Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa

Academic Year 2022/2023

Final Report - Project

In the course of "Computação em Núvem"

Work developed by:

Bruno Cotrim, nº54406

Christopher Anaya, nº60566

Inês Martins, nº54462

Miguel Carvalho, nº54399

INDEX

I – Motivation & Dataset Characterization

• Motivation

The driving force behind the project developed in this course was the aspiration to create a cloud-based application dedicated to movie and actor research. Our ultimate goal was to develop a platform that, with the addition of a frontend in future enhancements, could rival popular mainstream applications like Netflix. To achieve this, we embarked on a pursuit for a dataset that would possess distinctive attributes, including comprehensive information on movies, episodes, and actors, and that would also contain a vast number of records. For this purpose, we opted to use the IMDB public dataset as we consider it to be the most suitable and reliable dataset source for our project. This is due to its unique characteristics, such as:

- **Comprehensive Data:** The dataset offers a comprehensive collection of movie-related information, this includes details about movies, TV shows, actors, directors, ratings, reviews, genres, etc., enabling us many options to work with the data and allowing us to generate high quality data analysis.
- **Dataset Size:** The dataset is considerably large as it contains 6.7 GB of data with thousands of records, something that will allow us to perform comprehensive analysis, uncover patterns, and derive meaningful insights regarding movies, episodes, and actors.
- **Real Word Data:** IMDB is one of the most popular and widely used platforms for movie-related information and ratings, and by using this dataset we are analyzing real-world data that reflects user preferences, trends, and opinions.
- **Authenticity:** The selected dataset is also publicly available and can be easily accessed through the IMDB official website, something that confirms the authenticity of the chosen data.

• Dataset Characterization

Hereby follows the dataset characterization of the chosen IMDB public dataset:

- **Topic of Dataset:** Information about movies and TV shows (IMDB public dataset).
- **Summary of Dataset:** The chosen dataset is comprised of seven folders, each one representing different information:
  - **title.akas.tsv.gz:** This file contains alternate titles and region-specific title information for each movie or TV show in the dataset.
  - **title.basics.tsv.gz:** Contains basic information about each title, such as the title type (movie, TV show, etc.), the primary title, the original title, the release year, the runtime, the genres, and the average rating.
  - **title.crew.tsv.gz:** This file contains information about the directors and writers of each title in the dataset.
  - **title.episode.tsv.gz:** This file contains information about TV show episodes, including the season number, episode number, and title.
  - **title.principals.tsv.gz:** This file contains information about the cast and crew members who worked on each title.
  - **title.ratings.tsv.gz:** This file contains the IMDb rating and the number of votes for each title.
  - **name.basics.tsv.gz:** This file contains information about people involved in the production of movies and TV shows, such as actors, directors, and writers. Each record in the file represents a single person and includes their name, birth year, death year (if applicable), and a list of their known professions.

- **Date of Creation & Last Updates**: The IMDB dataset's date of creation is not publicly available, as the dataset has been compiled and updated over a period of many years. However, the dataset has been publicly available for research and non-commercial use since 2016, when IMDB released the dataset as part of their licensing agreement with Amazon Web Services. As stated in the official IMDB dataset webpage (https://www.imdb.com/interfaces/), there exists a daily refreshing of the datasets.
- **File Type:** The dataset is provided in a set of text files in TSV (Tab-Separated Values) format, which is an accessible and easy format to use in multiple technologies.
- **Size of Dataset:** The compressed size of the dataset is approximately 1.3 GB, and the uncompressed size is approximately 6.7 GB.
- **Link to Files:** https://datasets.imdbws.com/

II – Use Cases & API

- **Use Cases**

We hereby present the use case diagram followed by the list of use cases we implemented throughout the development of our application:

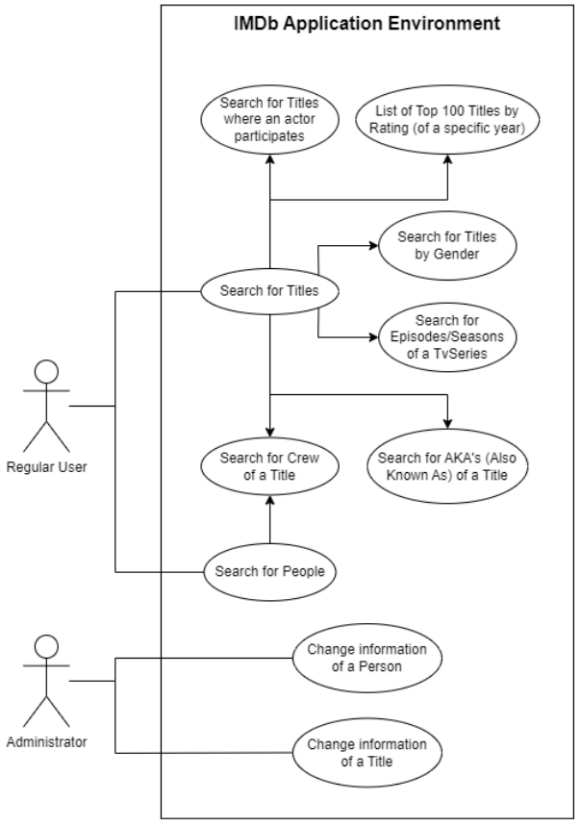

Figure 1 - Use Case Diagram

- Use Case List:
  - **Use case 1** - List of Films where a specific actor participates in;
  - **Use case 2** - List of the Top 100 films with the highest rating (in a chosen year);
  - **Use case 3** - List of episodes & seasons of a specific TV series;
  - **Use case 4** - List of Titles (Films/Series) by genre;

        - **Use case 5** - List of crew elements (people that participate in a Title) belonging to a specific Film/Series;
        - **Use case 6** - List of AKAs (Also Known As) of a specific Film/Series.
    - Use Cases (for Admin Users):
        - **Use case 1** - Change general information about a specific Title (Films/Series);
        - **Use case 2** - Change general information about a specific Person (that participates in Titles).

