Final Report

Computação em Nuvem

2020/2021

Grupo 4:
Daniel Ferrão 49606
Jaime Mendes 55919
Luís Gonçalves 55742

Índice

Motivation 3
DATASETS 3
Use Cases / API 5
Application and technical architecture 7
Application Architecture 7
Description 7
Technical architecture 8
Implementation 8
Microservices 8
Database 8
Kubernetes 9
Spark 9
Evaluation and Validation 9
Cost analysis 12
Discussion and Conclusions 12

Motivation

With the exponential growth of users on the various online streaming platforms, the number of films produced has also been increasing in response to this growing adhesion. Since movies are a big part of the current entertainment industry, there is a plethora of information available on the internet regarding this topic. However, the great majority of that information is incomplete, and covers only some aspects of movies, so it is necessary to integrate various datasets to answer specific questions that require data from various sources. In the present report, we will describe our approach to data integration and processing of two datasets concerning movies.

Taking into account the dataset, the services that seemed to be more interesting considering the type of data that we had were:

- In a given year, what are the films that are found on a specific streaming platform?
- What is the rating of a film in a given year?
- What is the budget for a film in a given year?
- What is the revenue for a film in a given year?
- How many movies are there on streaming platforms?
- What is the average length of a movie?

DATASETS

The group project will consist essentially of three datasets extracted from two different projects in the Kaggle platform, where one is the "movies_metadata.csv (hereby referred to as 'meta')[1] other is the "ratings.csv" (hereby referred to as ratings)[1] and the other one is "MoviesOnStreamingPlatforms_updated.csv (hereby referred to as 'stream')[2].

The meta dataframe contains information from 45.466 thousand films and has a size of approximately 900MB. This dataset includes films released on or before July 2017, 'cast', 'crew', 'keywords', 'budget', 'revenue', 'posters', 'release date', 'languages', 'production companies', 'production countries', 'genres', 'overview', 'popularity', 'runtime', 'status', 'tagline', 'TMDb votes' and 'average votes'.

The stream dataframe contains information from 16.744 films on streaming platforms and has a size of approximately 2MB. This dataset includes 'Title', 'Year', 'Age', 'Rotten Tomatoes', streaming platforms (Netflix, Hulu, Prime Video and Disney+), 'Type', 'Directors', 'Genres', 'Countries', 'Languages', 'Runtime' and the 'IMDb' on scale of 1 to 10.

The ratings dataframe has files containing 26 million ratings from 270,000 users for allhas a size of 45,000 movies. Ratings are on a scale of 1-5 and have been obtained from the official GroupLens website. The dataframe approximately 700MB and includes "userId", "movieId", "rating" and "timeStamp".

In each dataset the group decided to select the most relevant information for the project. In the meta dataframe only a few columns were selected, such as the 'id', 'title', 'budget', 'revenue', 'release date' and "runtime". In the stream dataframe we selected the 'Title', 'Year', 'IMDb', all streaming platforms (Netflix, Hulu, Prime Video and Disney+), 'Directors', 'Genres', 'Countries', 'Language' and 'Runtime'. In the ratings dataframe only the "rating" and "movieId" were selected.

In relation to data selection, there were three main reasons to eliminate the columns that we did not use: a high percentage of incomplete or missing data, unusable information, and repeated data between the two datasets.

For the incomplete and missing data, we eliminated the 'Rotten Tomatoes' and 'Age' columns from the stream dataframe. From the meta dataframe we eliminated the 'belongs_to_collection'.

Regarding the unusable information, we considered as part of this category data such as URL values, data that was repeated for the whole column, or simply information that cannot be used due to its structure. From de stream dataframe, we eliminated the columns 'ID' and 'Type' and from the meta dataframe we eliminated the columns 'adult', 'homepage', 'imdb_id', 'poster_path', 'status', 'video', 'overview', 'vote_average', 'vote_count', and 'original_title'. The tables "Links", "Keywords" and "Credits" were discarded for the same reason. From the ratings dataframe we eliminated the "userId" and "timeStamp."

