ADSDB

Data Science End-to-End Project

P1: Multi-Modal Data Management

Pelle Hvidbjørn Hartvig Andersen
Diego Velilla Recio
Alex Bueno León

30th October 2025

How to Run the Project

In order to run the project, follow these steps:

1. Clone the GitHub repository. (Note that the project uses a private *.env* file that has been sent via email to *Marc Maynou*, but it is not present in the public repository).
2. Move into the project folder. If not present yet, move the .env file into the project folder (Make sure it is named *.env*, not just *env*).
3. Boot up Docker inside your machine.
4. Run the following command to build the necessary Docker images:

   ```
   docker compose build
   ```
5. Run the following command to boot up the Streamlit interface:

   ```
   docker compose up -d streamlit
   ```
6. Open your browser and go to:

   ```
   http://localhost:8501
   ```
7. Run the pipeline from Streamlit's UI by clicking the *run pipeline* button. Due to the large amount of data managed, a single execution of the whole pipeline usually takes around 1h to complete.

   a) If the user can not afford to run the whole pipeline, we provide the final version of the exploitation zone data attached to the *.env* email, so testing the tasks is still possible. In order to do it, first log in into MinIO in the following URL using the credentials provided in the .env file:

   ```
   http://localhost:9001
   ```

   b) Then create a new bucket and name it *exploitation-zone*.

   c) Upload the uncompressed *json* and *media* folders inside the bucket. One can just drag them into the MinIO web UI.

   d) Go to the Streamlit UI and just run the *exploitation zone* part using the drop-down menu (5 min).

   e) Now one can run both the similarity search and chatbot without issues.
8. In order to run the similarity search pipelines, we have provided some sample data inside the *test_data* folder. However, the user is more than welcome to try with data of their own.
9. Once finished, run the following command to delete all Docker images:

   ```
   docker compose down
   ```

Contents

1. Introduction

This document details the design and development of an **end-to-end, multi-modal data pipeline**, built following DataOps principles. The pipeline follows a **zone-based architecture**, which progressively prepares data for multi-modal tasks across different layers: Landing, Formatted, Trusted, and Exploitation zones. In our specific use case, our main data source is Steam's video game API, which combines text, image, video, and tabular data.

In order to efficiently store the data to be processed in each zone inside the data warehouse, we have decided to locally host a **MinIO** instance, which is a lightweight and S3 compatible open-source storage system. In addition, since the final layer is designed to support three advanced, analytic tasks: same-modality similarity search, multi-modal similarity search and Retrieval Augmented Generation, we have the need of storing embeddings of our data in a vector database. For this, we have chosen to host a **ChromaDB** instance in the Exploitation Zone.

The instructions for running the project can be found in the **README.md** file inside the project folder.

2. Context and Data

This section defines the context and domain of this project, and describes the different data sources used in the pipeline.

2.1. Domain and Problem

This project focuses on the world of **video games**, specifically, the main data source is **Steam**, a digital platform of video game distribution developed by Valve Corporation. Steam is widely popular because it allows users to download and play any video game from its massive archive.

Steam is the largest platform for video game entertainment worldwide, as users can play whatever they want without having to step outside. However, for those who are more passionate, there is no way to fully explore the large number of games the platform has to offer by only using the available filters provided. For this reason, we have developed an application that allows Steam users to **find video games** based on a description, an image, or a video. Furthermore, we have also implemented a **Game Recommendation Assistant** that, given a user query and our repository of video game data, helps the user find the most suitable games to play.

2.2. Data Sources

This section defines the main data sources used in the data warehouse pipeline.

- **Steam API**: The Steam API is the official interface of Steam to get the information about the games in the platform. This API provides us with general information about the game, its contents and a score of each game. All multimodal data (text, image and video) has been obtained through this API. For a full listing of all Steam API's fields, refer to the Appendix A.
  - **Text**: Steam's API provides three different video game descriptions: *detailed description, about the game, short description*. Each one should have a different use and meaning. However, after a quick scan, we can see that these are not consistent. We can find several cases where these descriptions are being utilized to promote new content or game news, making them an unreliable source of the game context. To overcome this issue, we have used an external Large Language Model to engineer a new feature called *final description*, that compresses all useful game information into a single text field. This topic will be further explored when talking about the Exploitation Zone 3.1.

