

Deliveries – Final Report

Cloud COmputing 2022/23

Grupo 12

Bruno Susana | Nº 57408  
Davide Nogueira | Nº 60555   
Rafael Espadinha | Nº 58864   
Rui Silva | Nº 57643

Índice

[Deliveries – Final Report 1](#_Toc110346487)

[Índice 1](#_Toc1149538864)

[Data set Characterization 3](#_Toc614551210)

[Use Cases and REST API 4](#_Toc1084380506)

[1. Set of Use Cases 4](#_Toc1148458845)

[2. REST API 5](#_Toc2049663198)

[Architecture 6](#_Toc1200650034)

[Implementation 6](#_Toc1578952403)

[Deployment 7](#_Toc1878772652)

[Test and Evaluation techniques and results 8](#_Toc1852410847)

[Discussion 8](#_Toc1466057537)

[Conclusions 8](#_Toc1045336248)

Data set Characterization

Proposed dataset:

* **topic** 
  + Amazon Books Reviews
* **summary** about the information contained in the dataset
  + This dataset has information about 3M book reviews for 212.404 unique book and users who gives these reviews for each book.
* **date of creation or last update**
  + Dataset last update 5 months ago
* **file type**
  + 2 csv files
* **size**
  + The total size is 3GB
* **link to the files (URL)**
  + [Amazon Books Reviews | Kaggle](https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews?resource=download&select=books_data.csv)

Uma imagem com mesa

Descrição gerada automaticamente

Use Cases and REST API

## Set of Use Cases

Use Cases

U1 – Authenticates

U2 – Create Rating

U3 – Get Ratings by book ID

U4 – Update Ratings by rating ID

U5 – Delete Ratings by rating ID

U6 – Get Rating By Review Score

U7 – Create Book

U8 – Get Specific Book by ID

U9 – Update Specific Book by ID

U10 – Delete Book by ID

## REST API

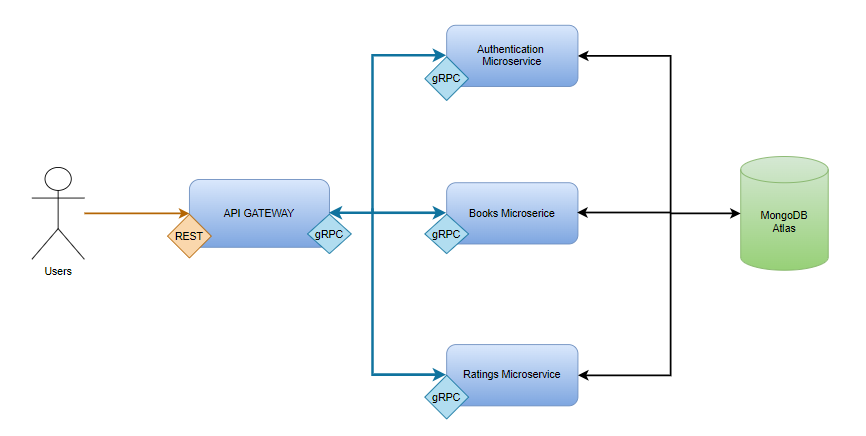
YAML File provided

The functional requirements are:

* Users can search Books by title, author or category
* Users can add reviews associated with a Book
* Users must be authenticated before adding a review
* Administrator must be authenticated before managing Books or Reviews
* Administrator can manage all books
* Administrator can search reviews and manage them

The following Diagram represents the applications components (services, Databases) and the interactions between them.

