Big Data Technology
Group Assessment 1

Movie Performance Analysis using TMDB Data

Group 6
Bernarda Andrade
Matias Arevalo
Pilar Guerrero
Moritz Goebbels
Tomás Lock
Allan Stalker

Table of Contents

1. Introduction

Success in the film industry is affected by many factors such as: production costs, actors, genre, and reviews. After searching the data available on this topic we realised that organizing this data into a structured database could provide a deeper and more structured look into all the data of the film business and uncover the factors that contribute to a movie's success. We found an extensive amount of data in the "The Movie Database" (TMDB), so we decided to compile and extract the data using their TMDB API, followed by creating tables that can be then connected to the database wich will serve as a tool to obtain insights. Seeking to identify trends and patterns that can support people interested in the movie industry in making data-driven decisions regarding movie creation, release strategies, and overall business planning.

Our first motivation for this project lies in creating a well-structured database, where key attributes such as genre, cast, budget, box office earnings, and audience ratings can be effectively organized and analyzed. With this organized structure we aim to facilitate the view of correlations that might be missed in the unprocessed data. For example, we want to explore whether an actor affects a film's success or if there are any insights into how different genres perform at the box office. By analyzing how these factors interact, we can help develop a more informed view of what drives success in the film industry which is valuable for the optimisation of resources of the movies budget.

For the first part of the project, we plan to extract data from the TMDB API using the API key to gather the data that is valuable to us, including data on genres, casts, production budgets, ratings, and revenue. Once we have this data, we will clean it and transform it into a structured format that is easy to analyze. With the cleaned data we will create tables that will hold key data in a way that optimizes querying. These tables will then be connected to a database to ensure that the data can be accessed and managed constantly. Through the design of this database, we hope to make the data accessible for future analysis, allowing us to identify valuable trends and insights over time. Our goal is not only to create this database, but also to establish an efficient data ingestion pipeline that ensures future data can be integrated into the system. All of this in order to ensure future scalability which will help to process new data as it becomes available and continue to serve as a tool for this analysis. It's important to highlight that the industry should not base all of their decision-making upon this particular tool but leave some space for improvisation and other inputs.

Our primary motivation is that we expect to provide producers, investors, and industry professionals with a tool that can help them understand what factors contribute to success in the movie industry. Through the database, we will be able to uncover trends related to box office success, audience ratings, and also risks that could affect a movie's success. This understanding will be important for decision-makers in many areas such as casting, genre selection, budget allocation, and marketing strategies. Furthermore, the insights gained

from this project can help optimize investment by identifying high-potential projects before they become hits.

Concluding that this project will create a valuable resource for the film industry as it will allow professionals to make more data-driven decisions and significantly improve the chances of success for future films.

2. The Dataset: Analysis

For the datasets for this project, we collected data regarding different details for both movies and tv shows that have been released over time. In these datasets we can find data that ranges from release date to budget, revenue to popularity scores, and more; all of which are useful to analyze the performance and success indicators of these media contents. In the following subsections there is more detailed data about the datasets' metadata and detailed data of what data can be found within each dataset.

2.1 Dataset Metadata

Actors Data

Table 1. *Actors Datasets Description*

File Name	Row Quantity	Column Quantity	Total Missing Value Count	Duplicate Rows Count
actor_director_data	21,944	6	27,808	5,771
actor_director_movies	160,736	3	9,047	55,150

Movies Data

Table 2. *Movie Datasets Description*

File Name	Row Quantity	Column Quantity	Total Missing Value Count	Duplicate Rows Count
movie_genres	1,078,451	3	0	41,279
movie_production_companies	711,514	4	245,755	28,985
movie_production_countries	675,367	3	66	25,134
movie_spoken_languages	700,252	3	23,010	26,563
movie_collection	27,927	3	0	1,214
movies_main	902,282	20	2,844,139	32,668

TV Shows Data

Table 3. *TV Show Datasets Description*

File Name	Row Quantity	Column Quantity	Total Missing Value Count	Duplicate Rows Count
TV_Show(Basic)	153,440	3	2	99
TV_Show(Airings)	153,486	4	892	3,474
TV_Show(Genre)	212,341	2	0	150
TV_Show(GenreID_Genre)	20	2	1	0
TV_Show(Overview)	153,440	2	64,891	89
TV_Show(Production)	153,460	4	277,492	53
TV_Show(Score)	153,460	2	0	21
TV_Show(SeasEps)	153,447	3	420	7,354
TV_Show(Status_Description)	6	3	0	0

2.2 Data Dictionary

Actor Data

Table 4. *Data Dictionary of actor_director_data Dataset*

Dataset:	*actor_director_data*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the person
Name	String	Name and last name
Date_of_Birth	datetime	Person's birth date
Nationality	String	Person's nationality
Biography	String	A brief description of the person
Role	String	Their role in the production

Table 5. *Data Dictionary of actor_director_movies Dataset*

Dataset:	*actor_director_movies*	
Column Name	**Data Type**	**Description**
Name	String	Name and last name
Movie/TV_Show	String	Name of the movie/show they participate
Role	String	Their role in the production

Movie Data

Table 6. *Data Dictionary of tmdb_movies_main Dataset*

Dataset:	*tmdb_movies_main*	
Column Name	**Data Type**	**Description**
movie_id	Integer	Unique code for the movie.
adult	Boolean	Indicates explicit content in the movie
budget	Integer	Movie allocated budget
original_language	String	Code of original language
overview	String	Movie's description
popularity	Float64	Popularity of the movie
release_date	datetime	Movie's release date
revenue	Float64	Movie's revenue earned
runtime	Integer	Duration of the movie in min
status	String	The current status of the movie
tagline	String	Catchy phrase of the movie
title	String	The movie's official title
video	Boolean	Indicates if it has an associated video
rating	Float64	Average user rating of the movie (0-10)
vote_count	Integer	Total number of votes or ratings

Table 7. *Data Dictionary of movie_genres Dataset*

Dataset:	*movie_genres*	
Column Name	**Data Type**	**Description**
movie_id	Integer	Unique code for the movie.
genre_id	Integer	Unique code for the genre.
genre_name	String	Name of the genre.

Table 8. *Data Dictionary of movie_production_companies Dataset*

Dataset:	*movie_production_companies*	
Column Name	**Data Type**	**Description**
movie_id	Integer	Unique code for the movie.
company_id	Integer	Unique code for the company.
company_name	String	Name of the company
origin_country	String	Country code for the company

Table 9. *Data Dictionary of movie_production_countries Dataset*

Dataset:	*movie_production_countries*	
Column Name	**Data Type**	**Description**
movie_id	Integer	Unique code for the movie.
iso_3166_1	String	Country code (abbreviation)
country_name	String	Country name

Table 10. *Data Dictionary of movie_spoken_languages Dataset*

Dataset:	*tmdb_movie_spoken_languages*	
Column Name	**Data Type**	**Description**
movie_id	Integer	Unique code for the movie.
iso_639_1	String	Language code (abbreviation)
country_name	String	Name of the spoken language

Table 11. *Data Dictionary of tmdb_movie_spoken_languages Dataset*

Dataset:	*tmdb_movie_spoken_languages*	
Column Name	**Data Type**	**Description**
movie_id	Integer	Unique code for the movie.
iso_639_1	String	Language code (abbreviation)
country_name	String	Name of the spoken language

TV Show Data

Table 12. *Data Dictionary of TV_Show(Basic) Dataset*

Dataset:	*TV_Show(Basic)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
Name	String	Name of the TV Show
Original Name	String	Name in original language

Table 13. *Data Dictionary of TV_Show(Overview) Dataset*

Dataset:	*TV_Show(Overview)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
Synopsys	String	A description of the TV Show.

Table 14. *Data Dictionary of TV_Show(SeasEps) Dataset*

Dataset:	*TV_Show(SeasEps)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
Seasons	Integer	Total number of seasons
Episodes	Integer	Total number of episodes

Table 15. *Data Dictionary of TV_Show(Production) Dataset*

Dataset:	*TV_Show(Production)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
Executive Producer	String	Ex. Producer Name
Production Companies	String	Production company's name
Network	String	Channel or streaming platform

Table 16. *Data Dictionary of TV_Show(Score) Dataset*

Dataset:	*TV_Show(Score)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
Popularity Score	Float64	Score given to the TV Show

Table 17. *Data Dictionary of TV_Show(Airings) Dataset*

Dataset:	*TV_Show(Airings)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
First Aired Date	datetime	Date of first airing
Last Aired Date	datetime	Date of latest airing
Status	String	Condition of the TV Show

Table 18. *Data Dictionary of TV_Show(Status_Description) Dataset*

Dataset:	*TV_Show(Status_Description)*	
Column Name	**Data Type**	**Description**
Status	String	Condition of the TV Show
Short Description	String	Brief description of the status
Long Description	String	Extended description

Table 19. *Data Dictionary of TV_Show(Genre) Dataset*

Dataset:	*TV_Show(Genre)*	
Column Name	**Data Type**	**Description**
ID	Integer	Unique code for the TV Show.
Genre ID	Integer	Unique code for the genre

Table 20. *Data Dictionary of TV_Show(GenreID_Genre) Dataset*

Dataset:	*TV_Show(GenreID_Genre)*	
Column Name	**Data Type**	**Description**
Genre ID	Integer	Unique code for the genre
Genre	String	Name of the show genre

2.3 Key Visualizations

Movies Visualizations

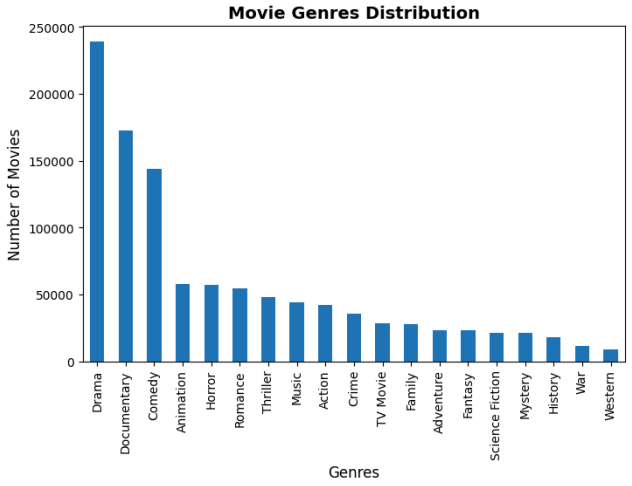


Figure 1. *Movie Genre Distribution*[1]

As seen in figure 1, movies categorized as “Drama” are the biggest group from the dataset, followed by “Documentaries” and “Comedy”. Meanwhile, “History”, “War”, and “Western” are the least popular genres of movies currently.

[1] Data extracted from the *movie_genres* dataset and visualized using Python (Matplotlib).

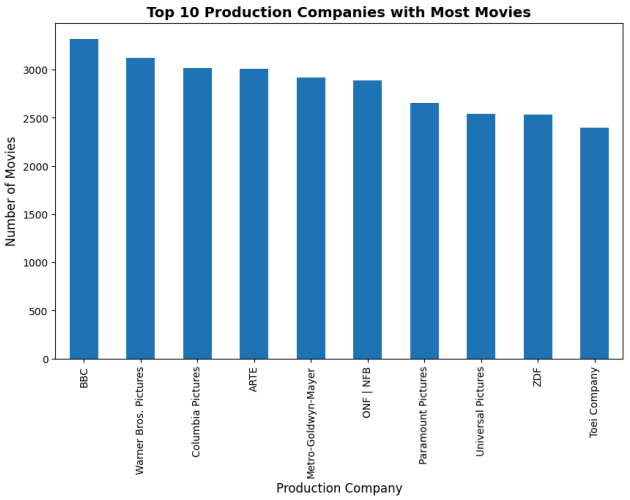


Figure 2. *Top 10 Production Companies with the Most Movies*[2]

In figure 2, we can see that BBC, Warner Bros. Pictures, and Columbia Pictures, are the three biggest production companies of movies. Meanwhile, streaming platforms with original content, like Netflix or Amazon Prime, are not in the top 10 production companies despite their increasing popularity in the past years.

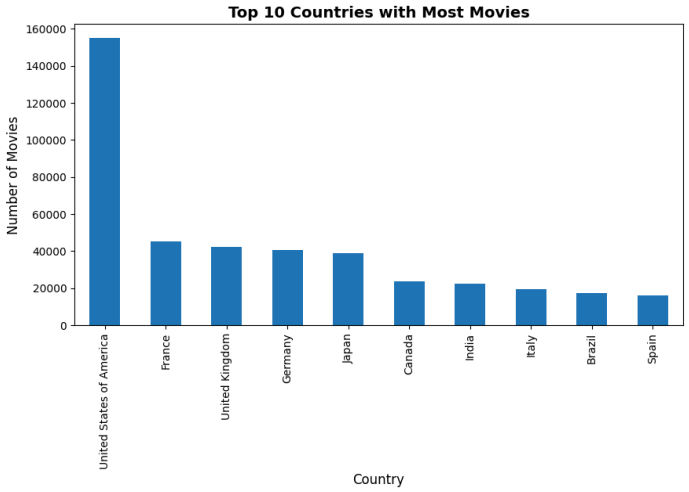


Figure 3. *Top 10 Countries with the Most Movies*[3]

In figure 3, we can see that, as expected, United States of America is the country that has produced the most movies. Nonetheless, European countries take the highest number of spots in the top 10 movie producing countries.

TV Shows Visualizations

[2] Data extracted from the *movie_production_companies* dataset and visualized using Python (Matplotlib).
[3] Data extracted from the *movie_production_companies* dataset and visualized using Python (Matplotlib).