  - **Images**: For our image data, we will take the first 5 screenshots available from the *screenshots* field. We have chosen to save only 5 images for storage limitations. However, this project can be further scaled and accept all game screenshots or even more by mirroring, rotating or applying other transformations to the initial set of images.
  - **Videos**: We will get our video data from the *movies* field. Just as stated before, we are only keeping 1 video, but this also can be changed.
- **SteamSpy API**: SteamSpy is an unofficial website that provides some Steam video game information that can not be directly accessed via the official API. With it, we will extend the tabular data obtained before by adding information about the popularity of each game with metric fields like *estimated owners* and *median playtime*.

It is also worth noting that, as of today, one can find more than 100,000 video games in Steam, each having its own images and videos. Since storing and processing all that data locally would require of dedicated hardware, we decided to narrow the scope of the project down to just the **100 highest rated games** in Steam, sorted by *Metacritic score*.

Lastly, the scraping tool used to get data from Steam and Steam API has been adapted from this open-source scraper.

2.3. Planned Usage

The obtained multi-modal data (JSON files, text, images, and videos) will be used as the foundation for the three analytical tasks implemented in the project. Our objective is to use this multi-modal data to build a system that helps the user find a new video game to play by completing these tasks:

- **Task 1 + 2 (Same-modality / Multimodal Similarity Search)**: We will use a CLIP model to generate multimodal embeddings for text, image and video. With this, the user will be able to input text, image or video data and set what type of data are they interested in between video game descriptions, screenshots or trailers. After some waiting, they will be presented with the most similar data stored in our database.
- **Task 3 (Generative tasks)**: We will combine our multimodal search capabilities with a state-of-the-art Large Language Model to build a chatbot tasked with helping the user find their next favorite game.

3. The Data Management

The data management pipeline has been separated in four different zones: **landing**, **formatted**, **trusted** and **exploitation zone**. Each zone has its own purpose, and therefore, its own subfolder inside the project. This isolation of each step of the pipeline really enhances the development process by simplifying debugging and improving maintainability. Inside each folder, one can find all the Python scripts that are used in the pipeline, as well as a bash scripts to run them sequentially. Also, the *global_scripts* folder contains one last bash file that runs every other bash script, running the whole pipeline from start to finish.

Most of the pipeline implementation is local, however, since there are not LLMs lightweight enough to fit in this project, all tasks that require LLM usage have been performed using **Google's Gemini Inference API**.

To present the final tool to the end user, we have also implemented a simple, yet charming, **Streamlit UI** where one can run either run the pipeline as a whole or by parts. Once the data has flown through our data warehouse, we encourage the user to check out the **similarity search tool** and the **game recommendation assistant** in order to find the best next game to play.

Lastly, while developing the code for this project, we have used this GitHub repository for version control and task management with GitHub issues.

3.1. Landing Zone

This initial zone is used to store the data in its **raw format**. It is divided into two sub-buckets: **temporal and persistent** landing. The temporal sub-bucket is where we store our data while it is being ingested from the APIs. Once the data has been ingested into the persistent landing, name conventions are applied, and this data is moved to the persistent landing, where it is stored forever. After moving the data to the persistent zone, the data from the temporal landing is deleted.

Buckets creation. We carried out the creation of the bucket for the landing zone, and the sub-buckets for the temporal and persistent landing. We also defined a **hierarchy of sub-buckets** inside the persistent landing: first split into JSON and media files, then the JSON sub-bucket is split into both sources: **Steam** and **SteamSpy**. In the same way, the media sub-bucket is divided into **image** and **video** sub-buckets.

