COMP/SOFT4902
Graduation Design Project

FEYZİYE SCHOOLS FOUNDATION
IŞIK UNIVERSITY

Project Title
AI Based Tv Show/Movie Recommendation System

TÜLİN EYLÜL ERDOĞAN 21COMP1010
IHSAN EREN ERBEN 22SOFT1055

REQUIREMENTS ANALYSIS DOCUMENT

1. Introduction

This document presents the completed requirements analysis for the **AI-Based Movie and TV Recommendation Web Platform (CineMatch)**. The initial version of this document described a system that was still in the planning and design phase, with several assumptions about how recommendations would be produced .

The current version reflects the completed and running implementation. All descriptions have been updated to match:

- The actual architecture (React frontend, Spring Boot backend, PostgreSQL database, Gemini + TMDB integrations).
- The real user workflows (favorites-based recommendations, persistent recommendation history, JWT-secured API).
- The final feature set is visible in the live system.

The platform is a full-stack web app that enables users to:

- Register and log in securely.
- Browse movies and TV shows..
- Build and manage a personal favorites list.
- Trigger **AI-based recommendations** using the Gemini API.
- Review and revisit **Old Recommendations** that have been stored in the database permanently.

When a user clicks the **"Get Recommendations"** button on the Recommendations page, the backend sends the titles of their favorite movies/series to the **Gemini API**. Gemini returns a list of similar titles per favorite. These titles are then matched against real content via **The Movie Database (TMDB)**. Successfully matched items are stored as full movie/series records in the PostgreSQL database and linked to an "old recommendation session" entity.

A differentiating feature of this system is persistence of recommendation sessions: generated recommendation sets are never “lost” after a single page visit. Users can go to the Old Recommendations page at any time and see their previously generated recommendation groups, again fully functional as content cards.

From an educational perspective, the project involves:

- Real-world backend development with **Spring Boot** and **Spring Security**.
- Modern frontend development with **React**, **Axios**, and **React Router**.
- Relational database design on **PostgreSQL**.
- Practical AI integration via **Gemini** and **TMDB**.

Thus, the final product is both a **working web platform** and a **demonstration of AI integration** in a modern web application.

1.1. Purpose of the System

The main focus of the system is to provide users with personalized and relevant movie and TV shows recommendations based on their previous preferences. Instead of depending on large-scale user rating datasets or traditional collaborative filtering, the system takes a **favorites-driven approach**:

- Users indicate what they like by adding movies/series to their **Favorites**.
- For each favorite, the system asks Gemini for 5 similar titles.
- These titles are then validated and enriched through TMDB, ensuring that only real, well-described content is presented.

In addition to this essential objective, the project also has a strong instructional and demonstrational purpose:

- To convert theoretical knowledge of artificial intelligence, software engineering, and web technologies into a real, developed product.
- To demonstrate how external AI services (Gemini) and content APIs (TMDB) can be safely integrated into a production-like Spring Boot backend.
- To demonstrate how persistent, AI-generated data can be modeled and managed in a relational database.

Ultimately, the system aims to improve the user experience of content discovery by offering:

- Recommendations grounded in the user’s actual favorites.
- A sense of continuity through **Old Recommendations** that can be revisited and managed over time.

1.2. Scope of the System

The scope of the system is centered on a web-based recommendation platform with the following capabilities:

- **User registration and login** , secured through JWT-based authentication.
- **Browsing movies and TV series** via dedicated pages (Movies, Series,Favorites).
- **Managing favorites** by adding or removing content directly from content cards and recommendation lists.
- **Getting AI-based recommendations** using Gemini, based on the user's favorites.
- **Viewing and managing Old Recommendations**, where all past recommendation sessions are stored and categorized.
- **Profile management**, where basic user information can be viewed.

Out of scope:

- Mobile-native applications (iOS/Android).
- Social/community features (friends, comments, ratings, reviews) beyond favorites.
- Payment, subscription, or premium plan management.
- Training custom machine learning models; instead, the system integrates **external AI (Gemini)**.
- Watching or streaming movies and TV shows within the platform; the system focuses only on recommendation, not content delivery.

From a project perspective, the main focus is:

- **AI and web integration**: connecting a modern web frontend, secure backend, relational database, and AI services into one coherent platform.
- **Persistence and usability**: ensuring that AI-generated recommendations are stored and reusable, not just transient API responses.

The system does not try to cover every possible feature of a commercial streaming platform; it is intentionally focused on a well-defined but realistic subset that is feasible within a graduation project and highlights the core concept of AI-based personalized recommendations.

1.3. Objectives and Success Criteria of the Project

Main Objective

To design, implement, and deploy a **full-stack AI-based recommendation platform** that:

- Accepts user favorites as input.
- Uses **Gemini** to generate similar titles.
- Uses **TMDB** to verify and enrich those titles.
- Stores and serves **persistent recommendation sessions** through a modern web interface.

Success Criteria

The project is considered successful if the following criteria are satisfied:

1. **Favorites-based Recommendation Generation**
    - The system generates recommendations directly from the user's **favorite movies and series**, not from generic popularity lists.
2. **Persistence of Recommendations**
    - Every recommendation session triggered by the user is stored in the database (OldMovieRecommendation, OldSerieRecommendation and their relations).
    - Users can access these sessions later via the **Old Recommendations** page.
3. **Usable and Intuitive Web Interface**
    - The React-based frontend offers a clear navigation structure: Home, Movies, Series, Favorites, Recommendations, Old Recommendations, Profile, Login, Register.
    - Content is displayed using card-based layouts and modals for details.
    - Users can efficiently add or remove favorites from multiple entry points, such as listing pages and recommendation results.

4. **Secure and Reliable Backend**

    - Authentication and authorization are implemented via **JWT** on top of **Spring Security**.
    - Only authenticated users can access protected resources such as favorites, recommendations, and old recommendations.
    - Passwords are stored securely using **BCrypt** hashing.

5. **Effective AI Integration**

    - The backend successfully communicates with **Gemini API**, sending favorite titles and receiving valid suggestion lists.
    - For each suggested title, the system queries **TMDB** and stores matching movie or TV series records in the database.
    - Generated recommendations are meaningful, relevant, and presented as complete content cards, including poster images, ratings, overviews, and release dates.

6. **Stable Data Management**

    - The PostgreSQL schema supports all required relations (User, Movie, Serie, Favorite, OldMovieRecommendation, OldSerieRecommendation, etc.).
    - No orphaned or inconsistent data is produced under normal operation.