Finally, for the repeated information we eliminated the "genres", 'spoken_languages', 'original_language', 'production_countries' and 'runtime' from the meta dataframe, since this information already exists on the stream dataframe. Despite the "Ratings" table containing different information about the movies' ratings when compared to the IMDb rating in the stream dataframe, they both cover the same information, and the latter requires less preprocessing. For that reason, we decided to drop the "Ratings" table.

Regarding possible problems about data quality in the columns we decided to keep, the group only found one column ('production_companies') containing a string representation of a JSON format object in the meta dataframe. In the stream dataset, the columns 'Genres', 'Directors', 'Languages', and 'Country' have different values grouped as a list, which is formatted as a string.

Year	IMDb	Netflix	Hulu	Prime Vid	Disney+	Directors	Genres	Country	Language	Runtime
2010	8.8	1	0	0	0	Christopher Nolan	Action,Adventure,Sci-Fi,Thriller	United States,United Kingdom	English,Japanese,French	148
1999	8.7	1	0	0	0	Lana Wachowski,Lilly Wachowski	Action,Sci-Fi	United States	English	136
2018	8.5	1	0	0	0	Anthony Russo,Joe Russo	Action,Adventure,Sci-Fi	United States	English	149
1985	8.5	1	0	0	0	Robert Zemeckis	Adventure,Comedy,Sci-Fi	United States	English	116
1966	8.8	1	0	1	0	Sergio Leone	Western	Italy,Spain,West Germany	Italian	161

id	budget	original_language	year	revenue	runtime	title
862	30000000	en	1995	3,74E+08	81.0	Toy Story
8844	65000000	en	1995	2,63E+08	104.0	Jumanji
15602	0	en	1995	0	101.0	Grumpier Old Men
31357	16000000	en	1995	81452156	127.0	Waiting to Exhale
11862	0	en	1995	76578911	106.0	Father of the Bride Part II

movieId	rating
110	1.0
147	4.5
858	5.0
1221	5.0
1246	5.0

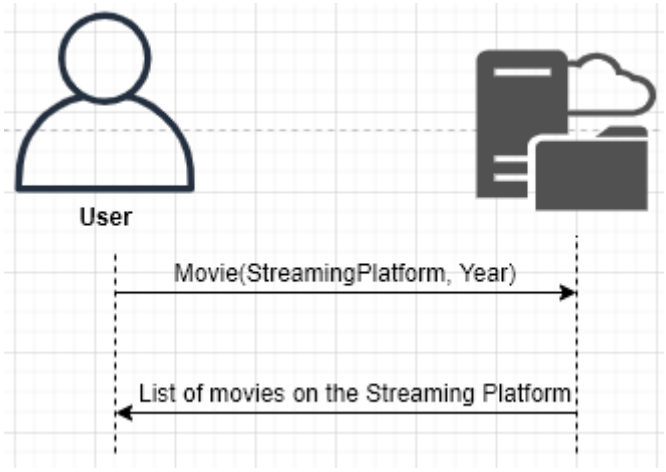
[1] https://www.kaggle.com/rounakbanik/the-movies-dataset

[2] https://www.kaggle.com/ruchi798/movies-on-netflix-prime-video-hulu-and-disney

Use Cases / API

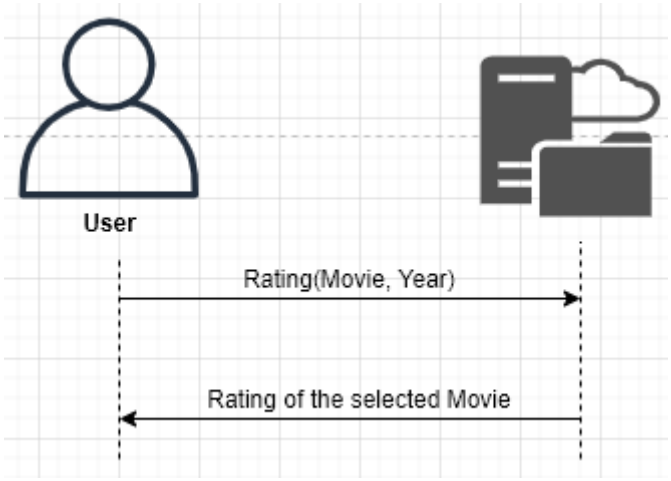
First use case: a user requests the list of existing movies on one streaming platform.

