AI Algorithms - Theory and Engineering

Movie Recommendation System

Documentation

- Submitted By -

Mohamed Amine Kina
mkina@uni-bremen.de

Prakunj Pratap Singh
o_3mvji4@uni-bremen.de

Karanjaspreet Singh
karanjas@uni-bremen.de

Taha Tabatabaei
seytab@uni-bremen.de

March 7, 2025

Contents

1 Introduction

This documentation provides a comprehensive overview of the codebase for our Movie Recommendation System.
The report delves into various aspects of the codebase, including the directory organization, microservices architecture, and system evaluation. It aims to offer clear and concise explanations of how different components interact and contribute to the overall functionality of the recommendation system.
Additionally, this documentation serves as a guide for evaluators to understand the technical structure and design decisions behind the system. By detailing the underlying architecture, technologies used, and the flow of data within the system.
This is not the Concise Design Document required as deliverables. For that document please refer to file named *Movie Recommendation System - Design Document*

2 Directory Organization

The directory organization tree for the project is structured as follows:

```
analytics_service/
auth_service/
database_service/
feedback_service/
kafka_service/
model_service/
telemetry_service/
ui_service/
web/
docker-compose.yml
```

This structure reflects a microservices-based architecture, where each major functionality is encapsulated within its respective service directory. Each folder represents a distinct service, such as analytics, authentication, feedback, and user interface, which promotes separation of concerns.

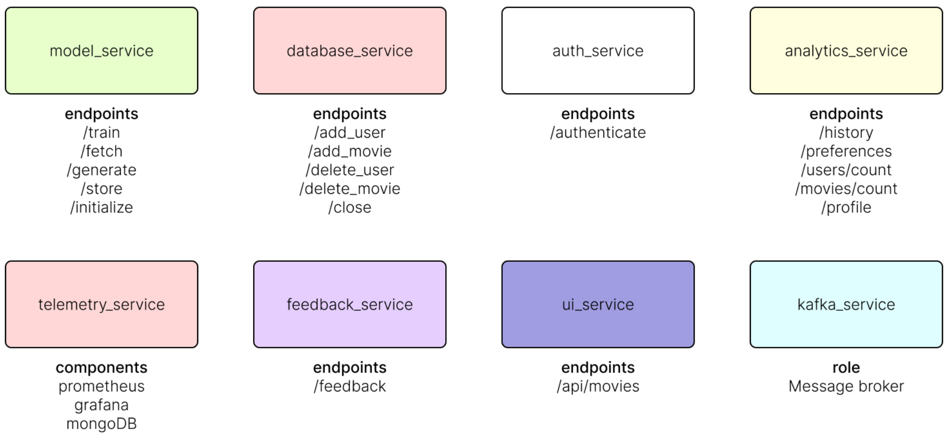


Figure 1: Overview of microservices

3 Microservices Architecture

The Movie Recommendation System is built on a microservices architecture, enabling modular development and deployment.
The system leverages containerization technologies such as Docker, ensuring consistency across different environments. Each microservice communicates through RESTful APIs or messaging queues, facilitating seamless data exchange and service coordination, as illustrated in Figure 2.

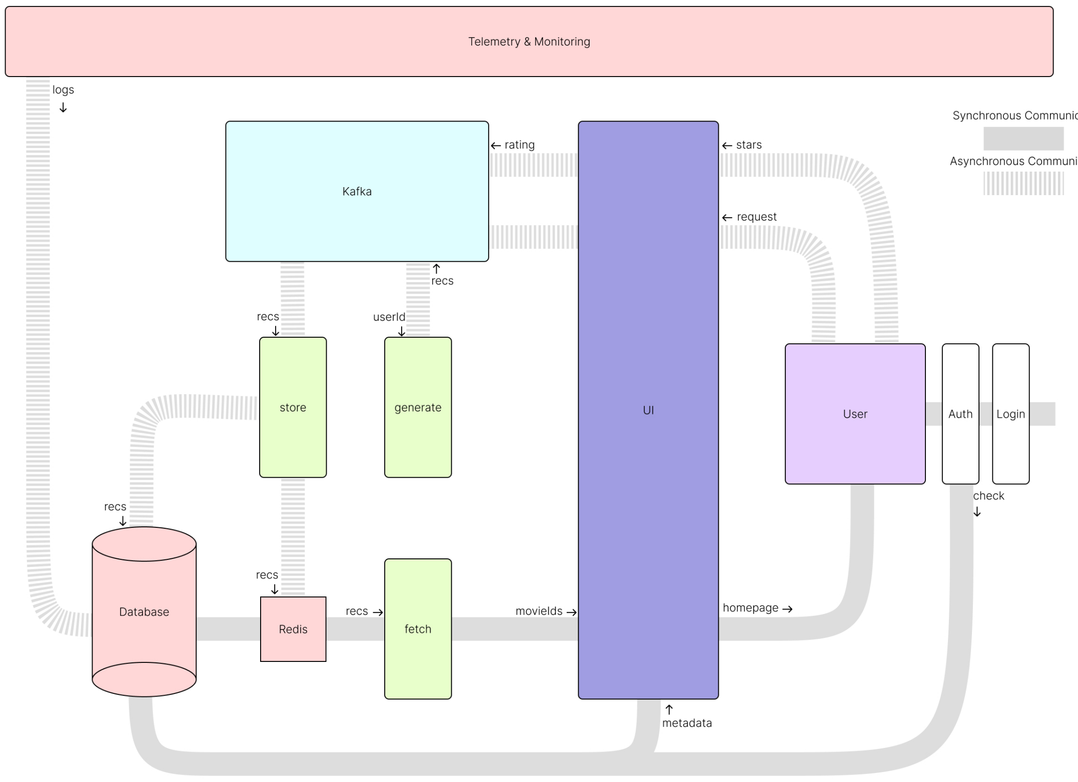


Figure 2: Overview of the communication between services

In the following section, each microservice will be examined in detail, highlighting its specific functionality and role within the overall system.

3.1 Analytics Service

This service interacts with a PostgreSQL database and a Redis cache to provide movie-related analytics and recommendations. Its primary purpose is to analyze user preferences, including movie viewing history, genre preferences, language preferences, and other aggregated statistics about users and movies.

The main functionalities of this service include:

- Retrieving a user's movie-watching history.
- Analyzing a user's genre and language preferences using a pre-trained PCA model.
- Counting the total number of users and movies in the database.

3.2 Authentication Service

This service manages user authentication for the system by relying on PostgreSQL database for user data storage. It provides functionalities for verifying passwords, authenticating users, and managing an admin account. This authentication module is essential for securing user access and maintaining account integrity.