7. **Academic and Technical Quality**

    - The project is documented clearly, covering requirements, architecture, design decisions, and implementation details.
    - It demonstrates not only programming skills but also system design, database modeling, and AI integration principles at a level appropriate for a graduation project

1.4. Definitions, Acronyms, and Abbreviations

- **Artificial Intelligence (AI):**
  The capability of computer systems to perform tasks that ordinarily require human intelligence, such as reasoning, learning, and pattern recognition.

- **Gemini API:**
  Google's generative artificial intelligence service used in this project to generate lists of similar movies and TV series based on textual title prompts.

- **TMDB (The Movie Database):**
  A widely used external movie and TV database that provides detailed metadata, including title, overview, poster, rating, and release date. In this project, TMDB is used to convert AI-generated text titles into real and structured content records.

- **JWT (JSON Web Token):**
  A compact and URL-safe token format used for authentication and authorization. After a successful login, the backend issues a JWT, which is then included by the frontend in the Authorization header of protected requests.

- **DB (Database):**
  A structured data storage system. In this project, **PostgreSQL** is used as the relational database to store users, movies, TV series, favorites, and recommendation history.

- **UI (User Interface):**
  The visual and interactive part of the application that users see and interact with, implemented in this project with **React**.

- **Recommendation Engine:**
  The overall mechanism that combines Gemini and TMDB to generate personalized recommendations. Gemini produces candidate titles based on user favorites, TMDB validates and enriches these titles with metadata, and the backend stores and returns the resulting items to the user.

- **Favorites:**
  TV shows or movies that a user has specifically marked as they like. The recommendation engine uses favorites as its primary input.

- **Old Recommendations:**
  Previously generated recommendation sessions stored in dedicated database tables, allowing users to revisit earlier sets of suggested movies and TV series.

Classical recommendation techniques such as Collaborative Filtering, Content-Based Filtering, and Embedding-based models are discussed in the literature review. However, in the implemented system, recommendations are generated through an external AI service (Gemini) rather than a locally trained machine learning model.

1.5. Overview

This Requirements Analysis Document is structured into the following main sections:

1. **Introduction** – Provides general background, motivation, purpose, and scope of the system, as well as definitions and acronyms.

2. **Current System** – Describes the initial situation and the absence of an existing solution, and summarizes the now completed implementation.

3. **Proposed System** – Presents the overall system concept, including architectural layers such as frontend, backend, database, and AI services, and explains how the final design meets the project objectives.

4. **System Requirements** – Presents overall system requirements and then a more detailed **Requirements** section with functional expectations.

5. **Use Cases and Scenarios** – Defines primary use cases, especially the **Get Movie Recommendations** use case, and includes textual scenarios covering normal and exceptional flows.

6. **Nonfunctional Requirements** – Specifies quality attributes such as usability, reliability, performance, supportability, implementation choices, interfaces, packaging, legal aspects, and other constraints.

7. **System Models** – Describes the conceptual models that support the analysis, including the use case model, object model, and dynamic and UI models (state machine, sequence diagram, navigation).

8. **Other Analysis Elements** – Includes a risk and alternatives analysis, as well as the project plan broken into work packages.

9. **Glossary and References** – Provides a glossary of key terms and lists the literature and technical documentation used during the project.

2. Current System

At the beginning of the project, there was **no integrated system** that:

- Combined AI-based text generation (Gemini) with TMDB metadata,
- Stored all recommendation sessions persistently,
- Offered a modern React frontend with secure JWT-based access.

Traditional movie/TV recommendation systems are usually part of large streaming services or limited to research prototypes. For this graduation project, we designed and implemented a new, custom web platform from scratch.

The current system, as implemented, consists of the following components :

- A **React-based frontend** providing an SPA-style user experience with pages for:
    - Home
    - Movies
    - Series
    - Favorites
    - Recommendations
    - Old Recommendations
    - Profile
    - Register/Login
- A **Spring Boot backend** exposing RESTful API endpoints, handling:
    - User registration and authentication with BCrypt and JWT .
    - Movie and series listing and retrieval.
    - Favorite management including adding and removing.
    - Recommendation workflows including Gemini API and TMDB API .
- A **PostgreSQL database** used for storing:
    - User accounts.
    - Movie and series records with TMDB-derived metadata.
    - User favorites.
    - Historical recommendation sessions and their Many-to-Many relationships with movies/series.

The platform is currently operational in a development environment successfully and meets the core objectives defined for the graduation project.

3. Proposed System

The system is an **AI-Based Movie and TV Recommendation Web Platform** designed to help users discover content that aligns with their taste by using explicit favorites and AI-generated suggestions.

Core Components

1. **Web Interface (Frontend)**
    - Built with **React** and **React Router**.
    - Uses **Axios** for HTTP communication with the backend.
    - Stores and sends the **JWT** in the `Authorization` header for protected endpoints.
    - Provides pages for:
        - Listing movies and series with card-based UI and modals.
        - Managing favorites directly via the content cards.
        - Triggering and viewing recommendations.
        - Browsing stored **Old Recommendations**.
        - Viewing a **Profile** page with basic user info.
2. **Backend (Spring Boot)**
    - Exposes RESTful endpoints under `/api`.
    - Main controllers:
        - **AuthController** – handles user registration and login; returns JWTs.
        - **FavoriteController** – manages adding/removing favorites for the authenticated user.
        - **MovieController** and **SerieController** – handle listing content, synchronizing data from TMDB, and retrieving old recommendation items.
        - **RecommendationController** – orchestrates the full recommendation flow:
            - Retrieves favorites from the database.
            - Calls Gemini for similar titles.
            - Calls TMDB for metadata.
            - Stores results and returns them to the client.

- Security implemented via **Spring Security** and a custom **JwtFilter** that:
  - Extracts and validates tokens.
  - Loads user identity into the `SecurityContext`.

3. **Database (PostgreSQL)**
   - Core tables:
     - **User**: `id`, `email`, `name`, `password`, `gender`, `birthDate`, `createdAt`, etc.
     - **Movie**: TMDB-based fields such as `title`, `overview`, `posterUrl`, `rating`, `releaseDate`, `category`, etc.
     - **Series**: Similar to Movie but for TV series (`name`, `firstAirDate`, etc.).
     - **Favorite**: Many favorites per user; each favorite links to either a movie or a series.
     - **OldMovieRecommendation** and **OldSerieRecommendation**: store recommendation sessions and maintain **Many-to-Many** relationships to Movie/Series entities.
   - This schema allows the system to:
     - Track which recommendations belong to which user and session.
     - Support future extensions (e.g., filtering old recommendations by date, favorite title, or category).

