Data Engineering – 2

Team members: Anton Jesu Vikranth

Lakshith Reddy Baddula

Kiran Mulawad

Abstract:

This project entails developing and deploying a comprehensive end-to-end data pipeline, leveraging the Google Cloud Platform (GCP) ecosystem, Dockers and Supabase.

The primary objective is to generate visualizations for a Label and Records company, so the Record producer can use these reports to shortlist artists, songs, when to release the songs and type of songs to released based on trends.

The workflow commences with the extraction of relevant data which is a simulation of a batch data using Spotify API. The data from Spotify is stored into kafka and Google Buckets. A notification function uses kafka to give song suggestions. Supabase is used as a database to store data from kafka that could be connected to Google Looker Studio for visualization.

Introduction:

This workflow involves a data pipeline that utilizes the Spotify API to extract relevant data, such as the top 50 ranks, recommended songs, song details, and analysis attributes. This data is temporarily stored in Kafka and refreshed every 24 hours with new viral songs. Lyrics of the songs, sourced from GeniusLyrics, are stored in Google Cloud Storage buckets.

A notification function checks Kafka daily for recommendations that match the user’s playlist. If matches are found, the user receives an email suggesting the addition of these viral songs to their playlist.

The primary database for this workflow is provided by Supabase, a Platform as a Service (PaaS). The data from Kafka is stored in a PostgreSQL database with a defined schema. This PostgreSQL database is then connected to Google Looker Studio for data visualization. This system ensures a dynamic, user-oriented music recommendation service with efficient data management and insightful visualizations.

User Story:

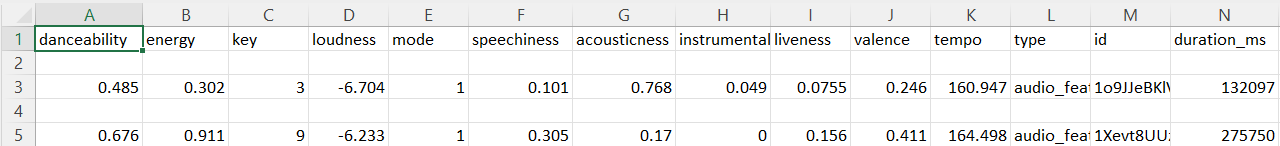
Record Producer:-

* Wants to hire trending artists based on spotify viral hits.
* Release the type of songs that are popular in the market at present.
* Wants to hire the next big thing and predict the music market based on previous results.

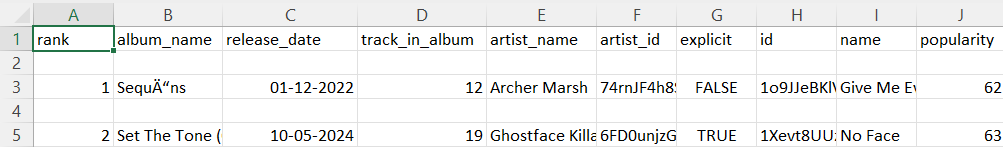
Music/Talent Analysts:-

* Analyze apps such as spotify to gain meaningful insights.
* Suggestion regarding artists and songs.
* Suggestions regarding when and what type of songs to release.
* Suggestions on artists to be hired.
* Suggestions on what type of songs and tracks that can be played at specific events.

Dataset:

Track Analysis:-

Track Details:-



Track Analysis datapoints:

Are more like music analysis attributes.

* **Danceability:** Describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity.
* **Valence:**Describes the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
* **Energy:** Represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale.
* **Tempo:**The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece, and derives directly from the average beat duration.
* **Loudness:**The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks.
* **Speechiness:**This detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value.
* **Instrumentalness:**Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”.
* **Liveness:**Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live.
* **Acousticness:** A confidence measure from 0.0 to 1.0 of whether the track is acoustic.
* **Key:** The estimated overall key of the track. Integers map to pitches using standard [Pitch Class notation](https://en.wikipedia.org/wiki/Pitch_class) . E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on.
* **Mode:** Indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
* **Duration:** The duration of the track in milliseconds.

