Cloud Computing

Masters in Informatic Enginnering
Project's Final Report

Group 02

Luís Viana (62516)
Guilherme Santos (62533)
João Magalhães (62546)
André Santos (62754)

Faculdade de Ciências, Universidade de Lisboa
`{fc62516, fc62533, fc62546, fc62754}@alunos.fc.ul.pt`

Teacher: Mário Calha
mcalha@ciencias.ulisboa.pt

June 18, 2024

Contents

1 Introduction

The goal of the project is to develop and deploy a cloud-native application that offers a set of services that provide relevant information extracted from a dataset as a set of business capabilities. The services will be provided through a **REST API** and are organized in two sets: regular user and premium user. The deployment of the cloud native application was done using the cloud service provider, **Google Cloud Platform** (**GCP**).

2 Motivation and dataset characterization

The chosen dataset refers to a collection of data such as artists, labels and releases, structured in a typical relational format. This data was obtained from **Beatport**, a renowned online music store specializing in electronic music and can also be used to match **Spotify** audio features through the **Spotify API**. It represents in total 7.36 GB and can be found at https://www.kaggle.com/datasets/mcfurland/10-m-beatport-tracks-spotify-audio-features/data.

The dataset is composed by a set of tables (multiple *.csv* files) containing audio features of tracks (duration, instrumentals, acoustics, etc.), artist´s names, **Beatport**'s web pages, tracks, media, releases, music genres and subgenres, labels, releases and tracks´ keys.

We decided to choose this dataset because initially we identified four possible business capabilities that could be interesting for the development of this project. These four business capabilities were:

- Identity and Access Management (Users Authentication, profiles)
- Track´s data analysis and summaries (Reports, Graphs of audio features)
- Artists Search (Search by browsing filters or tracks)
- Tracks Recommendations

3 Use cases and REST API

A set of functionalities was defined to be the basis of the cloud-native application. These functionalities are divided into two groups, functionalities that can only be accessed by premium users (authenticated users with a set of permissions associated to their role) and functionalities that can be accessed by any authenticated user. Table 1 list all different use cases and the respective necessary roles to access certain functionalities.

Roles	**Use cases**
Any	Search artists
Premium	Add a new artist
Any	Get an artist's releases
Any	Search music genres
Premium	Add music genres
Any	Search tracks of a genre
Any	Search tracks
Premium	Add and delete tracks
Any	Search releases
Premium	Add releases
Any	Search playlists created by users
Premium	Create and delete owned playlists
Any	Search a playlist's tracks
Premium	Add and remove tracks from owned playlists

Table 1: Roles and Use Cases

The implementation of these use cases was achieved by creating a **REST API** which was defined previously using an **OpenAPI** specification. Implementation aspects are presented at Section 5. Table 2 presents the various endpoints exposed by the developed **REST API**. All of the presented endpoints have a */api* prefix.

Method	**Endpoint**	**Description**
	Artists	
GET	/artists/{artist_id}	Search artist by their id
POST	/artists	Add new artist
GET	/artists/{artist_id}/releases	Get an Artist's releases
GET	/artists/{artist_id}/tracks	Get an Artist's tracks
	Genres	
GET	/genres	Read all genres
POST	/genres	Add new genre
GET	/genres/{genre_id}	Search genre by their id
PUT	/genres/{genre_id}	Update genre by their id
GET	/genres/{genre_id}/tracks	Get a Genre's tracks
	Tracks	
GET	/tracks/{track_id}	Search track by their id
DELETE	/tracks/{track_id}	Delete track by their id

Continued on next page

Continued from previous page

Method	**Endpoint**	**Description**
POST	/tracks	Add new track
GET	/tracks/{track_id}/genre	Get a track's genre
	Releases	
GET	/releases/{release_id}	Search release by their id
POST	/releases	Add new release
	Playlists	
GET	/playlists/{playlist_id}	Search playlist by their id
DELETE	/playlists/{playlist_id}	Delete playlist by their id
POST	/playlists	Add new playlist
GET	/playlists/{playlist_id}/tracks	Get a Playlist's tracks
PUT	/playlists/{playlist_id}/tracks	Add or delete track from playlist
	Auth	
GET	/auth/login	Create session using Auth0
GET	/auth/logout	End session
GET	/auth/callback	Callback endpoint for Auth0

Table 2: API Endpoints and Their Descriptions

4 Architecture

4.1 Application's architecture

In the context of the application developed for this project, we decided to follow a microservice based architecture. The structure of the application in this type of architecture is as a set of independently deployable, loosely coupled services. Each service consists of one or more subdomains and is owned by a team that owns the subdomains. This structure allows for independent teams to focus on a single domain of services and promotes simplicity of code, autonomy and fast deployment. Our architecture follows three different patterns. The **Database per Service** pattern describes how each service domain has its own database in order to ensure loose coupling. The **API Gateway** pattern defines how clients access the services in a microservice architecture. The **Remote Procedure Call** pattern defines how services can communicate. Figure 1 represents the application's architecture.