- **API**

We hereby present the list of APIs based on the use case list previously mentioned and our project requirements:

- **Titles Microservice:**
    - GET /findTitleAkas: Retrieves alternative titles (AKA's) for a given title;
    - GET /findByName: Searches for titles based on their name;
    - GET /titles/{page}: Retrieves a paginated list of titles;
    - PUT /titles: (Admin User Only) Updates information about a title;
    - GET /byGenre: Retrieves titles based on a specific genre;
    - GET /topHundredYear: Retrieves the top hundred titles for a given year.
- **Principals Microservice:**
    - GET /titleByActor: Retrieves titles in which a specific actor has appeared;
    - GET /charactersByActorTitle: Retrieves characters played by an actor in a specific title;
    - GET /actorsOfTitle: Retrieves actors who have appeared in a specific title.
- **People Microservice:**
    - GET /crewByTitle: Retrieves the crew members associated with a specific title;
    - GET /findByBirthYear: Searches for people based on their birth year;
    - GET /findByPrimaryName: Searches for people based on their primary name;
    - PUT /person: (Admin User Only) Updates information about a specific person.
- **Episode Microservice:**
    - GET /seriesEpisodes: Retrieves episodes of a chosen series;
    - GET /episodeNumber: Retrieves the episode number of a season given its ID;
    - GET /seasonNumber: Retrieves the season number of a series given its ID.

III – Architecture & Implementation

We hereby describe the defined architecture, its implementation and all progression changes that were made during the development of our project. Initially, in the first stage of the project, as our view and knowledge about cloud-related technologies was limited, we designed a simple architecture that was based on the usage of containers and their management by Kubernetes. As new insights into cloud computing were taught and researched, we understood the simplicity and the need to make some significant improvements to the project as we will discuss later in this section.

- **First Stage**

We now disclose the first stage architecture/implementation of our project via the following image and description:

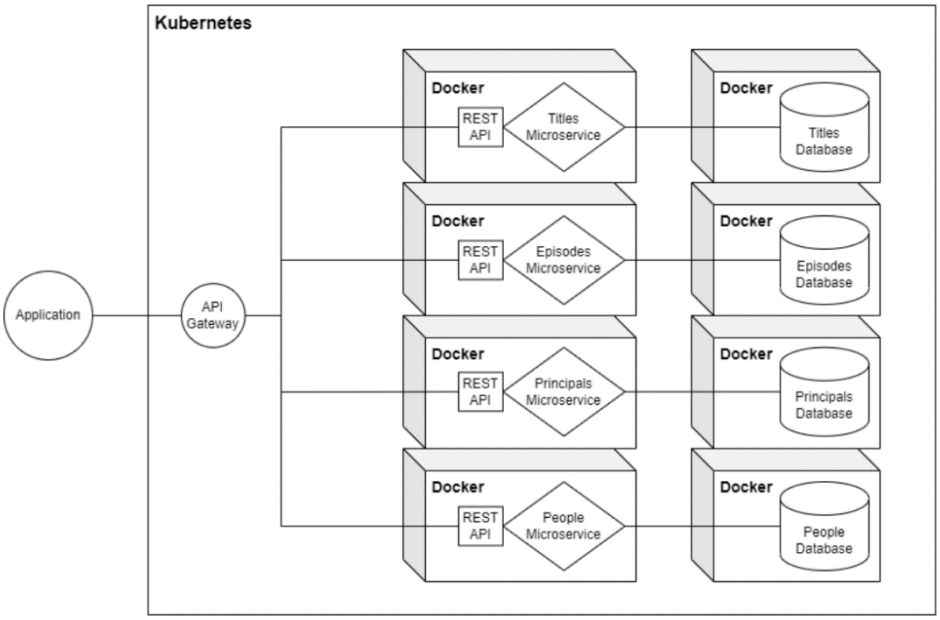

Figure 2 - First Stage Application Architecture

(First Stage) Architectural description: In a preliminary stage, the application architecture was to be defined as a set of four microservices, respective to the complexity of the information stored on the IMDb Dataset we are working with, where each microservice was also directly related to its own specific database. Each microservice and database was to be associated with its own specific Pod (Docker, in our case) and these elements were then to be managed by Kubernetes. To interact with the application and make use of its microservices, an API gateway was to be developed, and PostgreSQL was to be used as a database system for our project. Relative to the microservices, we considered using Java Spring due to the framework's unique capabilities, straightforward management on containerized environments, clean workplace for developing REST APIs, and simple database interaction. In relation to the API Gateway, Ingress was to be used as a tool to aid the development of this component. Finally, Swagger was to be used as a tool to test the application and its services.

Despite we considering that, at the time, the previously mentioned options were the best approach possible considering the context of our application, new insights and notions about cloud computing were revealed to us, enabling for some key changes to be made before the first discussion that took place in April, namely:

- **Change of the database configuration** from 4 independent databases that feed information to each microservice to a single shared database that would provide context to each microservice. This change was due to the well normalized and heavily dependent nature of our relational database;

- **Usage of Kubernetes Volumes and Replicas** in order to make a database cluster containing 2 machines to ensure reliability, data availability, and some fault resistance;
- **Addition of monitoring services** with Prometheus and Grafana.

Hereby follows the updated architecture of the system:

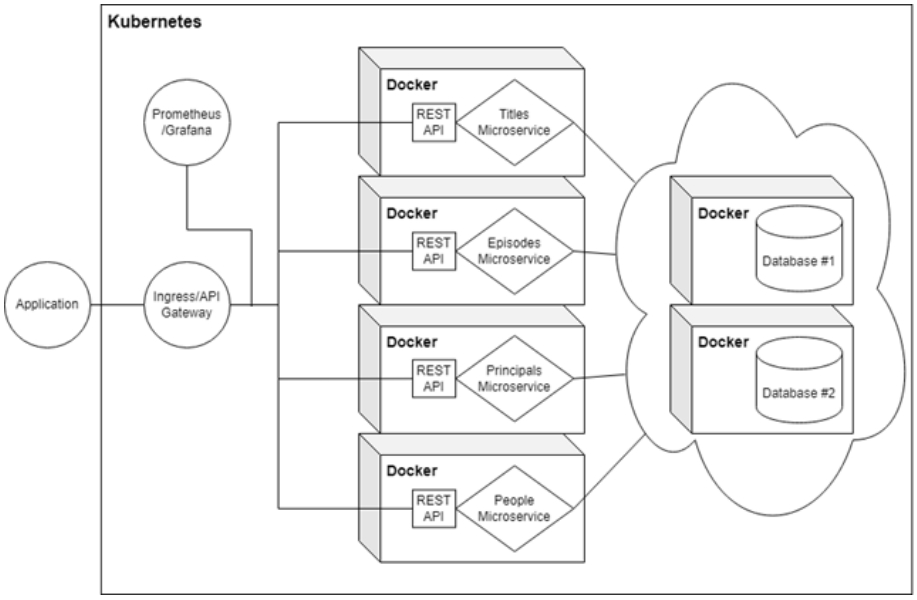

Figure 3 - First Stage "Updated" Application Architecture

- **Second Stage**

After the first project discussion and after the paper selection and presentation on the topic of Data Science, we understood that our project could be significantly enhanced via the utilization of powerful data processing frameworks such as Apache Spark. More concretely, we decided to improve our application with Apache Spark Elasticsearch, as now presented in the next application architecture diagram:

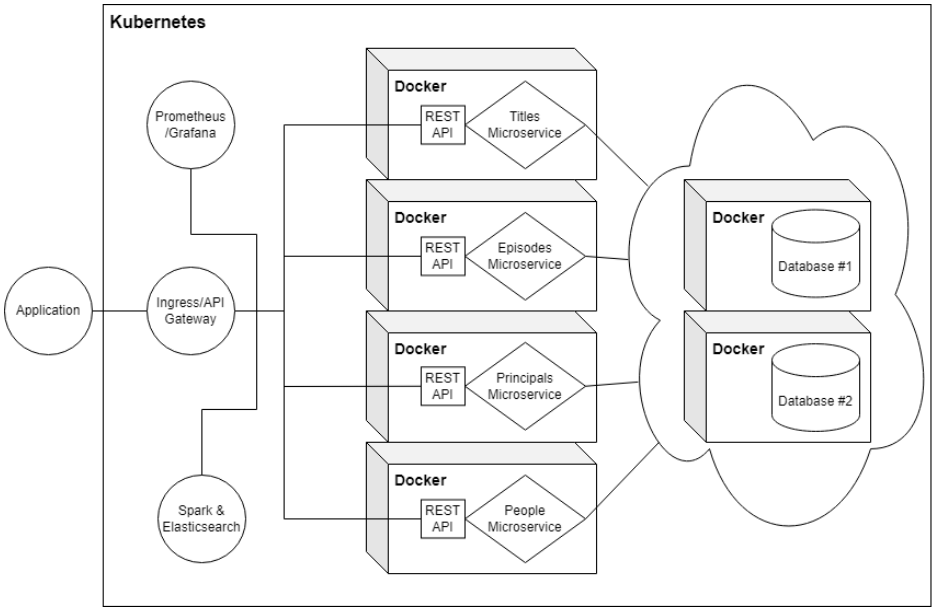

Figure 4 - Second Stage Application Architecture

(Second Stage) Architectural description: The application architecture maintained the defined set of four microservices and a cluster of two databases was configured (due to the existing large dependencies of data, to minimize resource usage and potentiate a more efficient ecosystem). We continued to use PostgreSQL as a database system, Java Spring for microservice implementation, and Ingress as API Gateway. Also, as shown in the first stage architecture, Prometheus and Grafana were used as tools for monitoring events and visualization of data relevant to identify patterns and observe the overall behavior of the system.

Focusing on the researched topic of Data Science, we aimed to use Spark and its distributed search and analytics engine Spark Elasticsearch and Kibana to store, search, and analyze the great volume of data that is transmitted and manipulated by the system. We also used logs over HTTP parameters and other relevant metrics that allowed for further extraction, assessment, and analysis of data (for example, to obtain the names of the most searched actors).

Finally, some topics regarding correctness and overall enhancement of the project that were also mentioned in the first project discussion, namely the update of the secrets config map, rolling update policies, dockerfile changes for correctness, expose command and usage of grpc were employed in this last project stage.

IV – Deployment

In terms of deployment plan, to construct a cloud native application we use google cloud and the various tools that it provides, such as the google container registry to store docker images, possible by the existing credits from the google accounts of each member of the group. The project itself is stored on a git repository and the start of the deployment is done by cloning the project in the google cloud virtual machine. After building it, we set up the google cloud credentials necessary to continue deployment. A Kubernetes cluster is then created, and specific Kubernetes files are run to complete the deployment of the project. After this, logs are analyzed, and liveness probes are verified to check the state of the system and verify if everything is running as expected. To note that our Kubernetes Engine includes some key points that are important to mention regarding the deployment of our application:

- **Scaling and Load Balancing:** Our project uses Horizontal-Pod-Autoscaling (HPA), provided by Kubernetes, and adjust the number of instances based on resource demand. Our Load Balancing and DNS Service are also provided by the Kubernetes Engine, allowing us to balance the load between the replicas of each service and simplify the DNS and service discovery that our services will require. This approach is also more secure due to the fact that our services are only accessible from within the cluster and are not exposed to the outside.
- **Availability and Fault Tolerance**: In terms of microservice deployment, we chose to only deploy one microservice (backed up by one replica) due to our limited google cloud budget, but we truly think the proper way to work would be to have every microservice deployed with two replicas. This way, if one fails, the other microservice will be able to handle the work even if limited by the load and give our Orchestration System time to instantiate a new replica, allowing us to have no service downtime.
- **Monitoring and Logging:** As previously mentioned, to monitor our application we use Prometheus and Grafana along with some metrics provided by the Google Cloud Engine that help us not only validate that our monitoring systems are giving us proper values but also add some additional information. Our logging is based on the ELK stack, more specifically Elasticsearch and Kibana, possibly by the development of a library that our microservices can use to make functional logs or any other type of logs. This helps us to understand and trace the behavior of our application with a centralized logging system and allows us to adapt and create new logs that can be used for data analytics and more complex dashboards in Kibana, creating a powerful mechanism for data science

purposes and overall visualization and interpretation of our system.

- **Rollback and Updates:** In order to have zero downtime, we use Kubernetes Rolling Policies and Rolling Updates in our deployment to allow for software updates and rollbacks while providing service at all times.
- **Environment Management:** As an additional measure in our deployment plan, we followed the twelve-factor application guidelines and decided to store our configs in the environment. However, for most simpler ones we used config-maps, and for more sensitive configs (such as secrets and passwords) we used so that values are encoded in our .yml file and encrypted when stored by Kubernetes.