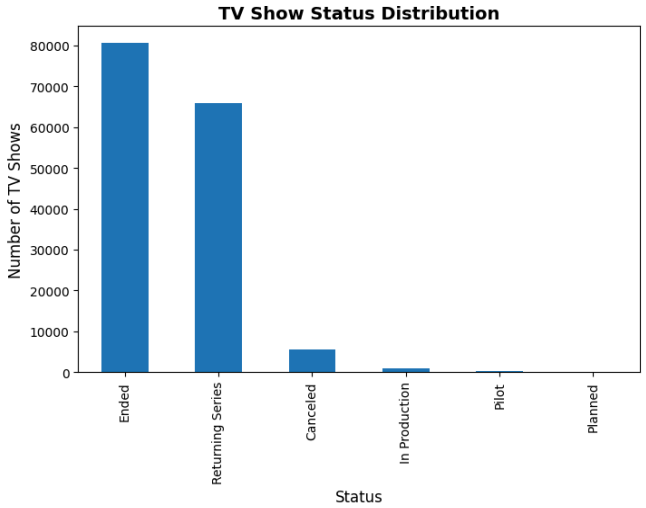


Figure 4. *TV Show Status Distribution*[4]

The majority of the TV Shows have already ended. The second biggest group is "returning series", which are those TV Shows that have new episodes arriving. There are very few shows that have the status "Pilot", which refers to a show only having its trial episode

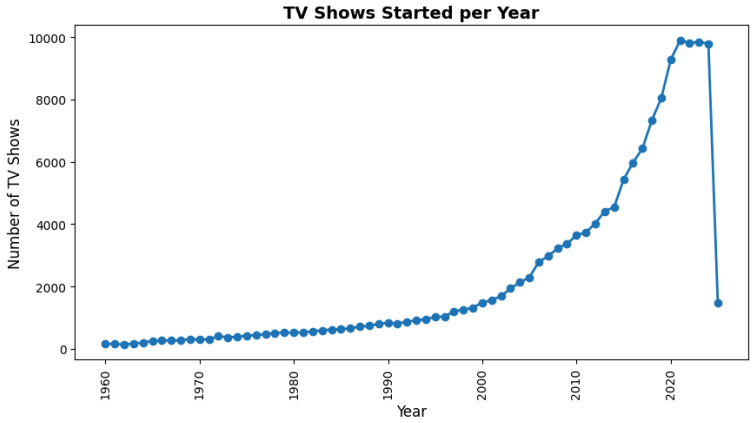


Figure 5. *TV Shows Started per Year*[5]

From figure 5 we can see that there has been an exponential increase of the number of TV Shows that have been started in the recent years compared to the ones in earlier years. Additionally, in the last six years, there has been the highest number of new shows.

[4] Data extracted from the *TV_Show(Status)* dataset and visualized using Python (Matplotlib).
[5] Data extracted from the *TV_Show(Airings)* dataset and visualized using Python (Matplotlib).

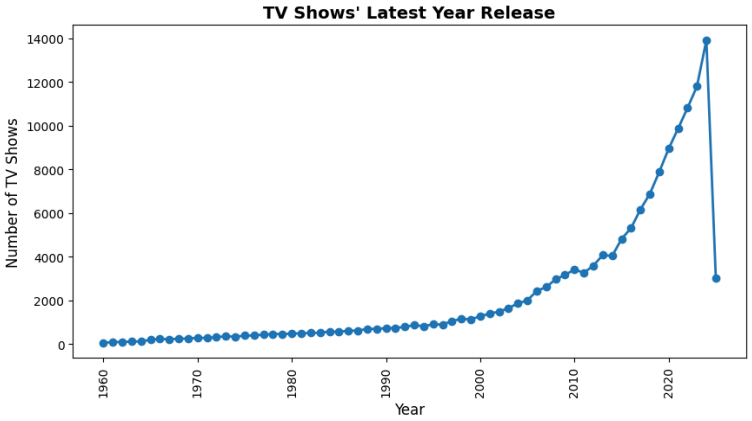


Figure 6. *TV Shows Latest Release Year*[6]

From figure 6, we can see that TV Shows have had their latest release in a similar trend as they have started. Furthermore, 2024 has been the year in which the most number of TV Shows have aired their latest episode.

[6] Data extracted from the *TV_Show(Airings)* dataset and visualized using Python (Matplotlib).

3. Data Architecture and Design

3.1 Overview of the Chosen Components

We needed an architecture that could efficiently handle large volumes of TMDB API data. After evaluating several options, we selected the following components:

Python became our primary programming language for the ETL process due to its ability to handle API calls and pandas for data manipulation. We found Python particularly effective for handling the nested JSON responses from TMDB's API. It also proved useful for converting text into ASCII format which was critical in loading the data as we encountered issues prior to the conversion.

PostgreSQL serves as our database system because of its reliability and excellent support for complex queries. Our movie dataset contains numerous many-to-many relationships and PostgreSQL handles them elegantly through its robust support for foreign keys and joins.

To manage the high volume of API requests efficiently, we implemented thread pooling with Python's concurrent.futures module. This approach allowed us to make multiple parallel requests to TMDB while carefully respecting their rate limits, significantly reducing our data collection time.

3.2 Data Flow and Integration Diagram

Our data pipeline follows a straightforward flow, beginning with TMDB API requests and ending with structured database tables. The process works as follows:

1. Our Python scripts send requests to the TMDB API, retrieving detailed data about movies and TV shows.
2. We then extract and flatten the nested JSON responses. This step was particularly important for handling complex elements like cast lists and production companies.
3. These flattened datasets are temporarily stored as CSV files to provide checkpoints in the pipeline.
4. Finally, we load these files into our PostgreSQL database using bulk import commands for efficiency.

The relationships between tables are visualized in our ER diagrams for both Movies (Figure A) and TV Shows (Figure B), which illustrate how we've normalized the data for optimal querying.

3.3 Tool Selection

When selecting technologies, we prioritized reliability, scalability, and maintainability for our specific use case:

Reliability: We encountered several challenges with TMDB's API, including occasional timeouts and rate limiting. PostgreSQL's transaction support ensures that our database remains consistent even when API calls fail mid-process. Python's exception handling allowed us to implement retry mechanisms when we hit TMDB's rate limits, ensuring complete data collection without manual intervention.

Scalability: Our initial dataset covers thousands of movies, but we designed the system to handle much larger volumes. The threaded API calls further allowed us to divide the workload efficiently. PostgreSQL's indexing capabilities have already proven valuable as our dataset grew, particularly for queries filtering by release date or genre.

Maintainability: We intentionally separated movie and TV show data structures, which simplified our code considerably. Our modular Python functions (like process_genres() and process_production_companies()) make the code easy to update when TMDB changes their API response format. Both Python and PostgreSQL have extensive communities, which has already helped us resolve several implementation challenges.

3.4 Sub-Architectures: Movies vs TV Shows

Movies and TV Shows: Separate but Connected

A key design decision was separating movie and TV show data structures. While both domains share similarities, they have fundamental differences that warranted distinct data models. However, one entity acted as a common bridge between movies and tv shows, this being the actors/directors from the ‘people roles’ which contains both actors and directors. This was important because there is a lot of crossover as most actors will be in movies and tv shows throughout their career. It subsequently allows us to evaluate how actors perform across different shows and movies, and whether a correlation in one domain is the same in the other.

Despite this bridge, we maintained separation in other aspects of the data model. Movies have distinct attributes like budget, revenue, and runtime, while TV shows track seasons, episodes, and airing schedules. Our separation allowed each domain to evolve independently. For instance, we can add TV-specific metrics like "episodes per season" without affecting the movie tables.

Overall, this approach allows us to use similar processing techniques for both domains, simplifying code maintenance while preserving the natural connection that exists with actors and directors.

3.5 Diagrams

Figure 7. *Movies ER Diagram*

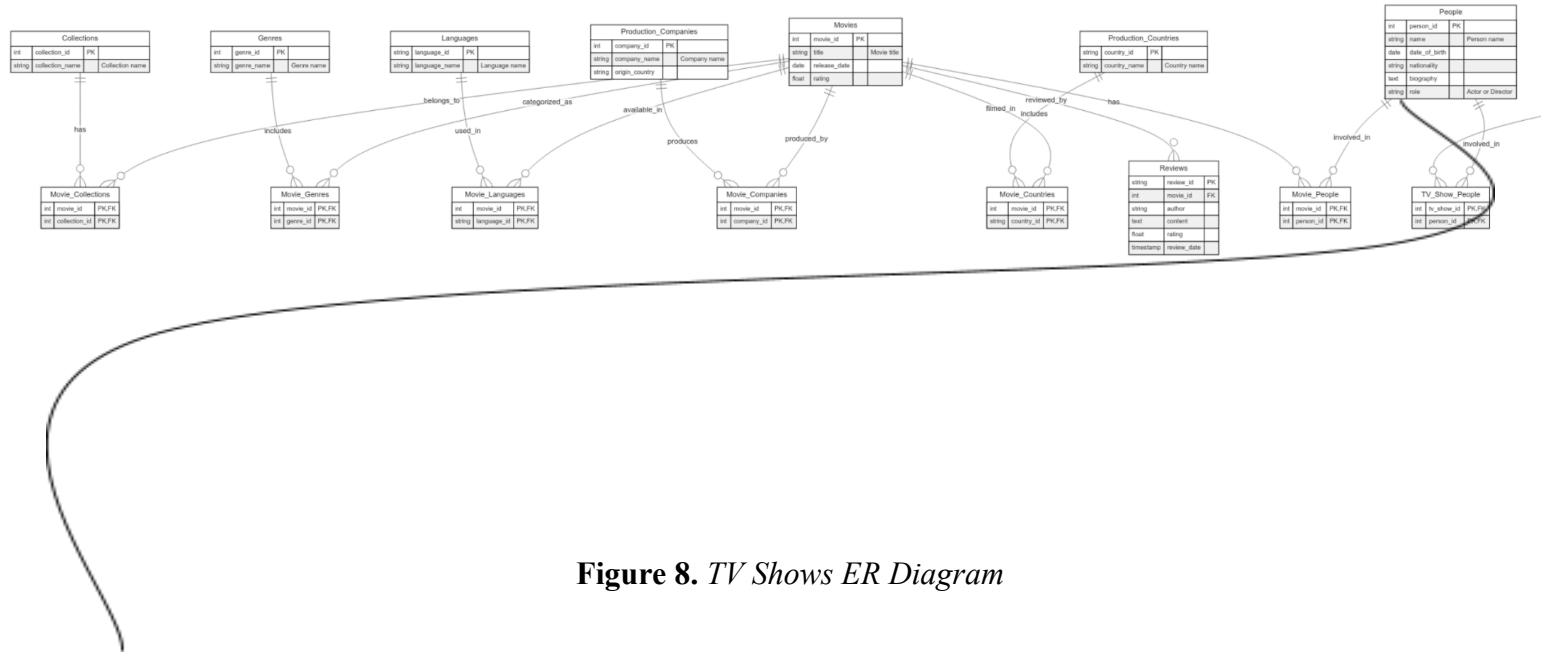


Figure 8. *TV Shows ER Diagram*

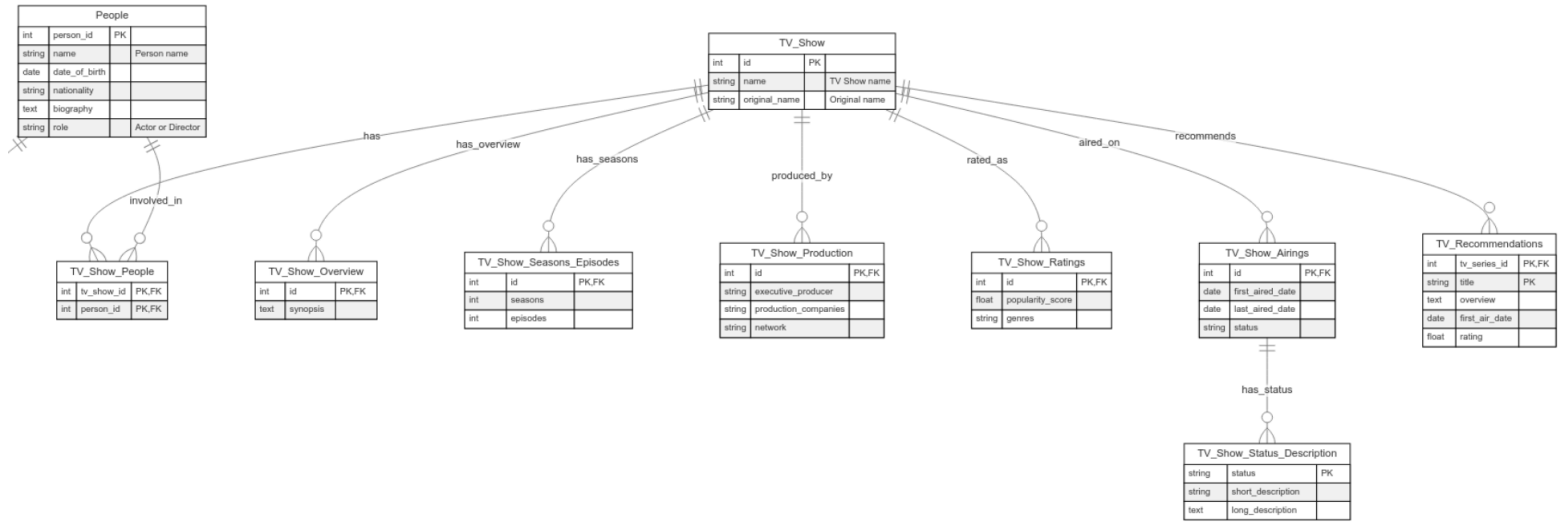


Figure #4 *ER Diagram, Full Size*
Find Full size diagram link clicking [Here]

3.6 Benefits to Reliability, Scalability, and Maintainability

Reliability:

- Using ACID transactions in PostgreSQL prevents partial data writes.
- Thread-managed data ingestion ensures partial failures (e.g., rate limit blocks) do not crash the entire process.

Scalability:

- Each sub-architecture (Movies, TV) can be scaled horizontally by either dedicating separate schemas or separate servers if data volume grows exponentially.
- Indexing specific columns (e.g., release_date, popularity, genre_id) allows faster queries, beneficial for large record counts.

Maintainability:

- The system uses standard, well-understood open-source tools, making it easier to find developers who can maintain or extend the pipeline.
- Clear boundaries between domains (movie- vs. TV-centric tables) prevent schema bloat, so new columns for, say, “episode runtimes” do not affect the movie domain.

3.7 Design Rationale Summary

Our architecture combines different tools to make a thorough data model to create a flexible, performance-oriented system. By separating the movie and TV show domains while using a consistent technical approach, we've built a foundation that supports our current analysis needs while remaining adaptable to future requirements.

The pipeline we've created not only efficiently processes TMDB data but also transforms it into a structure optimized for the specific analytical questions we're exploring about movie performance and success factors. This balance between processing efficiency and analytical utility is the central achievement of our design.

In the following sections, we'll examine how we implemented this architecture in practice and the data validation steps we undertook to ensure quality.

4. Data ingestion Pipeline

For the data ingestion processed we followed an ETL (Extract, Transform, Load) pipeline to gather the data from the TMDB API, processed it in an adequate form so that it’s ready for analysis. As a last step the data is loaded into PostgreSQL so it results in a database entity that can be queried.