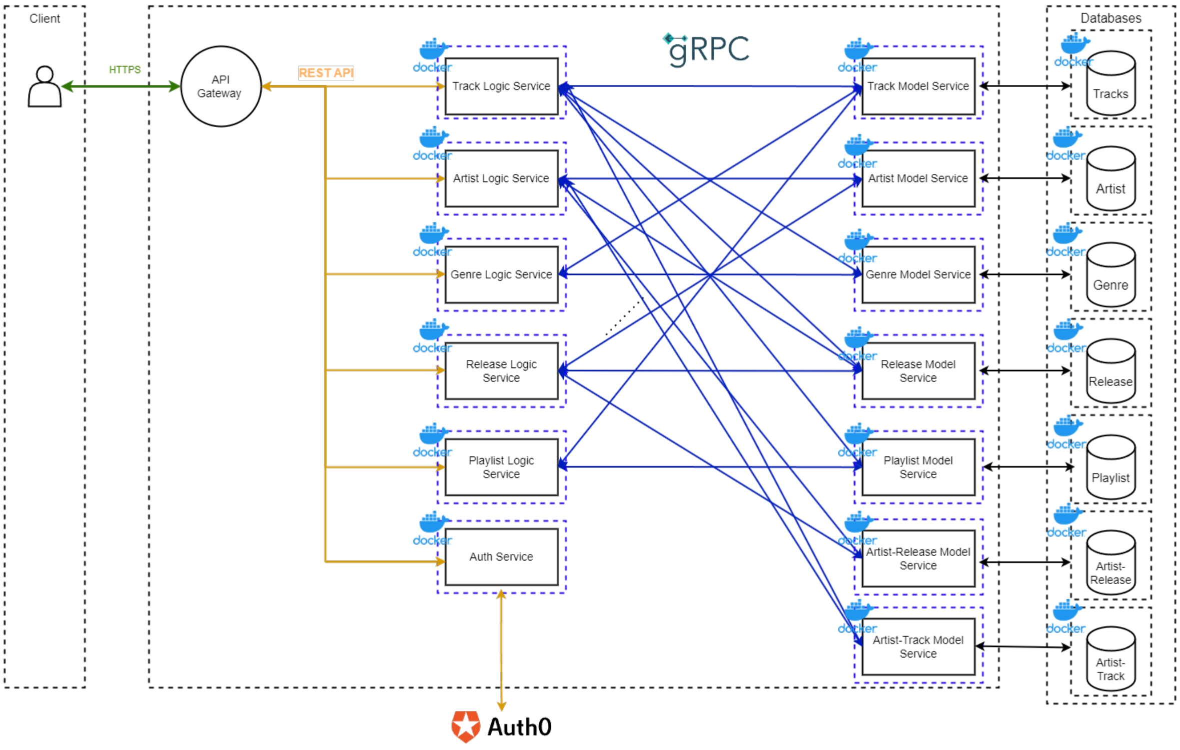


Figure 1: Application's architecture

There is a database for each service domain with the exception of the **Auth Service**. The communication between microservices is implemented using the **Google Remote Procedure Call** (**gRPC**) framework for the **Python** programming language. The lines connecting the different microservices represent the existing **gRPC** communications between them. The **API Gateway** exposes the **Logic Services**' and the **Auth Service**'s respective **REST API**'s endpoints to the outside using the **HTTPS** protocol. The architecture assumes each microservice to be containerized in one or more containers for replication and balancing purposes to guarantee service decoupling and isolation. In the same manner all databases are also isolated.

The application's architecture is composed by thirteen different microservices:

- **Track Logic Service** - Implements all **Track** related **REST API** endpoints.
- **Track Model Service** - Communicates with the **Track DB** to support **Track** related queries.
- **Artist Logic Service** - Implements all **Artist** related **REST API** endpoints.

- **Artist Model Service** - Communicates with the **Artist DB** to support **Artist** related queries.
- **Artist's Tracks Model Service** - Communicates with the **Artist's Tracks DB** to support queries that relate artists and tracks.
- **Artist's Releases Model Service** - Communicates with the **Artist's Releases DB** to support queries that relate artists and releases.
- **Genre Logic Service** - Implements all **Genre** related **REST API** endpoints.
- **Genre Model Service** - Communicates with the **Genre DB** to support **Track** related queries.
- **Release Logic Service** - Implements all **Release** related **REST API** endpoints.
- **Release Model Service** - Communicates with the **Release DB** to support **Track** related queries.
- **Playlist Logic Service** - Implements all **Playlist** related **REST API** endpoints.
- **Playlist Model Service** - Communicates with the **Playlist DB** to support **Track** related queries.
- **Auth Service** - Implements the Authentication process using the **Auth0** framework.

4.2 Technical architecture

The Technical architecture refers to the overall structure and design of the technology components, hardware, networks and data storage solutions that form the foundation of the cloud-native application. The Technical architecture of our cloud-native application can be found seen in Figure 2.

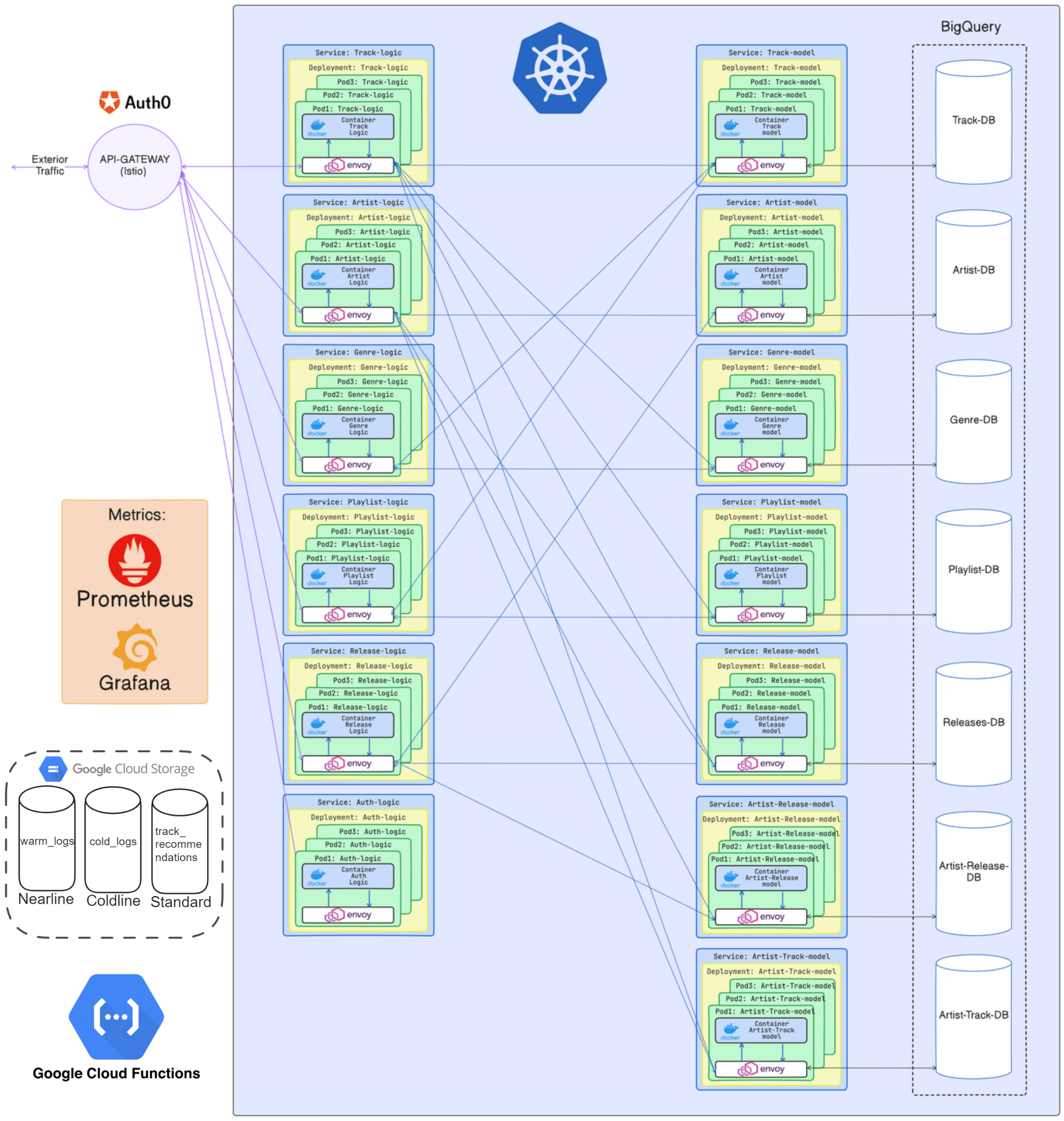


Figure 2: Technical architecture

To ensure that our application is robust, resilient and can handle varying loads, we decided to deploy three replicas of each microservice using **Google Kubernetes Engine** (**GKE**). This setup allows us to balance the load among replicas, provides fault tolerance and enables the use of scaling configurations. Each pod in Figure 2 represents one of those replicas being automatically managed by a **Deployment** instance of the **Kubernetes Engine**.