4. **AI Subsystem (Gemini + TMDB)**
   - **Gemini**:
     - Receives a prompt constructed from the user's favorites (e.g., "User liked X, suggest 5 similar movies…").
     - Returns a list of **candidate titles** as plain text.
   - **TMDB**:
     - The backend calls TMDB's search APIs using titles returned by Gemini.
     - When a match is found, the backend retrieves metadata and saves a Movie/Serie record in the database.
   - This approach replaces local model training with a **service-based AI strategy**, which is more practical for a graduation project and closer to modern production architectures.

A key functional characteristic of the proposed system is that **every recommendation set is persisted** as an “old recommendation session”. This provides continuity, reusability, and a richer user experience.

3.1. Overview

The AI-Based Movie and TV Recommendation Web Platform can be viewed as a **three-layer architecture**:

1. **Frontend (Web Interface)**
    - React SPA with routing and Axios-based API calls.
    - Handles presentation, user interactions, and local state management (favorites view, token storage, pagination, etc.).
2. **Backend (Application Layer)**
    - Spring Boot application implementing:
        - REST controllers.
        - Service layer (business logic).
        - Repository layer (JPA/Hibernate) to talk to PostgreSQL.
    - Integrates with:
        - **Gemini API** for AI suggestions.
        - **TMDB API** for metadata enrichment.
3. **Data Layer (Database + External Content Services)**
    - PostgreSQL database for all local, persistent data.
    - TMDB as an external data source for movie/series metadata.

From an interaction perspective:

- End-users interact with the **React UI**.
- The frontend communicates strictly with the **Spring Boot backend**.
- The backend is the single authority that:
    - Validates JWT,

- Manages user data,
- Calls Gemini and TMDB,
- Writes and reads database records.

System Requirements

At a high level, the system must:

- Allow new users to **register** and existing users to **log in**.
- Present a **browsable catalog** of movies and TV series, based on data stored in the local database (possibly synchronized from TMDB).
- Allow users to **add or remove favorites** directly from listing and recommendation views.
- Let users request **AI-based recommendations** based on their favorites.
- Store each **recommendation session** in the database so that the user can see it later on the **Old Recommendations** page.
- Ensure that only authenticated users can:
  - Access their own favorites.
  - Request personalized recommendations.
  - View old recommendation sessions associated with their account.

From the backend perspective:

4. The system must maintain a secure, reliable connection to the database and handle errors gracefully.
5. External API calls (Gemini and TMDB) must be implemented with proper error handling and timeouts.
6. Recommendation data should not be lost even if the user closes the browser; it should remain in the database.

6.1. Requirements

The system's high-level functional requirements for end users include:

- Registration and login using email and password credentials.
- Browsing Movies and Series pages that display content cards, including poster image, title, rating, and a short overview.
- Viewing detailed information about a selected movie or TV series in a modal dialog .
- Adding or removing items to and from the Favorites list from the following pages :
    - Movies
    - Series
    - Recommendations
- Viewing the **Favorites** page, which aggregates all favorite movies and series for the logged-in user.
- Navigating to the Recommendations page and triggering the AI-based recommendation process by clicking the **"Get Recommendations"** button.
- Navigating to the Old Recommendations page to review past recommendation sessions and interact with them, such as marking suggested items as favorites.

For administrative or maintenance purposes within the scope of the project, the backend provides endpoints to:

- Synchronize content from TMDB into Movie/Serie tables.
- Inspect, manage, or clear test data during development.

There is no separate end-user administrator interface in this version of the system. Administrative and maintenance operations are performed using backend utilities or development tools such as Postman and database administration tools.

6.1.1. Use Cases

Below is the list of primary use cases of the system with brief textual descriptions and priorities:

Name:	**Get Movie Recommendations**		
Primary Actor:	User	Secondary Actors:	Movie Recommendation System
Description:	This use case describes how a user receives personalized movie and TV show recommendations based on their watch history and preferences.		
Trigger:	The user clicks the "Get Recommendations" button or opens the Recommendations page.		
Preconditions:	• The user is **registered and logged in** (valid JWT). • The user has at least **one favorite** movie or series, stored in the Favorites table. • The system has connectivity to Gemini API • The Movie and Serie tables are already populated with TMDB-derived metadata		
Postconditions:	• The user sees a list of **recommended movies and series** on the Recommendations page. • The recommendations are stored persistently in the system. • Recommended items are linked to a newly created or updated **OldMovieRecommendation / OldSerieRecommendation** session for the user. • The user can access this recommendation session again via the **Old Recommendations** page.		
Normal Flow:	**1.0 Get Recommendations** The user opens the **Recommendations** page. The frontend sends a `GET` or `POST` request to the backend endpoint responsible for generating recommendations. The backend: 1. Extracts and validates the **JWT** from the request header. 2. Identifies the **current user**. 3. Retrieves the user's **Favorites** (movies and series) from the database.		

	For each favorite item: 1. The backend constructs a **prompt** for Gemini containing the title and possibly additional context. 2. The backend calls the **Gemini API**, requesting 5 similar movie or series titles. 3. Gemini returns a list of titles as plain text. For each title returned by Gemini: 1. The backend searches the **local database** (Movie/Serie tables) for a matching record. 2. If a suitable match is found, the backend retrieves the metadata (poster, rating, overview, release dates, etc.) from the database. 3. If no match is found, the title is skipped. The backend creates a new **OldMovieRecommendation** and/or **OldSerieRecommendation** session for the user and attaches all matched items via Many-to-Many relationships. The backend returns a structured response containing: ● The list of recommended movies and series. ● Possible grouping by favorite ("Because you liked X…"). The frontend renders recommendation cards and displays them to the user. From these recommendation cards, the user may add or remove items from Favorites.

Alternative Flows:	**1.2 No User Data**
	
	● Condition: The user has **no favorites** in the system.
	
	1. The user opens the Recommendations page and clicks **"Get Recommendations"**.
	2. The backend retrieves the user's favorites and finds **none**.
	3. The system displays a message indicating:
	    1. "You do not have enough favorites to generate personalized recommendations."
	4. The system may present one or both of the following options:
	    1. Redirect the user to the **Movies** or **Series** pages to add some favorites.