V – Tests, Cost Analysis & Evaluation

We hereby define and present all relevant cloud-related tests, cost analysis and overall evaluation of our project.

- **Security Testing**

In order to show how the authentication is implemented, we now present a series of images to highlight the existing feature of forbidden access to authentication endpoints.

Case 1:

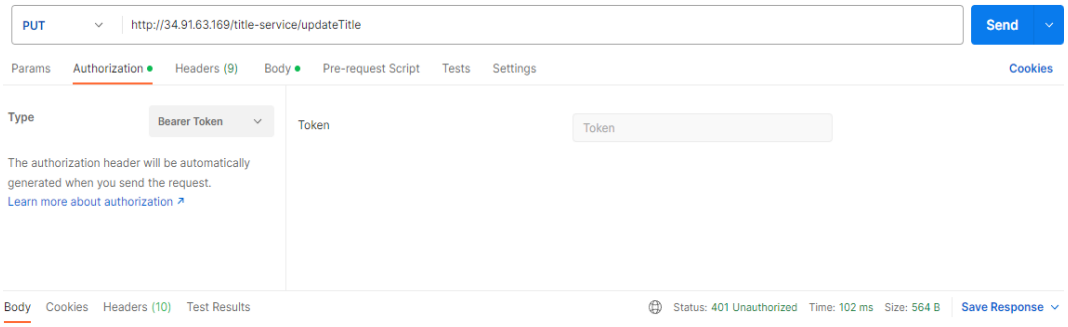

Figure 5 - No Access Token Is Present, so the resulting request returns 401 (Unauthorized)

Case 2:

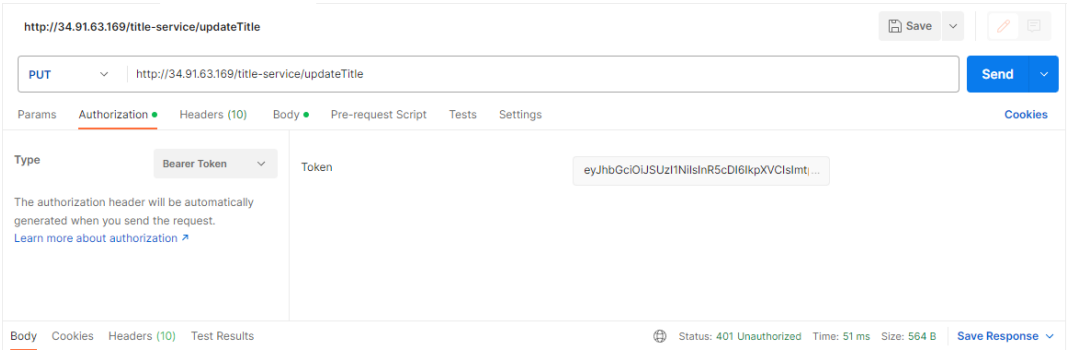

Figure 6 - Invalid or Expired Access Token, so the resulting request also returns 401 (Unauthorized)

Case 3:

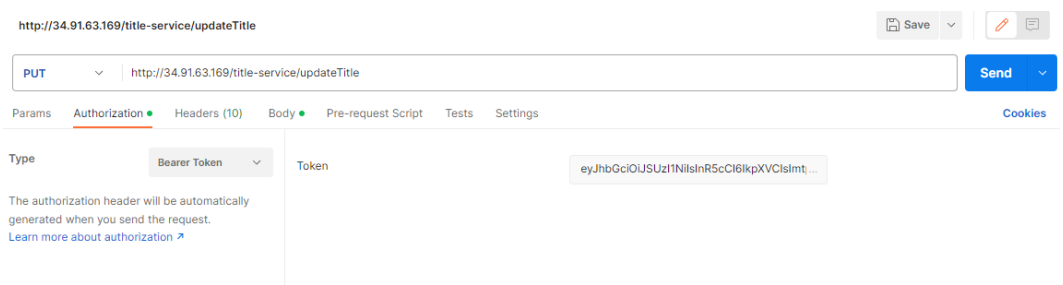

Figure 7 - Valid Access Token, so the expected result is 200 OK

- **Cross-site Reflection and SQL Injection**

Case 1: Store JavaScript Code in a database column

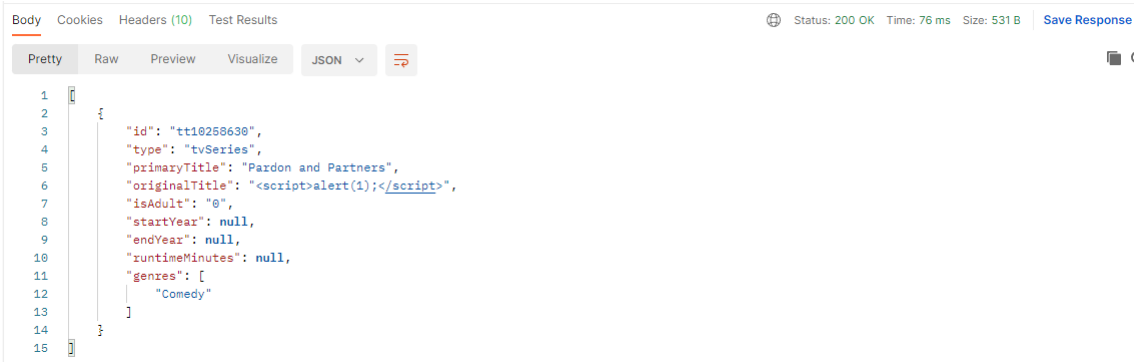

Figure 8 - Response from server (with JS script in "originalTitle" field)

Since we are using Spring Framework (a framework that uses a series of encoding techniques such as Prepared Statements to help prevent a variety of software vulnerability attacks), it allows us to detect and prevent code injections and XSS Attacks. This is because Spring does not run the input directly into a query in the database but instead uses Prepared Statements and encoding to detect malicious characters. Hereby follows the response from the above use case test:

$$[\{"id": "tt10258630", "type": "tvSeries", "primaryTitle": "Pardon\ and\ Partners",$$
$$"originalTitle":\<scripqt\>alert(1);\</script\>,isAdult:0,startYear: null,$$
$$"endYear": null, "runtimeMinutes": null, "genres": ["Comedy"]\}]$$

As we can see (highlighted in yellow), the malicious characters are successfully encoded by Spring so they won't be executed by the client's browser.