Application Architecture

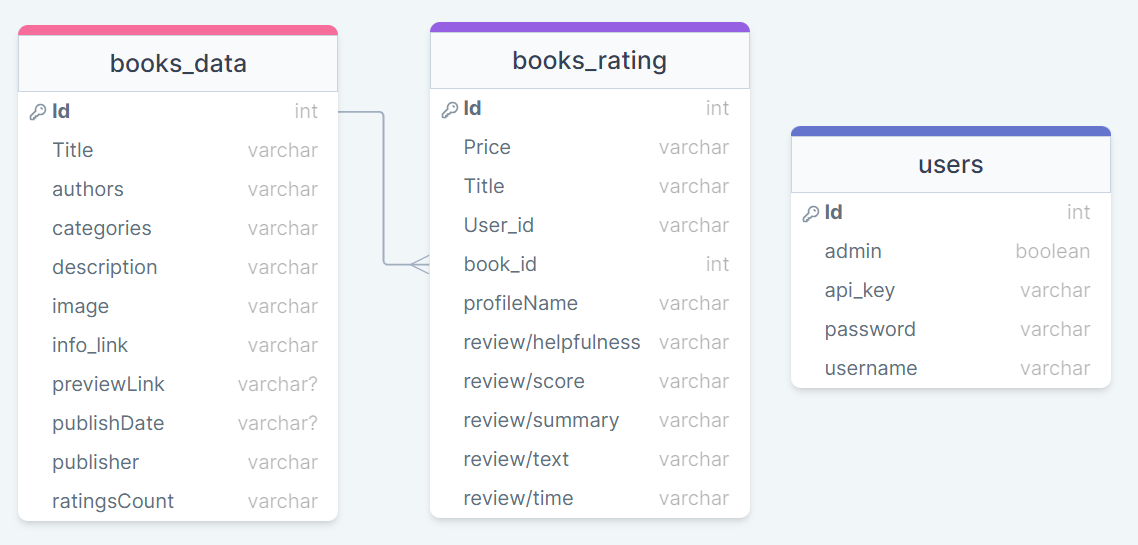


Architecture

The non-technical architecture is driven by the non-functional requirements. These are:

* Scalability: implementing a load balancer
* Security: usage of HTTPS instead of HTTP
* Maintainability: using Kubernetes dashboard and Grafana
* Performance: using a cloud database service
* Reliability: using Kubernetes HPA

Database



Implementation

To develop our application, we followed the architecture explained above, meaning that we divided our application into 3 microservices plus the api gateway. These microservices and the gateway communicate with each other using gRPC, but the gateway receives requests through REST. The two main microservices that provide the endpoints for almost all requests are the Books service and Ratings service. The endpoint exposed by the gateway are:

* Redirected to books microservice:
  + /books/{token} **(POST)**
    - Only administrator accounts with a valid token can post books
  + /books/{book\_id} **(GET)**
    - Everyone can make a GET request for a book id
  + /books/{book\_id}/{token} **(DELETE)**
    - Only administrator accounts with a valid token can delete books
  + /books/{book\_id}/{token} **(PUT)**
    - Only administrator accounts with a valid token can modify books
* Redirected to ratings microservice:
  + /ratings/{token} **(POST)**
    - Only users with a valid token can post a rating
  + /ratings/{book\_id} **(GET)**
    - Everyone can get the ratings for a specific book
  + /ratings/{rating\_id}/{token} **(PUT)**
    - Only admins can modify ratings
  + /ratings/{rating\_id}/{token} **(DELETE)**
    - Only admins can delete ratings
  + /ratingsReview/{review\_score} **(GET)**
    - Eveyone can get ratings by review score
  + /ratingsReviewByTitle/{book\_title} **(GET)**
    - Eveyone can get ratings by book title
  + /booksPrice/{book\_price} **(GET)**
    - Everyone can get ratings by price

All routes that require a token are protected, meaning that only an authenticated user can make the request. In addition, the microservice will check if the user has the necessary permissions to execute the request i.e., is an administrator. This token is generated making use of another endpoint:

* /login/{username}/{password}

This endpoint requires the user to insert a valid username and password that will match with one account in the database. If everything goes right, the microservice of the users will generate a token that will be stored in the database associated with that user. This token must be used in order the execute the request mentioned above.