	Alternative Flow Result: ● The use case ends after either redirecting the user to add favorites or showing generic content.

Exceptions:	**E1 – Gemini API Failure** The backend attempts to call the Gemini API, but the request fails (e.g., timeout, invalid key, or rate limit). The backend logs the error for debugging purposes. The system responds to the frontend with a user-friendly message such as: **"Recommendation service is temporarily unavailable."** In this case, no recommendations are generated and the use case ends. The user may try again later by re-triggering the recommendation process If a retry succeeds, the system returns to step 4 of the **Normal Flow**. **E2 – Local Db Matching Failure** Gemini returns a set of candidate titles, but the system cannot find reliable matches for some of them in the local database (e.g., titles are missing from the pre-synchronized dataset or have naming variations). For each unmatched title, the system skips it and does not include it in the recommendation set. The system may inform the user, for example: **"Some items could not be matched and are not shown."** The remaining matched recommendations are processed normally and displayed. **E3 – Database Connection Error** 1. While reading favorites or storing recommendations, the system encounters a **database error**. 2. The backend logs detailed error information for future debugging. 3. The frontend displays an error message such as: ○ "Unable to load data" 4. Once the database is available again: ○ The user can re-trigger the recommendation process, which returns to step 3 of the **Normal Flow**.

Priority:	• **High** – This is the central use case that delivers the primary value of the system.

Use Case Table

A complete tabular representation of all use cases (UC1: Register & Login, UC2: Browse Content, UC3: Manage Favorites, UC4: View Profile, UC5: Get Recommendations, UC6: View Old Recommendations, etc.) is defined in the broader project documentation.

In this Requirements Analysis Document, the **Get Movie Recommendations** use case is presented in full detail because:

- It is the core scenario that differentiates this system from a simple content browser.
- It integrates multiple components (frontend, backend, database, Gemini and TMDB based pre-synchronized data) in a single workflow.

6.1.2. Scenarios

Below are textual scenarios for some of the high-priority use cases.

Scenario 1 – Successful Recommendation Generation

- **Goal:** The user receives personalized movie and TV show recommendations based on their favorites.

Flow:

1. The user logs in and navigates to the **Recommendations** page.
2. The user clicks **"Get Recommendations"**.
3. The backend retrieves the user's favorites and calls Gemini with each favorite title.
4. Gemini returns 5 similar titles for each favorite.
5. For each title returned by Gemini, the backend searches the **local database** for matching Movie or **Serie** records that were previously synchronized.
6. The backend creates a new recommendation session and links all the matched items to it.
7. The frontend displays the recommendations as content cards, grouped by the original favorite (e.g., "Because you liked Inception…").
8. The user optionally adds some of these recommended items to **Favorites**.

Scenario 2 – No Favorites Available

- **Goal:** Provide meaningful behavior when the user has no favorites.

Flow:

1. The user logs in and opens the **Recommendations** page.
2. The user clicks **"Get Recommendations"**.
3. The backend checks the **Favorites** table and finds no records for this user.
4. The system informs the user that there are **no favorites yet**, so personalized recommendations cannot be generated.
5. The page suggests navigation options such as **"Go to Movies"** and **"Go to Series"** so that the user can add favorites.

Scenario 3 – Recommendation Service (Gemini) Failure

- **Goal:** Ensure graceful degradation when the external AI service is unavailable.

Flow:

1. The user opens the Recommendations page and clicks **"Get Recommendations"**.
2. The backend attempts to call the **Gemini API** but encounters an error (e.g., network issue, invalid credentials, or quota exceeded).
3. The backend logs the failure and returns an error response to the frontend.
4. The frontend shows a friendly message:
    - "Please try again later."
5. No recommendations are generated for this request, and the scenario ends.

Scenario 4 – Database Connection Error

- **Goal:** Maintain system stability when the database is temporarily unavailable.

Flow:

1. The user attempts to:
    - Open the Recommendations page, or
    - Add a favorite, or
    - View Old Recommendations.
2. The backend fails to read or write data due to a database connection issue.
3. The system returns a generic error message indicating that data cannot be loaded.
4. The user is not shown partial or inconsistent information; instead, they are asked to try again later.
5. Once the database is back online, the same action can be repeated successfully.

Scenario 5 – Missing Metadata

- **Goal:** Continue providing recommendations even when some metadata cannot be retrieved.

Flow:

1. Gemini returns a set of recommended titles.
2. For each returned title, the backend searches the **local database** for a matching Movie or Serie record.
3. For some matched records, certain metadata fields may be missing or incomplete (e.g., poster image).
4. The system does not discard these entries; instead, it continues processing them and provides recommendations using the available metadata.
5. If visual content such as a poster image is missing, the frontend displays the recommendation without a visual element while still showing textual information such as title, overview, rating, and dates.
6. The user is presented with a complete recommendation list, ensuring continuity and usability even when some visual data is unavailable.

Scenario 6 – Accessing Historical Recommendations

- **Goal:** Allow the user to revisit previously generated recommendation sessions.

Flow:

1. The user navigates to the **Old Recommendations** page.
2. The frontend sends a request to retrieve the user's historical recommendation sessions.
3. The backend queries the **OldMovieRecommendation** and **OldSerieRecommendation** tables, filtering by the current user.
4. The system loads the associated movies and series via Many-to-Many relationships.
5. The frontend displays:
    - A list of old sessions, potentially grouped by date or by source favorite.
    - Within each session, the actual recommended items as cards.
6. The user can again add/remove favorites from these cards, thus connecting past and present recommendation behavior.

Scenario 7 – Metadata Utilization from Pre-Synchronized TMDB Data

- **Goal:** Convert AI-generated text recommendations into rich, usable content cards using pre-synchronized metadata.

Flow:

1. For a given favorite movie or TV series, **Gemini** returns five suggested titles as plain text strings.
2. The backend searches the **local PostgreSQL database**, which already contains TMDB-derived metadata, for matching Movie or Serie records corresponding to the suggested titles.
3. For each successful match:
    - The backend retrieves the stored metadata such as poster path, rating, overview, release date, and other descriptive fields.
4. These existing Movie or Serie entities are reused and associated with the current recommendation session
5. The frontend receives the matched entities and renders content cards displaying:

    - Poster image (if available),
    - Title,
    - Rating,
    - Short description.
6. The user can interact with these cards in the same way as with catalog items, including adding them to **Favorites** or viewing additional details.