Track Details Datapoints:-

**Rank:** Rank of each song in the track.

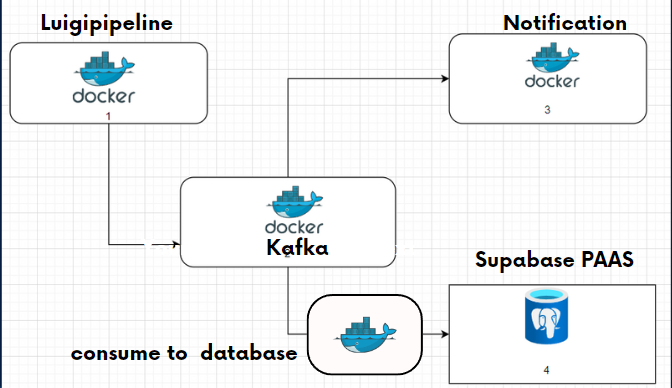
**Album\_name:** Name of the album of which the song is a part of.

**Release\_date:** Date at which the album was released.

**Track\_in\_album:** Specific song that is part of group of songs (album).

**Artist\_name:** Name of Artist.

**Artist\_id:** Unique id given to an artist.

Pipeline:

We deployed four docker containers.

The first container fetches data and pushes it into kafka and google storage bucket.

The second container is the kafka zookeeper instance.

The third container notifies user if a song in viral 50 suits his taste.

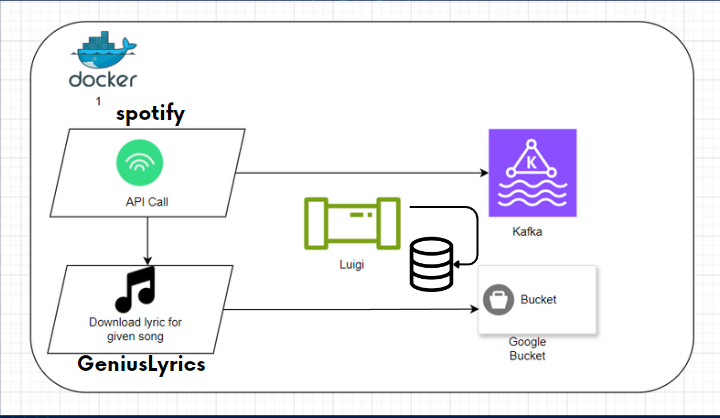
The Fourth container consumes data from kafka and inserts into supabase PaaS.

These four containers work together to create a dynamic, user-oriented music recommendation service. They ensure efficient data management, provide personalized recommendations, and enable insightful data visualizations through Google Looker Studio. This system is a great example of how various technologies can be integrated to deliver a comprehensive solution.

We choose to use docker because of its potability, consistency, scalability and deployment applications in different environments as we did not want to be dependent on a particular cloud platform.

Docker is also lightweight so we did not have to deploy a separate cloud instance for different microservice that we made.

Phase 1:

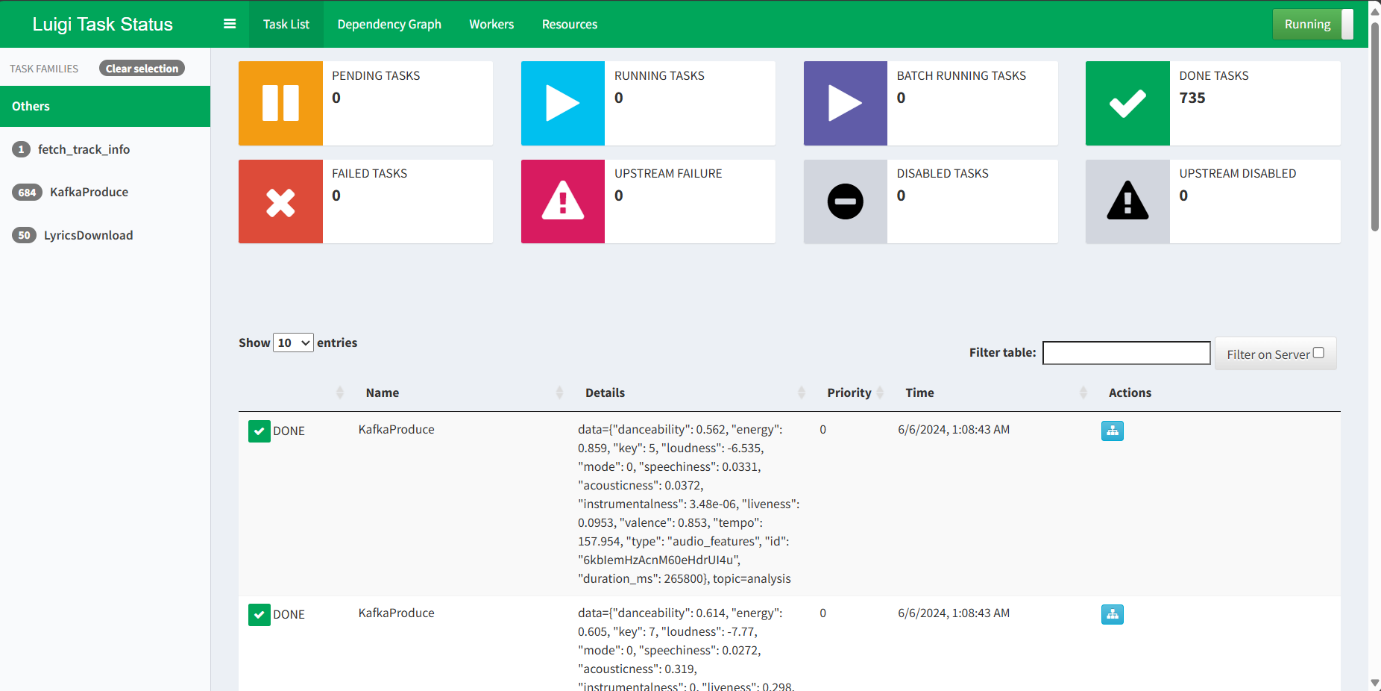


**Data Fetching Container**: The first container is responsible for fetching data from the Spotify API. This data includes information about the top 50 viral songs, song details, analysis attributes, and recommended songs. Once the data is fetched, it is pushed into two different storage systems. The first is Kafka, a distributed streaming platform that is used for storing and processing streams of records in real-time. The second is a Google Cloud Storage bucket, where the lyrics of the songs (sourced from GeniusLyrics) are stored for future use.

Luigi: -

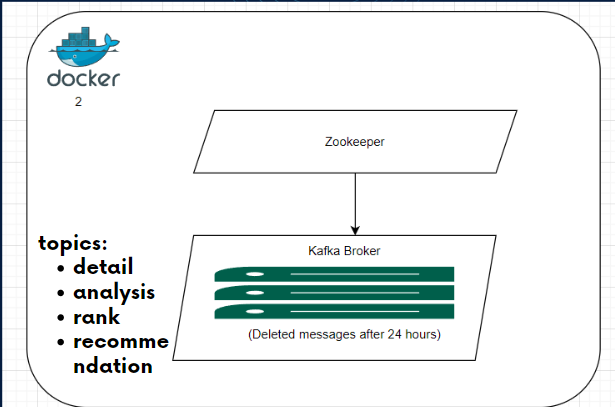
[Luigi](https://luigi.readthedocs.io/en/stable/index.html) is a [Python](https://www.python.org/) package that manages long-running *batch processing*, which is the automated running of data processing jobs on batches of items. Luigi allows you to define a data processing job as a set of dependent tasks. For example, task B depends on the output of task A. And task D depends on the output of task B and task C. Luigi automatically works out what tasks it needs to run to complete a requested job.

Overall Luigi provides a framework to develop and manage data processing pipelines. It was originally developed by [Spotify](https://www.slideshare.net/erikbern/luigi-presentation-nyc-data-science), who use it to manage plumbing together collections of tasks that need to fetch and process data from a variety of sources. Within Luigi, developers at Spotify built functionality to help with their batch processing needs including handling of failures, the ability to automatically resolve dependencies between tasks, and visualization of task processing. Spotify uses Luigi to support batch processing jobs, including providing music recommendations to users, populating internal dashboards, and calculating lists of top songs.

The above image is the screenshot from Luigid which is basically a visualization tool for luigi. It is seen that 735 tasks have been completed. These tasks include 1 fetch\_track\_info task, 684 KafkaProduce tasks and 50 LyricsDownload tasks.