Top 100 best games. In order to get all the game data needed, we first had to establish the top 100 highest rated video games in Steam. Since scraping all 100,000+ entries would be unfeasible, we leveraged an already existing Kaggle dataset containing all Steam video game data to create a list of IDs only from the games that made the top 100 highest rated.

Game data ingestion. Once we had the list of games we are only interested in, we modified the scrapping Python script used to generate the previous Kaggle dataset to only fetch information for our game IDs from both Steam and SteamSpy.

Media data recovery. As seen in Appendix A, the Steam API only returns the link to images and videos for each game, so we need to fetch it in order to store it locally. For this, we have taken advantage of **multi-threading** to get all media data from Steam and upload it to MinIO. This way, the process does not get stuck waiting for each response by **allowing parallel GET requests**. This has been possible because the Steam API is very permissive with rate limiting. However, we have implemented an exponential back-off system to keep retrying after waiting for a while to avoid access rejections caused by possible Steam restrictions. Note that multi-threading is only used in the media ingestion part of the pipeline and never more. As stated before, in this particular case, the API was very permissive. This does not hold for the SteamSpy nor the Gemini APIs, which have been very strict with their rate limits, even going as far as blocking some of our IP's during the project development.

Naming conventions. Finally, we move the files from temporal to persistent landing where we store them forever. For the JSON files we keep a copy for each time the data ingestion pipeline has been run, since some attributes like ratings or descriptions can change. For media data, since we assume it will not change, each time this step is executed, all previous records will be deleted and replaced with new ones. In this process, we apply the following naming conventions:

- JSON files: *<source>#<date>#games.<format>*, where *<source>* indicates if the source API is Steam or SteamSpy, *<date>* indicates the date following the format *YearMonthDay_HourMinuteSecond* and *<format>* is the format of the file (by default JSON). JSON files are also structured in two different sub-folders depending on their source: *Steam* and *SteamSpy*.
- Images/videos: *<date>#<game_id>#<media_num>.<format>*, where *<date>* indicates the date following the format *YearMonthDay_HourMinuteSecond*, *<game_id>* is the ID of the game corresponding to the image/video, *<media_num>* is the index of the image/video of the game, and finally *<format>* defines the format of the file (by default .png and .mp4 respectively). These media files will be separated into sub-folder depending on their type, image or video.

3.2. Formatted Zone

The formatted zone is modularized by types of data. Each type requires a different formatting. Below is explained the formatting applied in each.

JSON files. The formatting applied in JSON files from the Steam API and from the SteamSpy API is the same. Firstly, based on the assumption that JSON files may change, in every pipeline run, we only keep in the formatted zone **the most recent file** for Steam and for SteamSpy files. To do so, we compare the date of the file in the format with the one in the landing zone. If dates are the same [1] this means the data in the landing zone has not been updated, and we do nothing. Otherwise, we remove the JSON file from the formatted zone, and we move the new one from landing to the formatted zone. We provide support for **CSV, XML, YAML, JSON** input formats, then all files are converted to JSON.

Images. In this case, we assume that the images of the games are not updated. So in each run of the formatted zone, we move the images stored in the landing zone to the formatted zone. We provide support for all image formats taken care of by the function `io.BytesIO`. This function allows us to extract the bytes of the image and format it to the target format **PNG**.

Videos. In the case of videos, the pipeline follows the same approach as with images, we first delete the videos for the formatted zone to ensure synchronization with landing zone data. Then, we move new videos from landing to formatted. If videos have a format different from **MP4**, we use the `moviepy` Python library to download the file, re-encode it to MP4, and then upload the new formatted video into the formatted zone. This guarantees that all videos in the formatted zone follow MP4 format.

3.3. Trusted Zone

In this zone, we apply data quality preprocessing to the data coming from the formatted zone. The goal is to perform generic data cleaning to ensure it is reliable for further analytical tasks. We have followed the same architecture using *MinIO*, a bucket for the trusted zone itself, and a sub-bucket structure like in the formatted zone: we split into `json/steam/`, `json/steamspy/`, `media/image/`, and `media/video/`.