4.1 API Data Extraction

TMDB, also known as the movie database, is an online database which provides free access to data on movies and tv series. It’s a user driven platform so collaborators are constantly updating data making it a rich and growing source of data for our topic realm. As a

matter of fact it's a common source of data for movie recommendation systems which inclined us to select it as our data source.

In order to extract this large amount of data it was necessary to connect to TMDB's API, their web based service which allows access to their vast database and retrieved data in JSON format. We used python's request module to send HTTP GET requests to TMDB's over the internet in order to fetch the data. For this authentication was required through the use of an API key. The following extraction code for this data is entirely depicted in **Appendix Code 2.1 Data Processing Function.**

When handling these requests through the API we found out it returns paginated responses, breaking down the requests into multiple pages which each contained a subset of the data requested. As it will be shown in our extracting code, this requires us to loop requests in order to retrieve entire datasets. Additionally there was an imposed limit of 500 pages per query. To resolve this issue we used recursive date filtering for datasets with large queries by using binary search-like logic to split release movie dates dynamically and be able to extract the entirety of data in steps of 500 pages.

Another challenge we faced was the rate limiting management. Despite using threadpooling to carry out parallelised API requests, when exceeding the rate TMDB blocks access to the API for a short period, to avoid this issue we ensure a fixed delay within requests using time.sleep(). This is depicted in **Appendix Code 2.2 Recursive Date Range Processing with Parallelization.** Find below a snippet of the code which recursively processes date ranges for further understanding.

```
def process_date_range(gte, lte, *writers, flush_files, lock):
    print(f"\nProcessing movies released from {gte} to {lte}")
    discover = tmdb.Discover()
    params = {"page": 1, "primary_release_date.gte": gte,
"primary_release_date.lte": lte}

    try:
        total_pages = discover.movie(**params).get("total_pages", 1)
        print(f"  Total pages in this range: {total_pages}")
    except Exception as e:
        print(f"Error fetching first page for range {gte} to {lte}: {e}")
        return

    if total_pages >= 500:
        start, end = map(lambda d: datetime.datetime.strptime(d, "%Y-%m-%d"),
[gte, lte])
        if start >= end:
            print("  Date range too narrow to split further. Processing with
current cap.")
        else:
            mid = (start + (end - start) / 2).strftime("%Y-%m-%d")
            print(f"  Splitting range: {gte} to {mid} and {mid} to {lte}")
            process_date_range(gte, mid, *writers, flush_files=flush_files,
lock=lock)
            process_date_range(mid, lte, *writers, flush_files=flush_files,
lock=lock)
```

```
            return

    for page in range(1, total_pages + 1):
        params["page"] = page
        print(f"  Processing page {page}/{total_pages} for range {gte} to
{lte}...")
        try:
            movies = discover.movie(**params).get("results", [])
        except Exception as e:
            print(f"  Error on page {page}: {e}")
            continue

        with concurrent.futures.ThreadPoolExecutor(max_workers=10) as
executor:
            futures = [executor.submit(process_movie, m, *writers, lock) for
m in movies]
            concurrent.futures.wait(futures)

        if page % 500 == 0:
            for f in flush_files.values():
                f.flush()
            print(f"  Processed {page} pages; file buffers flushed.")
```

Thanks to this we were able to benefit from parallel execution of multiple requests running simultaneously yet within the API limit. Despite this parallelised optimisation, our main API request of 353,541,879 bytes ended up taking more than two entire nights to load. Despite being time consuming we ensured ourselves access to every TMDB movie's data ever released since 1900.

Guided by our big data use case we focused on retrieving the data which would be more relevant for analysing movie performance, this included:

- Movie and TV show metadata: (id, adult, backdrop_path, budget, homepage, imdb_id, original_language, original_title, overview, popularity, poster_path, release_date, revenue, runtime, status, tagline, title, video, vote_average, vote_count, production companies, production_countries, genres, collection_information & spoken_languages).
- Producer details (Company Name, Founding Year, Origin Country, Headquarters, Parent Company)
- Cast and crew data (directors, lead actors).
- User ratings and reviews (vote counts, average ratings).

This structured extraction ensures that our dataset is tailored for producers and industry professionals and contains all necessary and available data to identify the key determinants of a movie's success.

4.2 Storage

Requests to the TMDB API retrieved data in JSON format which entailed issues for the project; Raw JSON can be huge for big datasets and they're not easy to query like

relational databases. As shown in the last code snippet **Appendix 2.4 Main ETL Process and Loading** when saving the data to address this issue we flattened data from JSON into CSV.

To do this we identified nested lists such as genres, production_companies and spoken_languages of each movie and created preprocessing functions to flatten data and extract key elements into separate dictionaries and CSV files. For example movies can have different genres so we store each genre in a separate row in a many to many relationship.

```
def process_genres(details):
    """Extract the list of genres for a movie."""
    genres_list = details.get("genres", [])
    rows = []
    movie_id = details.get("id")
    for genre in genres_list:
        rows.append({
            "movie_id": movie_id,
            "genre_id": genre.get("id"),
            "genre_name": genre.get("name")
        })
    return rows
```

The code above ensures 1NF compliance when creating our csv file. A similar process was carried out for all nested attributes (e.g., production companies, collections, spoken languages) to transform JSON into CSV. From this extraction data was stored for further transformation and validation throughout the next steps of the ETL pipeline.

4.3 Data Transformation

Data transformation was a crucial step in our project to ensure that the dataset was clean, consistent, and optimized for our analysis. First we dropped any irrelevant columns that were not needed, such as “tagline” or “poster_link” from the movie data files, as well as any columns that primarily contained False values such as the “adult” or “video” column. The next step included cleaning text data to ensure compatibility when uploading into SQL. Furthermore, we converted non-ASCII characters into their closest ASCII equivalents and replaced special characters that couldn’t be converted with a placeholder. This is depicted in the snippet found in **Appendix Code 2.5 Transformation Data into ASCII code.** Afterwards, we had to take care of missing values for which we used different strategies based on their corresponding data type this is As a result, we replaced missing numerical values with “NA” and missing categorical and data values with “NULL” to align once more with SQL standards to avoid any possible incompatibilities. These transformations can be found in **Appendix Code 2.6 Transformation Correct Data Types** and **Appendix 2.7 Deal with Null values**

Overall this ensured improved dataset quality by making it cleaner and more structured, especially useful for further processing and storage in SQL.

4.4 Data Loading

To begin the data loading process needed the full ER diagram mentioned earlier, which defines all entities, attributes and relationships. First of all, we created the SQL database schema, ensuring proper primary keys (PK) and foreign keys (FK) to make sure our relationships would be implemented consequently. After we defined the data types for each column, also considering whether to allow NULL was necessary or not. As a result of these first steps it was important to check that the schema design aligned with the relational structure of the data, and whether all necessary transformations were done (4.3). Since both of these requirements were fulfilled and in addition the CSV files matched the schema, we could finally use them as the data source and finish the implementation using SQL Server Import Wizard to upload the data. This process ended up taking several hours due to the large dataset size, however once this was done we had a fully functioning database which now can be used for any operations.

Below is the resulting tables and relationships resulting from the loading of our SQL database:

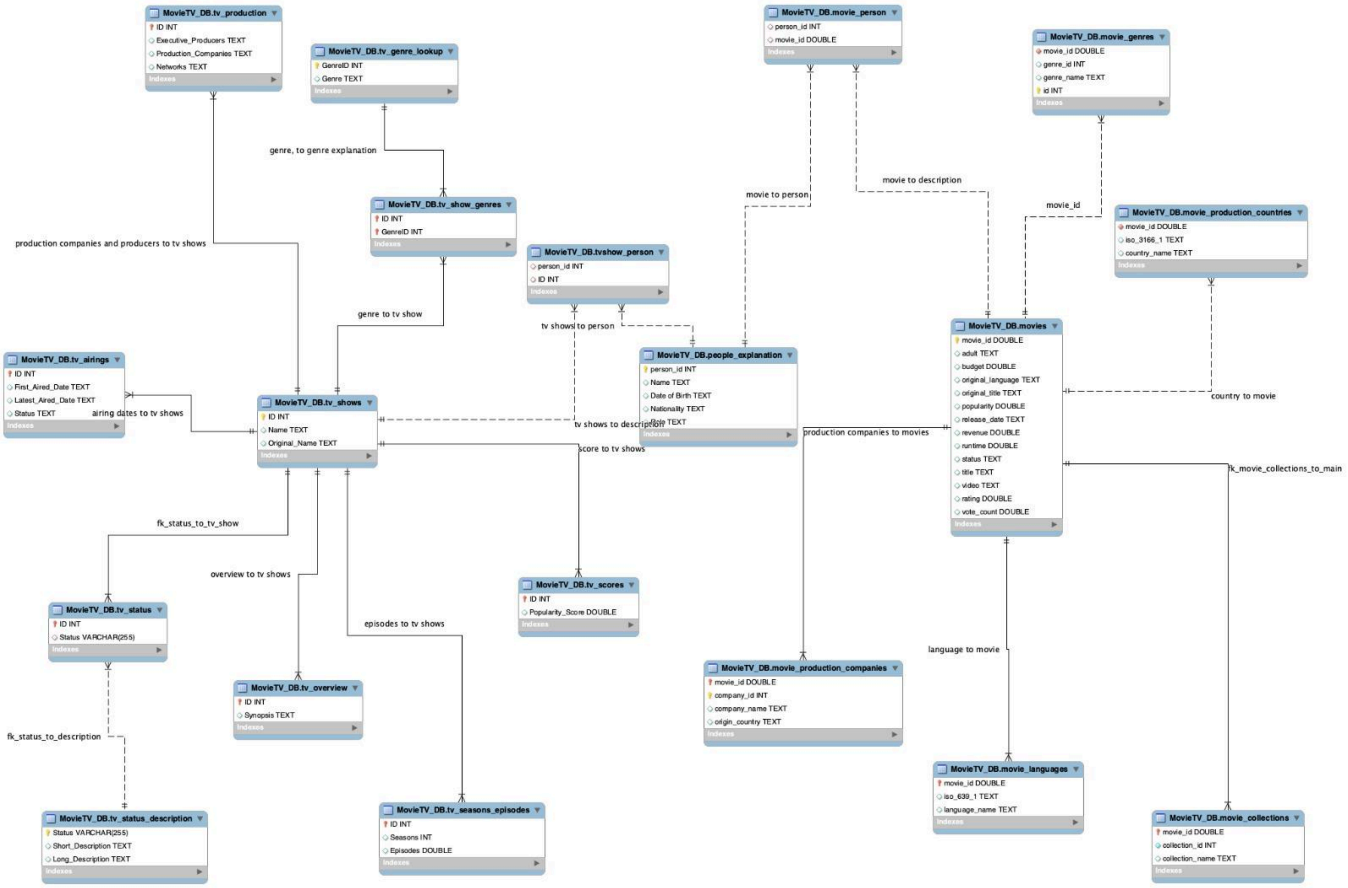


Figure #4 *Tables and Relationships from the SQL Database*
Depicts the Relationship tables in our loaded sql database downloadable through workbench.

5. Resulting Data Structures - Validated

The resulting data structure was built on a MySQL workbench. We utilized foreign key constraints, indexing, and junction tables; the database structure enables seamless integration of key entities such as movies, genres, reviews, production companies, and collections. This implementation ensures that the system remains scalable, query-efficient, and adaptable for analysis needs we may provide to movie companies to improve their production value.. The full code implementation is available in the **Appendix 2.8 MySQL Ingestion Code.** Below is some sample code.

```
CREATE TABLE Movies (
    movie_id INT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    release_date DATE,
    rating FLOAT
);

CREATE TABLE Movie_Collections (
    movie_id INT,
    collection_id INT,
    PRIMARY KEY (movie_id, collection_id),
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (collection_id) REFERENCES Collections(collection_id)
….
```

After creating the tables we imported the data from the csv files we had collected in the extraction and transformation part. We modified the files using python and pandas to match the schema of the database. Finally we used the import wizard to import the data directly into the tables.

The final schema is designed to ensure data integrity, efficient querying, as well as minimal redundancy. The schema is structured to support many-to-many relationships through junction tables and includes constraints to ensure data validity. We achieved this through normalising the database to the third normal form, avoiding redundancy and improving efficiency. The database satisfies the first normal form as it ensures that all attributes contain atomic values with no repeating groups, there is one value per entry.Additionally Each row has a unique identifier. Following that, the db ensures that all non-key attributes depend on the primary key in tables with composite keys, removing the partial dependencies of our database. Finally, in order to achieve a third normal form, the database ensures no transitive dependencies exist. For example, in the table Production_Companies, the column company_name depends only on company_id, not on origin_country. The database makes sure that each non-key attribute depends only on the primary key, achieving data integrity and eliminating redundancy.

In our database we decided to include some core tables which were crucial to ensure data integrity, support efficient queries, and be the foundation for meaningful insights. Regarding the main tables in the database, we created the main table which served as the central hub to our database as it contains key data about each movie as well as multiple tables linking back to these tables like: genres, collections, reviews, languages, production companies. This table has dependencies as other tables relate to movies directly or indirectly, making it essential for queries.

Regarding the general validation of our inputs we ensured that each movie had at least one genre and one rating. While each movie was confirmed to have a single release date (as specified when setting up the API), and every movie had one show rating, some TV shows originally had multiple genres. However, the data was transformed to accommodate this, allowing movies and TV shows to still be searchable by any of the genres they were originally assigned.