- **Performance and Scalability Evaluation**

To test the performance and scalability of our application, we ran a series of tests that allow us to measure the number of requests that our web server can process per second (the following image shows the logs of the test execution for 20 concurrent requests):

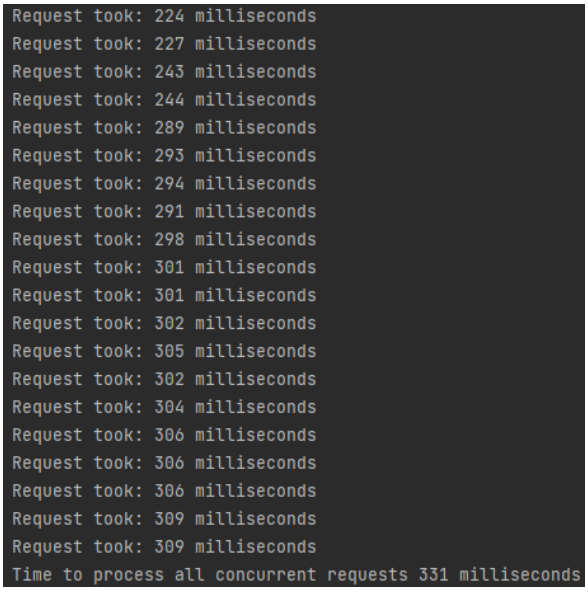

Figure 9 - Logs regarding 20 concurrent requests

We can then estimate the number of requests per second using the following formula:

$$x = \frac{20 * 1000ms}{331} = 60.4$$

, from where we understand that, on average, per second our system is capable of handling 60 concurrent requests per second. To verify this, we try to run 60 concurrent requests and check if the time it takes to process them is close to 1000ms (results in the following image):

```
Request took: 1101 milliseconds
Request took: 1102 milliseconds
Request took: 1104 milliseconds
Request took: 1107 milliseconds
Request took: 1108 milliseconds
Request took: 1110 milliseconds
Request took: 1105 milliseconds
Request took: 1110 milliseconds
Request took: 1111 milliseconds
Request took: 1105 milliseconds
Request took: 1115 milliseconds
Request took: 1162 milliseconds
Request took: 1163 milliseconds
Request took: 1164 milliseconds
Request took: 1165 milliseconds
Request took: 1170 milliseconds
Request took: 1171 milliseconds
Time to process all concurrent requests 1197 milliseconds
```

Figure 10 - Logs regarding 60 concurrent requests

We can notice from the result that the time it took to process 60 concurrent requests is somewhat similar to the expected (1 second ≈ 1.197 seconds), with some slight deviation due to network latency and the testing machine's hardware performance.

Using Prometheus, we can also estimate the time it takes to process each request individually on average, by analyzing the metrics given for the HTTP Total Requests and the HTTP Seconds Sum of a specific endpoint:

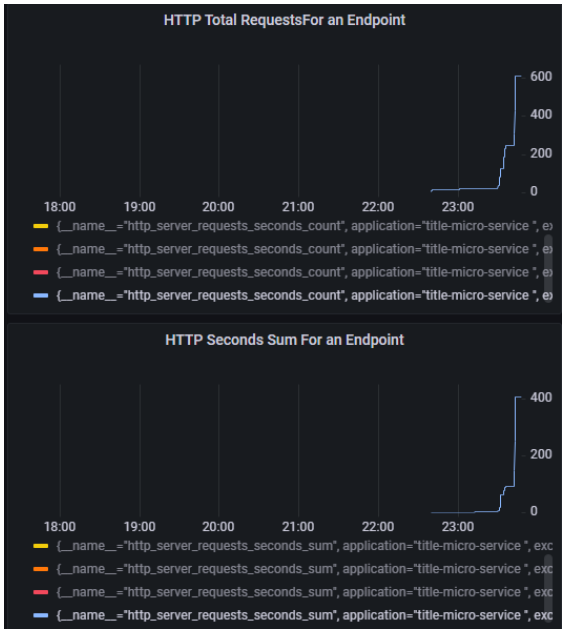

Figure 11 - HTTP Total and Sum of Requests for the title endpoint

From this last image, we concluded that a total of 608

requests from the load set were dealt in 407 seconds, making the average equivalent to 0.669 s/r (Seconds Per Request).

Using the ping command, we can also calculate average RTT:

```
Ping statistics for 34.91.63.169:
    Packets: Sent = 20, Received = 20, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 38ms, Maximum = 319ms, Average = 54ms
```

Figure 12 - Ping Command Results

Adding this to the average process time we get 0.669 + 0.054 = 0.723. Once again, the difference between this value and the expected 1 second threshold value of the system's performance for 60 requests can be explained by the test machine's hardware performance.