JSON Data Quality. For JSON files extracted from data in Steam and SteamSpy APIs, several validation and cleaning steps were performed (`process_json.py`):

1. **Schema validation:** Each game entry is checked to have a predefined list of required keys. Entries with missing keys are logged and skipped.
2. **Type validation:** In crucial fields, data types are validated against expected types defined in a dictionary. If the data type of a field does not match, it is converted to the correct one.
3. **Content validation:** Different rules in the contents are validated. For example, some fields are defined as non-negative, and are checked to ensure they are positive. If a negative value is found, it is corrected to 0 and logged.
4. **Missing data:** Fields that should be list or dictionaries but are None are converted to empty list or dictionaries, ensuring consistent structure.

A **Quality report** (`quality_report.pdf`) generated by `quality_report.ipynb` was created to analyze the results of the games in the dataset.

[1]Note that this only happens if we run the pipeline by zones separately

Image Data Quality. Image processing (`process_images.py`) consists of several steps:

1. **Format:** All images are converted to RGB format, ensuring all images have the same color channels.
2. **Brightness:** Histogram equalization is applied to standardize brightness among all images.
3. **Resolution:** Images are resized and padded to a resolution of 256x256.

Video Data Quality. We use the library `ffmpeg` to ensure data quality in video (`process_video.py`):

1. **Corruption heck:** Each video file is downloaded from the Formatted zone and is probed (`ffmpeg.probe` to check is not corrupted. Corrupted files are skipped.
2. **FPS standardization:** The frame rate is standardized to a target of 30 FPS.
3. **Resolution standardization:** The resolution of videos is also standardized to 1280x720, using scaling while preserving aspect ratio.

3.4. Exploitation Zone

In the exploitation zone, we prepare the processed data for the multi-modal, analytical tasks. Specifically, we create multimodal embeddings for text, images, and videos.

All embeddings have been generated using a CLIP model. This multimodal model, allows us to map all input data (text, image, video) into one unified **embedding space**. This mapping converts each data point into a single numerical embedding vector that contains all the semantic meaning from the original example. This allows us to compute distances, and by extension, similarities between them. In order to store all embeddings' data, we will locally host a ChromaDB instance.

We have selected the CLIP-ViT-B-32 model using the weights trained on the LAION 2B dataset, which is a subset of a larger dataset (LAION 5B) of text-image pairs.

Buckets creation. As always, we firstly create the MinIO buckets and sub-buckets corresponding to the exploitation zone. These will follow the same structure as in the trusted and formatted zones. Once the buckets and sub-buckets are created, we copy all data from the trusted zone to isolate the work.

Merge JSONs. Until now, we had two different JSON files, one for the Steam API and another for the SteamSpy API. Since both file only make sense together, we will unify the video game information into one single JSON file. Then, we will save them under the *json/* sub-bucket, getting rid of the folder division we had seen until now by source.

Enhancing video game descriptions. As mentioned before in the Data Sources section 2.2, the raw description fields we find for each game are not of a good enough quality to safely perform similarity search on, since they are not a reliable source of information and context about the game. To address this issue, we will use **Google's Gemini 2.5 Flash Lite** model via Gemini Inference API to create a general description of the game based on its descriptions and other informative data fields (*name* and *genres*).

Embeddings creation. Once we have consistent and semantically rich video game descriptions, we can start creating the embeddings for both text and image data. However, this CLIP model does not allow video as input. Upon further research on other CLIP models that allow video as input, like CLIP4Clip, we can see that the standard procedure is to sample frames from the video and average them. Although other more complex procedures, like using attention layers, can be also used to combine these. For this reason, in order to compute video embeddings, we will sample 10 evenly spaced out frames[2] and average them. Once all data has been

[2]We actually sample 12 evenly spaced out frames but leave out first and last since they are usually completely black frames.

preprocessed, we will create a **ChromaDB** collection for each modality and populate them with the according embeddings.

4. Multi-Modal Tasks

This section describes the implemented analytical multi-modal tasks based on the data processed with the pipeline. In the implemented UI, one can find a toggle button to change between similarity search tasks (Task 1 + Task 2) and a game recommendation assistant (Task 3).