- **GET /moviesOnPlatforms**



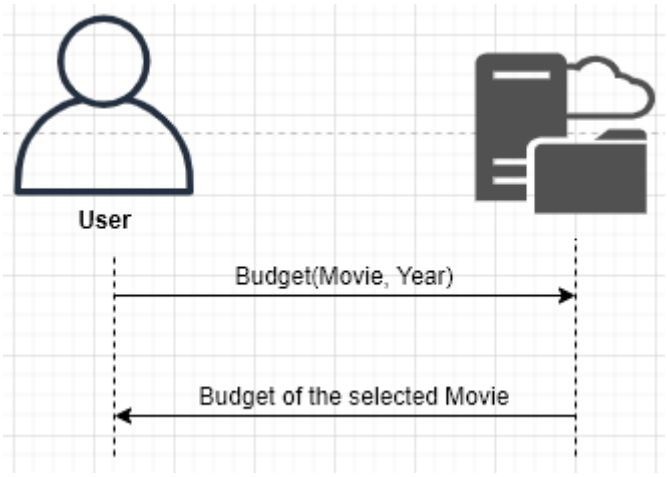
Second use case: a user requests the rating of one specific movie on one specific year.

- **GET /movieRating**



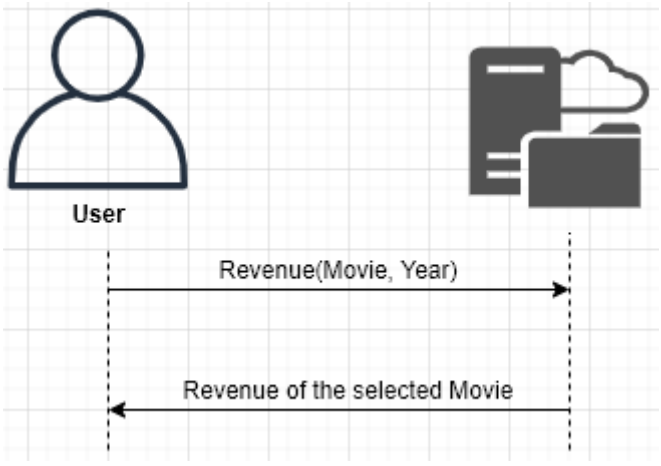
Third use case: a user requests the rating of one specific movie on one specific year.

- **GET /movieBudget**



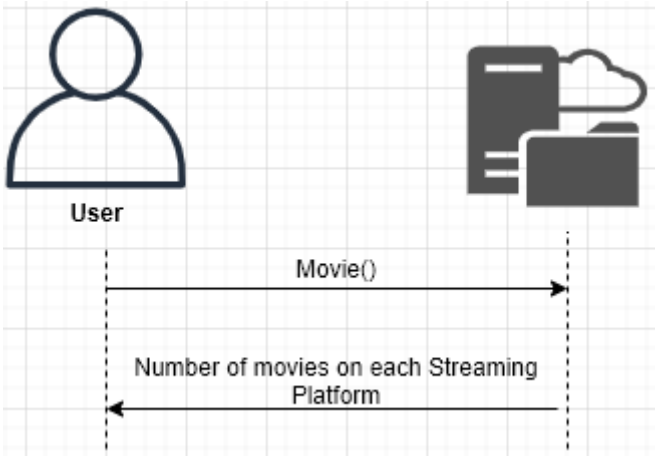
Fourth use case: a user requests the rating of one specific movie on one specific year.