6.2. Nonfunctional Requirements

This section describes the non-functional requirements of the system. These requirements define quality attributes and constraints that influence how the system behaves, rather than what the system does.

Usability

- The web interface must be **clean, intuitive, and responsive**, enabling users to perform all core actions (login, browse, favorite, get recommendations, view old recommendations) with minimal learning.
- Navigation must be consistent across pages, with a persistent **header or navbar** that provides links to:
    - Home, Movies, Series, Favorites, Recommendations, Old Recommendations, Profile, and Logout.
- Feedback messages (success/error) must be **clear and visible**, particularly when:
    - Adding and removing favorites.
    - Triggering recommendations.
    - Handling errors (AI failure, database errors).
- The UI should be usable on **standard desktop and laptop resolutions**.

Reliability

- The system targets **high availability** during normal operation in the development environment.
- Critical components (database, Gemini integration, TMDB integration) must have **basic error handling**:
    - Timeouts and exceptions are caught and logged.
    - The user receives meaningful messages instead of generic failures.
- User data (accounts, favorites, recommendation sessions) must be **stored persistently** and guarded against accidental loss:
    - Database backups or export scripts may be used during development.
- When Gemini or TMDB services are unavailable, the system must:
    - Avoid crashing.
    - Provide fallback behavior where possible (e.g., generic content lists).
- Concurrent operations (multiple requests from the same or different users) must preserve **data**

consistency, especially when managing favorites and recommendation sessions.

Performance

- For typical usage and a moderate number of favorites, initial recommendation generation should complete within a few seconds, primarily influenced by the latency of the external **Gemini AI service**.
- Page navigation (e.g., switching between Movies, Series, Favorites) should feel responsive, with page load times around **4-5 seconds or less** under normal network conditions.
- Database queries should be **indexed and optimized**, particularly:
    - Retrieval of favorites by user.
    - Loading old recommendations and their linked content.
- The system is designed to support future scalability by following a stateless backend architecture and using an external relational database. This design allows horizontal scaling if the application is deployed in a more demanding production environment in the future.

Supportability

- The codebase follows a **modular architecture**:
    - Separation between controllers, services, repositories, and models in the backend.
    - Separation between pages, components, and services in the frontend.
- The project includes **developer documentation** such as:
    - API endpoint descriptions (Swagger/OpenAPI or Postman collection).
    - Basic setup instructions (how to run backend, frontend, database, environment variables for Gemini and TMDB).
- External AI services are integrated via well-defined interfaces:
    - If a different AI provider is chosen in the future, the system can replace Gemini integration without rewriting the entire backend.
- Logging is implemented to record:
    - Errors from Gemini/TMDB.
    - Unexpected exceptions in the backend.
    - Important lifecycle events (login, recommendation generation).

Implementation

- **Frontend:**
    - Implemented in **React** (JavaScript).
    - Uses **React Router** for navigation.
    - Uses **Axios** for HTTP calls to the backend.
    - Stores JWTs securely in browser storage (e.g., localStorage) and includes them in `Authorization` headers.
- **Backend:**
    - Implemented in **Java** with **Spring Boot**.
    - Uses **Spring Security** and **JWT** for authentication and authorization.
    - Uses **JPA/Hibernate** for ORM and interaction with PostgreSQL.
    - Exposes RESTful endpoints under `/api`, adhering to common practices (HTTP verbs, status codes, JSON payloads).
- **Database:**
    - Uses **PostgreSQL** as the relational database.
    - Tables for users, movies, series, favorites, and old recommendation sessions have been designed using standard normalization principles.

External Interface

(Programmatic interfaces, not UI.)

- The system does not expose a public API for third-party developers in its current academic version.
- The backend communicates with **PostgreSQL** over a secure and authenticated connection (credentials stored in configuration).
- The backend interacts with:
    - **Gemini API**, using authenticated HTTP requests and a defined prompt format.
    - **TMDB API**, using an API key and adhering to TMDB's request limits and usage policies.
- Internal services (Gemini integration service, TMDB integration service) are encapsulated behind interfaces so that future changes in external APIs can be localized.

Packaging

- The platform is delivered as a **web application** accessible via a standard web browser.
- No additional installation is required for end users beyond access to the correct URL.
- The deployment consists of:
    - A **Spring Boot backend** (e.g., runnable JAR or deployed to a web server).
    - A **React frontend** (e.g., built static files served by the backend or by a separate web server).
    - A **PostgreSQL database** running in a managed service or container environment.

Legal

- The system stores **user email and profile data** securely and does not share these details with third parties.
- No copyrighted video content is stored or streamed; only **metadata** (titles, posters, overviews) and links derived from TMDB are used.
- All external libraries and frameworks (React, Spring Boot, PostgreSQL, etc.) are used under appropriate open-source or community licenses.
- TMDB and Gemini are used in accordance with their **terms of use**, especially regarding API key usage and data handling.

Other Constraints

Social Impact:

- The platform helps users navigate the overload of digital entertainment by providing targeted recommendations.
- It can surface **lesser-known titles** suggested by AI and discovered via TMDB, potentially supporting independent productions and a more diverse viewing experience.

Ethical Considerations:

- The recommendation process is based on **content favorites** only; it does not use sensitive personal data.
- The system avoids explicit collection of demographic or highly sensitive information.
- Where possible, the system behavior should be transparent: recommendations are explained as being derived from the user's favorites ("Because you liked…"), instead of an opaque black-box approach.

Environmental Impact:

- The system can be deployed on cloud infrastructure where **energy-efficient** or **carbon-aware** options are preferred, though this is not strictly enforced within the scope of the academic project.