4.1. Task 1 + 2: Same-Modality / Multimodal Similarity Search

The goal of this task is to allow our users to search through all of our data repository given any input data. In the UI, the user will be able to select any input data type (text, image, video) and will do the same for the output data type they expect. When running the similarity search pipeline, the user will also be faced with a small terminal-like text box in order to ease the wait time.

The pipeline process runs as follows:

1. The user selects input and output data types they desire from both the drop-down and multi-select menus.
2. Then, the user either writes the textual query or uploads a media file.
3. The process is started, and the user is faced with the terminal box where they can see at which exact moment of the similarity search process they are.
4. Inside, the pipeline gets the input data type, converts it into an embedding and then queries all selected ChromaDB collections. If the input is a video, however, there is a previous step, which involves getting 10 evenly spaced out frames and combine them into a single video embedding that will be used to query the collections.
5. Once the process ends, the most semantically similar content gets displayed in the UI. Note that videos must be downloaded first in order to watch them, since some browsers have incompatibilities with how we handle video data. In this case, the user can easily download them using the download button we provide.

This tool can be very useful to find video games that are similar to a specific topic or match the vibe of any given image or video.

4.2. Task 3: Generative Tasks (RAG)

The last task is a generative one. In here, we leverage a State-of-the-Art Large Language Model, specifically **Gemini 2.5 Flash Lite**, to create useful game recommendations. LLMs tend to hallucinate in their responses, even more when the domain they are dealing with has not been much present in their training data corpus. However, we could try to mitigate this issue by providing the LLM with the necessary context needed when querying. This is what **Retrieval Augmented Generation** or **RAG** is exactly about, context is retrieved by performing some kind of similarity search and the results are fed to the LLM as context.

In our case, we have tried to move past the baseline Query-Retrieve-Write typical RAG pipeline and have implemented more modern techniques that enhance the results obtained by this method. The entire pipeline process can be seen in Figure 4.2.2. The techniques implemented are known as **Hypothetical Document Embedding** or **HyDE** and **Filtering**.

HyDE is a very powerful RAG technique that tackles a very present problem in most RAG pipelines. In our case, the descriptions we have stored in our ChromaDB instance are very different from what a user might query, both in terms of length, and tone. Although the semantical meaning might be similar, the different structure of both text inputs can have a great effect on its results. In order to address this issue, we take the initial user query, and prompt an LLM to convert it into a Hypothetical Document. In our case, we ask it to write the description for a hypothetical video game that matches what the query looks for. Then, when performing similarity search, we are doing it between the same type of documents. For further information on this topic, we advise checking out HyDE's original paper.

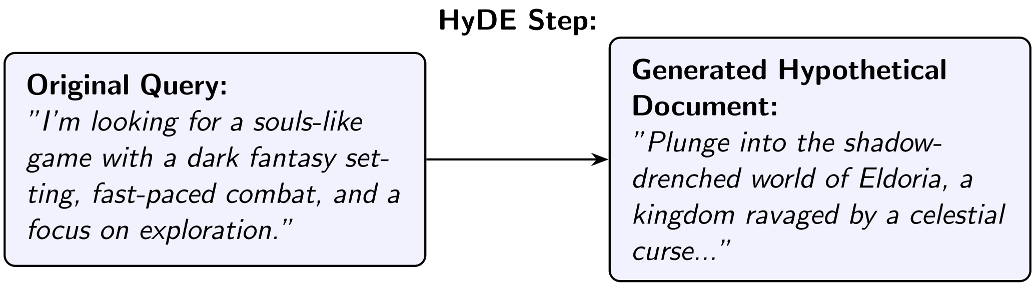
HyDE Step:
Original Query:
"I'm looking for a souls-like game with a dark fantasy setting, fast-paced combat, and a focus on exploration."
Generated Hypothetical Document:
"Plunge into the shadow-drenched world of Eldoria, a kingdom ravaged by a celestial curse..."