- **Memory Performance Evaluation**

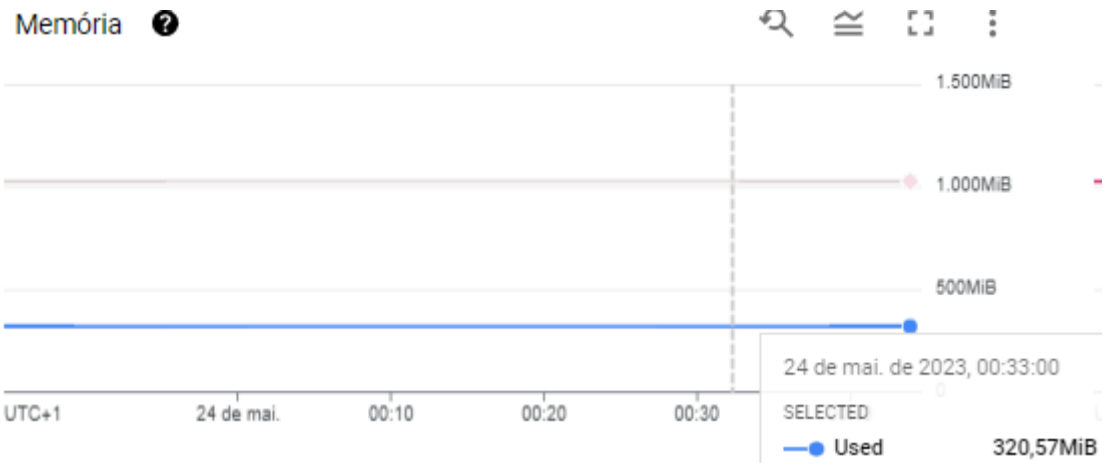

Figure 13 - Container Memory Consumption

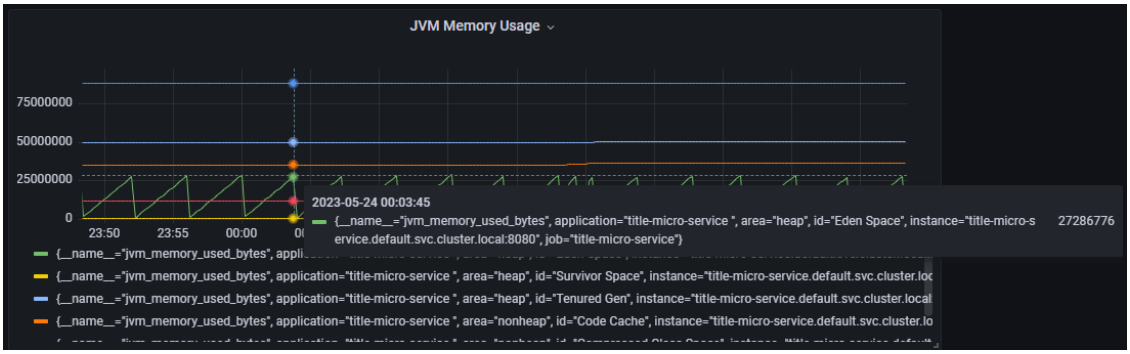

Figure 14 – JVM Memory Usage

From the above images, it is possible to observe that our JVM uses around 222 MB (sum of the memory points where the green line spike occurs) and 192MB of memory on standby. This means that our API Query consumes around 30 MB of memory (max of green line spikes), which for us seems like a reasonable value since our table has 2GB, from which 150 Entries are loaded in the result and later transformed into Java Objects (and consequent Query Results), along with other memory usage performed by the framework job itself. To also note that if we analyze the non-green lines (that present constant behavior), they are relatable to the standby memory usage of the respective component and the framework that it is using.

- **System CPU Usage Analysis**

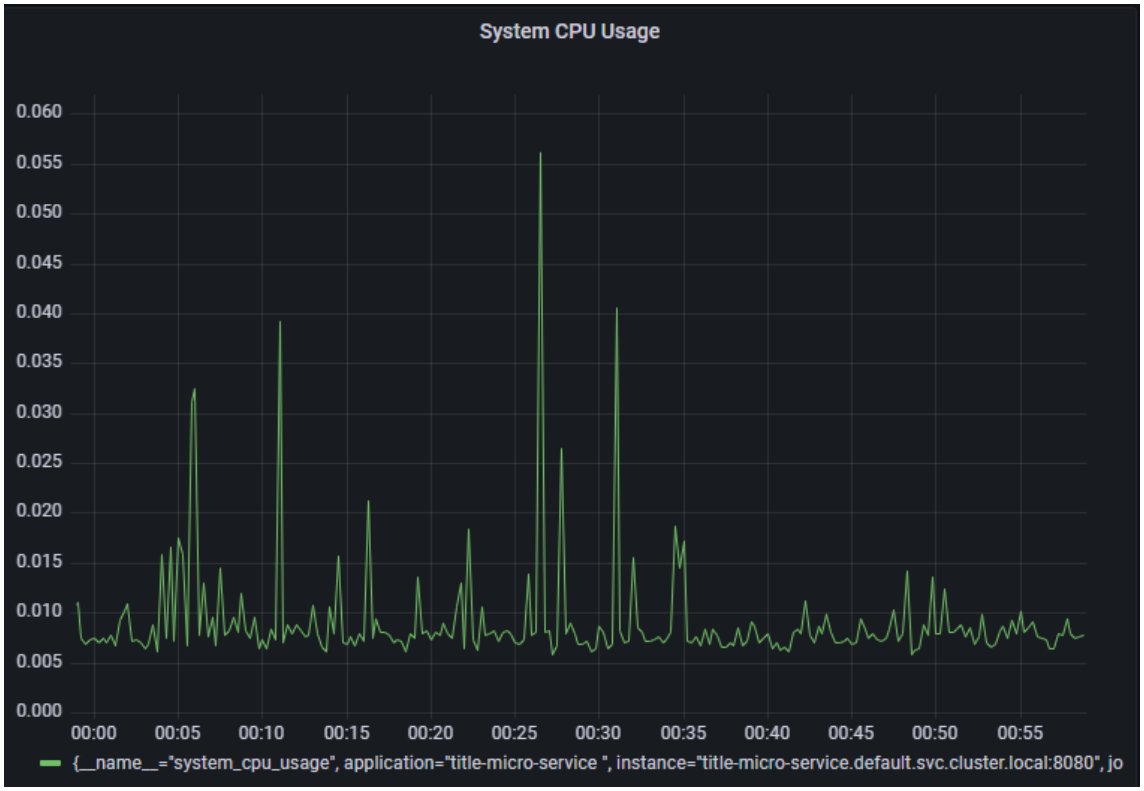

Figure 15 - System CPU Usage

The presented graphic related to the System CPU usage, from where we see that the max usage for our load tests lies between 0.030 and 0.060 “milicores” (a thousand of a core), which is the expected amount since our application does not process massive amounts of data, it only receives requests to the API controller, makes a database consult, sends the log to our Elastic Search, and finally returns the result to the user.