The main functionalities of this service include:

- Verifying user passwords.
- Authenticating users based on email and password.
- Authenticating an admin account.

3.3 Database Service

This service manages database operations, interacting with a PostgreSQL database tables and Redis namsespaces, as illustrated by Figure 3. It handles user and movie data, including user registration, movie entry processing, and vector embedding for recommendation models. The service is crucial for maintaining consistent and reliable data storage while enabling advanced analytics and recommendations.

The main functionalities of this service include:

- Adding and deleting users.
- Adding and deleting movies.
- Generating embeddings and processing movie data for machine learning models.
- Managing database connections and Redis caching.

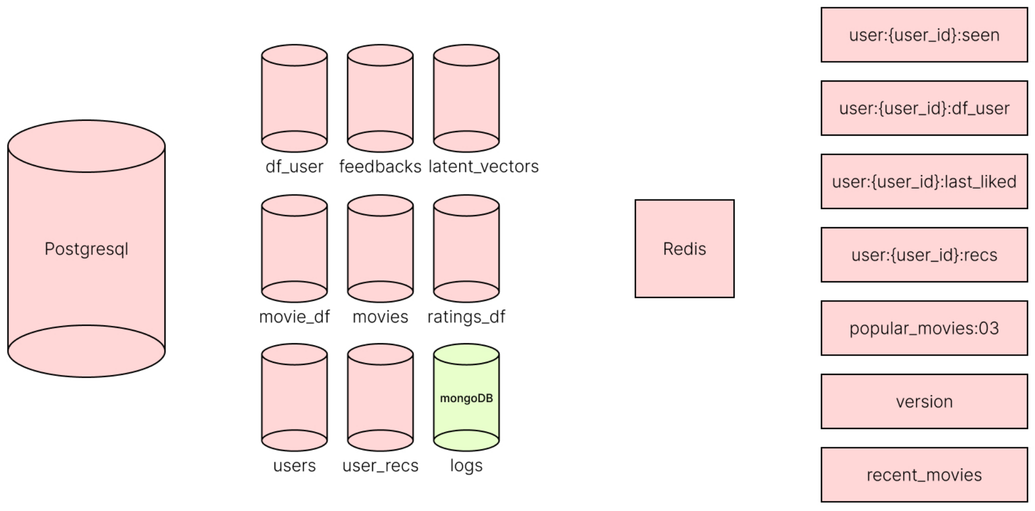


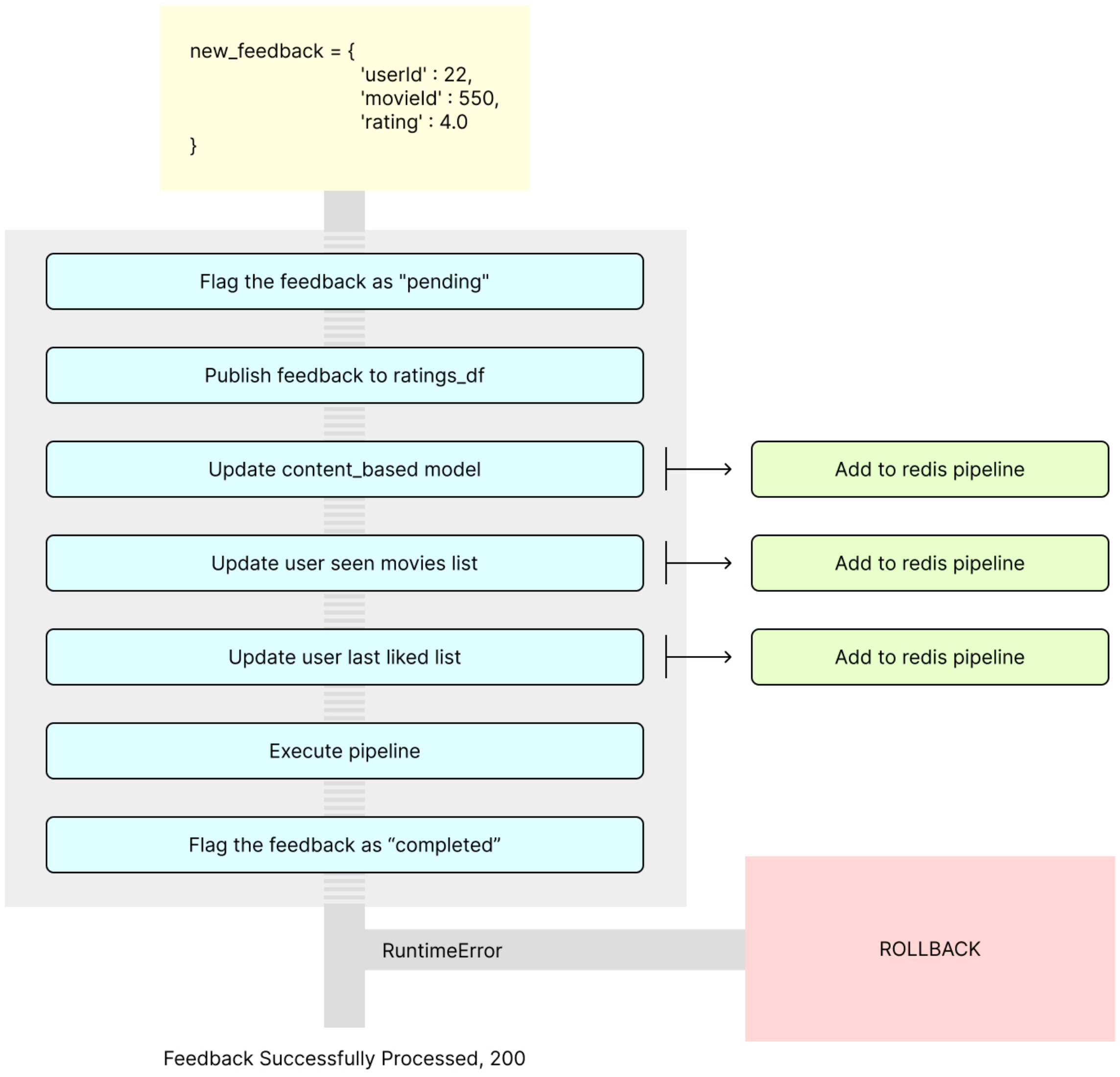
Figure 3: Organization of tables in database and namespaces in Redis

3.4 Feedback Service

This service handles feedback processing for a movie recommendation system, leveraging PostgreSQL for data persistence and Redis for caching. It efficiently manages user feedback on movies, updates user preference vectors, and maintains user history. To ensure consistency amidst frequent write operations, a robust transaction mechanism is implemented, enabling safe rollbacks to the initial state in case of any inconsistencies, as illustrated by Figure 4.