Business Constraints:

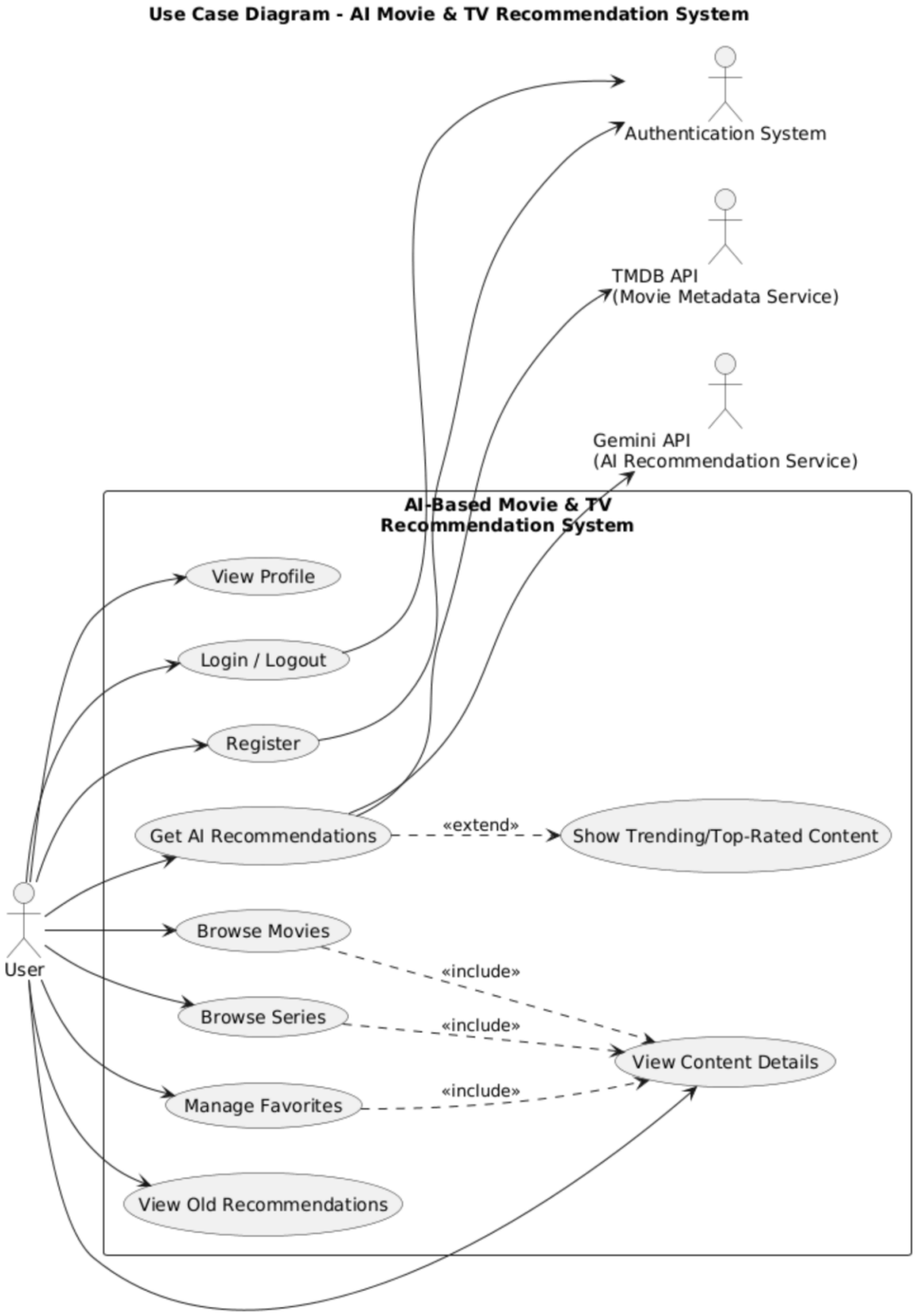
- This is a **graduation project**, not a commercial platform.
- No subscription, payment, or monetization features are included.
- API keys and hosting resources are constrained by academic and student-grade limits.

Regulatory Constraints:

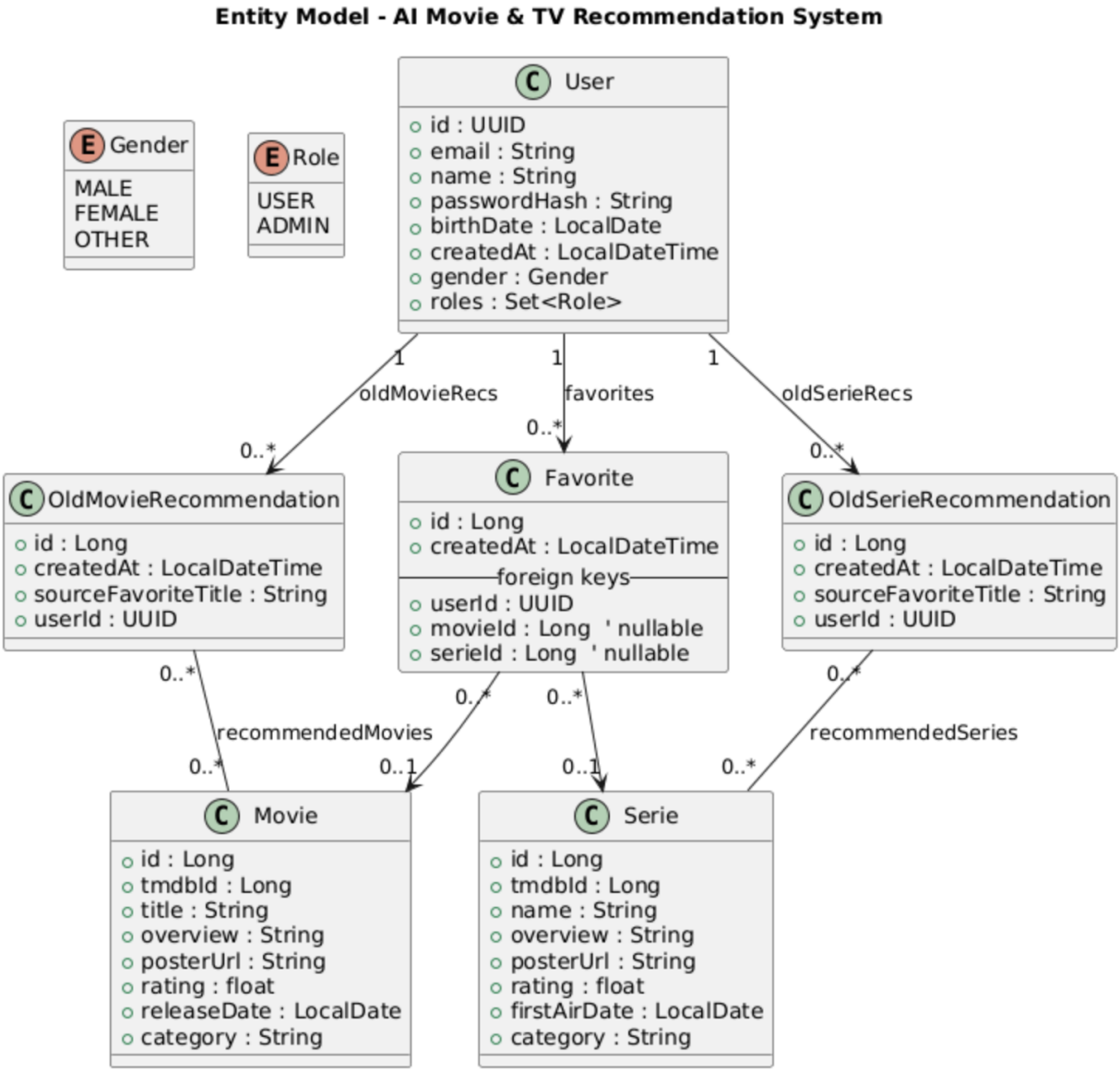
- The system must respect the **data usage policies** of TMDB and Gemini.
- If deployed in regions with strict privacy regulations, basic good practices (user consent, minimal data storage, secure passwords) should be followed, though a full legal compliance framework is beyond the project's scope