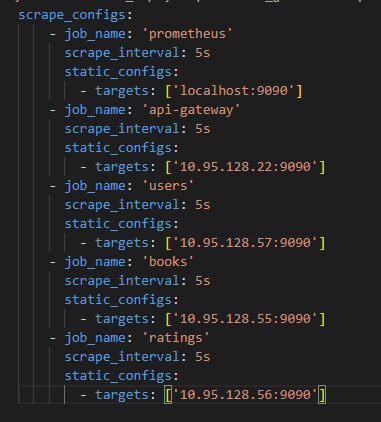
Regarding the database, and since it needed to be scalable, we decided to use a cloud database, namely MongoDB Atlas. All the 3 microservices connect to it. We also developed a script to populate it.

Deployment

Initially we were thinking about using a containerizing everything (microservices and database), after the first discussion we concluded that using a container with the database would come with more problems to handle and will increase the difficulty of scalability and maintainability, that´s why we changed to a mongoDB cluster, this way the replication of database is made by a third party (Atlas) and our containers are directly connected to it.

For deploying the application, we utilize a script that initiates the deployment of all the YAML files located in the "deployment\_files" folder, these files are responsible for starting the deployment with one replica of each microservice, Additionally, the YAML files located in the “service\_files” initiate the service and HPA (Horizontal Pod Autoscaler) for each microservice, the services files expose two ports internally to the other cluster pods: in case of API gateway 8080 for the application, 9090 for the Ppometheus metrics.

We also have a script ("startup\_prometheus\_grafana.sh") for deploying Prometheus, this script creates a configmap that contains the references to the Prometheus services running on their respective IP addresses as we can see on the picture below and starts the Prometheus server and Grafana server and expose GPafana to outside of cluster with a load balancer.

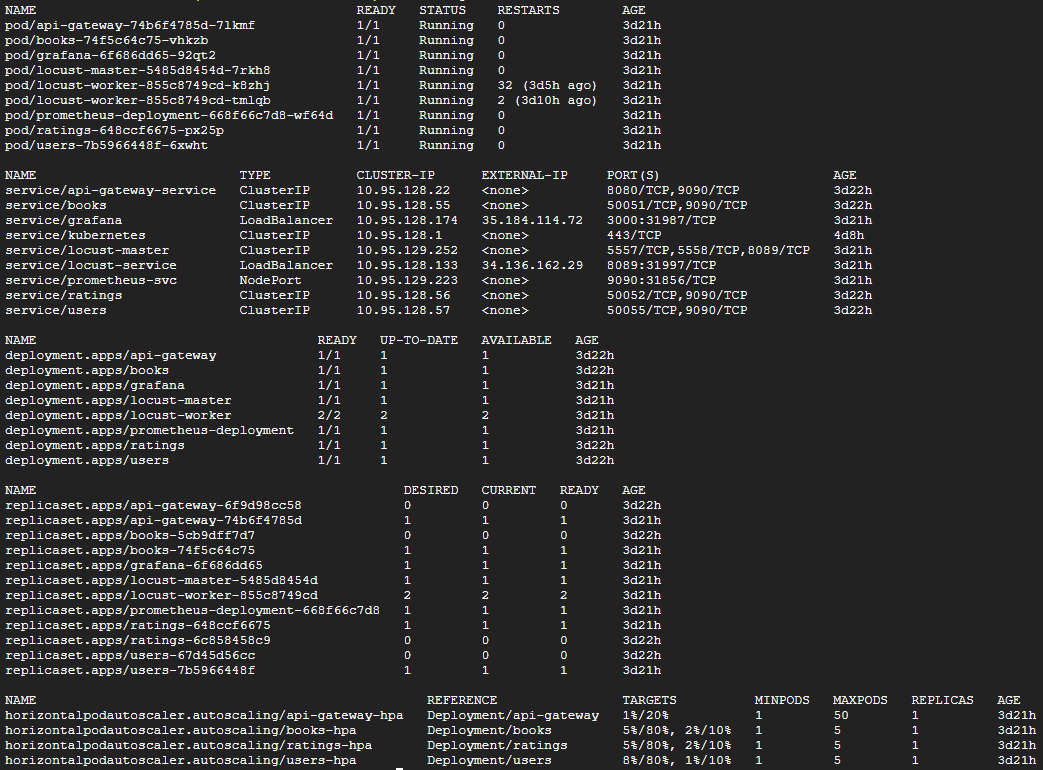
.