The main functionalities of this service include:

- Processing user feedback and updating corresponding vectors.
- Managing transactional consistency between PostgreSQL and Redis.
- Clearing and updating Redis caches.



```
new_feedback = {
                    'userId' : 22,
                    'movieId' : 550,
                    'rating' : 4.0
}
```

Flag the feedback as "pending"
Publish feedback to ratings_df
Update content_based model
Update user seen movies list
Update user last liked list
Execute pipeline
Flag the feedback as “completed”
Add to redis pipeline
Add to redis pipeline
Add to redis pipeline
RuntimeError
ROLLBACK
Feedback Successfully Processed, 200

Figure 4: Feedback transactional mechanism

3.5 Kafka Service

This service is implemented using Spring Boot and is designed for efficient messaging and event-driven communication between microservices. It ensures reliable data transmission and real-time data processing, leveraging Apache Kafka's high throughput and low latency features. Figure 5 is an example of how Kafka is used in our architecture.

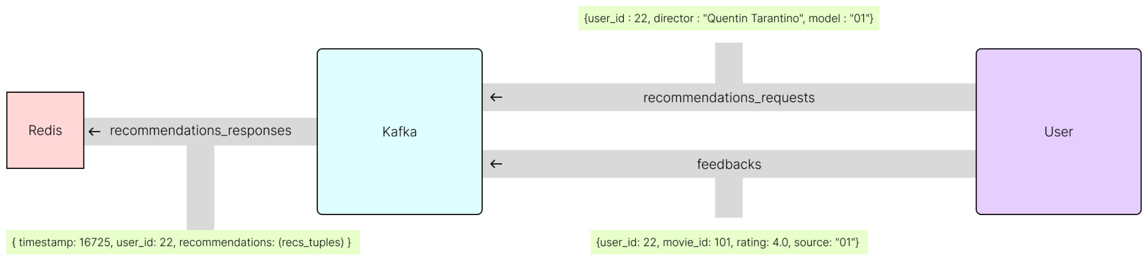

{user_id : 22, director : "Quentin Tarantino", model : "01"}
Redis
← recommendations_responses
Kafka
← recommendations_requests
← feedbacks
User
{ timestamp: 16725, user_id: 22, recommendations: (recs_tuples) }
{user_id: 22, movie_id: 101, rating: 4.0, source: "01"}

Figure 5: Kafka handling of messages

3.5.1 Topics

In the movie recommendation system, three Kafka topics are designed to manage and streamline the flow of data for generating and refining movie recommendations. These topics, *recommendations_ requests*, *recommendations_ responses*, and *feedbacks*, each serve distinct roles, ensuring efficient communication between different components of the system. Here's a detailed breakdown of each topic:

recommendations_requests : This topic handles incoming requests for movie recommendations. It initiates the recommendation computation process and can receive user-specific preferences. The message follows this structure:

~~~
‘‘‘json
{
    'user_id': 22,
    'director': "Quentin Tarantino",
    'model': "01"
}

‘‘‘
~~~

user_id: Identifies the user requesting recommendations. director: (Optional) Specifies a preferred movie director for personalized recommendations. If null, default recommendations are provided. model: (Optional) Indicates the recommendation model to be used for generating recommendations. It is specified by the user. If null, a default hybrid model is used.

recommendations_responses : This topic serves as a buffer for the computed movie recommendations before they are stored in the database systems, namely PostgreSQL and Redis. The message follows this structure:

```
‘‘‘json
{
    'timestamp': 16725,
    'user_id': 22,
    'recommendations': [
        [6812, "01"],
        ...,
        [32, "03"]
    ]
}

‘‘‘
```

timestamp: Captures the time when the recommendations were generated, ensuring accurate tracking and ordering. user_id: Links the recommendations to the specific user. recommendations: An array of recommended movie tuples, each containing: movie_id: Unique identifier of the recommended movie. source: Model's code responsible for generating that recommendation.

feedbacks : This topic collects user feedback on the recommended movies, allowing the system to learn and adapt its recommendation algorithms over time. The message follows this structure:

```
‘‘‘json
{
    'user_id': 22,
    'movie_id': 101,
    'rating': 4.0,
    'source': "01"
}
‘‘‘
```

user_id: Identifies the user providing feedback. movie_id: Specifies the movie on which feedback is given. rating: User's rating for the movie on a scale (1 to 5). source: Model's code responsible for generating that recommendation.

3.5.2 Controller

The Kafka controller within the movie recommendation system, exposes RESTful endpoints to facilitate communication between system services and Kafka topics. This setup ensures a decoupled and scalable communication pipeline for managing recommendation requests, responses, and user feedback. Below is the explanation of each enpoint:

- **POST** /recommend
  This endpoint receives a recommendation request from the client (UI), encapsulated in a MovieRequestModel object. The request may include user-specific preferences, such as user ID, director, and model code.

- **POST** /recommend/response
  This endpoint handles responses from the recommendation engine (gen_service). It receives computed movie recommendations as a raw JSON string.

- **POST** /recommend/feedback
  This endpoint collects user feedback from the client (UI) and encapsulates it in a FeedbackModel object containing user ID, movie ID, rating, and model code.

3.5.3 Producers

We use three Kafka producers *MovieRequestProducer*, *MovieResponseProducer*, and *FeedbackProducer* that are responsible for sending messages to their corresponding Kafka topics, ensuring seamless communication between system components.
Together, these producers decouple the system components, enhancing scalability and maintainability by efficiently processing and storing requests, recommendations, and feedback.

3.5.4 Consumers

We use three Kafka consumers *MovieRequestConsumer*, *MovieResponseConsumer*, and *FeedbackConsumer* are responsible for processing messages from their respective Kafka topics, ensuring efficient data flow and synchronization between system components.
These consumers enhance system reliability and scalability by asynchronously processing incoming data, allowing independent scaling of each component. They maintain data consistency and optimize performance by efficiently managing requests, recommendations, and feedback flows.

3.6 Model Service

This service is a core component of the movie recommendation system, responsible for generating and serving personalized movie recommendations to users. It leverages a hybrid recommendation approach, combining Content-Based and Collaborative Filtering models to enhance recommendation accuracy and relevance.

For a detailed discussion of the underlying hybrid model refer to section 4.