Main reason we choose luigi was because of its leightweight compared to Airflow as it does not contain robust UI and its own trigger system. That suited our use case as we wanted to build a docker container which occupies minimum resources.

Phase 2:



**Kafka Zookeeper Instance**: The second container hosts a Kafka Zookeeper instance. Zookeeper is a service used by Kafka for maintaining configuration information, providing distributed synchronization, and providing group services. It helps in coordinating and managing the Kafka brokers, ensuring the correct operation of your Kafka cluster.

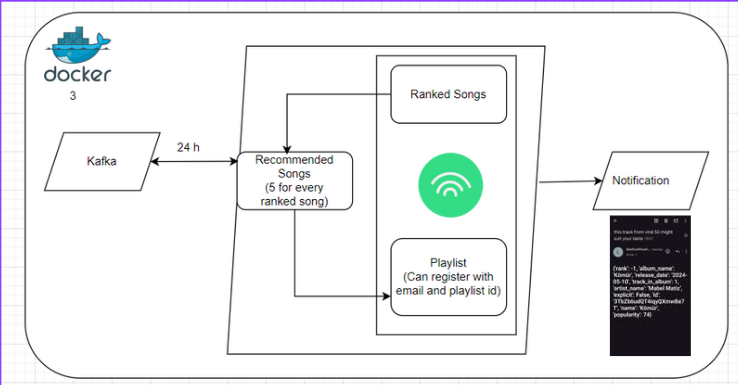
In this container we changed the retention period of message in kafka by modifying the docker compose file.

These are the topics we were broadcasting in kafka:-

* Detail
* Analysis
* Rank
* Recommended

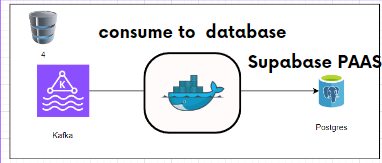
Here we are going with one broker configuration as there are only two consumers, and the data itself is not that critical as anyway we can resend a call to API if kafka fails. in future if we decide to scale up notification container as notification user might increase we might consider going for more than one broker.

A Kafka broker is a server that runs an instance of Kafka. It is responsible for maintaining the list of consumers for each topic, as well as managing the storage of messages for each topic. A Kafka cluster typically consists of multiple brokers, all of which work together to provide a fault-tolerant and scalable messaging system.

Phase 3:

**User Notification Container**: The third container is designed to enhance the user experience by providing personalized song recommendations. It checks Kafka daily to see if any of the viral songs match the user’s playlist. If a match is found, it triggers a notification to the user, suggesting that they might enjoy the viral song. This feature ensures that users are always up-to-date with the latest trends that suit their musical taste.

Each ranked song is paired with five recommended songs. The recommended songs are compared to songs in a playlist of a particular email id or spotify id. If the recommended songs are present in the playlist then the ranked song linked to said recommended songs are given as a notification to the user.

Phase 4:

**Data Consumption and Storage Container**: The fourth container is responsible for consuming the data from Kafka and inserting it into a PostgreSQL database hosted on Supabase, a Platform as a Service (PaaS). Supabase provides a scalable and flexible platform for application development, and in this case, it is used as the primary database for storing the fetched data. This container ensures that the data is readily available for further processing and analysis.

Data is loaded from dockerized kafka into postgres using PaaS (Platform as a Service).

We use supabase because:-

* Supabase is cheaper than GCP
* It has tier for experimentation
* We do not want to depend on a single cloud provider (GCP)