6. Tableau Dashboard Solution

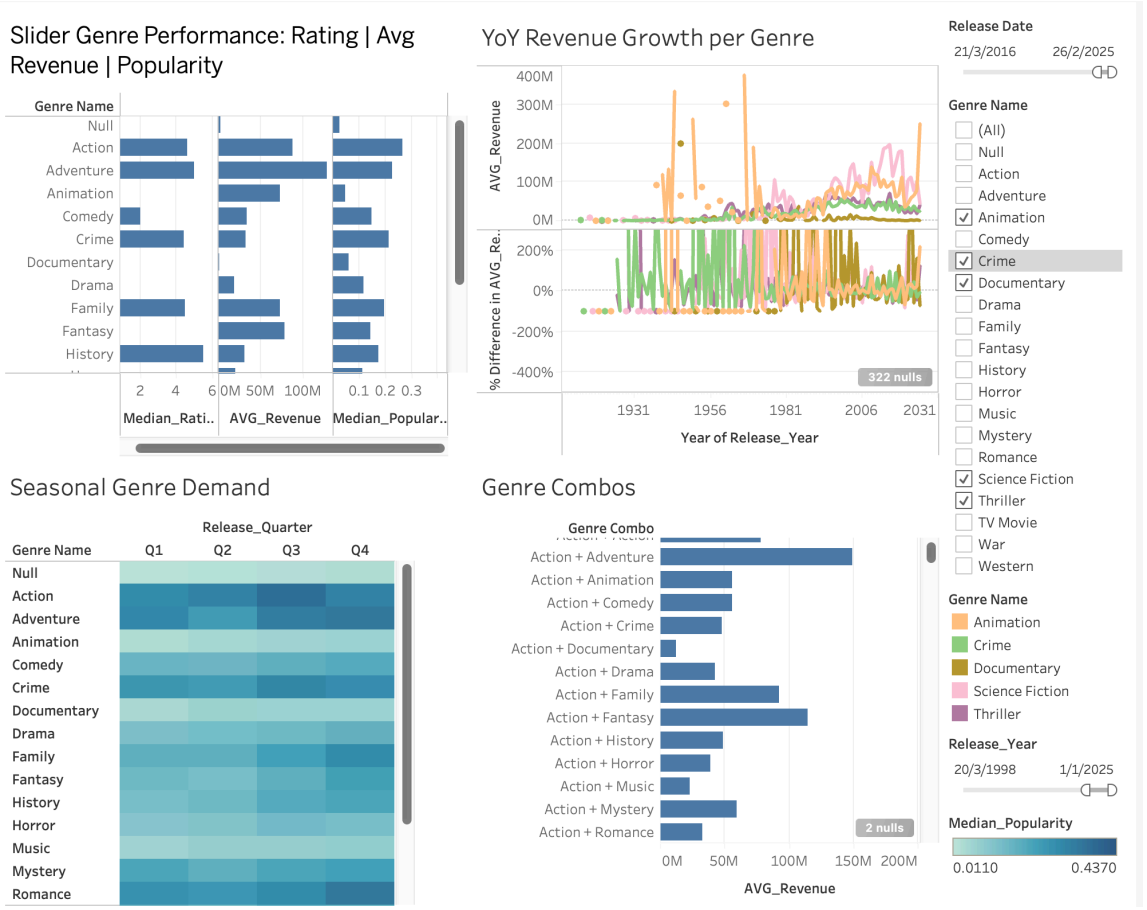
Our solution is a set of interactive tableau dashboards, they deliver a clear and visual representation of the relationships of multiple attributes in our dataset. The dashboard allows users (industry specialists) to filter and compare key variables. This system is valuable because it can help uncover trends that might be hidden in the raw data. Lastly, it can be monetized through the use of consulting or even as a SaaS product for big film producers.

We focused our analysis in four main research areas: genre and story selection, release timing and runtime, budget and ROI and casting and talent. We selected these research topics because we believe they were the most influential in the film's financial success.

6.1 Genre & Story Selection

With the objective of guiding directors, producers and other stakeholders in the selection of their stories and genres when developing new movies, we aim to provide them with a user-friendly dashboard (ideally updated in real time through the API). This dashboard will ensure users can easily visualise and stay up to date with the market’s growing genre niches, seasonality of certain stories as well as acknowledging the most popular and cost effective genres. Our guiding metrics for performance will be median popularity, median rating and average revenue. Median was chosen for the first two metrics as statistics were found to be skewed and median provided a more representative measure yet due to the null values in Revenue average was used instead for it. Below a preview is available of what the dashboard would look like.

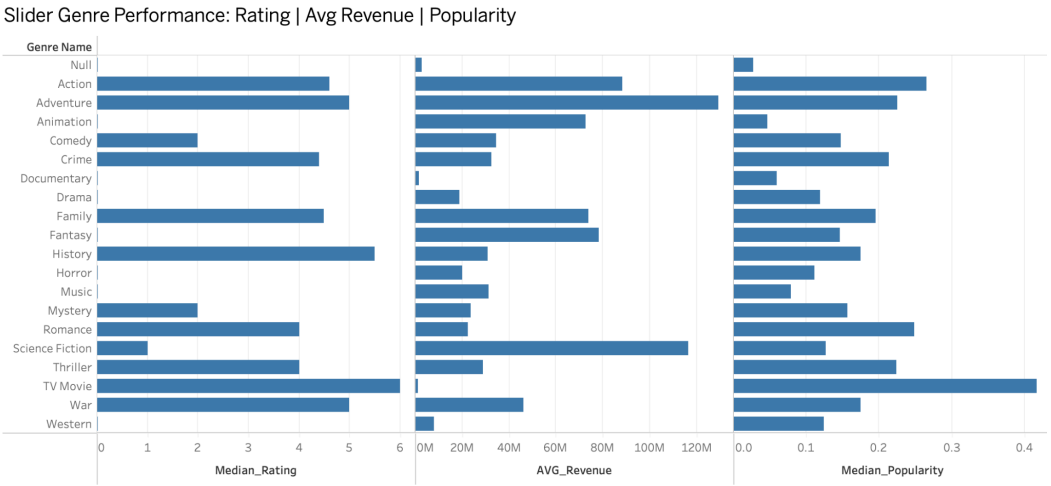
Figure 9: *Dashboard Genres Preview*



1. **Revenue & Rating by Genre:**

As a first crucial insight for our users, we offer a visualisation of the genres with highest median revenue and user rating. Tableau dashboard provides a sliding tab allowing users to select the time frame they wish to investigate.

Figure 10: *Performance per Genre*



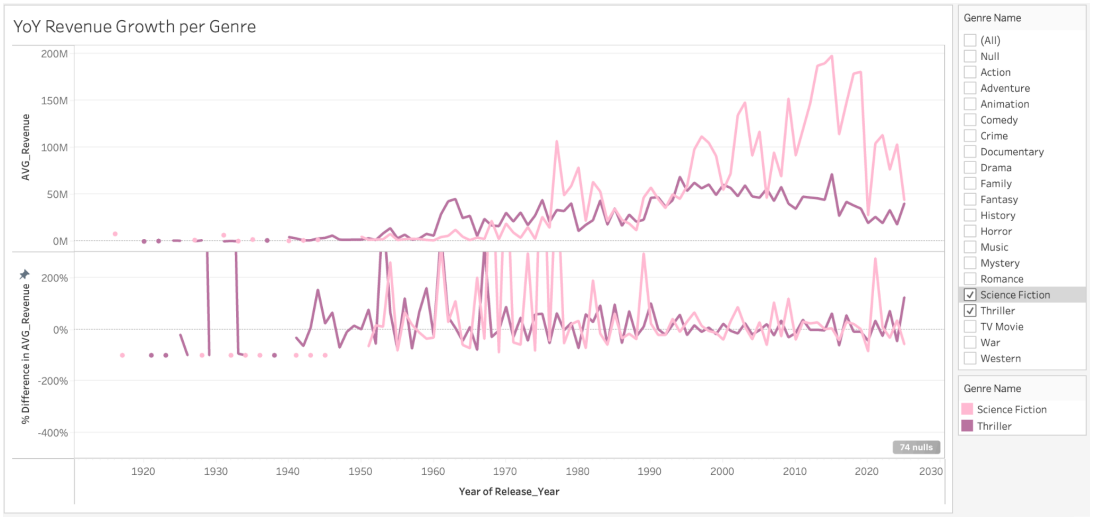
A snapshot of Genre Performance over the last 5 years already gives valuable insights. Over the last 5 years adventure seems like an all rounder with the highest average box office and offering a strong support in rating and popularity from viewers. It's also surprising how Science fiction despite leading the AVG Revenue lags audience support (median rating - 1). Explanation for this might be a few tentpoles like Marvel's Guardians, Star Wars which pump

the average revenue significantly even though most science fiction films make far less. The idea is to provide users with insights they can then explore deeper into to gain a competitive advantage.

2. **Evolution of Genres over Time**

As valuable as a static view might be, it's important for producers to acknowledge how tastes are evolving. At the top left users can plot the average revenue time series as well as YoY growth per genre. This can be critical in spotting emerging niche genres or lower volume genres that are showing rising revenue.

Figure 11: *Sci-Fi vs Thriller Revenue Evolution*

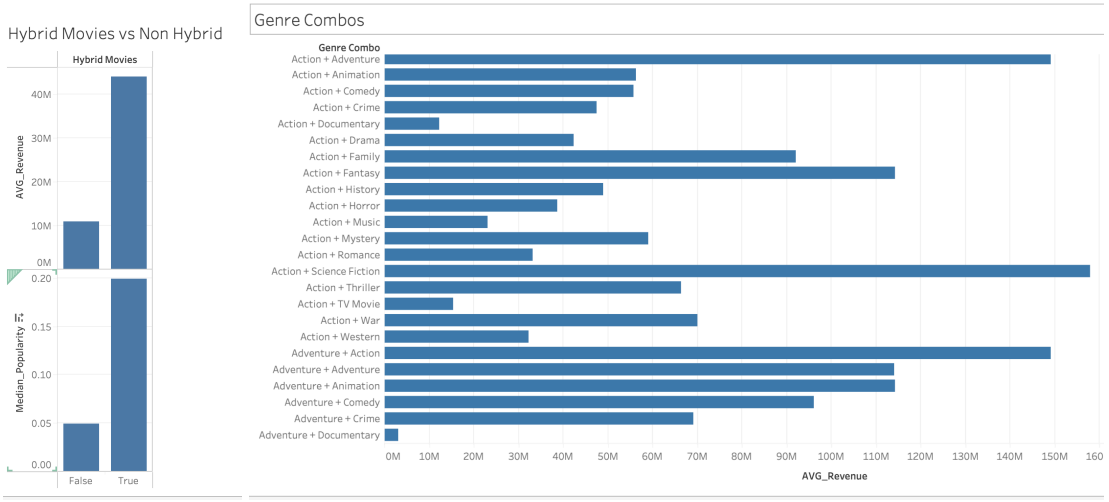


For a Head of Acquisitions Buyer persona which aims to find genres which are quietly building box-office momentum to decide which projects to greenlight, these plots can be very useful. The dashboard above filters “Thriller” and “Sci-fi” genre’s evolution is very insightful for a content strategist as it exposes Sci-Fi’s blockbuster boom with average revenue plummeting from under $20 M in the ’70s to peaks of $180 M in the 2010s. It also highlights thriller’s plateau with a steadier climb over the same decades. This plot contrasts volatility vs consistency. While Sci-Fi’s YoY swings consistently between +-200%, Thriller offers a dependable tight band almost always within ±20 % YoY. Head of acquisition strategies can effectively utilise these plots to spot emerging niches and balance risk and reward in its selection. Comparing genres further would grant us with more insights.

3. **Genre Hybrid Performance**

Given that our movies are tagged with multiple genres a possible research question for producers might be whether films tagged with multiple genres outperform single-genre films and if so in which combinations? An additional tab in the dashboard also allows for exploring.

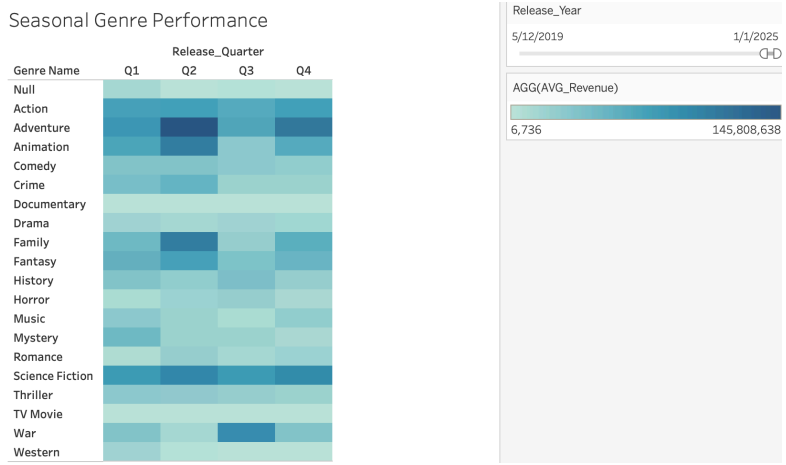
Figure 12: *Hybrid Genre Performance*



4. **Seasonal Genre Performance**

Finally it's important for producers and acquisition buyers to decide on release dates. Past data can be very useful in predicting when certain types of genres/ stories can perform better. An additional visualisation highlights whether certain genres perform better inspecific quarters or holiday windows. Allowing for a sliding window to include movies of only desired time frames.

Figure 13: *Heatmap for Seasonal Genre Revenue*



The heatmap above really brings how differently genres behave throughout seasons. Adventure, Science Fiction, Family and Action films are at their all time highest during quartile 2, with Adventure and Sci-Fi hitting some of the darkest cells in the entire map. Family movies in particular peak in spring suggesting studios probably release them based on march school breaks and to build momentum to ride into the full summer season. Quartile 4 seems like a strong second probably given the holiday audience surge. Evolution of these trends can also be observed sliding the release year bar at the left.

Ultimately, each producer, acquisitions buyer, content strategist or relevant stakeholder will have their own unique objectives and risk appetites, but this dashboard to empower decisions in terms of genre and story selection.

6.2 Realising Timing and Runtime

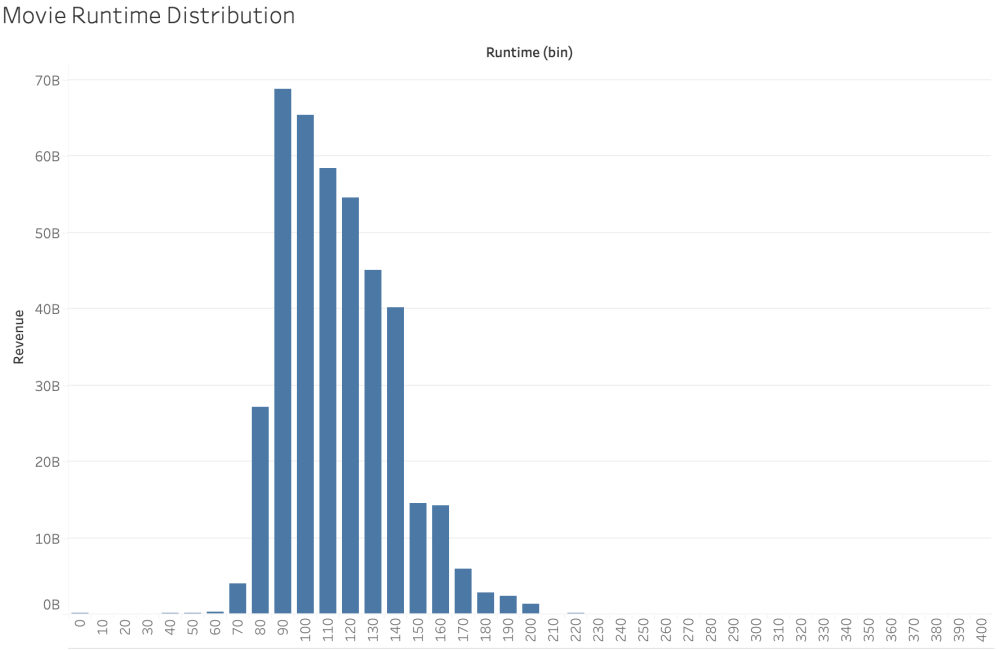
Moreover, we created another dashboard for release timing and runtime to see how it may influence movies success in terms of revenue and ROI. One would think these attributes might not affect the success of a movie but the trends show otherwise. Release timing can influence audience and revenue due to holidays, or competitive box office windows. In the same way the runtime affects the capacity in cinemas and can guide logistical decisions.