- **Scaling and Elasticity**

Now using a more aggressive stress test, we try and generate 1000 concurrent requests, from where the application maxes out its CPU cores (bottlenecking our general performance):

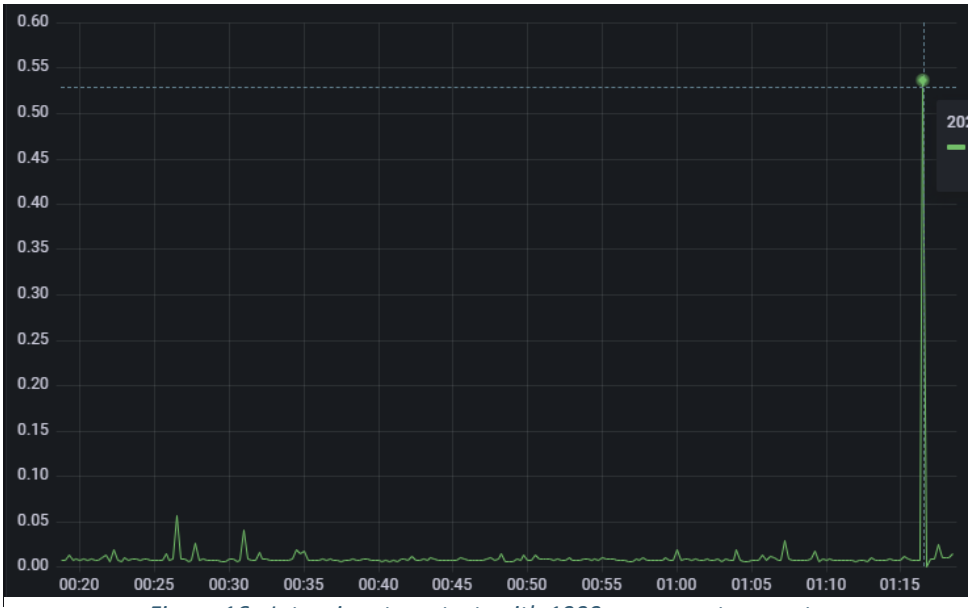

Figure 16 - Intensive stress test, with 1000 concurrent requests

It is possible to observe how the spike in the graphic is followed by a substantial drop. This is due to our Horizontal Pod Autoscaling, which softened the stress of the application when required and finally was dropped since it was no longer needed.

- **Observability**

In the last step of the project, we integrated Spark Elasticsearch and Kibana to our Kubernetes deployment, which allowed us to have a centralized infrastructure for our Microservice PODS. These logs are a custom library that we implemented so we can log custom messages as well as preview the status of the operation and send it to Elasticsearch where we can actually see the results (example in the following image):

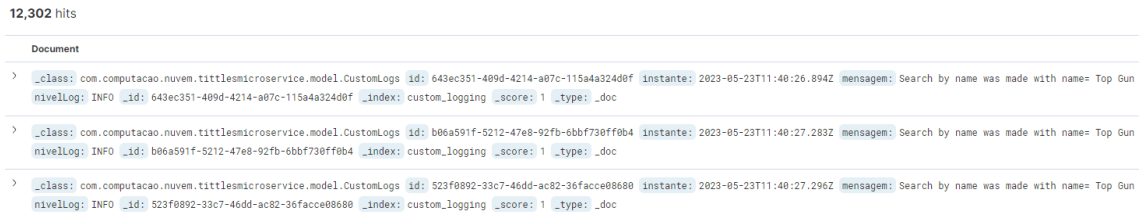

Figure 17 - ElasticSearch Centralized Custom Logging System

The presented Logs worked as expected, as we can also check the log level. The best part of this process is that this library is scalable, so at any point we can increase the number of parameters to our log index and add new info to be logged.

Using Google Cloud, we can see the traces of our component:

Figure 18 - Google Cloud Traces

For the metrics we also used Prometheus and Grafana, which worked as intended in our tests since most of our metrics were extracted from there and compared to the Google Cloud's metrics for integrity checking purposes.

- **Unit Testing**

To test the functionality of our code, we implemented some unit tests that assessed the API controllers responsible for the endpoints exposure and also the Services responsible for the logic and database querying:

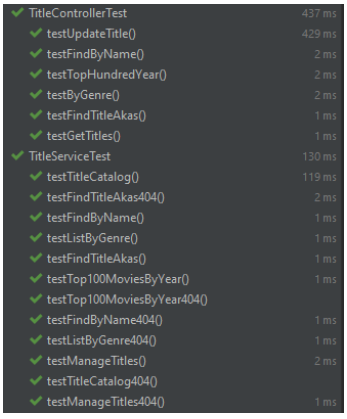

Figure 19 - Successful Unit Tests

- **Cost Optimization Strategies**

Relatively to cost optimization, in order to not spend all our credits, we scaled down from 4 to 1 microservice, focusing only on our title-microservice. In a general manner, we aimed to have as low as possible the CPU and Memory Requests:

Nome	Utilização da CPU (CPU)	Horas de CPU	Utilização da memória (GB)	Horas de memória	Cluster
alertmanager		0		0	hello-cluster
collector		0		0	hello-cluster
elasticsearch		0		0	hello-cluster
gmp-operator		NONE		NONE	hello-cluster
grafana		0		0	hello-cluster
ingress-nginx-admission-create	-	-	-	-	hello-cluster
ingress-nginx-admission-patch	-	-	-	-	hello-cluster
ingress-nginx-controller		0		1	hello-cluster
kibana-deployment		0		0	hello-cluster
postgresql-db		0		1	hello-cluster
prometheus-deployment		0		0	hello-cluster
rule-evaluator		0		0	hello-cluster
title-micro-service		0		0	hello-cluster

Figure 20 - Cost Optimization Table

If we look at the CPU bars, they are all relatively low mainly because CPU is not used in standby, only when load is increased the CPU usage will also respectively increase. We used around 0.250 “milicores” for each component except for Elasticsearch since it uses more CPU to process our logs and searches, and in terms of memory usage the same happens, from where we used around 500 Mibs for each component which seemed to be more than enough to cover our use, with some exceptions related to Elasticsearch which is heap heavy and we found that around 3 to 4 GB of RAM was required in order for it to work properly (our application uses pagination and caching in order to achieve fast speeds along with low memory consumption).

Note: Our fast speeds can not only be explained by the Elasticsearch indexing, our microservice caching and pagination but can also be explained from our created indexes for the database columns that improved drastically our response times from around 16 to 20 seconds to around 70 milliseconds for individual requests (possible due to the great size of our dataset).

- **Billing**

To finalize this section, we send in annex information relative to the billing of our system, generated by the Google Cloud Pricing Calculator.

