit account
POST	/user/login	User Login
GET	/user/logout	User Logout

4.2. Microservice - Searches

HTTP Method	URL	Description
GET	/searches/games	Search multiple games
GET	/searches/game	Get game
GET	/searches/reviews	Search multiple reviews
GET	/searches/review	Get review

4.3. Microservice - Admin Operations

HTTP Method	URL	Description
POST	/admin/games	Add game
PUT	/admin/games	Update game
DELETE	/admin/games	Delete game from URL
DELETE	/admin/games	Delete user

4.4. Microservice - Library

HTTP Method	URL	Description
GET	/library	List all games in library
POST	/library	Add game to library
DELETE	/library	Remove game from library

4.5. Microservice - Reviews

HTTP Method	URL	Description
POST	/reviews	Create Review
GET	/reviews	Get reviews list
GET	/reviews/{review_id}	Get review
PUT	/reviews/{review_id}	Update review
DELETE	/reviews/{review_id}	Delete review

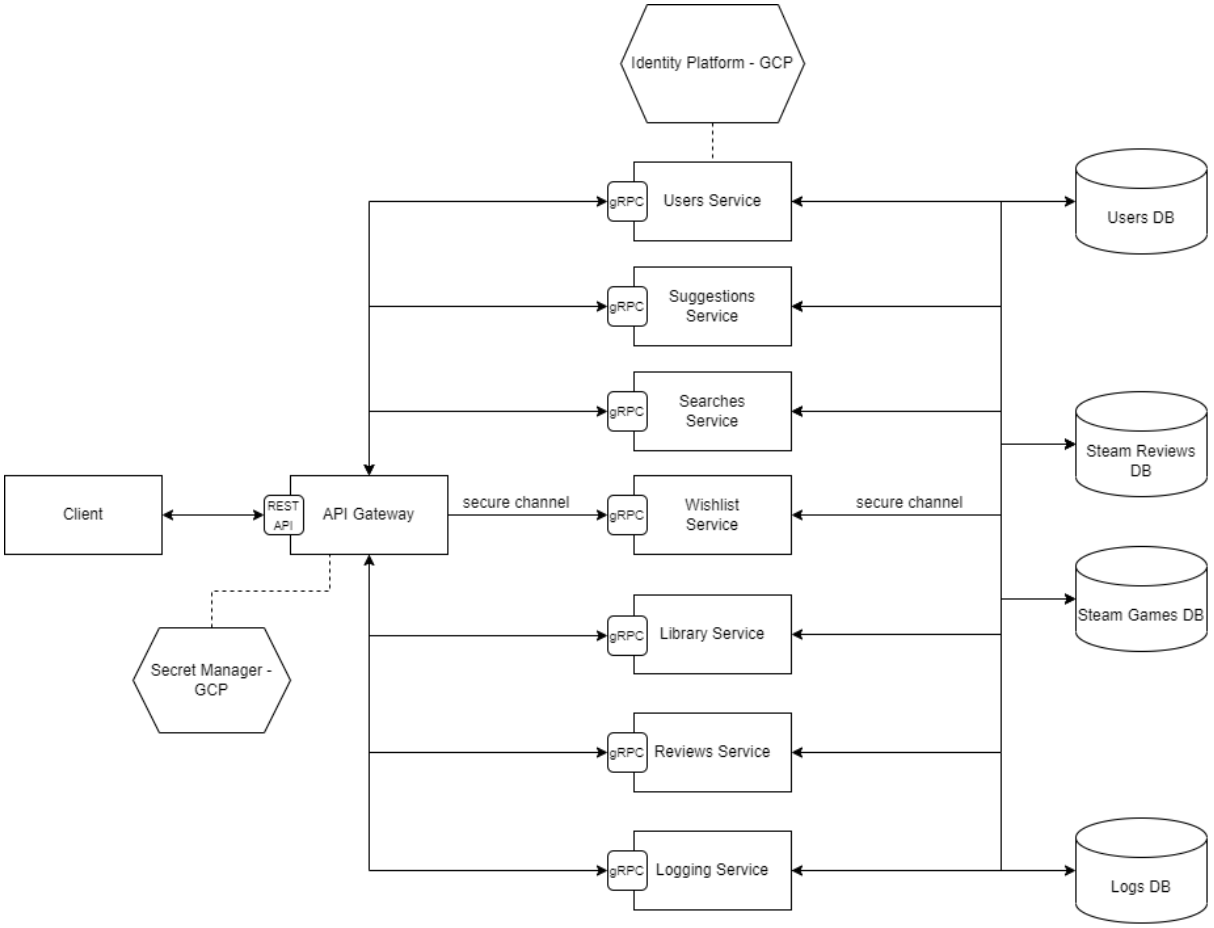
4.6. Microservice - Wishes

HTTP Method	URL	Description
GET	/wishes	List all games in wishlist
POST	/wishes	Add game to wishlist
DELETE	/wishes	Remove game from wishlist

4.7. Microservice - Suggestions

HTTP Method	URL	Description
GET	/suggestions/games	Call personalized top 10 games
GET	/suggestions/reviews	Call personalized top 10 reviews

5. Architecture



5.1 Client

On the client side, the user can access our API through the browser with the url

`http://{IP}.nip.io/`

where the IP it's the NGNIX external address.