Figure 14: *Dashboard Release timing and runtime*

1. **Revenue by Runtime :**

It is important to understand the relationship between runtime and revenue. Runtime does not only affect audience engagement by also shorter films can be screened more frequently which could increase ticket sales. By analyzing this relationship stakeholders can identify the optimal runtime.

Figure 15: *Distribution of the Movie Runtime and Revenue*

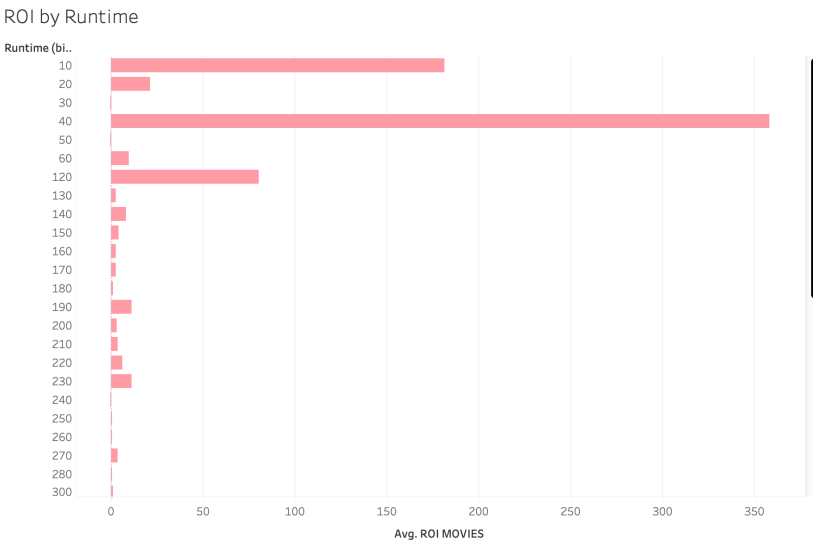


The bar chart above shows how movie revenue is affected by runtime. Helping identify the most financially successful duration. To improve the noise of individual munetes the runtime is grouped in 10 minute intervals to visualise the trend better. From the chart we can conclude that the optimal range appears to be between 80 and 120 minutes, this suggests that films that are within this duration might achieve a higher box office return.

2. **ROI by Runtime :**

Another valuable insight that can be analyzed is the ROI by runtime. This can help determine financial efficiency movies/tv series. Understanding this can help producers assess whether a certain film tends to deliver a better return. It can inform on budgeting, editing and more so resources are optimized.

Figure 16: *Distribution of the Movie Runtime and ROI*

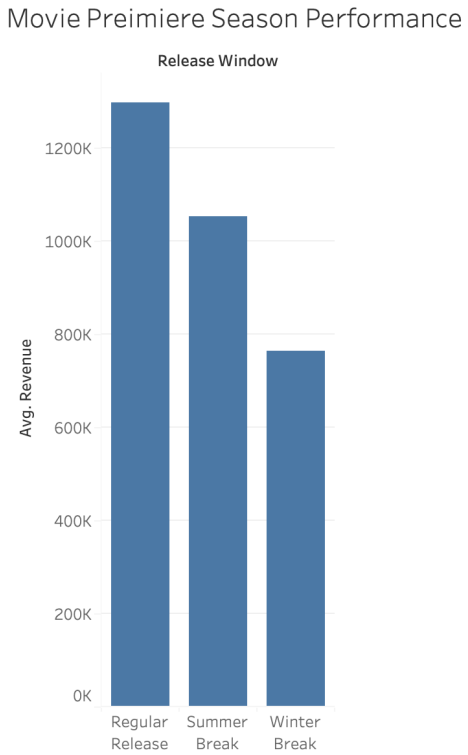


The chart reveals an unexpected pattern. The highest ROI between 40 min followed by 10 min and 120.This revealed that the spike at 40 min might reflect that films that had low budget for production and generated high revenues yet given the irregular pattern, it is evident that runtime alone cannot be a reliable predictor of ROI. Although there is a surprising is¡nsight, seen above, the second highest ROI was of 10 minute runtime which suggests that short films can perform very well in terms of ROI despite having lower absolut revenue. This can be given because of lower production costs and how even normal revenue can result in high ROI given that the initial investment was so small. This can give investors that do not have a large spending budget the opportunity to invest in multiple short films which show to have a better ROI and lower risk.

3. **Movie Premiere Season Performance :**

The season in which a movie premiers can influence the revenue given that if released during holidays and breaks, the films visibility can increase given that the audience's availability is higher. People are more prone to going to the movies during breaks and holidays than on random working days. This can be valuable to plan releases and maximize revenue.

Figure 17: *Movie premiere season performance*



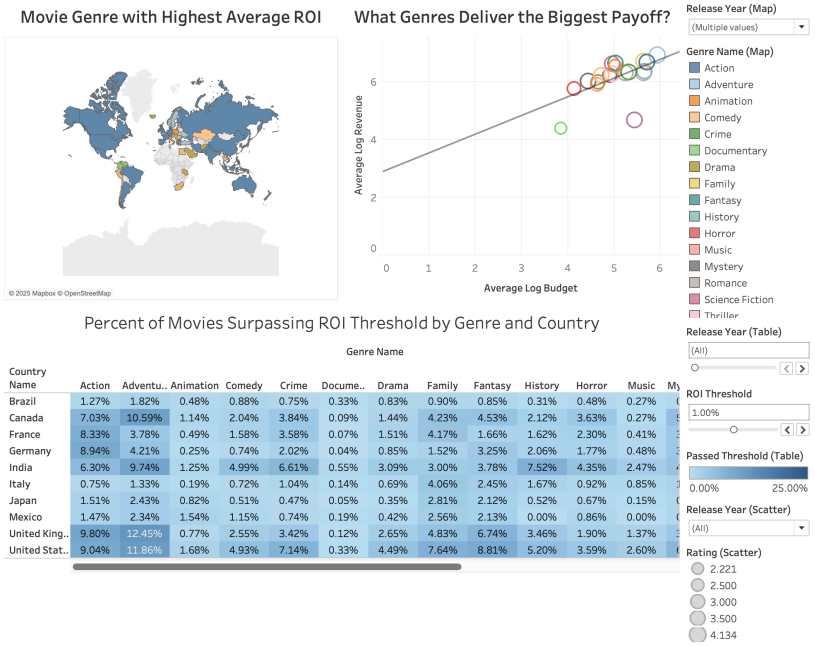
The graph above shows how even though the average revenue during a release in regular time is higher, we have to take into consideration that winter break and summer break are taking into account short periods of time that cannot account for all the other months. Showing how even in short periods like winter break given it accounts for less than a week, average revenue does perform relatively well. This suggests that despite the brevity these periods are highly lucrative windows for film premiers due to concentrated audience availability.

Investors can leverage this to maximise revenue potential, and align marketing spend with audience behavior.

6.3 Budget Allocation and ROIs

Furthermore, to deepen the support of financial decisions in movie production, we aim to provide a dashboard that focuses more on budget allocation strategies and expected return on investment (ROI). Building upon the insights from genre and story selection, this dashboard is designed to help producers and stakeholders identify optimal budget ranges that align with market trends and performance benchmarks, such as returns based on investment. By examining how different budgets correlate with revenue and popularity, with the addition of other aspects such as movie genre, locality and year since release date, the dashboard provides a data driven foundation for evaluating financial risk, forecasting potential gains, and setting realistic expectations from the production.

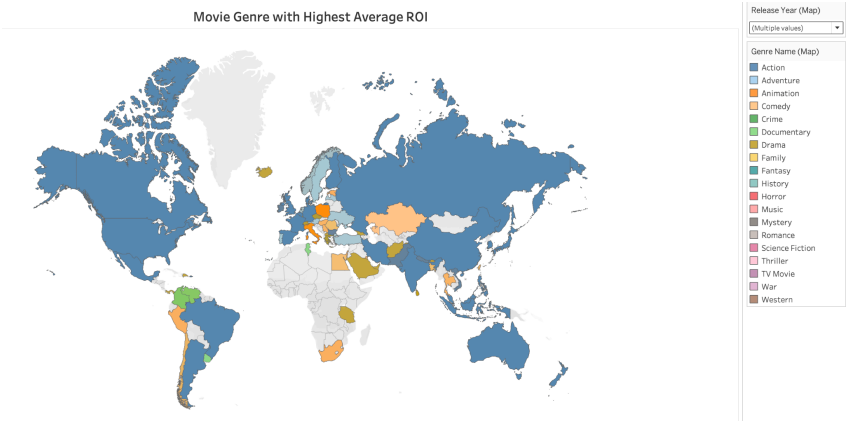
Figure 18: *Dashboard Financials Preview*



1. **Movie Genre with Highest Average ROI:**

As a first graph for the insights on financial aspects of movies, we provide a visualisation showing the genre with the highest average return on investment (ROI) in each country. Alongside it, we provide a dropdown menu for the user to further interact with the graph, allowing them to select a specific release year or custom range, to have a personalized visualization of the data.

Figure 19: *World Map showing Movie Genres with Highest Average ROI*



A snapshot of this, showing the results of movies released between 2016 and 2020, both inclusive, already showcase important insights useful for producers and directors. As of today, we can see that "Action" movies released in this period have consistently been able to achieve the highest average ROI in most countries around the world, suggesting that people enjoy high energy content. However, it's important to note that this trend may be heavily influenced by the presence of major international blockbusters within the genre. Regardless, in terms of financial returns, "Action" movies stand out as a reliable genre for studios aiming for profitability across diverse markets due to their broad audience appeal.

2. **Payoff and Rating per Genre:**

While the first graph offers valuable insights on the highest performing genre, it doesn't show the dynamics of all genres, for which we included a second visualization. This scatter plot compares revenue and budget while also incorporating average viewer ratings, which allows users to analyze how each genre performed both financially and critically. Similar to the first graph, this also contains a filter to allow the user to select the years they are interested in analysing. Additionally, for this graph we decided to use a the logarithm of both Budget and Revenue instead of the original values in order to reduce the impact of outliers.

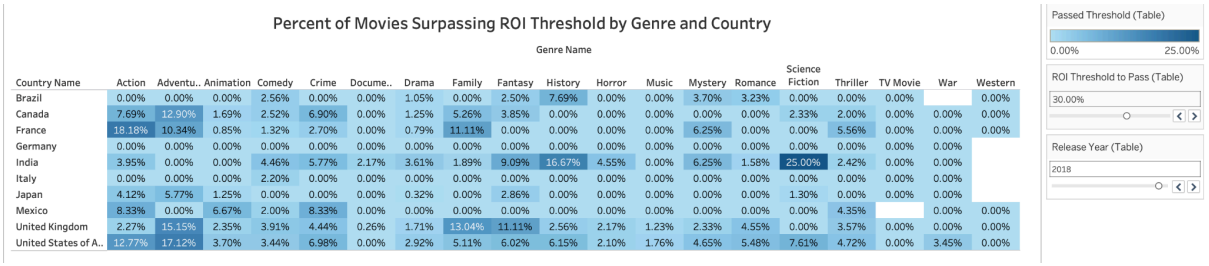
Figure 20: *Scatter Plot of Genres and Payoff*

In this snapshot from the same years as the first graph, from 2016 to 2020 inclusive, we can see that even though "Action" was the genre with the highest average ROI in most countries, almost all genres managed to achieve a relatively balanced relationship between budget and revenue. As most data points, representing each genre, cluster around the trend line, we can say that in general, higher budgets do lead to higher revenues. However, genres like "TV Movie" and "Documentary" fall below the trend line, suggesting that, even though they have lower production costs, they have a limited commercial performance. In contrast, genres such as "Adventure" and "Animation" not only perform well financially but also receive strong average viewer ratings, indicating both commercial and critical success.

3. **Percentage of Movies Surpassing a ROI Threshold by Genre and Country:**

Finally, we understand that one crucial aspect producers and companies consider when developing their productions is the likelihood of financial success. To support this, and to help set realistic goals that align creative decisions with current market trends, we've included a table showing the percentage of movies, by genre, that surpassed a specified ROI threshold. Users are able to interact with this table by selecting the release year and defining their desired ROI benchmark, allowing them to have a personalized table to see how realistic their goals and expectations are. We have limited the table to only show the top 10 countries with the highest number of movies in order to prevent it from cluttering.