VI – Discussion & Future Improvements

We opted for a simplified deployment configuration in which each service is represented by a single replica. However, as previously mentioned, due to budget constraints we made the decision not to deploy three of the total four defined microservices, despite them being implemented and fully functional. This approach allowed us to demonstrate and test our system while conserving our available credits and minimizing costs. Although this configuration satisfies our minimum requirements for performance and reliability, it represents a trade-off between functionality and cost, aligning with the project's purpose. Alternatively, an intermediate solution would involve increasing the resources, such as RAM and CPU, for each service replica. While this would enhance performance and system capacity, it would also incur higher costs as additional resources would need to be provisioned from the cloud provider. Although reliability would improve by handling increased loads and reducing crashes, this solution falls short of an ideal scenario.

For an ideal and more expensive solution, we would reconsider the available resources and increase the number of replicas to at least two, something that would offer improved performance and enhanced reliability, as one replica can seamlessly handle the service in case of failure or increased load. However, implementing this solution would significantly increase costs because it would require a potent Kubernetes cluster with more RAM and CPU cores.

In addition to these considerations, incorporating extras in the deployment configuration must be weighed against their associated costs. These extras may include Horizontal/Vertical Pod Autoscalers, backups, volume redundancy, externalizing the database from the cluster to the cloud, etc., all features that would further enhance the system but require careful evaluation of their impact on budget increase and overall project objectives.

As for future improvements, we collectively recognize the potential for further refinement of our project. Our focus extends beyond enhancing security measures to comprehend the goal of establishing an easily configurable infrastructure. To achieve this, we propose the utilization of Config Maps and Secrets in a distributed manner, implementing an external and secure Config Service, something that will most likely ensure heightened security and streamlined configuration processes.

Furthermore, we have deliberated on performance enhancements and devised a solution that entails leveraging Redis as an in-memory distributed cache. By adopting this approach, we anticipate accelerated retrieval of database entries and expedited updates. Specifically, frequently accessed entries would be then stored in Redis, facilitating faster access and manipulation of information. We consider this strategic implementation option to considerably improve our system's performance, particularly in scenarios demanding very high scalability.

VII – Contributions & Options Taken

Collectively, we share the unanimous feeling that each and every person within our group unequivocally invested in an equivalent measure of effort and dedication towards the completion of the presented project. In relation to the organization and definition of roles within our group, we also believe that there exists no individual who solely assumes a specific role, rather each one of us actively engaged in embodying and make use of the diverse responsibilities associated with the roles of Microservices Programmer, System Architect, Networking and Security Specialist, DevOps Officer, and Data Scientist.

As for the options taken, we feel that our decision-making process and selection of features have been in line with the project's objectives and requirements. The choices are summarized into the following table:

<table>
<tr><th colspan="3">Options Taken</th></tr>
<tr><th>Topic</th><th colspan="2">Motive for selection</th></tr>
<tr><td>Dataset, Use Cases & API</td><td colspan="2">Taking into consideration our use cases and the characteristics of the data within the chosen IMDB dataset, we feel that the amount of use cases, microservices and API calls is ideal when it comes to segregating functions effectively considering the requirements of our system.</td></tr>
<tr><td rowspan="5">(Some) System Frameworks</td><td>Spring</td><td>Between Flask/Django, .NET, and Spring, we feel that the last one makes it easy to create stand-alone, production-grade Spring based Applications with minimum fuss while requiring minimal Spring configuration, something ideal for our microservice architecture.</td></tr>
<tr><td>PostgreSQL</td><td>Instead of using MongoDB (a non-relational database manager), we opted to use PostgreSQL because of the relational nature of our data.</td></tr>
<tr><td>Google Cloud</td><td>From AWS, Microsoft Azure, and Google Cloud, we chose the last option due to the courses taught materials and provided usage credits.</td></tr>
<tr><td>Prometheus & Grafana</td><td>We had the option to use Google Cloud exclusively for monitoring, but we felt that by using Prometheus & Grafana we could provide a more complete monitoring experience.</td></tr>
<tr><td>Elasticsearch</td><td>We opted to use Elasticsearch due to its capabilities of tokenization, speedy search, simplified setup, and seamless integration with our cloud-driven application.</td></tr>
</table>

VIII – Conclusions

Considering our initial objectives for this project and our competence in delivering the intended application, we believe that, despite the constraints of a restrictive timeframe, we have successfully delivered what we set out to achieve. Our accomplishments include a multitude of key elements, including the successful deployment of a cloud-based application powered by microservices architecture, and the implementation of a dependable database system managed through pods and Kubernetes, ensuring optimal performance. In addition, the incorporation of various cloud-related frameworks, such as Prometheus, Grafana, Ingress, Apache Spark, and others, further solidifies our belief that we have realized the intended outcomes of our project with utmost success.

IX – References

1. Y. Huang, Z. Cheng, Q. Zhou, Y. Xiang and R. Zhao, "Data Mining Algorithm for Cloud Network Information Based on Artificial Intelligence Decision Mechanism," in IEEE Access, vol. 8, pp. 53394-53407, 2020, doi: 10.1109/ACCESS.2020.2981632.
2. A. Gupta and S. Jain, "Optimizing performance of Real-Time Big Data stateful streaming applications on Cloud," 2022 IEEE International Conference on Big Data and Smart Computing (BigComp), Daegu, Korea, Republic of, 2022, pp. 1-4, doi: 10.1109/BigComp54360.2022.00010.
3. P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, et. al., "Steering Query Optimizers: A Practical Take on Big Data Workloads", SIGMOD21 Proceedings of the 2021 International Conference on Management of Data, June 2021, pp. 2557-2569, doi: 10.1145/3448016.3457568.
4. H. Herodotou, Y. Chen and J. Lu, "A Survey on Automatic Parameter Tuning for Big Data Processing Systems", ACM Computing Surveys Volume 53 Issue 2, March 2021, Article 43, pp. 1–37, doi: 10.1145/3381027.
5. Shah, N., Willick, D. & Mago, V. A framework for social media data analytics using Elasticsearch and Kibana. Wireless Netw 28, 1179–1187 (2022). doi: 10.1007/s11276-018-01896-2
6. V. -A. Zamfir, M. Carabas, C. Carabas and N. Tapus, "Systems Monitoring and Big Data Analysis Using the Elasticsearch System," 2019 22nd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 2019, pp. 188-193, doi: 10.1109/CSCS.2019.00039
7. D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," in IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, Sept. 2014, doi: 10.1109/MCC.2014.51.