In our application, we used **Istio Service Mesh** to secure communication between the various microservices and to offer an ingress to expose the **REST API** implemented by those microservices. **Istio** enforces mutual **TLS** for service-to-service communication, ensuring that data transmitted between microservices is encrypted and authenticated. **Istio** automatically injects Envoy sidecar proxies into each microservice pod. These proxies handle the encryption and decryption of traffic, establishing secure mutual **TLS** connections between services. This secure communication is important since the **gRPC** channels used for messaging between microservices were not made secure. Additionally, the **Istio** ingress gateway provided a robust and secure way to expose our **REST API** to external users while allowing us to define authorization policies in conjunction with **Auth0**.

We decided to use **BigQuery** to host data for our cloud-native application because it integrated seamlessly with other **Google Cloud** services that could be used in the future. **BigQuery** also provided robust security features mainly encryption of data at rest and in transit.

To monitor the health and performance of our application, we utilized **Prometheus** and **Grafana** to gather and visualize metrics.

To implement long-term storage of logs we used **two Cloud Storage buckets** and **two Cloud Functions**. We utilized both **Coldline** and **Nearline** storage classes according to our needs.

5 Implementation

5.1 Microservices

The **Logic Services** that expose the **REST API**'s endpoints to the **API Gateway** were implemented using **Flask**, a **Python**'s framework for development of web applications. Like mentioned in Section 4, these microservices communicate with the **Model Services** to query database's data using **gRPC**. The **Model Services** must be able to access the application's data. For the **Model Services** to access the **BigQuery** service it was necessary to firstly, create all the necessary tables using the *.csv* files of the chosen dataset and secondly, configure a service account with permissions that enable the manipulation and retrieval of the data stored in the cloud provider. This service account returns a **JSON** key containing information of said account and its respective permissions. This **JSON** key is then used by the **Model Services** together with the **GCP** project identifier to access the data and tables stored in the **BigQuery** service. The **JSON** key is used to authenticate the **Model Services** accessing the **GCP**'s **BigQuery** service, while the project identifier is used to identify the tables of the database to be accessed by the microservices.

5.2 Authentication with Auth0

The **Auth Service** required the creation of an **Auth0** application to authenticate and authorize users and their requests. The creation of this **Auth0** application required the definition of allowed callback and logout URLs and the URL of the application domain. Besides the creation of the **Auth0** application, it was necessary to register an **API** for **Auth0**. This **API** enables the configuration of permissions in access tokens and roles following **Role Based Access Control** (**RBAC**). We defined the following two roles with these respective permissions:

- **User** - read:artists, read:genres, read:playlists, read:releases, read:tracks, write:playlists.
- **Premium** - read:artists, read:genres, read:playlists, read:releases, read:tracks, write:playlists, write:artists, write:genres, write:releases, write:tracks.

The authentication process starts with a user logging in. At this phase a session state is generated on the **Auth Service** side and stored in a session object and the user is redirected to the **Auth0** login page. After logging in, the **Auth0** service redirects the user to the application's callback URL and the **Auth Service** compares the previously generated state with the state present in the request's URL parameters to mitigate possible **Cross-Site Request Forgery** (**CSRF**) attacks. If the request is valid, the received code is swapped for the access token containing the user permissions and the user is redirected to the application. User's authorization policy was implemented by defining a **RequestAuthentication** resource in **Istio** which was used to enforce that **JSON Web Tokens** (**JWT**) in incoming **HTTP** requests are valid and issued by the identity provider. The configuration made verifies a **JWT** present in a cookie created after a user is logged in. Additionally, the configuration of **AuthorizationPolicy** resources allowed the implementation of the cloud-native application's functionalities control access by users' roles and permissions present in the **JWT** token.

5.3 Secure Channels with Istio

Furthermore, **Istio** was integrated into the system to enhance security over the microservices communication. Firstly, **Istio** was deployed in the Kubernetes cluster managing the microservices. The deployment involved installing Istio's core components, including the **Istio control plane** and enabling **automatic sidecar injection** for the application's namespaces. This setup ensures that every pod in the namespace gets an Envoy proxy sidecar, which handles the communication between microservices. In order to secure communication between microservices with **Istio**, it was necessary to configure a **Virtual Service** object per microservice as well as a **PeerAuthentication** resource which is used to configure mutual **TLS** settings for all services in the mesh when using the STRICT mode, requiring that all service-to-service communication within the mesh be encrypted and authenticated using mTLS.

5.4 Cloud Monitoring

To monitor the health and performance of our application, **Prometheus** and **Grafana** were integrated. **Prometheus** needed its certificates to be able to communicate with the Istio Envoy sidecar proxies to scrape metrics. Additionally, we configured the **Grafana** reverse proxy to keep the original host header, ensuring that **Grafana** could get **Prometheus** as a data source.

5.5 Long-term Log Storage

We implemented **Long-term Log Storage** by following the definition of a Round Robin Database using different storage classes depending on the age of the logs, and aggregation to lower the level of the detail of older logs. Even though this kind of implementation normally aims to handle time series data such as network bandwidth, temperatures or CPU load, we found relevant to give some focus to logs, since it was one of the twelve factors that we weren't significantly exploring. This solution aims to manage old logs with fewer costs than **Google Cloud Logging**. It is based on **two Cloud Storage buckets**, warm_logs (Nearline storage class and Deletion rule 30+ days since the object was created) and cold_logs (Coldline storage class), **two jobs in Cloud Scheduler** (one daily, and another that runs every sunday) and **two Cloud Functions**:

- **daily_logs -** retrieves filtered logs with severity >= 'ERROR' from **Google Cloud Logging** and aggregates repeated logs in a single line containing the log message, number of occurrences, first and last timestamp; storing it in the warm_logs **Cloud Storage bucket** with the day in the **JSON** file name. Several filters were applied in the retrieval query, since some error logs weren't relevant and since some text payload summary logs also included timestamps, impossibilitating aggregation.

- **weekly_logs -** this **Cloud Function** retrieves the logs from the past week (Sunday to Saturday) by accessing the seven files stored in the warm_logs bucket in **Cloud Storage** and aggregating them in a single **JSON** file to the cold_logs bucket.

For the system storage footprint to remain constant over time, a deletion rule could be added to the Coldline bucket but considering the time of the project it wasn't considered relevant, storing it forever as it stands.

The filter query of the logs for the daily_logs cloud function had to undergo frequent updates, due to the unpredictability of log analysis. Some problems occurred due to quota exceeding, while others clearly reflected some possible improvements in our system implementation. One of these was the frequent categorization of GET operations to health or any other endpoints like tracks, etc; with

”ERROR” severity. If this wasn't the case, the query filter of the daily_logs would have to be changed to not only errors, since the system's requests need to be aggregated for the implementation of Content Popularity.

5.6 Content Popularity based in Analytics Aggregation

The function weekly_logs receives the number of top tracks to register in the BODY of the POST method. Since it aggregates the weekly logs counting their repetition occurrences, it is easy to filter any logs containing ”GET /api/tracks/” and selecting, by default, the 20 track ids with higher ”count” value associated. These values are stored in **track_recommendation Cloud Storage bucket**.

5.7 User Based Track Recommendation

We implemented **Track Recommendations** to each user that has playlists, based on the system's trending tracks and the user's playlists recent activity. It executes once a week, recommending 6 tracks, focusing on genres of tracks added to the users playlists in that week (up to the 20 most recent). If the top trending tracks do not contain up to 6 tracks of any of the genres previously selected from the user's playlists, then random tracks of any of the genres selected (excluding any track already selected from the trending) will be added to the recommendation.