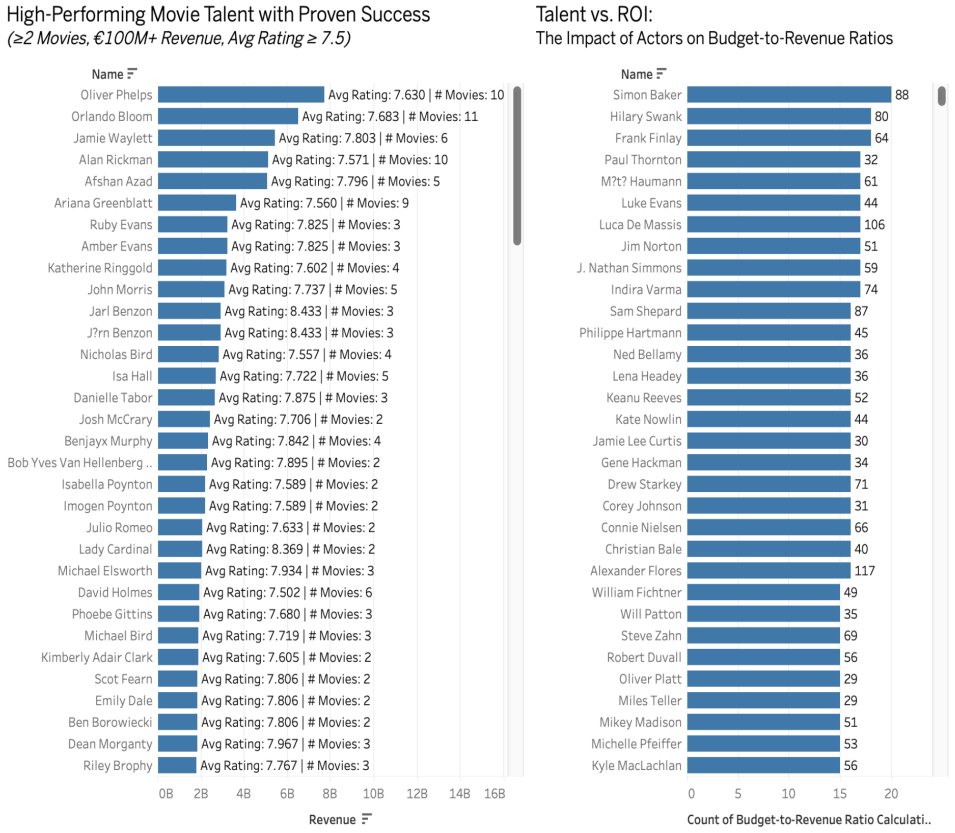
Figure 21: *Heat map of Movies surpassing ROI threshold by Genre and Country*



With this sample snapshot of the table, which selected the movies released in 2018 and an ROI of 30%, which means movies that earned at least 30% more than what its production cost was, we can already see some insights that can be interesting for producers to know. Firstly, we can see that 25% of "Science Fiction" movies from India met this benchmark, making it one of the highest from the table. Additionally, 17% of "Adventure" and 12% of "Action" movies produced in the United States also met this threshold, which means that these genres were still successful to the eyes of the user. Furthermore, it highlights that idea that more nice genres, such as "History" and "Western", released in the selected year, did not manage to reach the desired ROI threshold, indicating lower commercial success in comparison to other genres. With this tool, we believe that producers will be able to gain a clearer understanding of the market and use this information to make more informed decisions about what type of productions will most likely be able to deliver the desired results and meet their expectations.

6.4 Casting & Talent

Figure 22: *Actors Dashboard (1/2)*

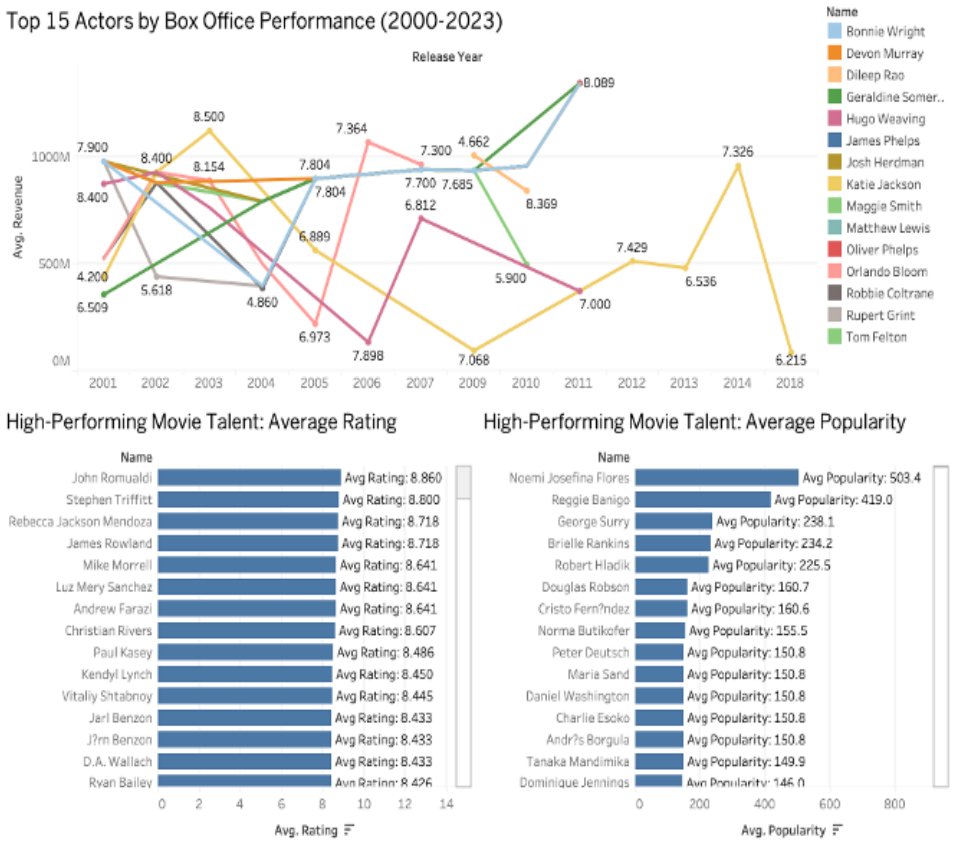


Moreover, producers, casting directors, and studio executives will gain insights about the top actors as well as underrated rising talent.

This dashboard gives them a wide perspective on talent performance. It explores both famous actors and rising talent focussing on critical success components and their financial returns. Figure 22 (High-Performing Movie Talent with proven Success) includes movie actors who have appeared in at least two movies with earnings exceeding €100M and an average rating of 7.5 or higher. This provides a strong reference for inexpensive and reliable lead or supporting cast. Notable actors like Orlando Bloom and Alan Rickman provide high consistency and success with 11 and 10 qualifying films, respectively. Meanwhile, Danielle Tabor and Afshan Azad possess few but extremely high-rated films, representing low-risk, high-quality casting options for targeted projects. As a result, this allows viewing two extremes. On the one hand, scaling up with famous talent while on the other hand optimizing for quality with selected less famous actors.

Furthermore, Figure 22 (Talent vs. ROI) provides an ROI-based casting analysis by ranking actors according to the number of times their films achieve favorable results on budget-to-revenue ratios. This measure is key to independent studios or productions with budget constraints. As one would expect, famous actors like Simon Baker and Hilary Swank rank very high on this list. However, it also includes lesser known actors like Alexander Flores or Luca De Massis. This insight enables stakeholders to optimize the balance between box office revenue and profitability, thereby enhancing financial outcomes in limited funding scenarios by choosing talents like Alexander Flores.

Figure 23: *Actors Dashboard (2/2)*

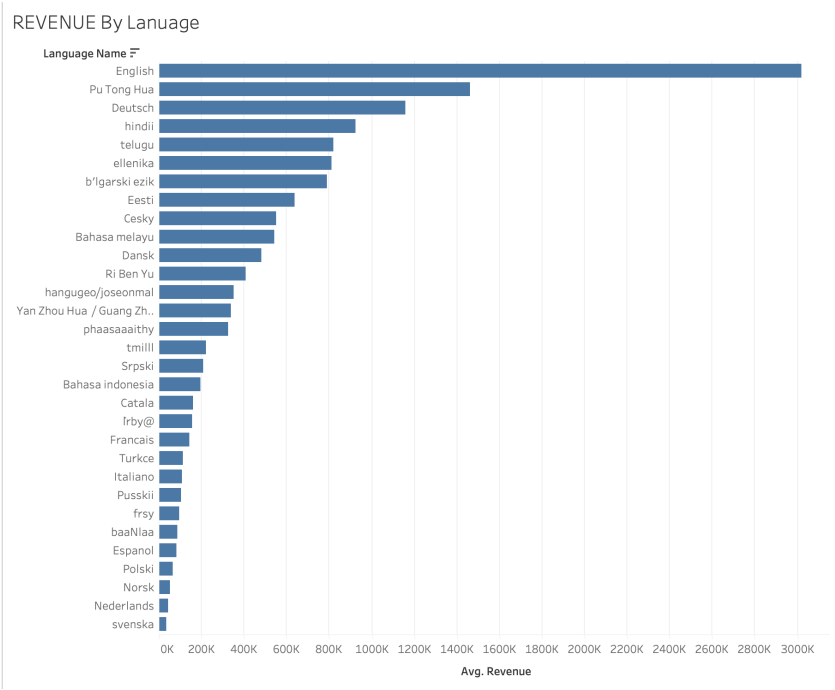


In order to deepen the understanding on the importance of specific actors, the second dashboard aims to serve production companies monitoring both individual actor career trajectories and their contribution to success and popularity. This dashboard provides a temporal evaluation of top-performing talent, average rating, and popularity over time. Figure 23 (Top 15 Actors by Box Office Performance) displays the top 15 actors by box office performance from 2000 to 2023. This allows the assessment of longevity and appeal over time. Only a few actors such as Devon Murray, Geraldine Somerville, and James Phelps showcased steady performances tied to release dates, while some of the rest seem to experience steeper declines, which indicates volatility of the movie industry. Furthermore, Figure 23 (High Performing Movie Talent: Average Rating) ranks actors by their average movie ratings, giving directors insights into consistent critical success of actors. John Romualdi is leading the chart with an average rating of 8.86 and is closely followed by Stephen Triffitt with 8.80, and Rebecca Jackson Mendoza with 8.718. The insights about these actors would be crucial for studios seeking big prestigious releases, for which they need to identify high artistic talent. Finally, Figure 23 (High Performing Movie Talent: Average Popularity) highlights actors with the highest average popularity, which is in this case often associated with streaming views, social media interactions and other engagements. In this case Noemi Josefina Flores and Reggie Banigo are at the top of the list, showcasing very high average popularity of their movies. Furthermore, taking a closer look at these names, one might wonder that neither Noemi nor Reggie are featured at the top of the ROI or other rating charts. This shows that financial returns might not necessarily correlate with marketing value, especially for niche movies.

6.5 Language & Strategy

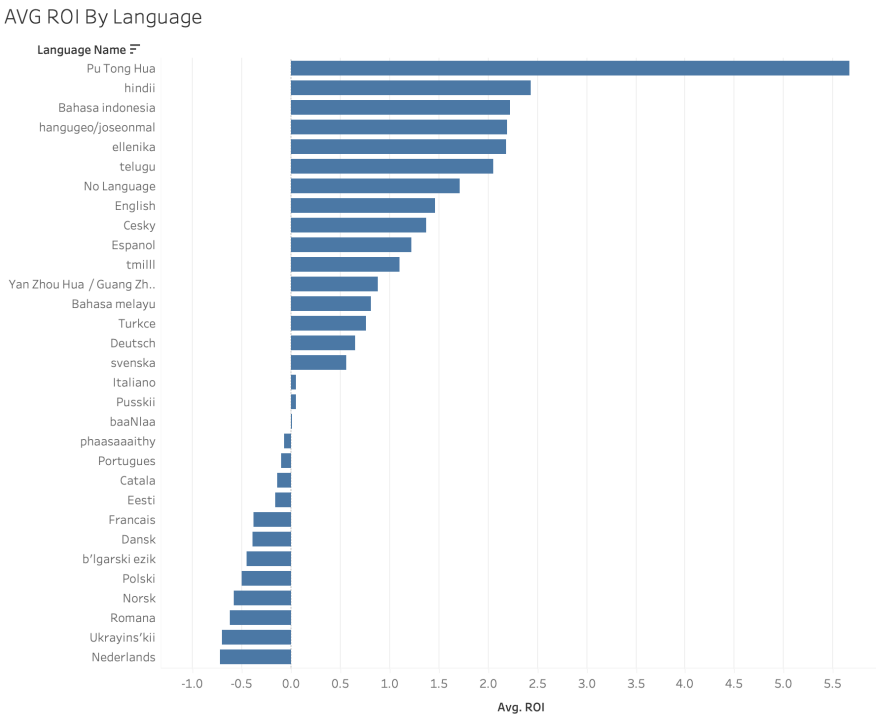
Our final dashboard set focuses on language-driven dynamics, which languages supply the largest catalogues, deliver the best financial returns, and partner most effectively with individual genres. The four visuals below provide decision-makers and stakeholders with a helpful guide for planning at the international level.

Figure 23: *Language Dashboard (1/2)*



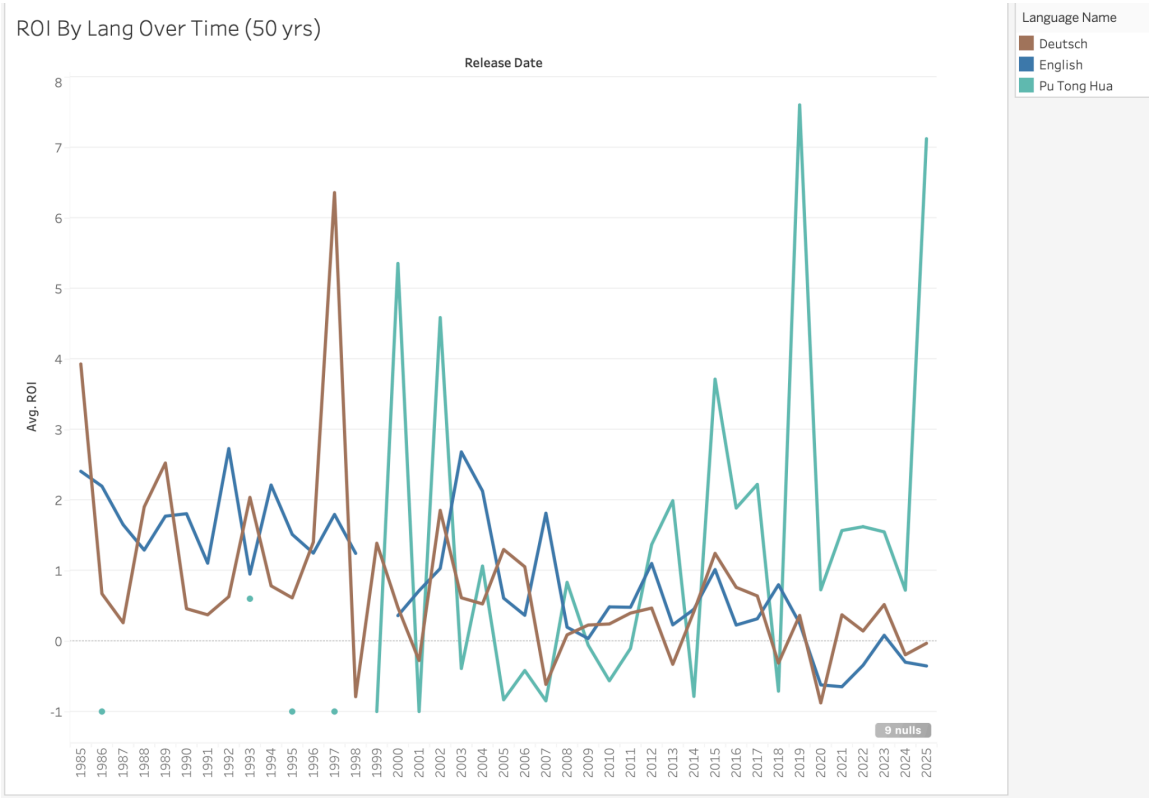
The first panel ranks languages by the average amount of revenue made across those movies. We filtered movies that had languages of at least 10’000 films to avoid including overly niche dialects. English, Chinese (Pu Tong Hua), Hindi and Deutsch are the most prominent. For distributors, this can help highlight where subtitle or dubbing investment will secure the broadest reach. It’s important to note that this is more a metric of popularity and not profitability, as we avoid taking into account the budget used in each measurement.