5.2 API Gateaway

After the client connects and sends a request, it's the API Gateway that is responsible to receive them as **REST** and send them to the respective service.

5.3 Services

Our application has a total of **8 microservices** deployed in a single **Kubernetes cluster** at the **Google Cloud Platform** where, if necessary, they communicate with each other to fulfill all the requests. Each service is responsible to process their own request and access the data from the database.

5.3 Databases

We have **4 databases** hosted by **MongoDB** that save and manage all the required information for our application to work.

The data from users like email, games, wishlist, between others, are held in the Users DB. The reviews and games datasets are, respectively, in the Reviews DB and Games DB and finally all the audit logs in the Logs DB.

5.4 External Services

Besides using our own services, we made use of services provided by Google Cloud like the **Identity Platform** and the **Secret Manager**. With the Secret Manager service we could store all the sensitive data of our application like keys, passwords and certificates in a secure way. The Identity Platform service provides authentication and custom tokens for our users with **JSON Web Token**.

6. Implementation & Deployment

6.1 MicroServices

Our microservices are implemented in Python. Each microservice has a server and an API.

The API uses different technologies like: **flask**, to create the REST API and to be able to receive requests, **protobufs**, to serialize structured data, **gRPC,** to connect the API to the server (all communications use secure channels).

The Server also uses different technologies such as: **gRPC,** to receive communications from the API, **protobufs**, to serialize structured data, **pymongo,** to communicate with the Database and send queries, **HTTP Server,** to allow data to be fetched by **Prometheus**.

6.2 Docker

We use docker to containerize each Microservice. Each microservice creates 2 containers: the API and the Server. We use the DockerFile to generate the protobufs, to expose the necessary ports and to install libraries like: flask, grpcio-tools, Jinja2, google, protobuf, pymongo and prometheus_client.

Each container will be pushed to the cloud and stored on the **Google Container Registry**.

6.3 Gcloud Commands

We decided to use only gcloud commands to deploy the application on the cloud. We are able to deploy our application with just one script. That script is divided into multiple phases that we will now detail:

A. Variables - We set multiple variables to use on the script, like the project id;
B. Enable Google Cloud Services;
C. Create Cluster;
D. Fetch the certificates for each team from the Secret Manager;
E. Build and Push the docker containers;
F. Create Namespaces;
G. Configure the configmaps;
H. Create secrets for the Database;
I. Deploy the application using .yaml files;
J. Setup the networking for each namespace;
K. Setup the HPA;
L. Setup Prometheus and Grafana;
M. Create Role-based access control on Kubernetes for each Team.

6.4 Database

We decided to use MongoDB to save the dataset because of:

1. Flexibility: MongoDB uses documents that are flexible.
2. Schema-less model: We can control different properties directly from the application.

The Database is deployed inside the Kubernetes Cluster due to budget reasons. If we used Google Cloud Databases our budget would not be sufficient to maintain the application. With this approach we are able to maintain a stable database without increasing our budget.

Complementary was implemented a **Persistent Volume**, allowing us to persist the data as long as the cluster is running.

To write the data to the Persistent Volume we use a **Job** on the Kubernetes. That job is a docker application with a python script that loads the data from the datasets and creates the documents on the MongoDB. That Job only runs once.

6.5 Kubernetes

We use Kubernetes to deploy and maintain our application (docker containers) on the Google Cloud Platform. We create:

A. Deployments

For each container we detail: **namespace**, number of **replicas**, the strategy for **rolling update**, the image location on the **container registry** and finally: **livenessProbe**, **readinessProbe**, **startupProbe**.

B. Services

Each container contains a ClusterIP to expose the necessary ports.

C. ConfigMaps

To keep our application code separated from our configuration like: ports and name services.

D. Namespaces

Each Microservice has its own namespace. We use .yaml files to create each namespace.

E. HPA

We define policies to autoscale our microservices using HPA.

F. Networking

We use services to create External Names to be able to communicate between namespaces. We implemented nginx-ingress to expose an external IP to receive requests from the outside.