The *model_service* is designed as a modular microservice, ensuring scalability, maintainability, and ease of integration with other system components. It is composed of four main subservices: *train_service*, *gen_service*, *store_service*, and *fetch_service*. Each subservice plays a crucial role in the lifecycle of serving recommendations.

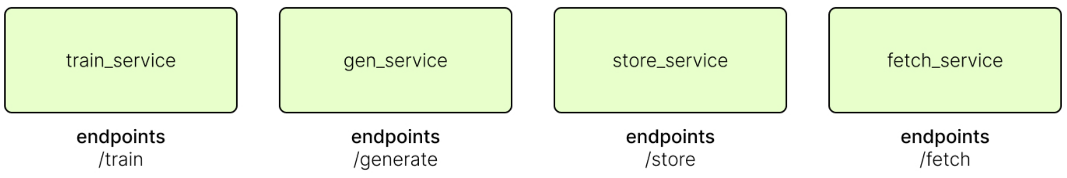


Figure 6: Model subservices

3.6.1 Train Service

This subservice is responsible for training the hybrid recommendation models. It ingests user interaction data and movie metadata to build both content-based and collaborative filtering models. It supports continuous learning by retraining models periodically with fresh data, ensuring that the recommendations remain relevant over time.

Model updates can be performed and deployed to *gen_service* nodes, responsible for computing recommendations, with minimal downtime and consistent user experience. This is done by updtaing model version key on redis. The *gen_service* routinely check for updates. During an update, nodes safely switch to serving default recommendations (popular movies) to maintain system availability. Once the update is complete, *gen_service* resumes serving personalized recommendations.

3.6.2 Gen Service

This subservice is responsible for generating personalized movie recommendations. It is designed to work efficiently with the hybrid recommendation models built by the *train_service*, leveraging both content-based and collaborative filtering approaches, as well as popular model. This service ensures that users receive relevant and engaging movie suggestions based on their interactions and preferences.

The *gen_service* receives recommendation requests through Kafka, enabling asynchronous and scalable processing of user queries. It supports multiple recommendation modes, ensuring a flexible and tailored user experience:

By default, the system generates 20 hybrid recommendations, consisting of:

- 10 content-based recommendations, denoted as CB, that utilize movie metadata such as genres, actors, and plot descriptions.
- 10 collaborative filtering recommendations, denoted as CBF, that leverage user interaction data (i.e., ratings).

$$\text{Default Recommendations} = CB + CBF = 10 + 10 = 20$$

If the request specifies a director, the system adjusts its approach as follows:

- 5 content-based recommendations are specifically from the specified director, denoted as $CB_{director}$.
- 5 general content-based recommendations using standard metadata filtering, denoted as CB.
- 10 collaborative filtering recommendations as usual.

$$\text{Director-Specific Recommendations} = CB_{director} + CB + CBF = 5 + 5 + 10 = 20$$

When a request specifies a particular model code (e.g., for A/B testing or personalized experimentation), only the corresponding model is used to generate 20 recommendations. Let M_{code} denote the recommendations from the specified model:

$$\text{Model-Specific Recommendations} = M_{code} = CB : 01, CBF : 02, PB : 03$$

The recommendation modes can be summarized as:

$$\begin{array}{ll} \text{Default Mode:} & CB + CBF = 20 \\ \text{Director-Specific Mode:} & CB_{director} + CB + CBF = 20 \\ \text{Model-Specific Mode:} & M_{code} = CB : 01, CBF : 02, PB : 03 \end{array}$$

This flexible structure ensures a tailored user experience by dynamically adjusting recommendations based on request parameters. Additionally, to maintain a consistent 20-movie recommendation list after removing duplicate movies generated by content-based and collaborative models, we resort to padding the list with popular movie recommendations by the Popular Model, ensuring a complete set of 20 unique recommendations.

After generating the recommendations, the *gen_service* sends them to Kafka through its controller. These recommendations are then consumed and stored in Redis and PostgreSQL by *store_service*. This architectural choice decouples the recommendation generation process from the storage operations, minimizing potential issues that might arise when writing to Redis and the database directly from the *gen_service*. By offloading the storage responsibility, the *gen_service* remains focused on continuously computing recommendations as requests are received, ensuring high availability and performance.

3.6.3 Store Service

The *store_service* is responsible for efficiently storing the recommendations that are produced by the *gen_service* and sent through Kafka. The *store_service* consumes the recommendation messages from Kafka and handles the persistence of these recommendations in two storage systems: Redis and PostgreSQL. This dual-storage strategy optimizes both performance and durability, catering to different access patterns and system requirements.

In Redis, recommendations are cached with a Time-To-Live (TTL) of 6 hours. This short-term caching mechanism ensures rapid access to recently generated recommendations, minimizing latency for end-users. In contrast, PostgreSQL stores the recommendations for a longer duration of 24 hours with a TTL mechanism that automatically expires outdated data. By leveraging Redis for fast, temporary access and PostgreSQL for more reliable, longer-term storage, the *store_service* achieves a balance between speed and persistence, enhancing the overall scalability and reliability of the recommendation system.

3.6.4 Fetch Service

The *fetch_service* is responsible for delivering movie recommendations to the client (UI) by retrieving them from Redis or PostgreSQL. This service acts as the intermediary between the storage layer and the user interface, ensuring that users receive their personalized recommendations efficiently and seamlessly. The *fetch_service* is designed to minimize latency and optimize the user experience by leveraging a hierarchical caching strategy.

When a user reloads the home page, the *fetch_service* first checks if the recommendations are available in Redis. If found, they are served instantly, ensuring low-latency access due to Redis's high-speed caching capabilities. If the recommendations are not present in Redis, the service then queries PostgreSQL for the data. This two-tiered approach balances speed and durability, retrieving cached recommendations quickly from Redis and resorting to the more persistent PostgreSQL storage when necessary.

In cases where recommendations are not available in both Redis and PostgreSQL, the *fetch_service* serves a default set of recommendations generated by the Popular Model. This ensures that the user experience remains uninterrupted and relevant even when personalized recommendations are not yet available. Additionally, every time the user reloads the home page, it triggers the generation of new recommendations by the *gen_service*, which are then stored in Redis and PostgreSQL by the *store_service*. This ensures that fresh, personalized recommendations are continuously generated and made available for subsequent requests, maintaining a dynamic and engaging user experience.

3.7 Telemetry Service

The telemetry service is implemented using Prometheus and Grafana for monitoring and observability of the system. It tracks key performance and database metrics to ensure system reliability, optimal performance, and real-time alerting. The integration allows for detailed insights into the behavior of microservices and database operations, helping to maintain high availability and user satisfaction.