The **Cloud Functions** responsible for this feature are weekly_logs, for content popularity documentation, and test, for the rest of the operations.

The test **Cloud Function** executes queries to **BigQuery** using the secret stored in the Secret Manager.

It is assured that the recommendations will not include any track added to the user's playlists in the week being analyzed. However it is possible to recommend tracks added to a playlist more than a week ago. This is because the calculation has in mind recent activity for the recommendation, querying genres of only recent tracks added, while still knowing the specific tracks. The removal of all tracks in playlists from being possible recommendations would require a new query. Since it wouldn't bring any new knowledge, being repetition of previously done steps, and since the cost could be considerably higher, we established the nostalgia feature to let the users remember and re-listen old favorites.

6 Deployment

6.1 GKE

The cloud-native application was deployed by configuring a **GKE** cluster with a with a machine-type=n1-standard-4 and managing different resources. The **GKE** cluster serves as the foundation for deploying the containerized microservices. Deployment and containerization of these microservices required generating the correspondent images, storing and building them. To generate and later build these images, a **Dockerfile** was written for each microservice.

6.2 Artifact Repository

The **Docker** images were then built and pushed to a **GCP** artifact repository service, **Artifact Registry**, where the images are stored. When the **GKE** cluster is initialized, the **Kubernetes** configuration will pull the **Docker** images from **Artifact Registry** to create the various replicas of the microservices. To set up the **Artifact Registry** service, it was necessary to enable the **Artifact Registry API** in **GCP** and create a service account with enough permissions to push and pull the images stored. This service account returns a **JSON** key containing information of said account and its respective permissions. This **JSON** key is stored in a **Secret** object and then used by the **Kubernetes** configuration to pull the necessary images for pod's initialization.

6.3 Data Cleaning

The first step taken to clean the dataset before importing it into **BigQuery** was to choose the *.csv* files and respective columns that were the most relevant for the project and the microservices defined. All the lines that contained empty columns were removed as well as . other any entries in other CVS's that were dependent from those. The final step was the substitution of the original **Beatport's** user ID's by our application's identifiers while also being careful with other CVS's that were dependent on those identifiers. The genre and sub-genre CVS's were merged into one.

6.4 ConfigMap

In our **Kubernetes YAML** configuration, we utilized **ConfigMaps** to provide the necessary information for our **Logic microservices** to discover the services of the **Model microservices** as environmental variables. This setup allowed the logic microservices to establish **gRPC** stubs for communication.

6.5 Deployments and Services

For each microservice there is a **Deployment** that manages a set of pods. These pods are part of a **ReplicaSet** created by the **Deployment**. The **Deployment**'s configuration determines:

- The number of replicas of the **ReplicaSet**.
- The strategy for updates: The **RollingUpdate** strategy allows updating a **Deployment** incrementally by replacing pods one at a time, ensuring that some pods remain available to handle traffic during the update process.
- The **Secret** containing the necessary permissions to pull images from the **Artifact Registry**.
- The image to be pulled from the **Artifact Registry** and pull-policy to create the microservice.

- A liveness probe to check pod's health status.
- The amount of memory and CPU resources to be used by each pod.

Additionally, for each microservice there is a **Service** used to expose them, or, expose the multiple pods running the microservice in the cluster.

6.6 Horizontal Scaling

We used **Horizontal Pod Autoscaler** (**HPA**) that automatically adjusts the number of pod replicas based on observed CPU and memory utilization metrics. This helps ensure that the application scales according to the load it experiences, improving performance and resource efficiency.

6.7 Credentials Management

To securely manage the credentials, we created **Kubernetes Secrets** and .env files (The Twelve Factor App - 3rd factor Config) which were used to store sensitive data such as service account keys, API keys, and other credentials securely within the **Kubernetes** cluster.

6.8 Automatization of the Deployment

Finally, the deployment was automated with a bash script `deploy_cluster.sh` which created a cluster with a machine-type=n1-standard-4, created the **Kubernetes** secrets, installed **Istio** and deployed `kubernetes.yaml`, `istio-gateway.yaml`, `prometheus.yaml` and `grafana.yaml` .

6.9 Domain

To provide a stable external endpoint for the **API Gateway**, we created a static external IP address. We then used No-IP DNS services to map this static IP to `cngroup02.ddns.net`.

7 Test and Evaluation techniques and results

The cloud-native application **REST API** was tested by creating a **Postman**'s request library. This library includes all possible requests that can be made to the cloud-native application deployed on the cloud. This library can be easily imported into **Postman** and can be found at `https://github.com/fcul-cn/group02/blob/main/docs/services.postman_collection.json`. It is important to notice that this library requires defining a cookie for the application's domain containing a valid **JWT** access token and the static IP address of the cloud-native application.

To ensure the robustness and scalability of the cloud-native application, we used **Locust**, an open-source load testing tool to perform a comprehensive stress test. Locust allows us to simulate a high number of concurrent users and generate realistic traffic patterns to assess the application's performance under load. We defined a user behavior **Python** script to mimic a usage scenario that causes a substantial and scalable load generation. After executing this script we observed the application scaling with increasing load. By other words, the number of pods per microservice increased with the increasing load showing the **HPA** configuration taking effect. The results can be seen in Figures 3, 4 and 5. From the results obtained we concluded that the application demonstrated perfect reliability since the service was not denied during the overload period. On the other hand, performance was not so good since the average response time of microservices was always greater than one second. This average time also increases considerably when the number of requests processed simultaneously goes up.

8 Discussion about possible deployment configurations their performance/reliability/costs

The cost evaluation for hosting the application was conducted using the GCP Cloud Service Provider's cost calculator. Following, we present the costs associated with hosting the application:

GKE Configuration

- 3 nodes running for a total of one month (2190 total hours per month)
- 1 Single-zone Cluster
- 0 Regional Clusters
- Machine type: n1-standard-4
- Region: europe-west-4

BigQuery Configuration

- Edition: Standard
- Region: europe-west-4
- Active logical storage: 2 GiB
- Maximum Slots: Small (100 slots)
- Average utilization of autoscale slot: 0%

Based on these configurations, the **GKE** configuration equalled an amount of 542.30$ per month while the **BigQuery** configuration does not add any additional costs. During the cost optimization process, the number of **GKE Regional Clusters** was reduced from 3 to 1, as the previous number of clusters was deemed unnecessary. The pods were using an excessive amount of memory and CPU for the size of the project, with memory: "500Mi" and CPU: "200m". Therefore, we downgraded the machine from n1-standard-8 to n1-standard-4 and reduced the memory and CPU values to 100Mi and 80m, respectively. The pods were using around 70Mi of memory, so we considered using 100Mi to have some margin.