- **GET /movieRevenue**



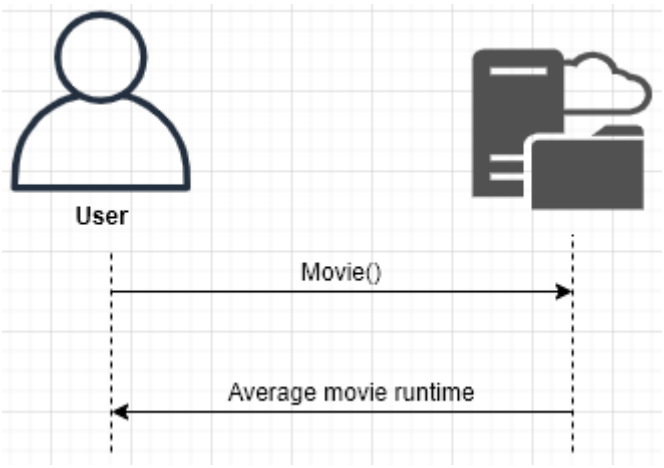
Fifth use case: a user requests the rating of one specific movie on one specific year.

- **GET /numberOfMovies**



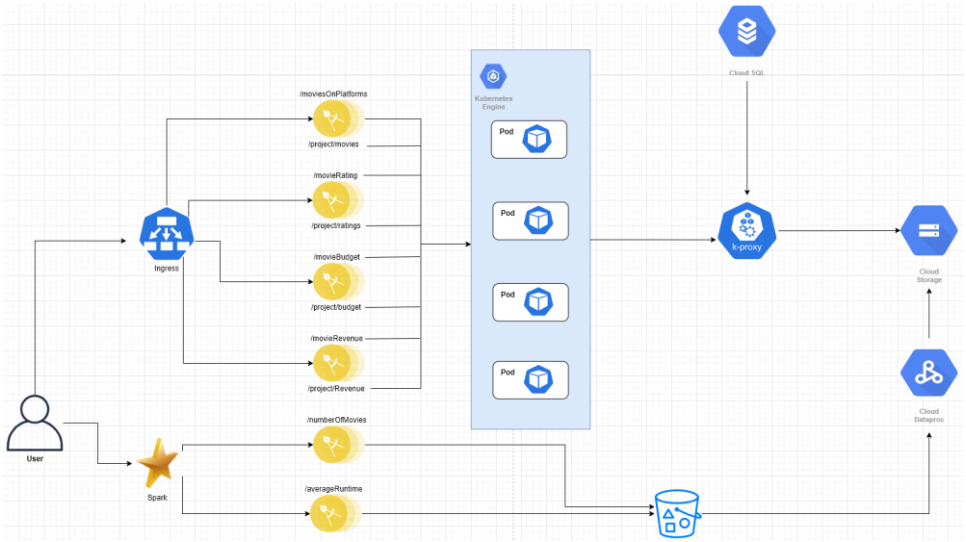
Sixth use case: a user requests the rating of one specific movie on one specific year.

- **GET /averageRuntime**



Application and technical architecture

Application Architecture



Description

The user makes a request for a specific resource or list of resources which is received by the Ingress. Next, the Ingress is going to redirect the request to the correct target group of the microservice and then the request is going to be fulfilled by the proxy that makes the connection between the Cloud SQL and the Kubernetes cluster, accessing a database of the requested resource (meta, stream, ratings). To make use of the Spark the user makes again a request for a specific resource and then the request is going to be fulfilled by the Cloud Dataproc that will get the information from the created bucket on the cloud that contains the csv files with the necessary information.

Technical architecture

Configuration Management - We used the cloud service provider Google Cloud and the services that we learned and used in class so that the configuration management is the simplest possible.

Response Time - We will use SQL Cloud database which performed the best given our datasets, using the PostgreSQL. We will also use different tables for each resource type (meta, stream, rating) which is going to decrease the response time.

Readability - The output should be easily read by the user. To make this possible, we are going to format the output using a table, so that the information is well organized and readable.

Usability - The resource requests will be extremely easy to make and are going to be executed using the command line.

Implementation

Microservices

We divided development in four microservices: moviesOnPlatform, movieRating, movieBudget and movieRevenue which implement the use cases 1, 2, 3 and 4 respectively. All the microservices where written with python.