To expose the API-gateway on port 80 and allow connection to the API through the ingress, we have another script specifically for starting the ingress during deployment (“startup\_ingress.sh”).



To establish the CI/CD pipeline, we developed a script (“update\_images.sh”) that builds the images and pushes them to Docker Hub. Consequently, when the deployment files are executed, they fetch the updated image from Docker Hub.

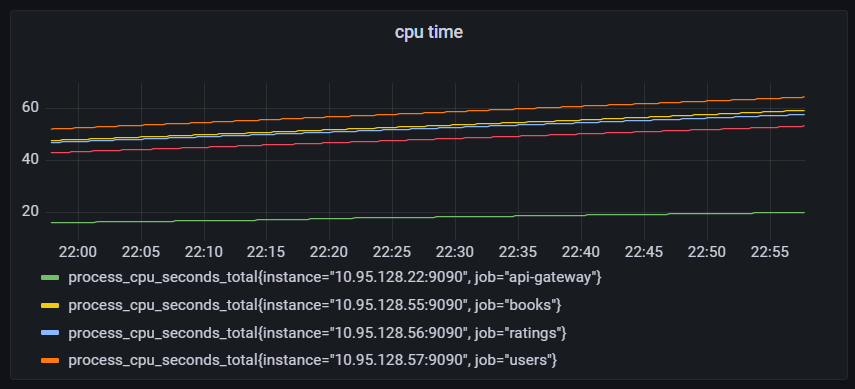
Overall print of deploy.