Cloud Storage Configuration: warm_logs bucket

- Location Type: Multi-Region
- Location: eu
- Storage Class: Nearline Storage
- Total amount of storage: 1 GiB
- Data retrieval amount: 1 GiB
- Data Transfer within Google Cloud: 1 GiB

Cloud Storage Configuration: cold_logs bucket

- Location Type: Region

- Location: europe-west1
- Storage Class: Coldline Storage
- Total amount of storage: 10 GiB
- Data retrieval amount: -
- Data Transfer within Google Cloud: -

Cloud Storage Configuration: track_recommendations bucket

- Location Type: Region
- Location: europe-west1
- Storage Class: Standard Storage
- Total amount of storage: 0.1 GiB
- Data Transfer within Google Cloud: -

The amount of storage for each **Cloud Storage** bucket was not extensively calculated since the number of logs and the number of users are unpredictable. Since the weekly_logs **Cloud Function** retrieves the logs from the warm_logs bucket, this value was associated. The estimate of costs was 0.07$ for warm_logs bucket, 0.04$ for cold_logs bucket and 0.02$ for track_recommendations bucket, totalling 0.13$ for **Cloud Storage**.

Cloud Function Configuration: daily_logs

- Cloud Function Version: 1st gen
- Region: europe-west1
- Requests per month (millions): 0.00003
- Average execution time per request (ms): 45000

Cloud Function Configuration: weekly_logs

- Cloud Function Version: 1st gen
- Region: europe-west1
- Requests per month (millions): 0.000005
- Average execution time per request (ms): 3500

Cloud Function Configuration: test

- Cloud Function Version: 1st gen
- Region: europe-west1
- Requests per month (millions): 0.1
- Average execution time per request (ms): 11000

The estimate was done having in mind that daily_logs runs once a day, averaging 30 executions a month. The execution time varies a lot depending on the system traffic. On some days we had about 2000 logs, executing in about 4000-6000 ms. In other days due to intense testing 37000 logs were generated, taking the cloud function around 40000-45000 on average to finish execution. The cost associated to daily_logs function is 0.01$. The weekly_logs function executes once every week. Since the number of executions is extremely low and the execution time is no where near the necessary to have costs involved, this cloud function cost is inapplicable. The track_recommendations function's cost is completely arbitrary since it depends mostly in the number of users. Estimating 25000 users, whose recommendations are calculated once a week, the cost associated is 5.06$.

Since Google Cloud Scheduler's free tier has three free jobs, it didn't have any costs associated.

Secret Manager only manages the key for **BigQuery** access so it also doesn't have costs associated. However, if the number of accesses exceeded 10000 (each user recommendations retrieves 4/5 times a month, so more than 2500 users would suffice), then it would have a small cost.

With this, the total monthly cost associated with hosting the application on **GCP** was 547.50$ per month.

9 Conclusions

With the making of this project we were able to develop a cloud-native application that ended up touching all topics of the **Twelve-factor app** methodology. We reached all previously defined and our own objectives related to deployment, service exposure, load balancing, automatic scaling, role-based access, authentication using an identity provider, metrics and logs monitoring and cost evaluation. The architecture of the system allowed us to explore three different architectural patterns of microservice based cloud-native applications. In the other hand, the **Technical architecture** offered us a whole bunch of possibilities of different topics of specialization but we to focus on the topic of security justifying our choice to use **Istio**. **Istio** offered end-to-end encryption and identity verification for service-to-service communication and in conjunction with the **Auth0** framework was used to implement authorization policies based on user permissions and roles.

Despite achieving our objectives, there are still possible improvements or contributions that could have been made. These include, the creation of more automatic tests and the automation of **Docker** images deployment to the **Artifact Registry**. The categorization of GET operations to health or any other endpoints with "ERROR" severity might have also undergone some revisions, to improve the logs analysis and consistency. Additionally, we did not make use of some of the business capabilities listed in the beginning phases which caused our base application to be simple in concept. To compensate, we focused our efforts in the technical and configuration of the cloud environment which is the main focus of this project. Also, another existing problem is the use of **BigQuery** for our database. The biggest issue with **BigQuery** is that it takes time to register an update to the data. For example, if we insert new data, we have to wait a certain amount of time so that the data can be updated or deleted. A possible fix is using **Cloud SQL**, but this will come with more costs.

10 Contributions

Luís Viana (62516)

- Implemented Microservices
- Established Kubernetes cluster (Deployments, Services)
- Integrated Ingress (later removed)
- Implemented long-term storage of logs using Cloud Storage, Functions and Scheduler
- Implemented Content Popularity Documentation and Track Recommendation
- Calculated cloud costs

Guilherme Santos (62533)

- Implemented Microservices
- Established Kubernetes cluster (Deployments, Services, HPA, RollingUpdate)
- Integrated Istio Service Mesh
- Incorporated Prometheus and Grafana Monitoring
- Automated the deployment of the cluster
- Performed Stress-Testing with locust

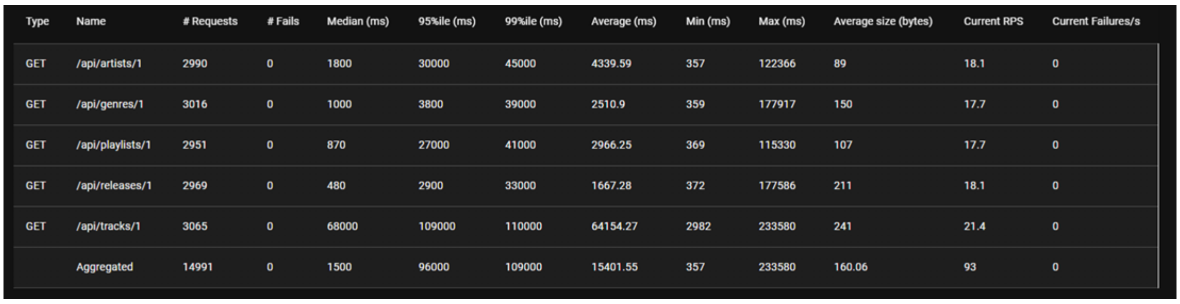
João Magalhães (62546)

- Implemented Microservices
- Deployed microservices in Docker containers
- Performed CSV data cleaning
- Integrated BigQuery
- Integrated Auth0

André Santos (62754)

- Implemented Microservices
- Deployed microservices in Docker containers
- Defined Authorization policies based on permissions
- Integrated Auth0
- Performed Stress-Testing with locust
- Calculated cloud costs (FinOps)

11 Appendix



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/api/artists/1	2990	0	1800	30000	45000	4339.59	357	122366	89	18.1	0
GET	/api/genres/1	3016	0	1000	3800	39000	2510.9	359	177917	150	17.7	0
GET	/api/playlists/1	2951	0	870	27000	41000	2966.25	369	115330	107	17.7	0
GET	/api/releases/1	2969	0	480	2900	33000	1667.28	372	177586	211	18.1	0
GET	/api/tracks/1	3065	0	68000	109000	110000	64154.27	2982	233580	241	21.4	0
	Aggregated	14991	0	1500	96000	109000	15401.55	357	233580	160.06	93	0