It was also implemented some **Network Policies** so that we could control the traffic flow in each pod.

G. Secrets

We defined secrets to access the database.

6.6 Security

We choose to increase the security on our application. Implementing multiple Kubernetes security options and using services from the google cloud.

A. RBAC

Each microservice is divided into namespaces and each microservices has its own team. We implemented Role-based access control to create specific roles for each team.

We use the Secret Manager from the Google Cloud to save the following files:

a. Team Key
b. Team Certificate

The certificates are signed by the Kubernetes using: CertificateSigningRequest. To the following usage: digital signature, key encipherment, server auth and client auth.

So each team has its own key and certificate that they can fetch from the secret manager to be able to perform actions on the kubernetes. It's possible to implement teams on the google cloud and then associate each secret to a specific team. We tried to implement teams but we need to be a company to implement that. So we just created the secrets without teams on the cloud level using the secret manager.

We defined Roles to all resources available on the kubernetes. We were able to specify different verbs to different resources like: "update pods", "get configmaps". And finally we bind each role to the user (team). With this approach we are able to isolate each team in its own namespace and with specific restrictions inside the namespace.

So some example would be:

a. Can a team create a pod outside its namespace: NO
b. Can a team list all secrets: NO
c. Can a team update pods inside the namespace: YES

B. Protection DoS

Our first idea was to force each user to get a JSON web token from the UserManagementService so that we could control the number of requests made by each user. We started implementing this feature but we noticed that the UserManagementService was overloaded by so many requests.

After some research we decided to change our approach completely and use the Nginx Ingress to limit the number of requests, with just a few lines of configuration we were able to limit the requests by limiting the number of rpm allowed on the ingress.

C. Secure Channels

We decided to implement secure channels on all communications. It was a real challenge to achieve this goal! Once again we used the secret manager to save all keys and certificates. We use **CFSSL**, a TLS PKI tool to generate and sign all the files.

D. Audit Logging

To keep records of all the operations performed in the pods, who performed them, when and its result, we built a database with documents containing all the events of the application. This logs will be useful to monitor and observe any abnormal behavior and detect possible breaches,

E. Network Policies

While creating Network Policies, we are able to isolate pods and filter traffic within the cluster. We identify which ingress traffic is allowed to the nginx pod.

F. Broken Access Control and Inputs injection Checks

Each request made to our API is validated based on the JWT (JSON Web Token) authorization header issued by Google. This ensures that nobody would be able to tamper with the data inside the token unless they have access to the google private key.

For requests to the database, we have configured parameterized queries which are going to ensure that user input isn't going to be interpreted as a command by our server.

6.7 Authentication

A. Identity Platform

We are using Identity Platform to manage the user's authentication. Each user needs to give an email and password. We send that information to the cloud to perform the authentication, after we receive the OK from the google cloud we ask google cloud to give us an JSON web token signed by them. The user then needs to use that JSON web token to access the library and wishlist and also to perform admin operations.

6.8 Prometheus & Grafana

We implemented prometheus on our deployment to be able to monitor the following metrics:

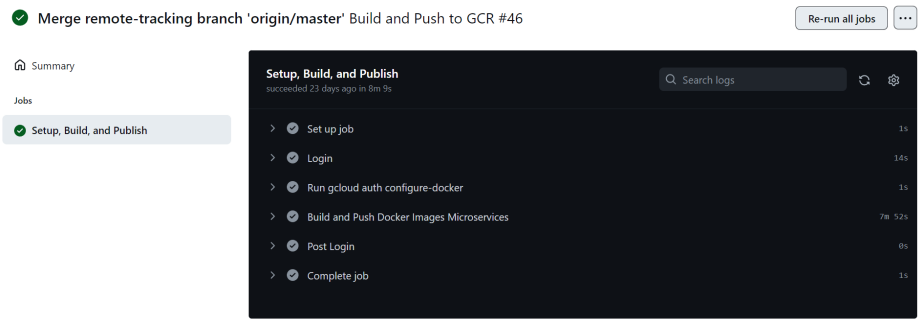
a. CPU load;
b. Memory Usage;
c. Number of requests per microservices;
d. Time each request takes.

The configuration was simple.

We decided to not use the prometheus UI to monitor the metrics and implement Grafana to get a better UI. The grafana implementation was also very simple and we were able to create a dashboard with all the metrics. We used Grafana to monitor our results during stress testing.

6.9 CI

We implemented a workflow on GitHub so that every time a push happens the flow would build and push the containers to the Container Registry. We used that flow for several days during deployment but then we received a warning from the google cloud that we were sending too much data. We want to change the workflow to send the containers only when a release happens and not when a push happens.