This microservices were implemented on the Google Cloud Platform (GCP), where the group used the free credits provided by course. To enjoy the microsevices, this were implemented on Kubernetes, more specific, on a kubernetes cluster seeking information from our database.

Database

The group started by using the MariaDB database but later on the group migrated to the PostgreSQL database for being more friendly user and having a better graphical interface. We created two databases because they are files with a reasonably large size and further on that will take a long period time to create and populate the tables. One database contains the information of the movies on streaming platforms and the other the information of the metadata end ratings.

The group had two strategies for the creation of the database:

The first one, that takes most time to be created, it consists of dumping the database that was already locally on the machine, then inserting that dump into the bucket within the GCP and create the database with the dump that is in the bucket.

The second one, that takes the less time to be created, consisted, after creating the SQL instance on GCP, adding the public IP address on the Connections option after that, on the Postgres GUI, pgadmin4, we create a new server that makes the connection between our local pc and the SQL instance, using the IP of that instance.

In order to populate the database, using the three csv files, we created a small shall script named **create_and_populate.sh**. The shell script consists of running the created python files that makes the connection with the Postgres and populates the tables with the attributes of the csv file.

Kubernetes

In this moment of the development process, we had the microservices working individually on our machines, the next step was to start building the means to run the whole application in a kubernetes cluster on Google Kubernetes Engine (GKE). In order to do so, we started by moving the microservices into docker containers.

Next step was deploying these services on Pods in a Kubernetes Cluster, the challenge here was providing appropriate credentials for the Pods to access the database as well as abstracting the id of the GCP project. To access the database the group created a proxy for each microservice on a YAML file, the proxy makes use a secret that the group also created that adds two roles (Cloud SQL client and Editor) and at the end the created secret key with the GCP credentials JSON file. To access the credentials for our database this were put on the ingress file.

With the deployments of the microservices setup, the only thing left to be done is directing outside requests to the correct microservice. To achieve this we used, again, the Ingress file with the correct forwarding rules.

The script **create_services.sh** automates the process of creating the GKE cluster, pushing the docker images to GCR, deploying the services and running the application.

The script **delete_services.sh** automates the process of deleting all the things created by the script **create_services.sh**.

Spark

For the Big Data side of things, we first built the two queries in PySpark on our local machines, and once that was done, we tried to figure out the best way to integrate these new features onto the project application.

We chose to create a dedicated Dataproc cluster to run the PySpark jobs by accessing the various dataset files from a Cloud Storage bucket.

Each service has the capability of sending the respective PySpark job to the Dataproc cluster to process, the server waits for the job completion and returns the complete output of the job in question.

The script **spark_services.sh** automates the process of submitting the spark job and obtaining the result.

Evaluation and Validation

Since the microservices created for the scope of the project contain only one main function where our query is performed, it will not make sense to perform unit tests, but only to perform acceptance tests.

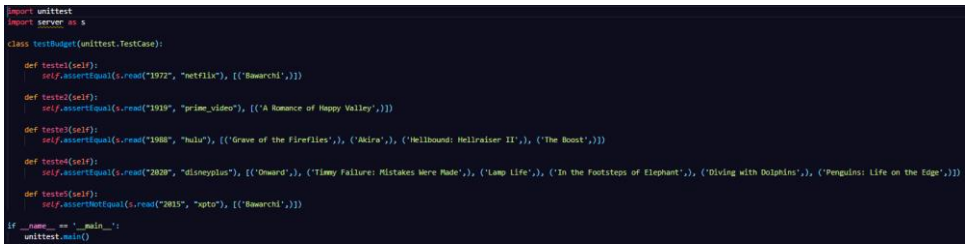
For the three microservices, we introduce the two required inputs with a specific choice and test is the output is the correct answer.