Figure 3: Locust stress testing number of requests

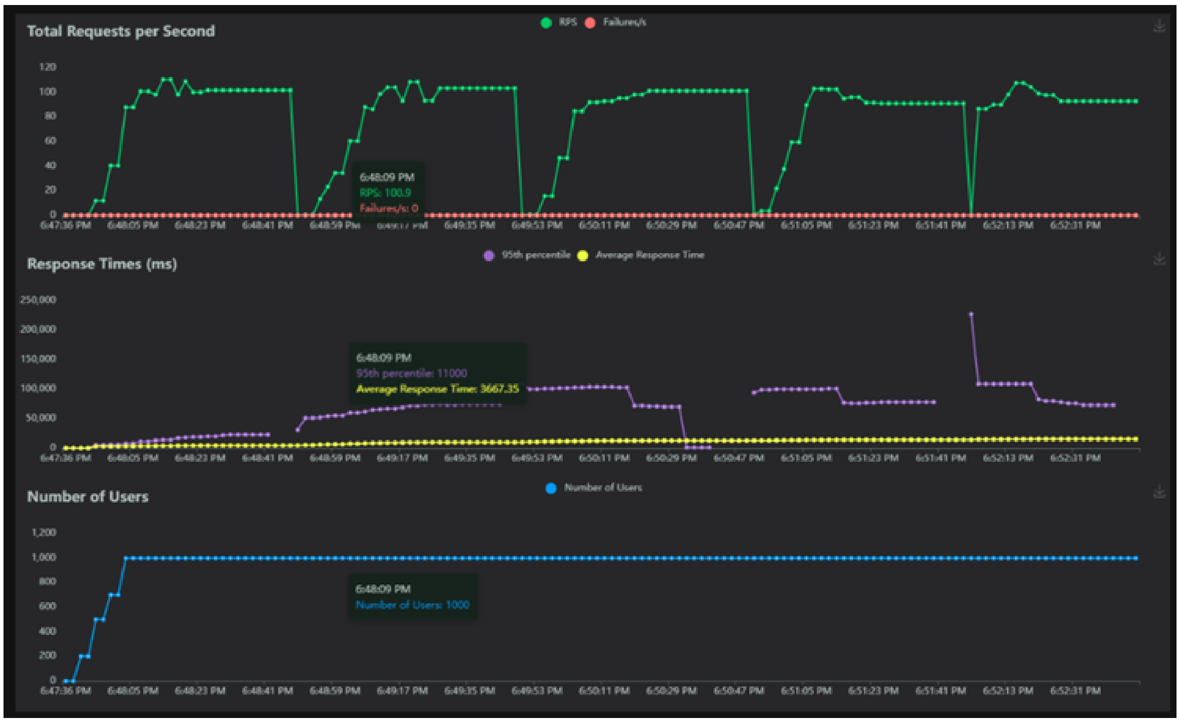
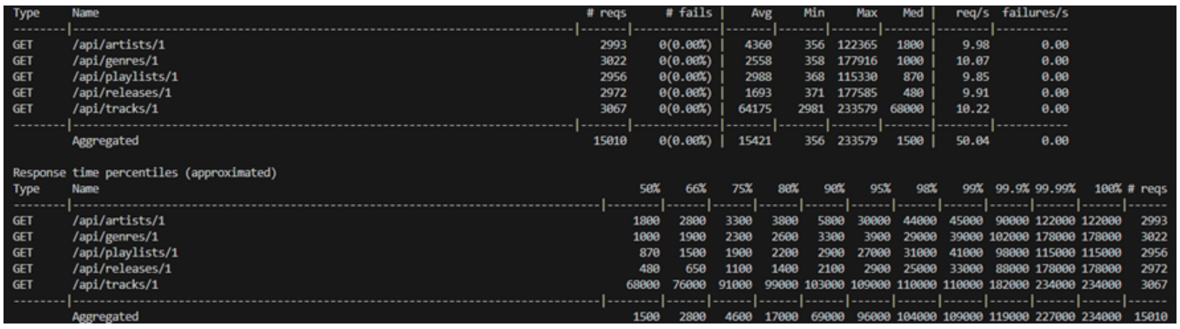


Figure 4: Locust stress testing response total requests



```
Type     Name                                                                          # reqs      # fails |    Avg     Min     Max     Med |   req/s  failures/s
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
GET      /api/artists/1                                                                  2993     0(0.00%) |   4360     356  122365    1800 |    9.98        0.00
GET      /api/genres/1                                                                   3022     0(0.00%) |   2558     358  177916    1000 |   10.07        0.00
GET      /api/playlists/1                                                                2956     0(0.00%) |   2988     368  115330     870 |    9.85        0.00
GET      /api/releases/1                                                                 2972     0(0.00%) |   1693     371  177585     480 |    9.91        0.00
GET      /api/tracks/1                                                                   3067     0(0.00%) |  64175    2981  233579   68000 |   10.22        0.00
--------|----------------------------------------------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
         Aggregated                                                                     15010     0(0.00%) |  15421     356  233579    1500 |   50.04        0.00

Response time percentiles (approximated)
Type     Name                                                                                  50%    66%    75%    80%    90%    95%    98%    99%  99.9% 99.99%   100% # reqs
--------|--------------------------------------------------------------------------------|--------|------|------|------|------|------|------|------|------|------|------|------
GET      /api/artists/1                                                                       1800   2800   3300   3800   5800  30000  44000  45000  90000 122000 122000   2993
GET      /api/genres/1                                                                        1000   1900   2300   2600   3300   3900  29000  39000 102000 178000 178000   3022
GET      /api/playlists/1                                                                      870   1500   1900   2200   2900  27000  31000  41000  98000 115000 115000   2956
GET      /api/releases/1                                                                       480    650   1100   1400   2100   2900  25000  33000  88000 178000 178000   2972
GET      /api/tracks/1                                                                       68000  76000  91000  99000 103000 109000 110000 110000 182000 234000 234000   3067
--------|--------------------------------------------------------------------------------|--------|------|------|------|------|------|------|------|------|------|------|------
         Aggregated                                                                           1500   2800   4600  17000  69000  96000 104000 109000 119000 227000 234000  15010
```