The main metrics tracked include:

- Uptime Percentage: Monitors the availability and reliability of the microservices.
- Request Duration (P50, P90, P99): Measures the latency distribution, helping to identify performance bottlenecks.
- Request Rate (RPS): Monitors the rate of incoming requests per second.
- Connection Metrics: Tracks the number of active and idle database connections to ensure efficient resource usage.
- Slow Queries: Identifies queries that take longer than expected, aiding in performance optimization.
- Index Usage: Monitors how often database indexes are utilized to optimize query execution.

In addition to tracking system performance, an alert system has been set up to monitor the average rating of each machine learning model. If the rating of the Collaborative Model falls below a defined threshold (e.g., 2), the system automatically triggers the *train_service* to retrain the model. This ensures continuous model performance and adapts to changing data patterns, improving the overall reliability of recommendations.

3.8 User Interface Service

This service manages the user interface interactions for a movie recommendation system. It serves as a bridge between the frontend and backend, retrieving movie data, normalizing ratings, and fetching new releases. It integrates with a Redis cache for performance optimization and uses a movie model for database queries.

The main functionalities of this service include:

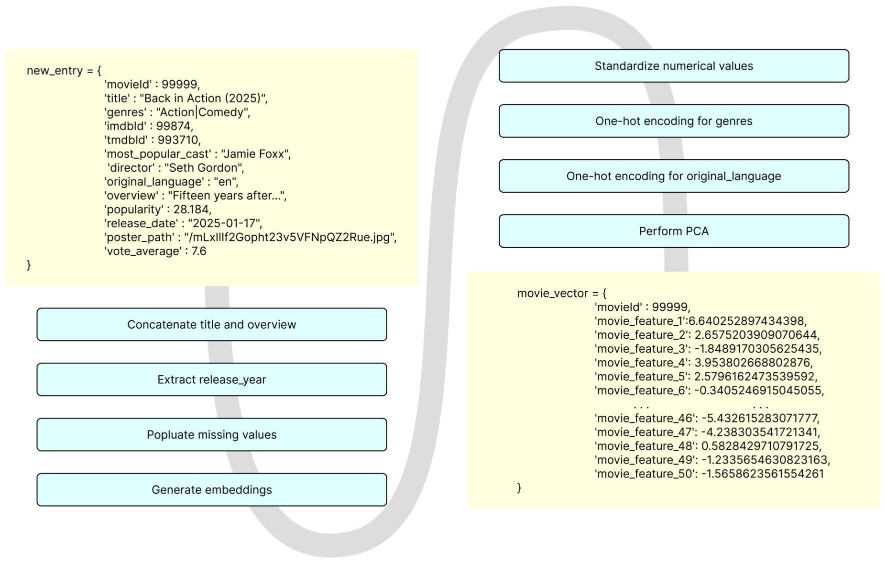
- Retrieving movie posters and additional details.
- Fetching and displaying new movie releases.

4 Hybrid Model

In designing a robust and personalized movie recommendation system, leveraging the strengths of multiple models proves to be highly effective. This hybrid approach integrates three distinct methodologies—Content-Based Filtering, Collaborative Filtering, and Popularity-Based Recommendations—to deliver tailored movie suggestions that cater to both individual preferences and broader audience trends.

4.1 Content-Based Model

The Content-Based model focuses on recommending movies that are similar to those the user has liked or interacted with in the past. It does this by analyzing movie attributes such as genres, directors, and actors.



```
new_entry = {
                'movieId' : 99999,
                'title' : "Back in Action (2025)",
                'genres' : "Action|Comedy",
                'imdbId' : 99874,
                'tmdbId' : 993710,
                'most_popular_cast' : "Jamie Foxx",
                 'director' : "Seth Gordon",
                'original_language' : "en",
                'overview' : "Fifteen years after...",
                'popularity' : 28.184,
                'release_date' : "2025-01-17",
                'poster_path' : "/mLxIIIf2Gopht23v5VFNpQZ2Rue.jpg",
                'vote_average' : 7.6
}
```

Concatenate title and overview
Extract release_year
Popluate missing values
Generate embeddings
Standardize numerical values
One-hot encoding for genres
One-hot encoding for original_language
Perform PCA

```
movie_vector = {
                'movieId' : 99999,
                'movie_feature_1':6.640252897434398,
                'movie_feature_2': 2.6575203909070644,
                'movie_feature_3': -1.8489170305625435,
                'movie_feature_4': 3.953802668802876,
                'movie_feature_5': 2.5796162473539592,
                'movie_feature_6': -0.3405246915045055,
                       . . .                     . . .
                'movie_feature_46': -5.432615283071777,
                'movie_feature_47': -4.238303541721341,
                'movie_feature_48': 0.5828429710791725,
                'movie_feature_49': -1.2335654630823163,
                'movie_feature_50': -1.5658623561554261
}
```

Figure 7: Indexing a new movie

User profiles are constructed by aggregating the profiles of previously rated movies, with each movie's profile scaled by the user's rating to capture the intensity of individual preferences. This method ensures that the recommendations are aligned with the user's historical viewing habits and tastes.

$$\text{User Vector} = \text{Movie Vector} \times \log(\text{Rating} + 1)$$

Humans perceive rating scales non-linearly, as described by the Weber–Fechner law [1], which suggests that perceived differences depend on the relative rather than absolute changes. Applying a logarithmic transformation to ratings aligns the model with this perceptual pattern, ensuring that variations in ratings better reflect users' subjective experiences.

4.2 Collaborative Filtering Model

The Collaborative Filtering model relies on the collective preferences of users to generate recommendations. It utilizes the GLocal-k [2] architecture, which focuses solely on item vectors and leverages the combined strengths of local and global kernels. It effectively encodes user tastes without requiring any side information, achieving high accuracy in low-resource settings.

$$\text{User Vector} = \sum_{i=1}^{N} \left(\text{Movie Vector}_i \times \frac{\exp(-i)}{\sum_{j=1}^{N} \exp(-j)} \right), \quad \text{where } N = \text{len(last\_liked)} = 5$$

User Vectors are dynamically constructed by exponentially averaging the latent features of the *last_ liked* movies they have positively interacted with (i.e., rating ≥ 4), effectively capturing evolving user preferences.

4.3 Popular Model

The Popular Model recommends movies that are widely popular and well-liked, independent of individual user preferences. It calculates a combined score for each movie based on both popularity (number of ratings) and likeability (average rating), using the formula:

$$\text{Popularity-Likeability Score} = 0.5 \times \text{Popularity} + 0.5 \times \text{Likeability}$$

The top 50 movies with the highest scores are selected and stored in Redis for quick retrieval. This model ensures that trending and highly rated movies are always accessible, maintaining recommendation consistency when needed.