6.3. System Models

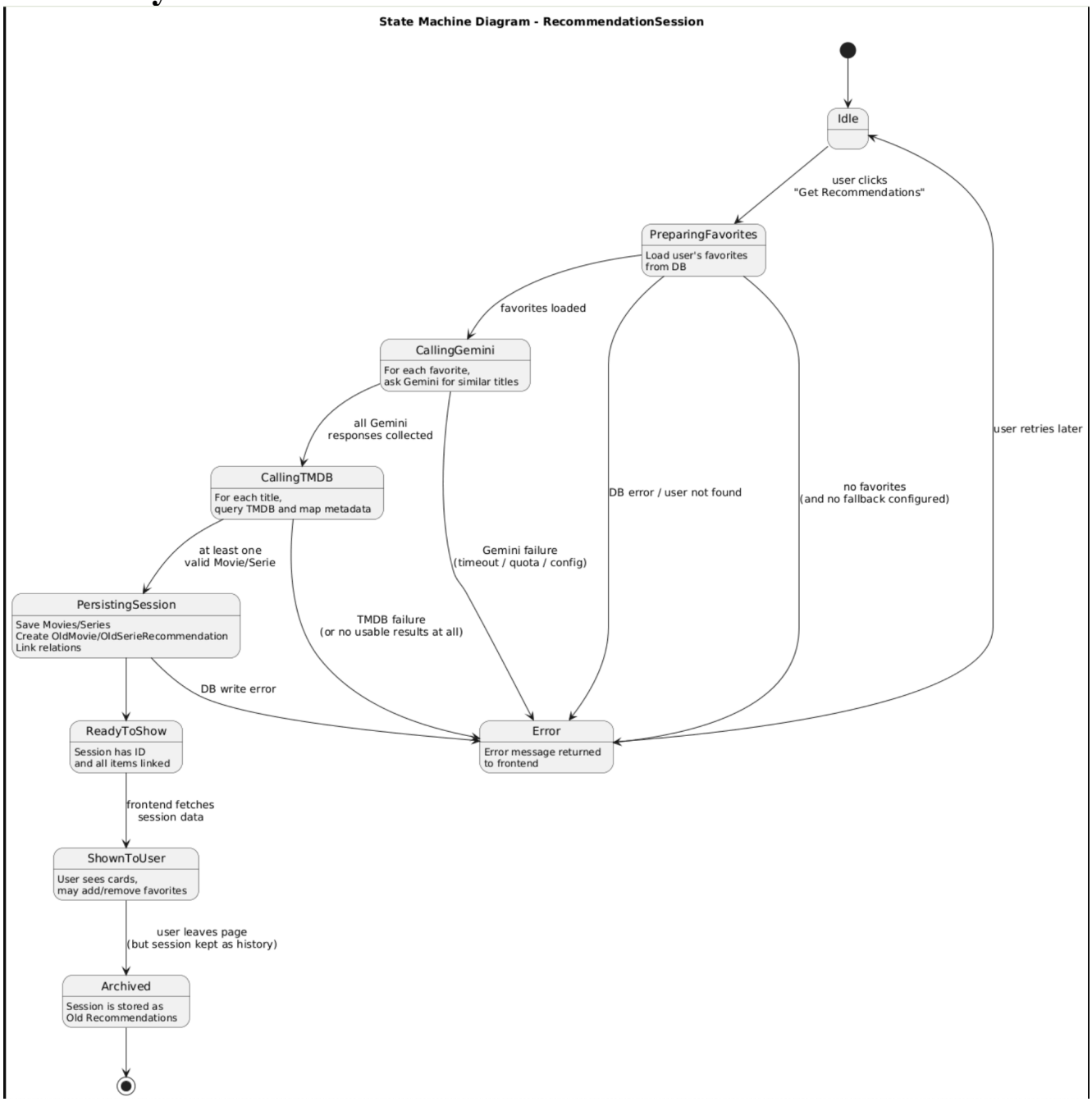
6.3.1. Use Case Model