For this reason, we included the same graph but using ROI as the average metric. A different hierarchy emerges. Medium-sized markets—Korean, Telugu and Indonesian—outperform English-language productions on a percentage basis, suggesting leaner cost structures and strong domestic audiences. Studios seeking **high-margin co-production partners** may gain more by financing local content in these languages than by pursuing already saturated English releases.



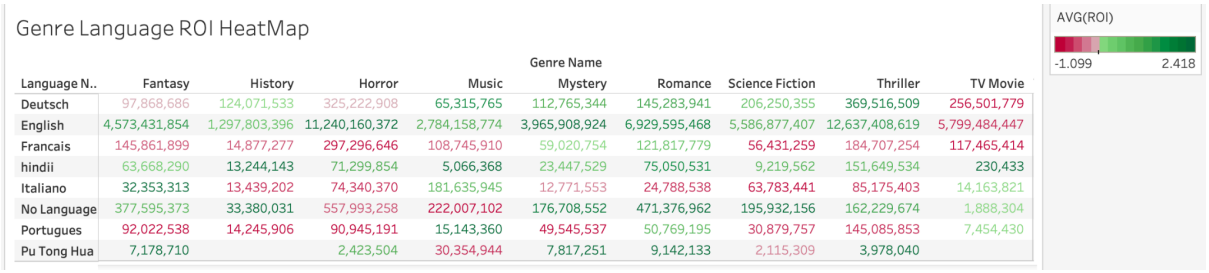
When implementing this perspective, a different ranking is apparent. Medium sized markets such as Korean, Hindi and Indonesian outperform English language productions on a percentage basis which suggests leaner cost structures and strong domestic audiences. Studios seeking high margin production partners may gain more by financing local content in these languages than by pursuing already saturated English releases.

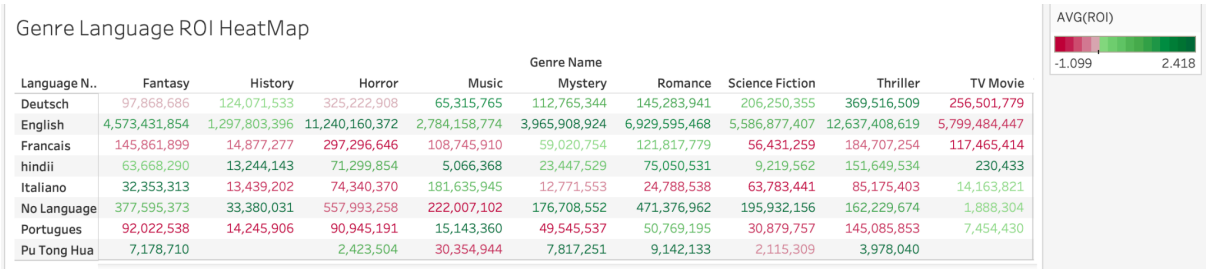
Figure 24: *Language Dashboard (2/3)*



For our second dashboard, we did a similar analysis of language and ROI but this time we took the last 50 years as a measurement so we could garner a sense of momentum], direction or trend in languages. It revealed that Chinese ROIs have climbed steadily since the mid-2000s, mirroring the rise of the Chinese entertainment industry at the turn of the century. On the other hand, English ROI has had a slight downward trend since 2015, squeezed by ballooning blockbuster budgets. For simplicity of interpretation we are limited to three languages, where we can see Deutsch has had a similar trend to that of English. For planners, this chart informs long range localisation road maps by showing where to expand dubbing and marketing budgets next, and which languages and cultures require caution upon entering.

Figure 25: *Language Dashboard (3/3)*





The final panel cross references genre with language to expose sweet spots. The visual square heat map can illustrate clear patterns in content specialisation across different markets. Music and Romance titles are overwhelmingly English or German, reinforcing those markets' broad commercial reach. Thriller and Science Fiction content is surprisingly strong in Portuguese, whereas German and English are helpful for platforms chasing pan-regional or global audiences. Historical and Documentary films skew towards English, German and French, reflecting both catalogue depth and structured funding models in those regions. These intersections can help guide acquisition. For example, a US streamer targeting Latin America could prioritise Portuguese-language thrillers, while a European broadcaster might co-produce French documentaries to diversify its slate.

7. Legal Implications

Regarding the legal compliances of our project, we took for reference the principles of the General Data Protection Regulations. This ensured that our solution would comply with the data protection law in the European union. Given that we were not working with personal data we still implemented the GDPR design principles to ensure that our solution is transparent and scalable under major data governance regulations.

Our data type ensures privacy, given that we only use public non personal metadata from the TMDB API. This means we are not identifying any personal data directly. Moving on with the purpose, we had it clear that from the beginning that the data we were going to use would be exclusively for generating the insights to help guide film industry specialists in future investment decisions. This is aligned with the purpose limitation principle referenced in the GDPR. Our approach also minimizes data, as we only extracted the relevant data necessary for analysis. This avoids collecting excessive and unrelatable fields. Regarding accountability and liability, the dashboard profive only informational support. We ensure there is transparency by emphasising that our solution is a tool that is meant to support decision making but cannot back up any outcome.

Concluding that, while the GDPR is not directly established for projects that use non personal data, we have followed the principles to ensure we are legally robust so this can become a tool that is trustworthy and scalable.

8. Conclusions and Further Work

7.1 Conclusions

Through the process of creating the database for movie performance analysis, we realized the value of constructing a well-structured database to ensure scalability, making it capable of handling massive amounts of data. With this in mind, we focused on designing an efficient schema that would allow the data to be queried effectively for future analysis. Then we saved the data in PostgreSQL by extracting movie metadata from the TMDB API and structured it into a normalized format optimized for efficiency. However, during implementation, we encountered issues that prevented SQL from functioning as expected due to the complexity of one of the Actors tables which did not enable us to extract the data correctly, after brainstorming we were able to find the way of merging that table into the SQL. This adjustment allowed us to successfully integrate the necessary data and proceed with building accurate, fully functional dashboards.

One of the key challenges was translating raw data, particularly in JSON format, into a structured relational format. Flattening complex nested structures such as genres, production firms, and spoken languages, it was essential to creating discrete tables connected via primary and foreign keys. However, despite our efforts to standardize the data and optimize performance, we faced technical limitations that prevented successful execution in SQL. These challenges highlighted the complexities of relational database management and reinforced the importance of carefully designing data transformations.

Additionally, we encountered issues such as rate limiting when extracting data from the TMDB API and the need for recursive date filtering to retrieve the complete dataset. These obstacles demonstrated the necessity of building robust data ingestion pipelines capable of handling large-scale operations while accounting for the reliability of external data sources.

The separation of movie and TV show domains was another significant design decision, allowing us to capture the unique attributes of each medium without overcomplicating the structure. This approach ensured flexibility for future expansions of each category. While our current implementation provides a foundational framework for analyzing movie performance and storing the data, we know several opportunities for improvement in future work, particularly in refining the database structure and overcoming the limitations that we faced during the SQL implementation.

Finally with our tableau dashboard solution, we were able to convert raw data into actionable insights for content strategists, producers, acquisition buyers, movie investors and various other stakeholders. This solution empowers decision makers, especially in the key research areas we've focused on.

7.2 Further Work

Looking ahead, the next steps for this project will focus on deepening the analysis that can uncover new patterns. Refining the dashboards to offer more unseen relationships, and integrating forecasting and machine learning in order to enhance the tools capabilities to become a predictive decision support platform.

Future work:

Enhanced Data Integration: Incorporating additional data sources to significantly enrich the analysis capabilities. Box office data from sources like Box Office Mojo, critic reviews from Metacritic or Rotten Tomatoes, and social media sentiment analysis would provide complementary perspectives on movie performance beyond TMDB's metrics.

Advanced Analytics Implementation: Building predictive models to forecast movie performance based on factors like cast, director, genre, and release timing represents a valuable next step. Machine learning approaches could identify non-obvious patterns in historical data that correlate with box office success.

Real-Time Data Processing: Upgrading our batch processing system to incorporate streaming data would enable monitoring of movie performance as it happens. This could be particularly valuable for tracking social media response during opening weekends.

SaaS platform: Create a specific tailored platform similar as tableau but tailored specifically to the customer. Whether it's film studios, distributors or production companies. This would enable the tool to be specifically useful for individual companies transforming it to a tailored strategic planning solution.

By pursuing these enhancements, we could transform our current decision support system for film industry professionals into one that is real time updated with new movies and includes advanced analytics implementation making it a SaaS product.

Appendix

Academic References

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

Foidl, H., Golendukhina, V., Ramler, R., & Felderer, M. (2024). Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers. *Journal of Systems and Software, 207*, Article 111855. https://doi.org/10.1016/j.jss.2023.111855

Kim, A., Trimi, S., & Lee, S.-G. (2021). Exploring the key success factors of films: A survival analysis approach. *Service Business, 15*(4), 613–638. https://doi.org/10.1007/s11628-021-00460-x

Lash, M. T., & Zhao, K. (2016). Early predictions of movie success: The who, what, and when of profitability. *Journal of Management Information Systems, 33*(3), 874–903. https://doi.org/10.1080/07421222.2016.1243969

Ridzuan, F., & Wan Zainon, W. M. N. (2019). A review on data cleansing methods for big data. *Procedia Computer Science, 161*, 731–738. https://doi.org/10.1016/j.procs.2019.11.177

Serbout, S., El Malki, A., Pautasso, C., & Zdun, U. (2023). API rate limit adoption – a pattern collection. In *Proceedings of the 28th European Conference on Pattern Languages of Programs (EuroPLoP 2023)*. ACM. https://doi.org/10.1145/3628034.3628039

The Movie Database. (n.d.). *TMDB API documentation*. Retrieved April 3, 2025, from https://developer.themoviedb.org/docs

Code

2.1 Data Processing Function.

```
# NECESSARY TO ESTABLISH CONNECTION FIRST
tmdb.API_KEY = 'PERSONAL_KEY'

# ----------------------------
# Data Processing Functions
# ----------------------------
def get_movie_details(movie_id):
    """Fetch detailed data for a movie by its ID."""
    movie = tmdb.Movies(movie_id)
    try:
        details = movie.info()
        return details
    except Exception as e:
        print(f"Error retrieving movie {movie_id}: {e}")
        return None

def process_main_data(details):
    """Extract single-valued (non-normalized) fields for the main movies
table."""
    return {
        "id": details.get("id"),
        "adult": details.get("adult"),
        "backdrop_path": details.get("backdrop_path"),
        "budget": details.get("budget"),
        "homepage": details.get("homepage"),
        "imdb_id": details.get("imdb_id"),
        "original_language": details.get("original_language"),
        "original_title": details.get("original_title"),
        "overview": details.get("overview"),
        "popularity": details.get("popularity"),
        "poster_path": details.get("poster_path"),
        "release_date": details.get("release_date"),
        "revenue": details.get("revenue"),
        "runtime": details.get("runtime"),
        "status": details.get("status"),
        "tagline": details.get("tagline"),
        "title": details.get("title"),
        "video": details.get("video"),
        "vote_average": details.get("vote_average"),
        "vote_count": details.get("vote_count")
    }

def process_collection(details):
    """Extract collection info (belongs_to_collection) into its own
record."""
    coll = details.get("belongs_to_collection")
    if coll:
        return {
            "movie_id": details.get("id"),
```

```
            "collection_id": coll.get("id"),
            "collection_name": coll.get("name")
        }
    return None

def process_genres(details):
    """Extract the list of genres for a movie."""
    genres_list = details.get("genres", [])
    rows = []
    movie_id = details.get("id")
    for genre in genres_list:
        rows.append({
            "movie_id": movie_id,
            "genre_id": genre.get("id"),
            "genre_name": genre.get("name")
        })
    return rows

def process_production_companies(details):
    """Extract production companies for a movie."""
    companies = details.get("production_companies", [])
    rows = []
    movie_id = details.get("id")
    for comp in companies:
        rows.append({
            "movie_id": movie_id,
            "company_id": comp.get("id"),
            "company_name": comp.get("name"),
            "origin_country": comp.get("origin_country")
        })
    return rows

def process_production_countries(details):
    """Extract production countries for a movie."""
    countries = details.get("production_countries", [])
    rows = []
    movie_id = details.get("id")
    for country in countries:
        rows.append({
            "movie_id": movie_id,
            "iso_3166_1": country.get("iso_3166_1"),
            "country_name": country.get("name")
        })
    return rows

def process_spoken_languages(details):
    """Extract spoken languages for a movie."""
    languages = details.get("spoken_languages", [])
    rows = []
    movie_id = details.get("id")
    for lang in languages:
        rows.append({
            "movie_id": movie_id,
```

```
            "iso_639_1": lang.get("iso_639_1"),
            "language_name": lang.get("name")
        })
    return rows

# ----------------------------
# Helper: Process a Single Movie
# ----------------------------
def process_movie(movie, main_writer, coll_writer, genres_writer,
                  prod_comp_writer, prod_countries_writer, spoken_writer,
lock):
    """
    Process a single movie: fetch details, write data to CSV (within a lock),
    and then sleep to help respect rate limits.
    """
    movie_id = movie.get("id")
    details = get_movie_details(movie_id)
    if details is None:
        return
    with lock:
        main_data = process_main_data(details)
        main_writer.writerow(main_data)

        collection_data = process_collection(details)
        if collection_data:
            coll_writer.writerow(collection_data)

        for row in process_genres(details):
            genres_writer.writerow(row)
        for row in process_production_companies(details):
            prod_comp_writer.writerow(row)
        for row in process_production_countries(details):
            prod_countries_writer.writerow(row)
        for row in process_spoken_languages(details):
            spoken_writer.writerow(row)
    # Sleep to respect API rate limits.
    time.sleep(0.25)
```