4.4 Recommendations

Recommendations are computed using FAISS [3], an efficient similarity search library. For the Content-Based model, movie vectors are generated from the movie attributes and indexed using FAISS on the movie_df, enabling fast retrieval of similar movies based on user profiles. For the Collaborative Filtering model, latent vectors representing user-movie interactions are indexed in FAISS, facilitating rapid nearest-neighbor searches to identify movies favored by users with similar tastes. This dual indexing approach ensures efficient and accurate recommendation generation by leveraging vector similarities for both models.

For the model evaluation please refer to section 5.1.

5 Evaluation Overview

This section covers the evaluation strategies used for both the model and the system, ensuring high performance, accuracy, and reliability. It also provides a justification for the architectural choices made in the system design.

5.1 Model Evaluation

This section draws heavily on the methodology presented in *"Recommender System Performance Evaluation and Prediction: An Information Retrieval Perspective"* by Alejandro Bellogín [4]. Their approach to evaluating model performance provided valuable insights that informed the design and implementation of our evaluation framework.

5.1.1 Metrics

To evaluate out models we have used two Precision-based metrics, namely Precision@k and Recall@k. More specifically, precision accounts for the fraction of recommended items (k) that are relevant, whereas recall is the fraction of the relevant items that has been recommended in k.

$$P@k = \frac{1}{|U|}\sum_{u\in U}\frac{|\text{Relevant@}k|}{k}$$

$$R@k = \frac{1}{|U|}\sum_{u\in U}\frac{|\text{Relevant@}k|}{|\text{Relevant}|}$$

5.1.2 Experimental Setup

It is crucial to clearly outline the experimental setup decisions before presenting the evaluation results, as variations in experimental configurations can significantly influence the interpretation of the results. Without a consistent and transparent setup, the findings may be easily misinterpreted or lead to misleading conclusions.

The train-test split is performed by first grouping ratings by user and then filtering them based on a relevance threshold (i.e., rating ≥ 4), ensuring that only ratings above this threshold are considered relevant. For each user, relevant ratings are sorted chronologically to preserve the temporal sequence of interactions. If a user has a sufficient number of relevant ratings (i.e., relevant ratings ≥ 5), the last 20% of these ratings are selected for the test set, maintaining at least one rating whenever possible. The remaining ratings are assigned to the training set. Conversely, if a user does not meet the minimum threshold for relevant ratings, all ratings are placed in the training set, leaving the test set empty for that user.

During evaluation, recommendations are generated based on the union of movies present in the test dataset, ensuring that only potential items are considered. This constraint helps mitigate potential biases in the evaluation of recommender systems by preventing the inclusion of items that have no chance of being relevant to the users.

5.1.3 Results

We compared the performance of the three recommendation models — Content-Based, Collaborative, Popular, as well as a Random model that serves as a baseline, as illustrated by Figure 8.

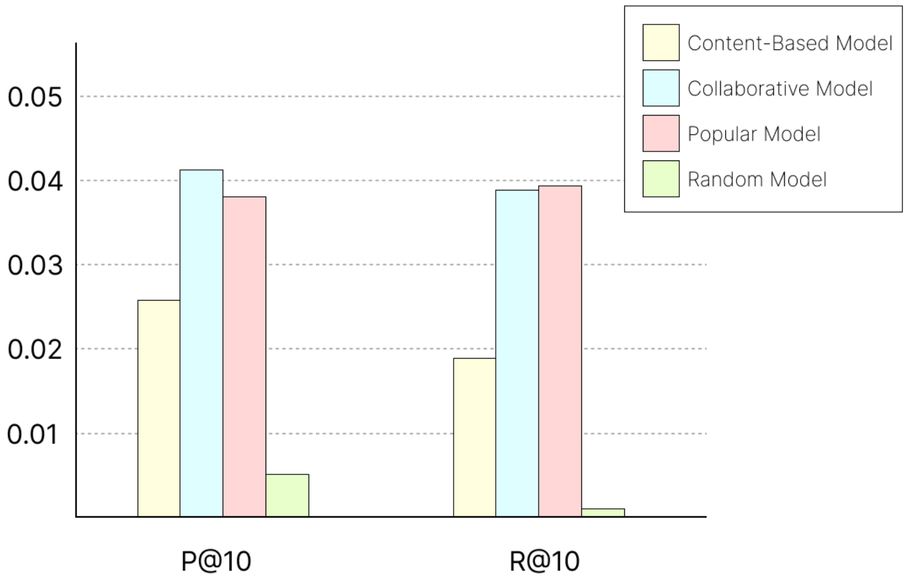


Figure 8: Precision and Recall @ 10 comparison

The Collaborative and Popular models outperform the others in both metrics, showing the highest values for both P@10 and R@10. The Content-Based model performs moderately well, trailing behind Collaborative and Popular models but significantly outperforming the Random model. The Random model shows the lowest performance across both metrics, highlighting its inefficiency in producing relevant recommendations.

5.1.4 Popularity Bias

Although the Popular Model shows high precision and recall (comparable to the Collaborative Model), it is not the best choice when it comes to diversifying recommendations or catering to individual user preferences.

Popularity-driven models maximize precision by catering to the tastes of the majority. This means they perform well on metrics like P@10 and R@10 because they recommend items that are already widely liked. However, they overlook the preferences of users with unique or niche tastes, leading to a lack of personalization.

Moreover, Popular models tend to repeatedly recommend the same set of popular items, reducing the variety in recommendations. This results in a homogenized experience for users, where they see similar or identical recommendations, regardless of their individual preferences. Consequently, users who might be interested in less mainstream content are underserved.

5.2 System Evaluation

Evaluating the performance of such a system requires comprehensive monitoring and analysis. In this section, we focus on the evaluation of the *fetch_service*, the primary microservice responsible for delivering movie recommendations to users. This evaluation was conducted using Prometheus and Grafana to collect and visualize system metrics.

5.2.1 Performance Evaluation of Fetch Service

The *fetch_service* was chosen as the representative microservice for evaluating system performance. It handles the main request load, processing movie recommendations for users. The service was tested under simulated load using Locust, a tool for load testing and performance measurement.