Figure 5: Locust stress testing response times

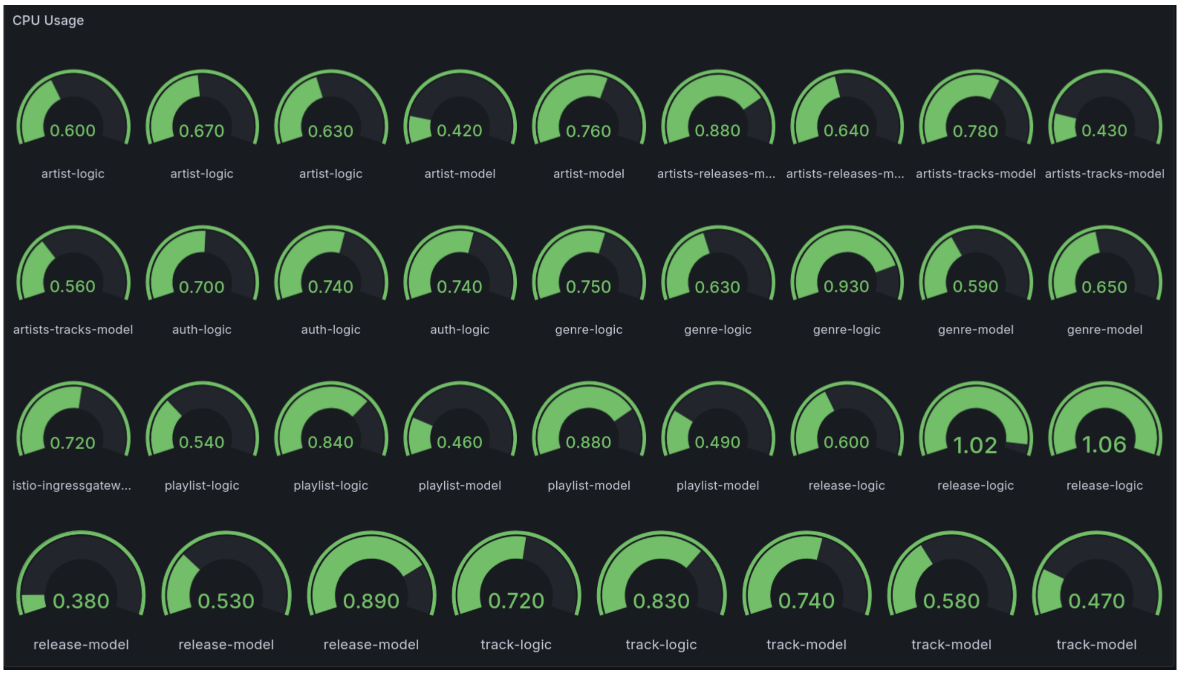


Figure 6: CPU Usage metrics of each pod replica

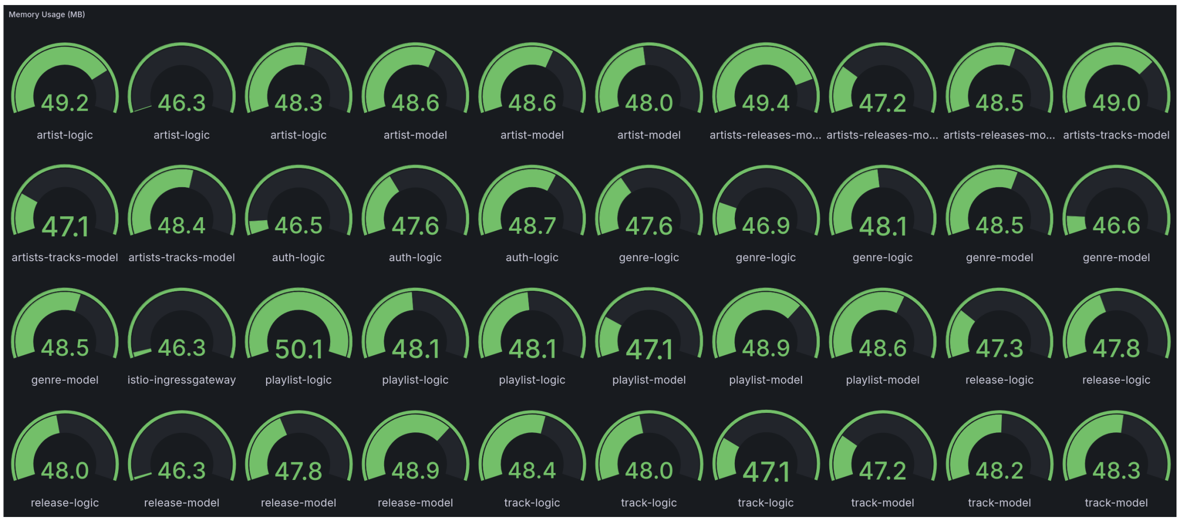


Figure 7: Memory Usage metrics of each pod replica (MB)