2.2 Recursive Date Range Processing with Parallelization

```
def process_date_range(gte, lte, main_writer, coll_writer, genres_writer,
                       prod_comp_writer, prod_countries_writer,
spoken_writer, flush_files, lock):
    """
    Process movies released between gte and lte.
    If the TMDB discover query returns 500 or more pages, split the range.
    For each page, process movies in parallel and flush file buffers after
every 500 pages.
    """
    print(f"\nProcessing movies released from {gte} to {lte}")
    discover = tmdb.Discover()
    params = {
        "page": 1,
```

```
        "primary_release_date.gte": gte,
        "primary_release_date.lte": lte
    }

    try:
        first_page_response = discover.movie(**params)
    except Exception as e:
        print(f"Error fetching first page for range {gte} to {lte}: {e}")
        return

    total_pages = first_page_response.get("total_pages", 1)
    print(f"  Total pages in this range: {total_pages}")

    # If we hit the 500-page cap, split the range.
    if total_pages >= 500:
        start_date = datetime.datetime.strptime(gte, "%Y-%m-%d")
        end_date = datetime.datetime.strptime(lte, "%Y-%m-%d")
        if start_date >= end_date:
            print("  Date range too narrow to split further. Processing with
current cap.")
        else:
            mid_date = start_date + (end_date - start_date) / 2
            mid_str = mid_date.strftime("%Y-%m-%d")
            print(f"  Splitting range: {gte} to {mid_str} and {mid_str} to
{lte}")
            process_date_range(gte, mid_str, main_writer, coll_writer,
genres_writer,
                               prod_comp_writer, prod_countries_writer,
spoken_writer,
                               flush_files, lock)
            process_date_range(mid_str, lte, main_writer, coll_writer,
genres_writer,
                               prod_comp_writer, prod_countries_writer,
spoken_writer,
                               flush_files, lock)
            return

    pages_processed = 0
    # Process each page in the current date range.
    for page in range(1, total_pages + 1):
        params["page"] = page
        print(f"  Processing page {page}/{total_pages} for range {gte} to
{lte}...")
        try:
            response = discover.movie(**params)
        except Exception as e:
            print(f"  Error on page {page}: {e}")
            continue

        movies = response.get("results", [])
        # Process movies in parallel using a ThreadPoolExecutor.
        with concurrent.futures.ThreadPoolExecutor(max_workers=10) as
executor:
```

```
            futures = [executor.submit(process_movie, movie, main_writer,
coll_writer,
                                       genres_writer, prod_comp_writer,
prod_countries_writer,
                                       spoken_writer, lock)
                       for movie in movies]
            # Wait for all submitted tasks to complete.
            concurrent.futures.wait(futures)

        pages_processed += 1
        # Flush file buffers after every 500 pages.
        if pages_processed % 500 == 0:
            for file_obj in flush_files.values():
                file_obj.flush()
            print(f"  Processed {pages_processed} pages; file buffers
flushed.")
```

2.4 Main ETL Process and Loading

```
def main():
    # Define output CSV filenames.
    main_file = "tmdb_movies_main.csv"
    collection_file = "tmdb_movie_collection.csv"
    genres_file = "tmdb_movie_genres.csv"
    production_companies_file = "tmdb_movie_production_companies.csv"
    production_countries_file = "tmdb_movie_production_countries.csv"
    spoken_languages_file = "tmdb_movie_spoken_languages.csv"

    # Define CSV headers for each file.
    main_headers = [
        "id", "adult", "backdrop_path", "budget", "homepage", "imdb_id",
        "original_language", "original_title", "overview", "popularity",
        "poster_path", "release_date", "revenue", "runtime", "status",
        "tagline", "title", "video", "vote_average", "vote_count"
    ]
    collection_headers = ["movie_id", "collection_id", "collection_name"]
    genres_headers = ["movie_id", "genre_id", "genre_name"]
    production_companies_headers = ["movie_id", "company_id", "company_name",
"origin_country"]
    production_countries_headers = ["movie_id", "iso_3166_1", "country_name"]
    spoken_languages_headers = ["movie_id", "iso_639_1", "language_name"]

# Open CSV files for writing.
    with open(main_file, mode="w", newline="", encoding="utf-8") as main_csv,
\
         open(collection_file, mode="w", newline="", encoding="utf-8") as
coll_csv, \
         open(genres_file, mode="w", newline="", encoding="utf-8") as
genres_csv, \
         open(production_companies_file, mode="w", newline="",
encoding="utf-8") as prod_comp_csv, \
         open(production_countries_file, mode="w", newline="",
encoding="utf-8") as prod_countries_csv, \
```

```
         open(spoken_languages_file, mode="w", newline="", encoding="utf-8")
as spoken_csv:

        main_writer = csv.DictWriter(main_csv, fieldnames=main_headers)
        coll_writer = csv.DictWriter(coll_csv, fieldnames=collection_headers)
        genres_writer = csv.DictWriter(genres_csv, fieldnames=genres_headers)
        prod_comp_writer = csv.DictWriter(prod_comp_csv,
fieldnames=production_companies_headers)
        prod_countries_writer = csv.DictWriter(prod_countries_csv,
fieldnames=production_countries_headers)
        spoken_writer = csv.DictWriter(spoken_csv,
fieldnames=spoken_languages_headers)

        # Write headers to each CSV file.
        main_writer.writeheader()
        coll_writer.writeheader()
        genres_writer.writeheader()
        prod_comp_writer.writeheader()
        prod_countries_writer.writeheader()
        spoken_writer.writeheader()

        # Dictionary of file objects for flushing.
        flush_files = {
            "main": main_csv,
            "collection": coll_csv,
            "genres": genres_csv,
            "prod_companies": prod_comp_csv,
            "prod_countries": prod_countries_csv,
            "spoken": spoken_csv
}

        # Create a lock for thread-safe CSV writing.
        lock = threading.Lock()

        # Define the full date range: from 1900-01-01 to today.
        start_date = "1900-01-01"
        today_str = datetime.datetime.today().strftime("%Y-%m-%d")
        process_date_range(start_date, today_str,
                           main_writer, coll_writer, genres_writer,
                           prod_comp_writer, prod_countries_writer,
spoken_writer,
                           flush_files, lock)

        # Final flush after all processing is complete.
        for file_obj in flush_files.values():
            file_obj.flush()
        print(f"\nCompleted processing movies from {start_date} to
{today_str}. Files flushed to disk.")

if __name__ == "__main__":
    main()
```

2.5 Transformation Data into ASCII code

```
def clean_text(value):
    if isinstance(value, str):
        ascii_equiv = unidecode(value)  # Convert accented characters
        return "".join(char if ord(char) < 128 else '?' for char in
ascii_equiv)  # Replace non-ASCII with '?'
    return value

# Change Daa types
# Convert movie_id to integer (or keep as string if it has leading 0s)
df['movie_id'] = pd.to_numeric(df['movie_id'], errors='coerce')
```

2.6 Transformation Correct Data Types

```
# Convert adult and video to boolean
df['adult'] = df['adult'].astype(str).str.lower().map({'true': True, 'false':
False})
df['video'] = df['video'].astype(str).str.lower().map({'true': True, 'false':
False})

# Convert budget and popularity to numeric
df['budget'] = pd.to_numeric(df['budget'], errors='coerce')
df['popularity'] = pd.to_numeric(df['popularity'], errors='coerce')

# Convert release_date to datetime
df['release_date'] = pd.to_datetime(df['release_date'], errors='coerce')

# Optional: Drop rows where conversion failed (NaNs introduced)
# df.dropna(subset=['movie_id', 'budget', 'popularity', 'release_date'],
inplace=True)
```

2.7 Deal with Null values

```
# Fill missing 'title' with a placeholder
df['title'] = df['title'].fillna('[No Title]')

# Fill missing 'original_title' with 'title' if available, else a placeholder
df['original_title'] = df['original_title'].fillna(df['title'])
df['original_title'] = df['original_title'].fillna('[No Original Title]')

df['release_date'] = df['release_date'].fillna(pd.Timestamp('1800-01-01')) #
place holder

df[['vote_count', 'rating', 'popularity', 'budget', 'revenue']] =
df[['vote_count', 'rating', 'popularity', 'budget', 'revenue']].fillna(pd.NA)
```

2.8 MySQL Ingestion Code

```
DROP DATABASE IF EXISTS MovieTV_DB;
CREATE DATABASE MovieTV_DB;
USE MovieTV_DB;


-- Movies Table
CREATE TABLE Movies (
    movie_id INT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    release_date DATE,
    rating FLOAT
);
```

```
-- Collections Table
CREATE TABLE Collections (
    collection_id INT PRIMARY KEY,
    collection_name VARCHAR(255) NOT NULL
);

-- Many-to-Many Relationship: Movies and Collections
CREATE TABLE Movie_Collections (
    movie_id INT,
    collection_id INT,
    PRIMARY KEY (movie_id, collection_id),
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (collection_id) REFERENCES Collections(collection_id)
);

-- Genres Table
CREATE TABLE Genres (
    genre_id INT PRIMARY KEY,
    genre_name VARCHAR(100) NOT NULL
);

-- Many-to-Many Relationship: Movies and Genres
CREATE TABLE Movie_Genres (
    movie_id INT,
    genre_id INT,
    PRIMARY KEY (movie_id, genre_id),
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (genre_id) REFERENCES Genres(genre_id)
);

-- Languages Table
CREATE TABLE Languages (
    language_id VARCHAR(10) PRIMARY KEY,
    language_name VARCHAR(100) NOT NULL
);

-- Many-to-Many Relationship: Movies and Languages
CREATE TABLE Movie_Languages (
    movie_id INT,
    language_id VARCHAR(10),
    PRIMARY KEY (movie_id, language_id),
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (language_id) REFERENCES Languages(language_id)
);

-- Production Companies Table
CREATE TABLE Production_Companies (
    company_id INT PRIMARY KEY,
    company_name VARCHAR(255) NOT NULL,
    origin_country VARCHAR(10)
);

-- Many-to-Many Relationship: Movies and Production Companies
```

```
CREATE TABLE Movie_Companies (
    movie_id INT,
    company_id INT,
    PRIMARY KEY (movie_id, company_id),
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (company_id) REFERENCES Production_Companies(company_id)
);

-- Production Countries Table
CREATE TABLE Production_Countries (
    country_id VARCHAR(10) PRIMARY KEY,
    country_name VARCHAR(100) NOT NULL
);

-- Many-to-Many Relationship: Movies and Production Countries
CREATE TABLE Movie_Countries (
    movie_id INT,
    country_id VARCHAR(10),
    PRIMARY KEY (movie_id, country_id),
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (country_id) REFERENCES Production_Countries(country_id)
);

-- Reviews Table
CREATE TABLE Reviews (
    review_id VARCHAR(50) PRIMARY KEY,
    movie_id INT,
    author VARCHAR(255),
    content TEXT,
    rating FLOAT,
    review_date TIMESTAMP,
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id)
);

-- TV Show Table
CREATE TABLE TV_Show (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    original_name VARCHAR(255) NOT NULL
);

-- TV Show Overview Table
CREATE TABLE TV_Show_Overview (
    id INT PRIMARY KEY,
    synopsis TEXT,
    FOREIGN KEY (id) REFERENCES TV_Show(id)
);

-- TV Show Seasons & Episodes Table
CREATE TABLE TV_Show_Seasons_Episodes (
    id INT PRIMARY KEY,
    seasons INT,
    episodes INT,
```

```
    FOREIGN KEY (id) REFERENCES TV_Show(id)
);

-- TV Show Production Table
CREATE TABLE TV_Show_Production (
    id INT PRIMARY KEY,
    executive_producer VARCHAR(255),
    production_companies VARCHAR(255),
    network VARCHAR(255),
    FOREIGN KEY (id) REFERENCES TV_Show(id)
);

-- TV Show Ratings Table
CREATE TABLE TV_Show_Ratings (
    id INT PRIMARY KEY,
    popularity_score FLOAT,
    genres VARCHAR(255),
    FOREIGN KEY (id) REFERENCES TV_Show(id)
);

-- TV Show Airings Table
CREATE TABLE TV_Show_Airings (
    id INT PRIMARY KEY,
    first_aired_date DATE,
    last_aired_date DATE,
    status VARCHAR(50),
    FOREIGN KEY (id) REFERENCES TV_Show(id)
);

-- TV Show Status Description Table
CREATE TABLE TV_Show_Status_Description (
    status VARCHAR(50) PRIMARY KEY,
    short_description VARCHAR(255),
    long_description TEXT
);

-- TV Show Recommendations Table
CREATE TABLE TV_Recommendations (
    tv_series_id INT,
    title VARCHAR(255),
    overview TEXT,
    first_air_date DATE,
    rating FLOAT,
    PRIMARY KEY (tv_series_id, title),
    FOREIGN KEY (tv_series_id) REFERENCES TV_Show(id)
);

-- People Table (Actors & Directors)
CREATE TABLE People (
      person_id INT PRIMARY KEY,
      name VARCHAR(255) NOT NULL,
      date_of_birth DATE,
      nationality VARCHAR(255),
```

```
      biography TEXT,
      role ENUM('Actor', 'Director') NOT NULL
);

-- Many-to-Many Relationship: Movies & People
CREATE TABLE Movie_People (
      movie_id INT,
      person_id INT,
      PRIMARY KEY (movie_id, person_id),
      FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
      FOREIGN KEY (person_id) REFERENCES People(person_id)
);

-- Many-to-Many Relationship: TV Shows & People
CREATE TABLE TV_Show_People (
      tv_show_id INT,
      person_id INT,
      PRIMARY KEY (tv_show_id, person_id),
      FOREIGN KEY (tv_show_id) REFERENCES TV_Show(id),
      FOREIGN KEY (person_id) REFERENCES People(person_id)
);
```

ER Diagram Full Size: Here
https://www.mermaidchart.com/raw/16a05f04-e995-43a4-8d4d-4e83db7318a1?theme=light&version=v0.1&format=svg