For the budget testing we introduce one movie and one year and verify if the expected output is correct.

```python
import unittest
import server as s

class testBudget(unittest.TestCase):

    def teste1(self):
        self.assertEqual(s.read('Jumanji', '1995'), "65000000")

    def teste2(self):
        self.assertEqual(s.read('Grumpier Old Men', '1995'), "0")

if __name__ == '__main__':
    unittest.main()
```

```
Querie correu bem
.Querie correu bem
.
----------------------------------------------------------------------
Ran 2 tests in 0.086s

OK
```

For the movie testing it was made an acceptance test for each platform introducing one streaming platform and one year whit the purpose of verifying if the list of movies for each platform is correct.



```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.336s

OK
```

For the rating testing we introduce one movie and verify if the expected output is correct.

```python
import unittest
import server as s

class testRating(unittest.TestCase):

    def teste1(self):
        print(s.read('Jumanji'))
        self.assertEqual(s.read('Jumanji'), "3.7601626016260163") #Local

    def teste2(self):
        print(s.read('Heat'))
        self.assertEqual(s.read('Heat'), "3.8700331125827815") #Local

if __name__ == '__main__':
    unittest.main()
```

```
Querie correu bem
.Querie correu bem
.
----------------------------------------------------------------------
Ran 2 tests in 2.577s

OK
```

For the revenue testing we introduce one movie and one year and verify if the expected output is correct.

```python
import unittest
import server as s

class testRevenue(unittest.TestCase):

    def teste1(self):
        self.assertEqual(s.read('Waiting to Exhale', '1995'), "81452156") #Local

    def teste2(self):
        self.assertEqual(s.read('Tom and Huck', '1995'), "0") #Local

if __name__ == '__main__':
    unittest.main()
```

```
Querie correu bem
.Querie correu bem
.
----------------------------------------------------------------------
Ran 2 tests in 0.085s

OK
```

All the acceptance testing was made locally, because the database is the same on the Cloud instance on GCP.

Cost analysis

The cost analysis of the project has to have in consideration the multiple services used. The services used are Compute Engine (with Dataproc), Kubernetes Engine, Cloud Storage and Cloud SQL for Postgres. The list and image below show the cost associated with this services:

- Compute Engine
    - Dataproc
        - Quantity: 1
        - Type: n1-standard-1 (lowest setting)
        - Use Period: 24 hours * 30 days
        - Price: **24,27** USD($)

- Google Kubernetes Engine
    - Quantity: 3
    - Type: n1-standard-1 (lowest setting)
    - Use Period: 24 hours * 30 days * 3 nodes
    - Price: **72,82** USD($)

- Cloud Storage
    - Type: cn2021-projeto
    - Total amount of storage: 3GB
    - Use Period: 24 hours * 30 days * 3 months
    - Price: **0.24** USD($)

- Cloud SQL
    - Quantity: 1
    - Type: db-f1-micro (lowest setting)
    - Use Period: 24 hours * 30 days * 3 months
    - Price: **33.72** USD($)

For the entire project the total amount spent were **131.05** USD($).

Discussion and Conclusions

The first challenge that the group faced was the lack of communication, that led in a group separation, remaining, for the first part of the project, only two participants instead of five. This separation generated a much higher workload than expected resulting in some gaps in the first delivery, which we tried to correct in the final delivery.

For the second part of the project, there was the integration of a new member in the group with the intent of lightening the workload and being able to help in the realization of the project, this integration became relevant because it helped to eliminate some gaps in the first phase as helped to carry out the second part of the project.

The biggest challenges in the development of the application were to understand how to solve the configuration problems resulting from the use of the cloud, and the existing documentation was often unable to help in solving the problems that appeared.

Before using our initial datasets, we needed to clean it before implementing anything, but we were able to implement the desired functionality of the API, even though some improvements are needed such as: looking for a different solution from ingress for the distribution of backends in order to avoid waiting for the time that ingress takes to be ready and add some security measures.

Lastly the management of the credits provided by the course was not the best since most of the project was carried out at GCP instead of being initially done locally, thus saving many resources. With this mismanagement, two of the 50 credits provided were spent, but one had already been used on another course, as well as part of the 250 credits offered by GCP.