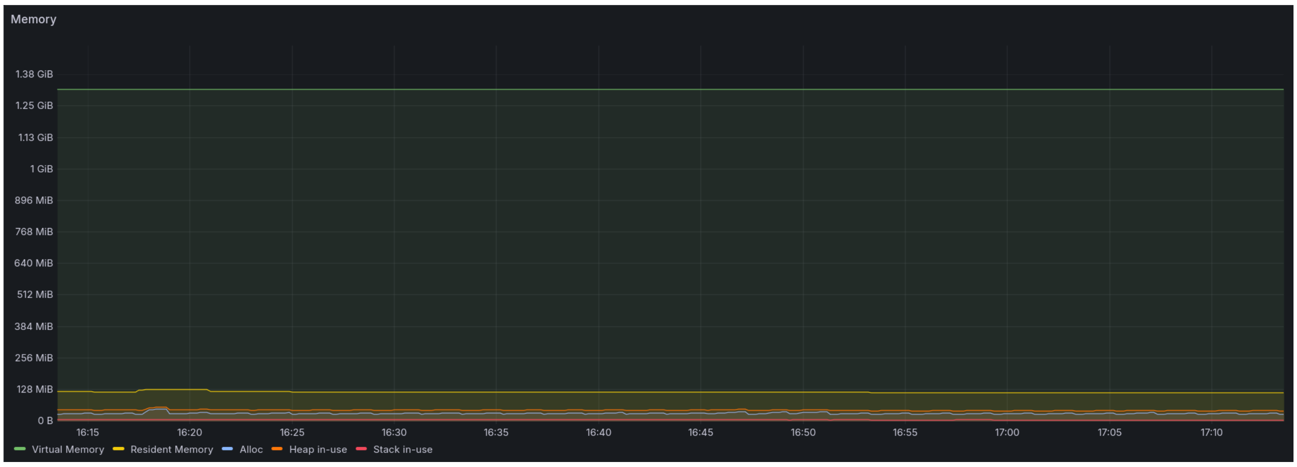


Figure 8: Total Memory Usage

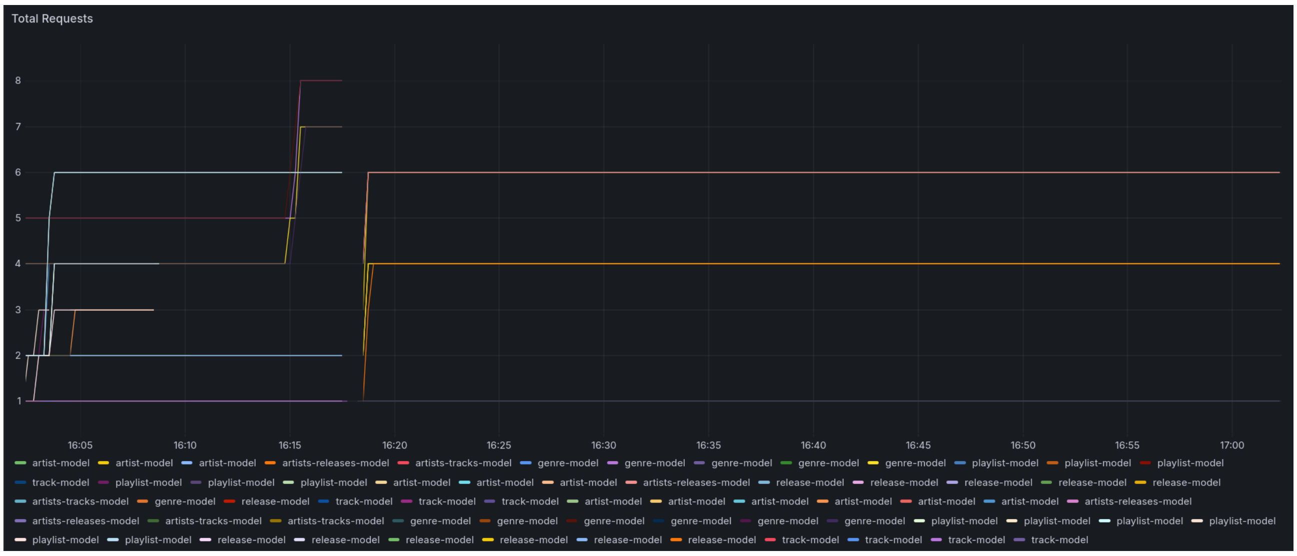


Figure 9: Total Requests of each Pod

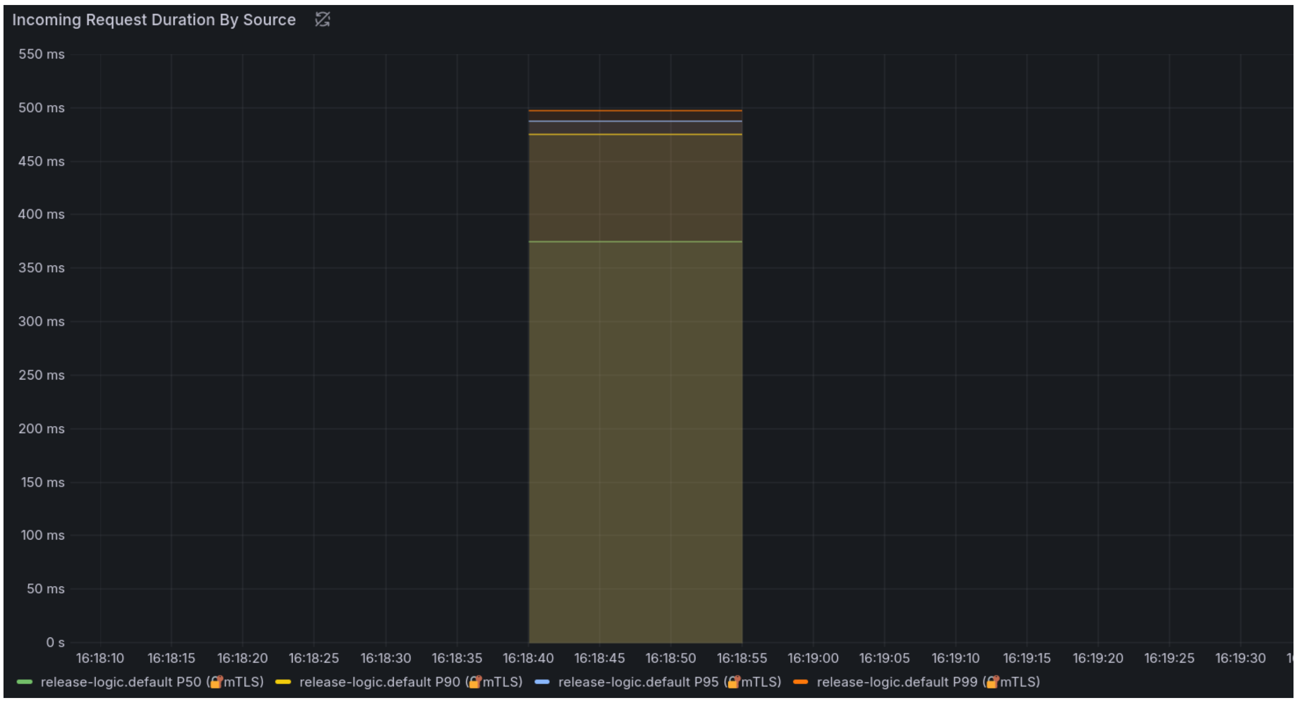


Figure 10: Request Duration (ms)

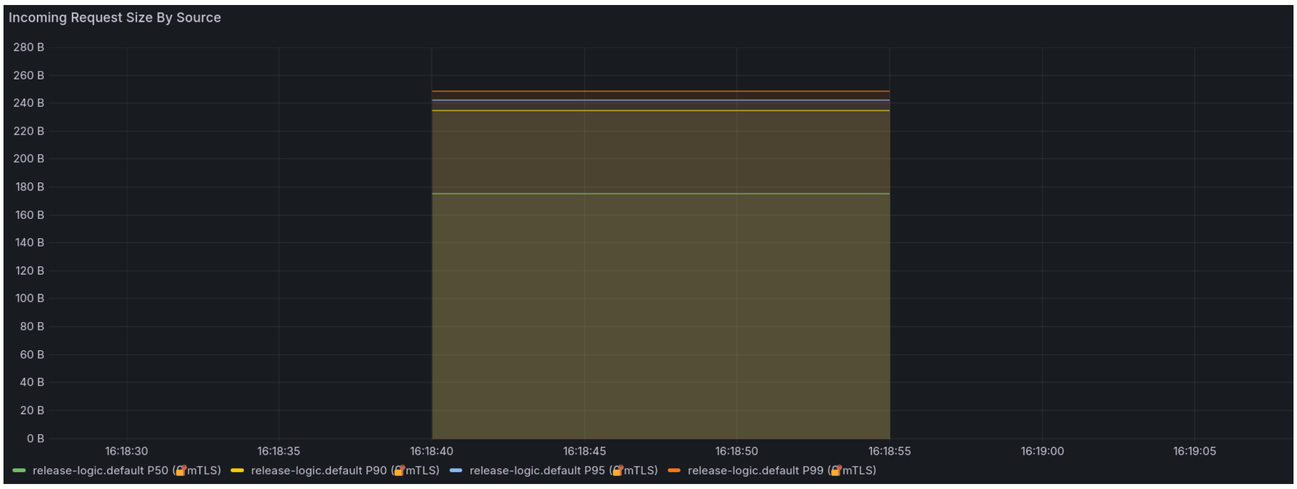


Figure 11: Request Size (B)

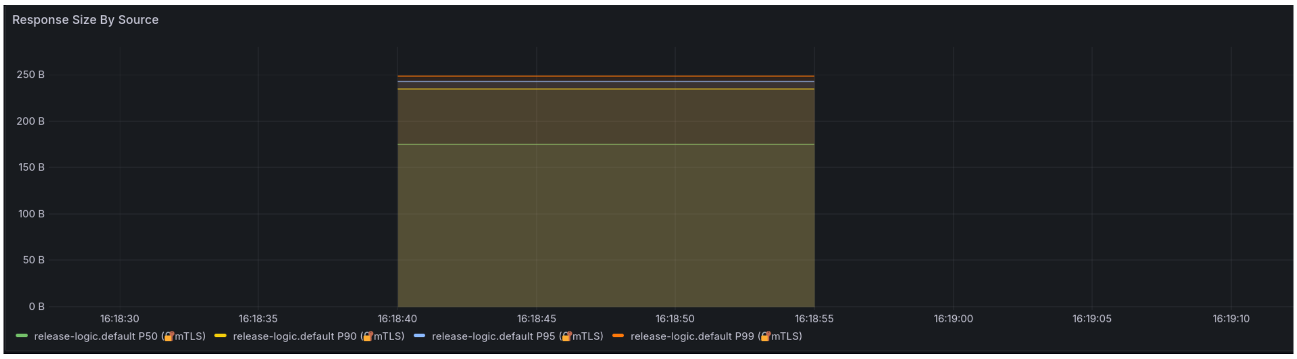


Figure 12: Response Size (B)