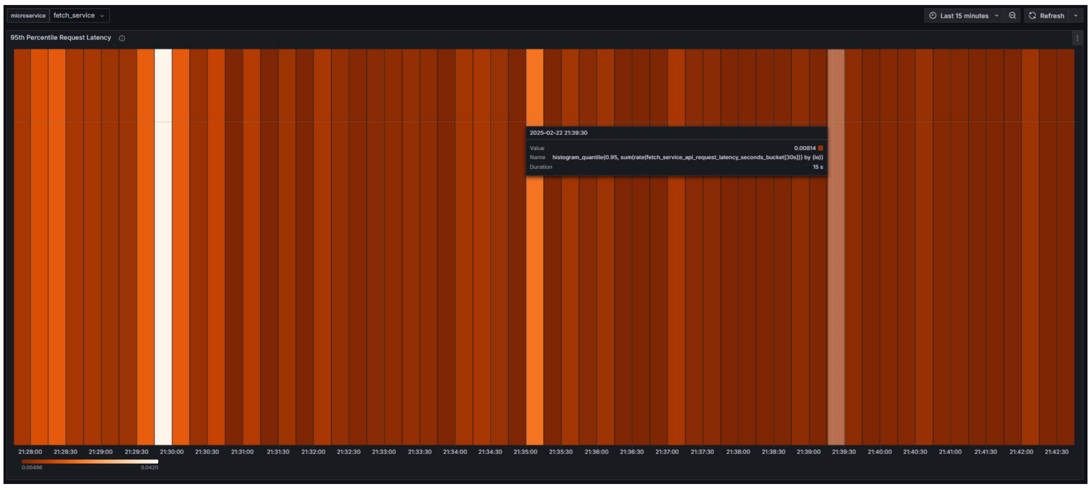
Testing Setup

- Test Environment: The *fetch_service* was deployed on a single node for performance testing. It is important to note that evaluating the entire system's performance would require a distributed testing environment. Running all services on a single machine could lead to resource contention and may not accurately represent a production environment.

- Test Configuration: The system was tested with 100 concurrent users to simulate high traffic scenarios. Locust was used to generate the load and measure the response times and request rates.

5.2.2 Observations from Grafana Dashboard

From the Grafana dashboard:



2025-02-22 21:39:30

Value 0.00814

Name histogram_quantile(0.95, sum(rate(fetch_service_api_request_latency_seconds_bucket[30s])) by (le))

Duration 15 s

Figure 9: 95th Percentile request latency of Fetch Service

- Uptime Percentage: The service maintained an uptime of 100%, indicating high availability during the test period of 15 mins.

- Total Requests: The *fetch_service* processed 25,419 requests during the test, demonstrating its capacity to handle significant traffic volumes.

- Request Duration and Latency: The P50, P90, and P99 percentiles for request duration showed consistent performance, with occasional spikes in the P99 latency. Average Request Latency was generally low, with the median latency being 5ms.
- Request Rate (RPS): The request rate fluctuated between 23 and 27 requests per second, indicating steady traffic handling. This translates to an approximate throughput of 1,380 to 1,620 requests per minute, showing that the service can efficiently handle a substantial volume of requests.
- 95th Percentile Request Latency: The heatmap showed consistent latency.

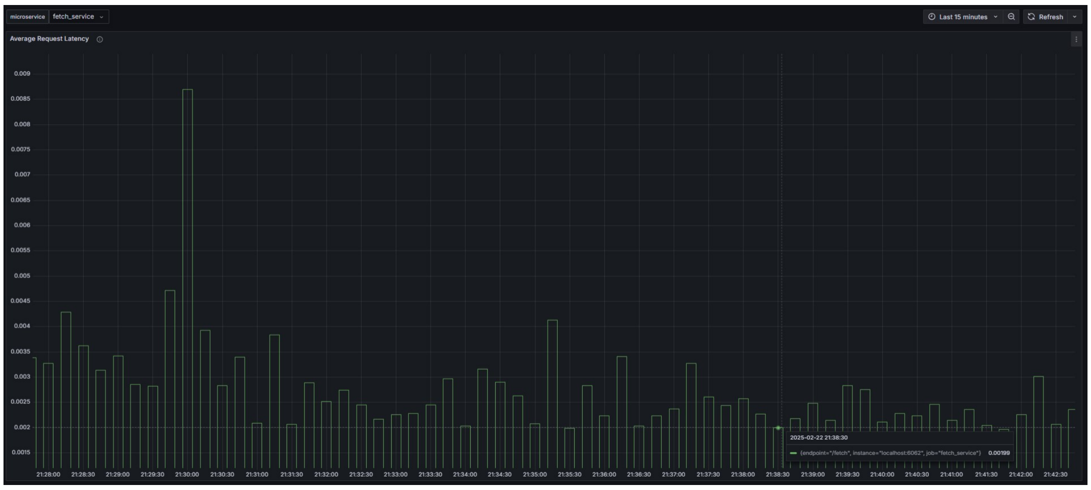


Figure 10: Average request latency of Fetch Service

5.2.3 Considerations for Scaling

The *fetch_ service* demonstrated reliable performance on a single node, but a distributed environment is necessary to fully understand the system's scalability, resilience, and performance under real-world conditions.

6 User Experience

This section explores the user experience of UBFlix, focusing on how users interact with the platform and the features that enhance their journey. It provides an in-depth look at the system's usability, including the ease of navigation, the flow of discovering and selecting movies, and the efficiency of personalized recommendations. By examining how users engage with the website, this section highlights the intuitive design choices that facilitate a seamless and satisfying experience for users.

6.1 User Page

Upon logging in, users are directed to the homepage, which serves as the central hub for movie recommendations. The layout prominently displays a curated selection of movies tailored to user preferences, enhancing the discovery experience. Each movie is presented with visually engaging cover art, title, release year, rating, and director's name, providing

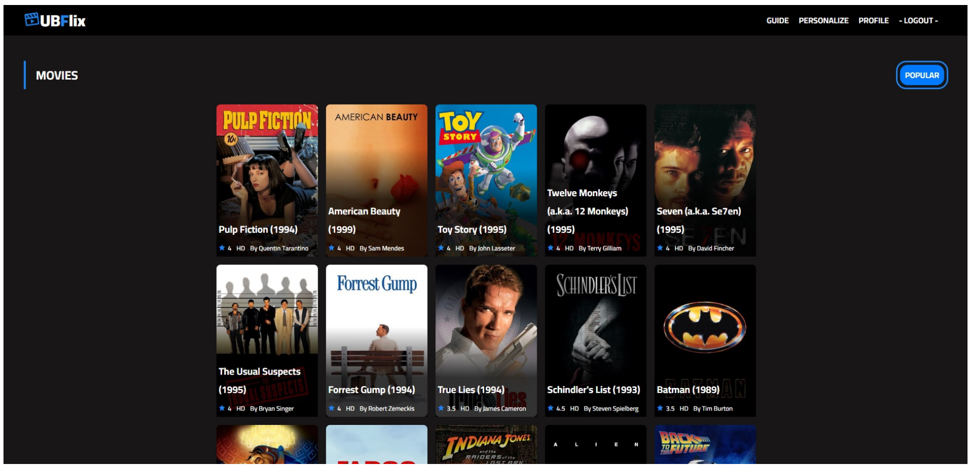


Figure 11: User Interface of our prototype UBFlix

essential information at a glance, as illustrated by Figure 11.