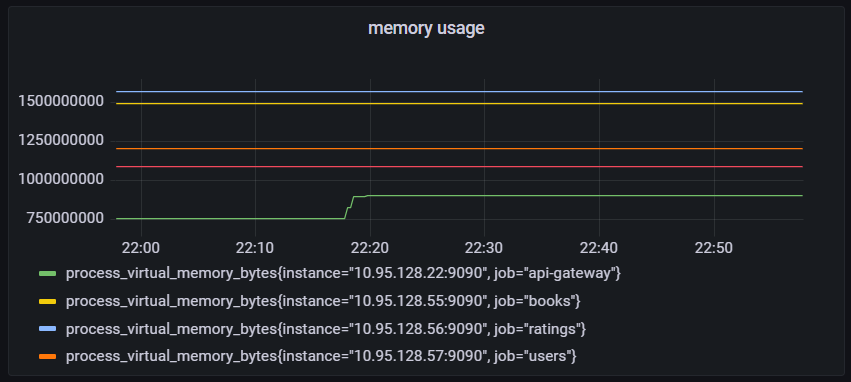


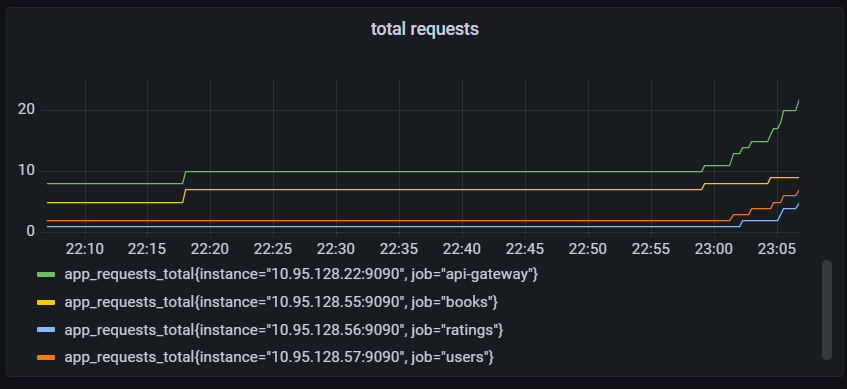


Test and Evaluation techniques and results

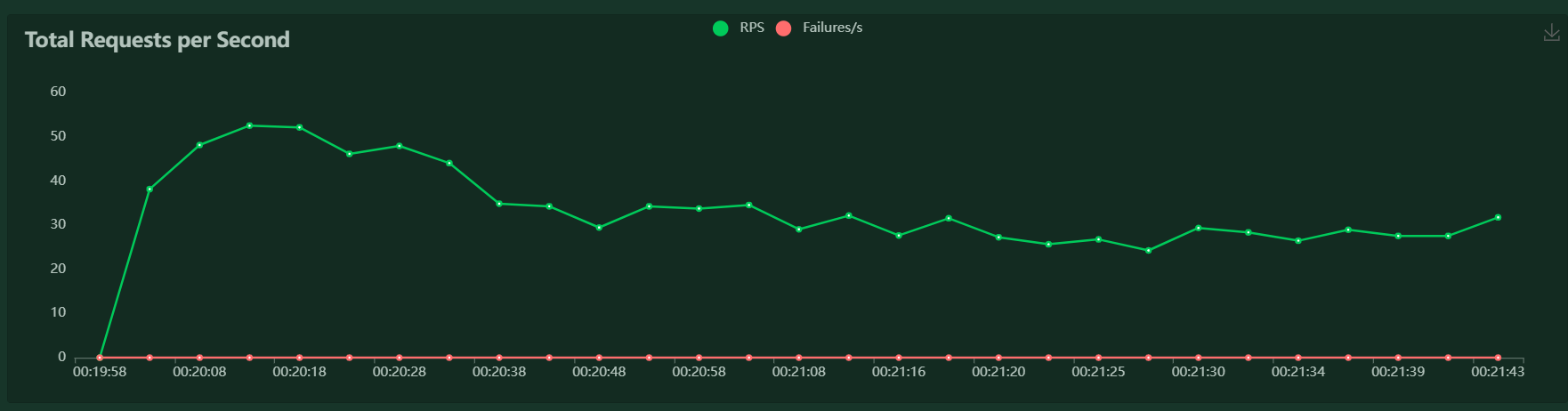
In order to measure some metrics of our application, we started a Prometheus server in each pod. These Prometheus servers measure the CPU time, memory usage and the total number of requests for each pod. Beyond this we also have a pod running Grafana. Grafana gets the data from Prometheus and shows it in graphics, on the web. In the 3 graphics below, we can see these metrics as we increased the number of requests made to each microservice.