Figure 4.2.1. HyDE example.

After retrieving the five most similar results across all collections, we might still find some noise in them. That is why, instead of generating a response with them blindly, we will prompt an LLM, along with the original query, in order to binary classify each game description into a useful or useless result. This way, we are more confident on the final recommendation we will give to the user. If none of the games pass the filter, the LLM will politely apologize to the user for not having any useful recommendations.

Retrieval Augmented Generation Pipeline

1. The LLM greets the user, stating that its purpose is to help them find a new game to play.
2. The user tells the LLM what type of games do they like or what are they looking for.
3. The user's query gets converted into a hypothetical video game description in order to enhance similarity with other descriptions.
4. All three collections are queried for the three most similar entries in each one.
5. All results are combined, sorted and discarded if not in the top five across all modalities.
6. Since some of them might not be textual, we get the description of each game.
7. The LLM filters the top five most similar list descriptions based on the original user query, and we only take the useful recommendations.
8. Lastly, with the final filtered list of games, the LLM writes a recommendation message reasoning the match for each selection. If the list happened to be empty, the LLM would apologize and politely inform the user that we don't have games that fit that criteria in our database.

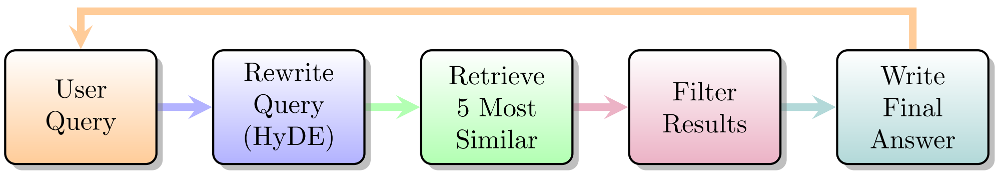

Figure 4.2.2: Graphical representation of implemented RAG pipeline.

5. Operations

This section details the implemented operations to automate the end-to-end data pipeline by following **DataOps principles of orchestration and reproducibility**.

Once we had all each zone working with multiple Python files, we needed to convert those isolated scripts into a **sequential**, **orchestrated workflow** that could be executed in **any environment**. Our solution addresses this by using Docker containers.

5.1. Pipeline Orchestration

To implement the fully orchestrated pipeline, we have developed the code following two main operational designs:

1. **Code Modularization:** The code was modularized by zones, organizing it into different folders and Python scripts, where each has a well-defined responsibility. For example:

   - `landing_zone/ingest_games.py`
   - `formatted_zone/format_images.py`
   - `exploitation_zone/create_embeddings.py`

   This modularity ensures that each component is reusable and maintainable, following software engineering best practices.

2. **Sequential Orchestration:** We developed several bash scripts to orchestrate each zone. These scripts sequentially run the individual Python scripts, thus managing the flow of data from one stage to the next inside a zone.

   The main orchestrators are:

   - `landing_zone.sh`: Executes the creation of MinIO buckets, ingests data from Steam and SteamSpy API, fetches all image and video files, and finally moves all raw data from the temporal to the persistent landing zone.
   - `formatted_zone.sh`: Sequentially runs the format conversion scripts for JSON, images, and videos. This ensures all data in the formatted zone has its respective target formats (JSON, JPG, and MP4).
   - `trusted_zone.sh`: Applies data quality and cleaning process to the data in the formatted zone, such as validating JSON structures, standardizing images, and videos. Finally, moves the preprocessed data to the trusted zone buckets.
   - `exploitation_zone.sh`: Runs the tasks to prepare data for analytical tasks. This includes merging Steam and SteamSpy JSON files, generating new descriptions, and finally creating multi-modal embeddings in ChromaDB.
   - `run_pipeline.sh`: Sequentially runs all previous bash scripts in order to complete a full step of the data ingestion process.

5.2. Environment reproducibility

To ensure the pipeline can be properly run in any environment, we have created three different Docker containers to isolate different services.