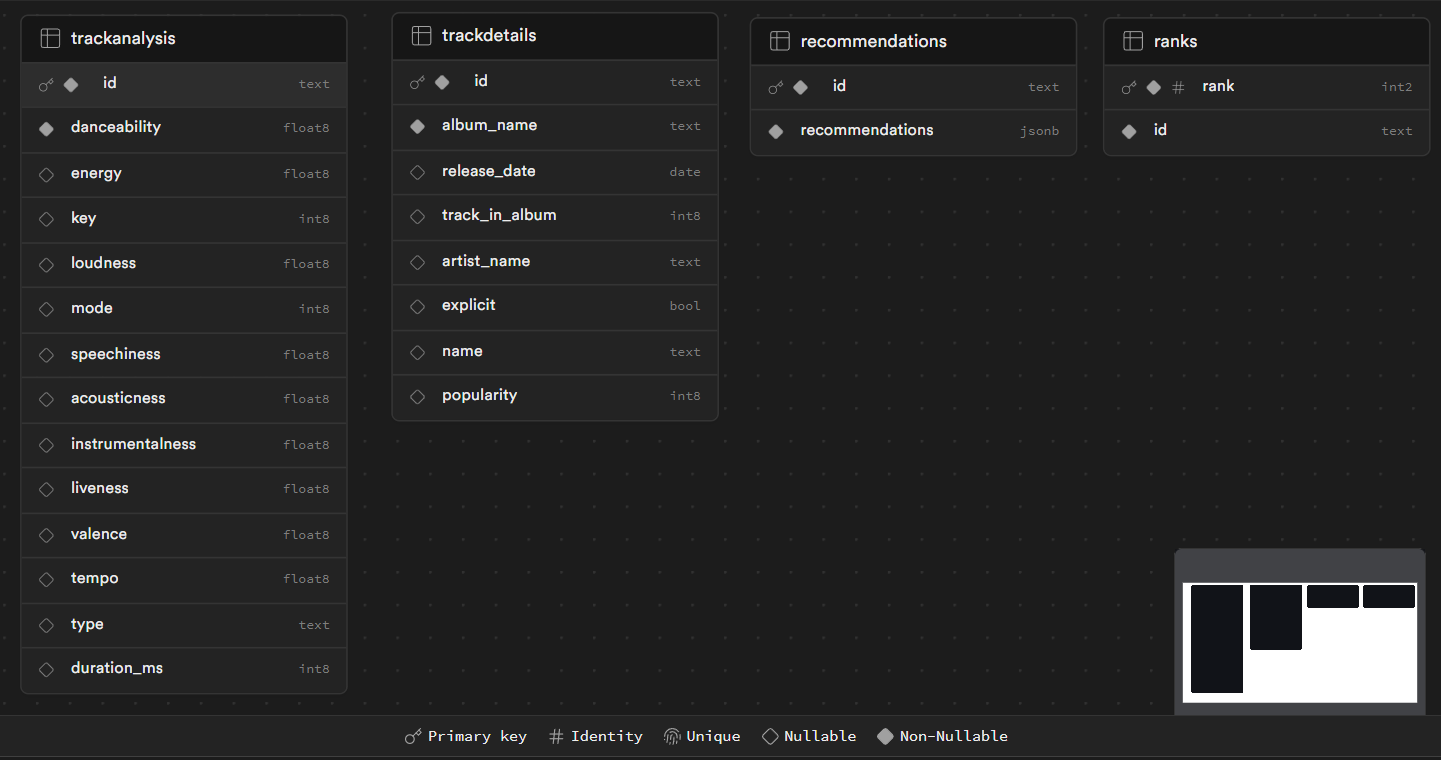
Schema (four tables):-

Rank: Consists of rank and id (top 50 ranked songs).

Recommendations: Recommended songs and their ids (5 recommended songs per ranked song).

Trackdetails: Consists of general information such as album\_name, artist\_name etc. Consists of both ranked and recommended songs.

Trackanalysis: Consists of music analytic attributes such as dancability, energy, loudness, speechiness etc.

Fig. Schema

Major Problems Faced:

-Using track lyrics data stored in google bucket which had the .txt file extension. The track lyrics were uploaded from the geniouslyrics api into a google bucket. When we tried to use the said bucket in BigQuery we Could not select the entire bucket folder as one. Instead we could only select the lyrics which were in .txt format one at a time. We tried to convert the entire bucket into .csv format using a Cloud Function feature.

-While using Spotipy which is an unofficial python SDK, it was caching the api token which was created by client id and client password that we had given to it. This access token had an expiration time that we were not aware of. Since it was caching those token , after the second day the luigi pipeline container stopped working. After extensive research we found this out and deleted the cached file everyday before the luigi pipeline would run.

-We had a problem integrating that data from poastgres into Looker Studio. We could