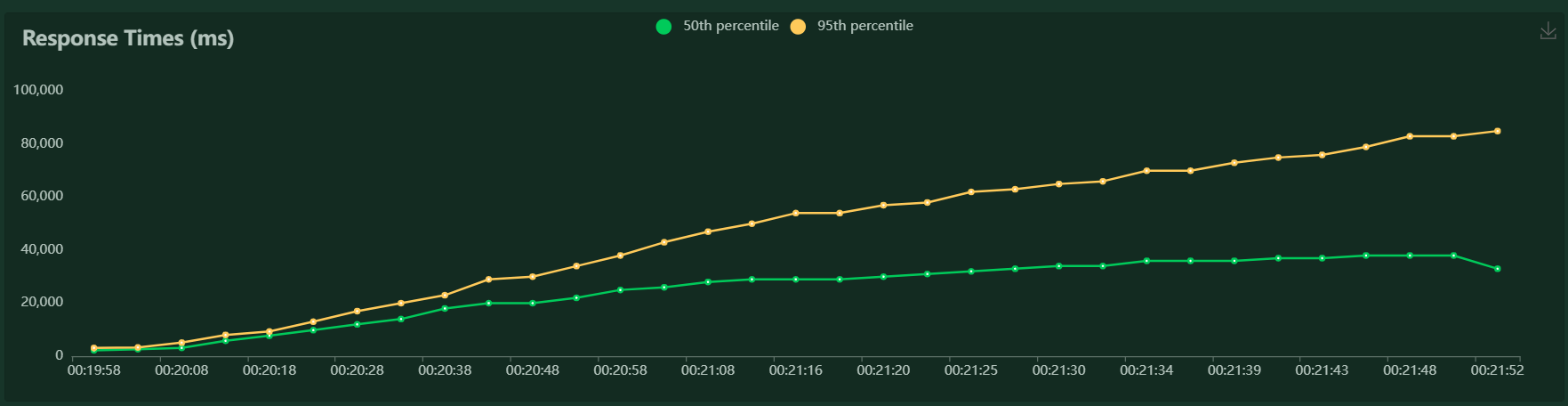


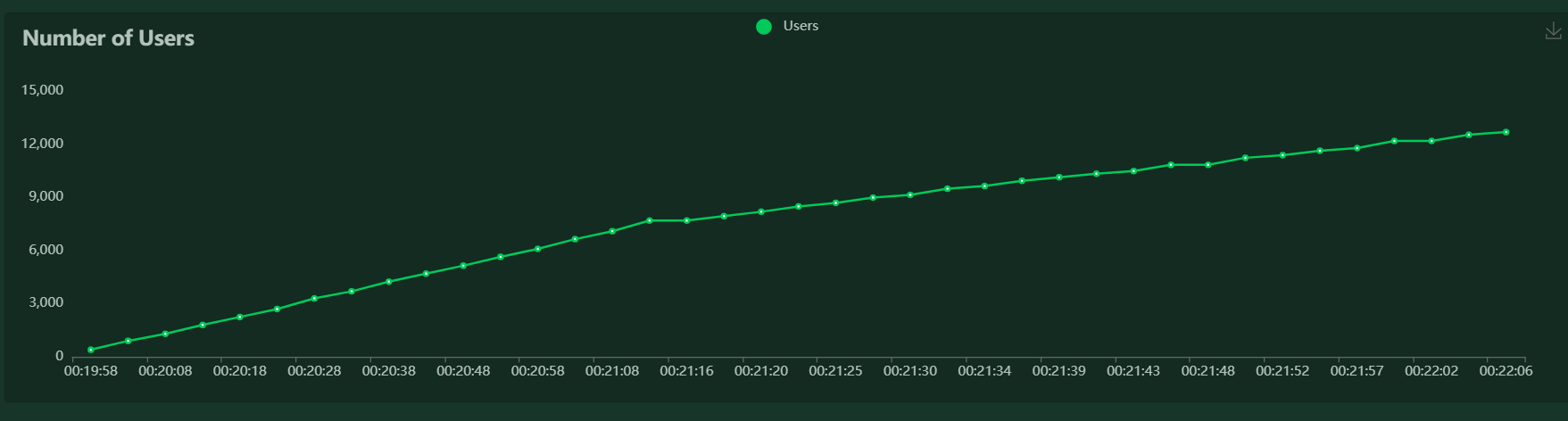




Once we ensured the correct functioning of the Prometheus server, we set up Locust to execute load tests in the application. We had 2 pods with Locust workers running. Once we started Locust and it started spawning more and more users, as we can see in the 3rd graphic below the HPA started creating more pods to satisfy the number of requests received.







Conclusions and discussions

We think we achieved a good result, although we could have improved some points. One of these points is the parameters used in the HPA. Another point that can be improved is the auditing and monitoring of the microservices, as well as its own recovery when something goes wrong. Another improvement Overall, our application meets most of the requirements, from integrating microservices with gRPC to as external cloud database that handles all the issues related with it, the containerized environments where our microservices run, separated from each other as well as managing all of them to work together. Finally, we also have some basic monitoring of the microservices, showing us some basic metrics.