- **Container Definitions:** The `docker_compose.yml` file defines the complete project environment. It deploys three independent containers that communicate to each other using Docker's internal network:

  1. **MinIO container:** Hosts the MinIO instance.
  2. **ChromaDB container:** Hosts the ChromaDB instance.
  3. **Streamlit container:** Hosts the Streamlit UI application and runs bash scripts that either execute the data pipeline or one of the three multi-modal tasks.

  This configuration ensures that all necessary services are automatically provided and deployed.

- **Dependency Management:** The `requirements.txt` file defines our project's Python dependencies (`boto3, chromadb, google-genai`, etc). Moreover, since PyTorch is one of the bigger dependencies we have, in order to expedite Docker's build time, we have decided to use a PyTorch base docker image for the Streamlit container instead of a basic Python one and then having to install PyTorch in it. This ensures that the code can be executed with the exact library versions it was developed with, thus avoiding conflicts.

- **Execution Instructions:** The project's `README` file provides the exact commands needed to start and stop the services and execute the orchestration by zones. Also, the Streamlit UI guides the user with helpful messages and logs to understand what is happening at each moment and what to can they do.

6. Future Work

Even though our current operations environment is aligned with the project requirements, some improvements could be made:

- **Advanced orchestration:** We could use more robust orchestration tools (e.g., Apache, Airflow, or Prefect) to provide better error handling, automatic retries, or a visual UI for monitoring pipeline executions.
- **Error handling and testing:** The current pipeline handles errors by just checking them; it could be improved by adding unit tests for data transformations and integration tests to ensure correct data flow among zones.
- **Monitoring and quality control:** Future work could involve integrating Prometheus for monitoring container performances and SonarQube for code analysis to ensure code quality.
- **Further enhancing RAG pipeline:** Since the HyDE step heavily relies on the quality of the input query, we could also add a previous step where the LLM takes the original query and converts it to a set of more exhaustive queries. In addition, we could also insert a final step where the LLM is prompted to sort the filtered result in order to present the user with the most useful information first. These two techniques are known as Query Expansion and Reranking, for further information on these topics we advise to read the original papers (Query Expansion paper, Reranking paper).

A. API Data Fields

Steam API:

- *name*: Name of the game.
- *realease date*: Release date of the game.
- *required age*: Required age to play the game.
- *is free*: Boolean indicating if the game is free.
- *price*: Price of the game.
- *dlc count*: Number of DLCs.
- *detailed description*: Detailed description about the game.
- *about the game*: Description about the game.
- *short description*: Short description about the game.
- *reviews*: List of game reviews.
- *header image*: Link to the header image.
- *website*: Link to the game website.
- *support url*: Link to the support website.
- *support email*: Support email.
- *windows*: Boolean indicating Windows support.
- *mac*: Boolean indicating Mac support.
- *linux*: Boolean indicating Linux support.
- *metacritic score*: *Metacritic* average score.
- *metacritic url*: Link to the *Metacritic* review.
- *achievements*: Number of achievements.
- *recommendations*: Number of recommendations.
- *notes*: List of public notes about the game.
- *supported languages*: List of available text languages.
- *full audio languages*: List of available audio languages.
- *packages*: List of available game extra packages.
- *developers*: List of developers.
- *publishers*: List of publishers.
- *categories*: List of game categories.
- *genres*: List of game genres.
- *screenshots*: List of links to game screenshots.
- *movies*: List of links to game videos.

SteamSpy API:

- *user_score*: Mean game score voted by users.
- *score_rank*: Position in the global ranking sorted by *user_score*.

- *positive*: Number of positive votes.
- *negative*: Number of negative votes.
- *estimated_owners*: Range of estimated owners.
- *average_playtime_forever*: Average user playtime.
- *average_playtime_2weeks*: Average user playtime in the last 2 weeks.
- *median_playtime_forever*: Median user playtime.
- *median_playtime_2weeks*: Median user playtime in the last 2 weeks.
- *discount*: Percentage of discount of the game at the moment.
- *peak_ccu*: Peak concurrent players.
- *tags*: List of tags related to the game.