Additionally, users can easily access other key sections of the platform, including:

- Guide: A comprehensive overview of the platform's features.
- Personalize: A section where users can customize their movie recommendations based on their favorite directors or favorite model.
- Profile: Where users can explore insights on their genres preferences and viewing history.

This design ensures that users can seamlessly navigate between different sections, promoting an engaging and personalized movie-watching experience.

6.2 Admin Page

The Admin Page is accessible through a dedicated link located at the footer of the homepage. For demonstration purposes, we keep simple and default credentials as follows:

- **Username**: admin
- **Password**: admin

Once logged in, administrators are directed to the Control Panel, which provides comprehensive management functionalities, including:

- Dashboard Overview:

  Displays key statistics such as the total number of users and the total movie count. Offers a quick summary of platform activity, enabling administrators to monitor user engagement and movie library growth at a glance.

- User Management:

  Allows administrators to search for users using their first name.

- Movie Management:

  Provides options to search for existing movies or add new ones to the database.

The Admin Page is designed with a minimalist and intuitive layout, prioritizing functionality and ease of use. This approach allows administrators to efficiently manage platform content and user activities while maintaining a seamless workflow.

6.3 User Story

The user story that we will tackle states:

```
As a user , I want my feedback to be stored locally when the service
is unavailable or when I am offline , and for it to be synchronized
and accounted for once the connection is restored .
```

To ensure a seamless user experience even during connectivity issues, the feedback system has been designed to handle offline scenarios gracefully. When users provide feedback on a movie (i.e. ratings), the following workflow is executed:

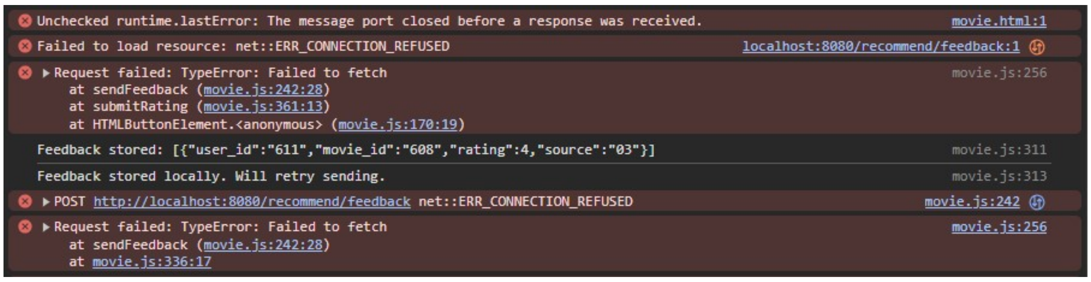
- Local Storage Mechanism:

  If the communication with Kafka fails, the feedback data is saved locally using the browser's localStorage. This ensures that user inputs are not lost and can be synchronized once the connection is restored.

- Periodic Retry and Synchronization:

  The UI is designed to periodically retry communication with Kafka. Upon re-establishing the connection, all locally stored feedback is automatically synchronized with the server. This process is seamless and requires no additional user intervention, ensuring that feedback is accurately accounted for even after a network disruption.

This can be view in Figure 12.



```
Unchecked runtime.lastError: The message port closed before a response was received.              movie.html:1
Failed to load resource: net::ERR_CONNECTION_REFUSED                     localhost:8080/recommend/feedback:1
▶ Request failed: TypeError: Failed to fetch                                                       movie.js:256
    at sendFeedback (movie.js:242:28)
    at submitRating (movie.js:361:13)
    at HTMLButtonElement.<anonymous> (movie.js:170:19)
Feedback stored: [{"user_id":"611","movie_id":"608","rating":4,"source":"03"}]                     movie.js:311
Feedback stored locally. Will retry sending.                                                       movie.js:313
▶ POST http://localhost:8080/recommend/feedback net::ERR_CONNECTION_REFUSED                        movie.js:242
▶ Request failed: TypeError: Failed to fetch                                                       movie.js:256
    at sendFeedback (movie.js:242:28)
    at movie.js:336:17
```

Figure 12: Handling of feedback error

7 Conclusion

The Movie Recommendation System presented in this documentation uses thoughtfully designed microservices architecture. Rather than relying on monolithic design, we've separated concerns into specialized services—from authentication to PostgreSQL data management— enabling independent scaling and targeted optimization where needed most.

Our recommendation algorithm distinguishes itself through a three-pronged approach: content-based filtering identifies thematic patterns in viewing history, collaborative filtering surfaces unexpected gems from similar users, and popularity metrics ensure users stay connected with cultural touchpoints. The system's Kafka backbone enables asynchronous and scalable communication across services, ensuring efficient data processing. The *fetch_service* plays a critical role in delivering recommendations with low latency, utilizing a caching mechanism that optimizes performance. Through Prometheus and Grafana, the system provides real-time monitoring and alerting, ensuring reliability and proactive maintenance.

In summary, this project successfully delivers a high-quality recommendation system that balances performance, scalability, and user engagement. With continuous improvements and optimizations, it has the potential to provide an even more personalized and efficient recommendation experience.

References

[1] Wikipedia contributors, "Weber–Fechner law – Wikipedia, The Free Encyclopedia," 2025, [Online; accessed 24-February-2025]. [Online]. Available: https://en.wikipedia.org/wiki/Weber%E2%80%93Fechner_law

[2] S. C. Han, T. Lim, S. Long, B. Burgstaller, and J. Poon, "Glocal-k: Global and local kernels for recommender systems," *CoRR*, vol. abs/2108.12184, 2021. [Online]. Available: https://arxiv.org/abs/2108.12184

[3] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," *arXiv preprint*, vol. abs/2401.08281, 2024. [Online]. Available: https://arxiv.org/abs/2401.08281

[4] A. Bellogín, "Recommender system performance evaluation and prediction: An information retrieval perspective," 2012, [Online; accessed 24-February-2025]. [Online]. Available: https://abellogin.github.io/thesis/thesis-bellogin.pdf