7. Test and Evaluation techniques and results

7.1 Tests

We created a python script to stress test the application. We performed 3 tests:

Test ID	Number of Active Users (simultaneously)	New Users Entry Frequency (seconds)	Requests Frequency (seconds)	Microservices to be tested	Application Total Requests
1	50	0.2	0.2	6	300
2	1 000	0.2	0.2	6	6 000
3	2 500	0.1	0.1	6	15 000

7.2 Results

We used grafana to monitor the application, the results are the following:

Test 1 – With a reliability of 100% the CPU usage was low with only 20% of usage, the time per request was 150 milliseconds.

Test 2 – With a reliability of 100% the CPU usage was low with only 45% of usage, the time per request was 220 milliseconds.

The UserManagement Service was the service more overloaded with CPU going to 100%. This behavior is happening because most of the services depend on the usermanagement to perform authentication checks. We should start this service with more replicas.

Test 3 – With a reliability of 98.76% the CPU usage was low with only 97% of usage, the time per request was 300 milliseconds.

7.3 Discussion Results

Regarding GitHub's continuous integration, it was very interesting and useful to have an automation that put our containers at the Cloud. However, in the future we would like to change the script to just perform this workflow when a release happens and not whenever there is a push.

In security, we have achieved very concrete goals and we are very satisfied with the results. We were able to prevent several attacks and create security policies in the Kubernetes. It was very interesting when we were doing the stress testing that we had to temporarily change our security policies because our stress testing was being identified as a DoS attack.

Regarding stress testing, we use Grafana to evaluate our performance. In the first two tests the results and our implementation were as expected. In relation to the last test, we had a reliability of 98.76% that we considered acceptable. Our pods have never crashed and always managed to recover even in periods of high load.

8. Costs

Service	Costs
Kubernetes	**50$**
	50$
	14,75$
	27$
	Total: 141,74$

We used multiple accounts. All budgets within limits.

9. Group organization

System and API Architect (CEO) - Martim

Networking - Catarina

Security Specialist - Thomas

DevOps Officer - Tomas

Data Scientist - André

9.1 Microservices

Martim - Admin Operations & User Management
Catarina - Wishlist & Library
Tomás - Reviews
Thomas - Suggestions
André - Searches

9.2 Google Cloud

(1) deploy the containers to a kubernetes cluster on the cloud (ALL)
(2) deploy the databases to Kubernetes volumes - (Catarina)
(3) use an HTTP(s) ingress to connect each external service to the cloud load balancer (Tomás)
(4) configure kubernetes policy for scalability (HPA and if required VPA and Cluster) (Thomas)
(5) expose only the services that really need to be accessed from the outside (ALL)
(6) use a managed authentication service and implement the planned authorization policies (Martim)
(7) configure resource utilization through requests and limits (try to be as cost-effective as possible) (ALL)
(8) setup metrics per pod for monitoring with Prometheus (Martim)
(9) setup your own probes for liveness, readiness and start-up (André)
(10) implement rolling updates and rollback (Catarina)

9.3 Security

(1) Check the need for config maps (Catarina/Martim)
(2) Actions with Git CA (Martim)
(3) Managing multiple namespaces. Create different RBAC policies for each namespace. (ALL)
(4) Check if the current configuration of the network architecture isolates critical points from the public Internet.
(5) Secure channels in all communications. (Martim)
(6) Creation of logs (Audit Logging). (Tomás)
(7) Creation of NetworkPolicy. (Catarina)
(8) Protection for DoS, each request needs an access token, to monitor maximum requests per user. (Martim/Catarina)
(9) Check for attack Broken Object Level Authorization (when sensitive fields within an object are incorrectly exposed), test for database inputs injection. (Thomas)

10. Conclusions

As it is a considerably long project, it is worth noting the increase in difficulty encountered as each phase progressed. This forced us as a group to adopt clear working methods as well as a proactive stance in order to complete the different tasks proposed.

We should also highlight the enormous amount and diversity of concepts, methodologies and technologies involved in Cloud Computing, in which we had the possibility of deepening our skills.

About the work done, to highlight the deployment phase that was quite challenging given the robustness of the configurations needed for the correct functioning between the services, arising with some frequency problems related to communication between them. The implementation of the monitoring services and the adjustment of the respective settings also required some extra attention. The use of Google Cloud Platform added other setbacks such as the management of the different APIs, the budget made available for this work and the increased complexity of monitoring and debugging the project. The design phase of the proposed improvements, namely in the field of system security, proved to be equally laborious, highlighting the implementation of RBAC policies.

In general, we can consider that all the essential goals of each of the project phases were achieved.

Thank you!