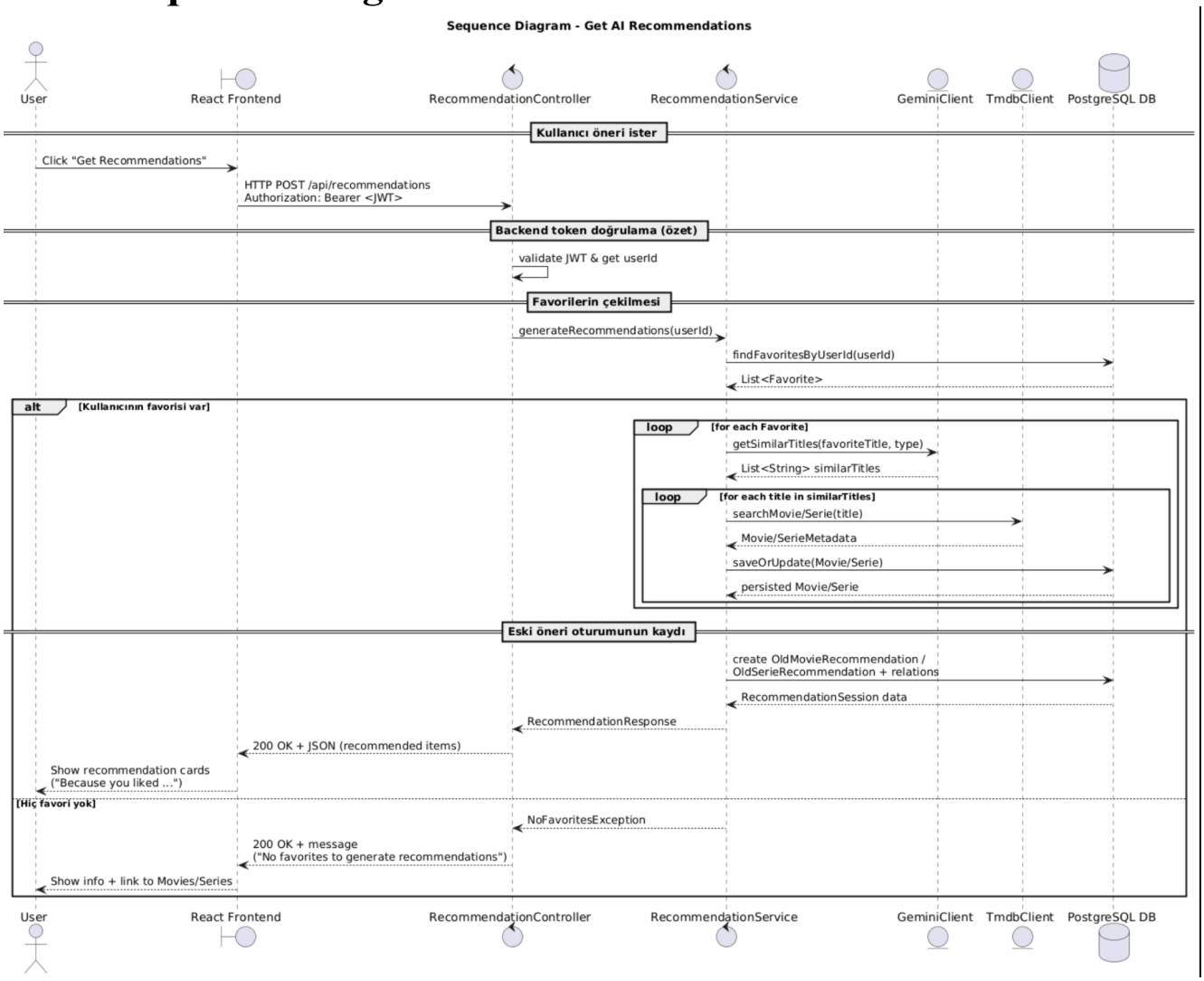
6.3.2. Object Model



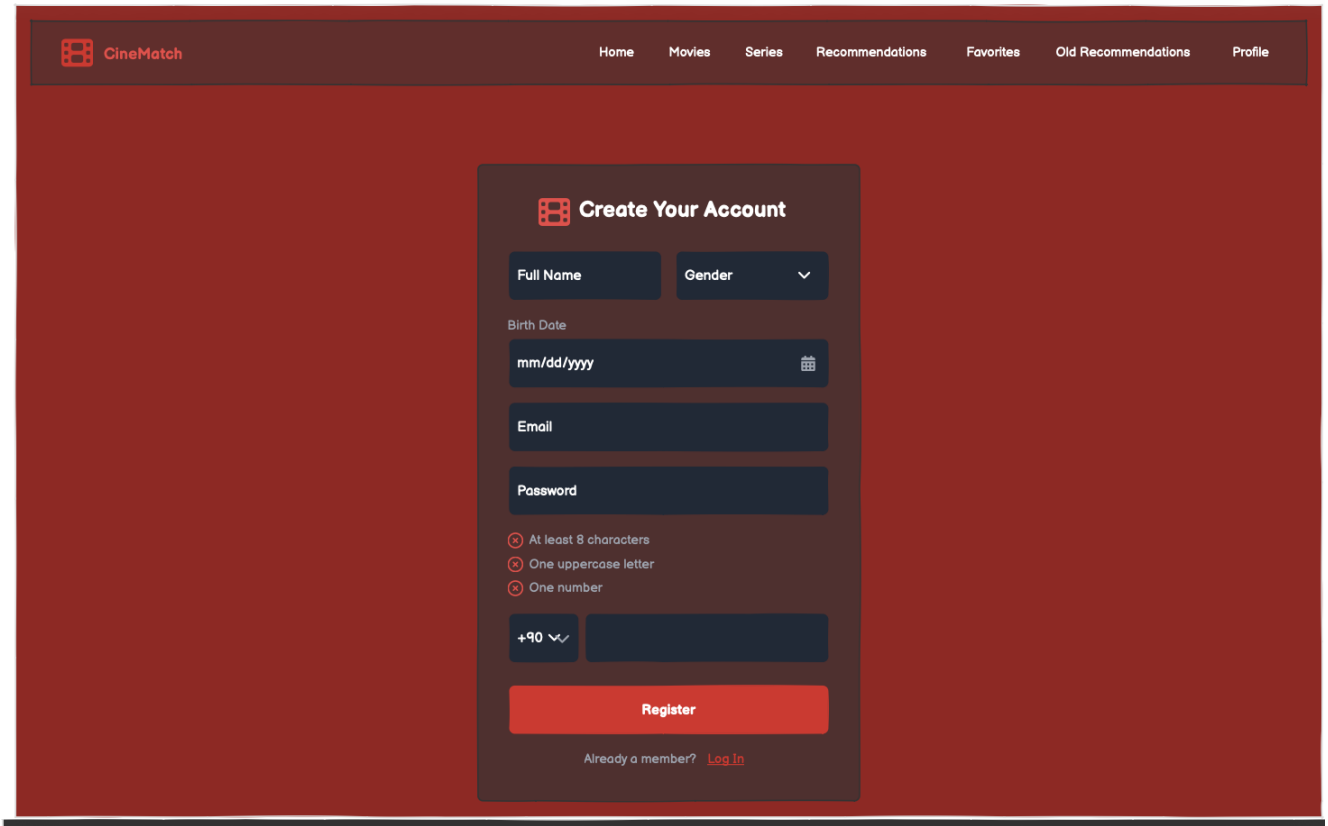
6.3.3. Dynamic Model

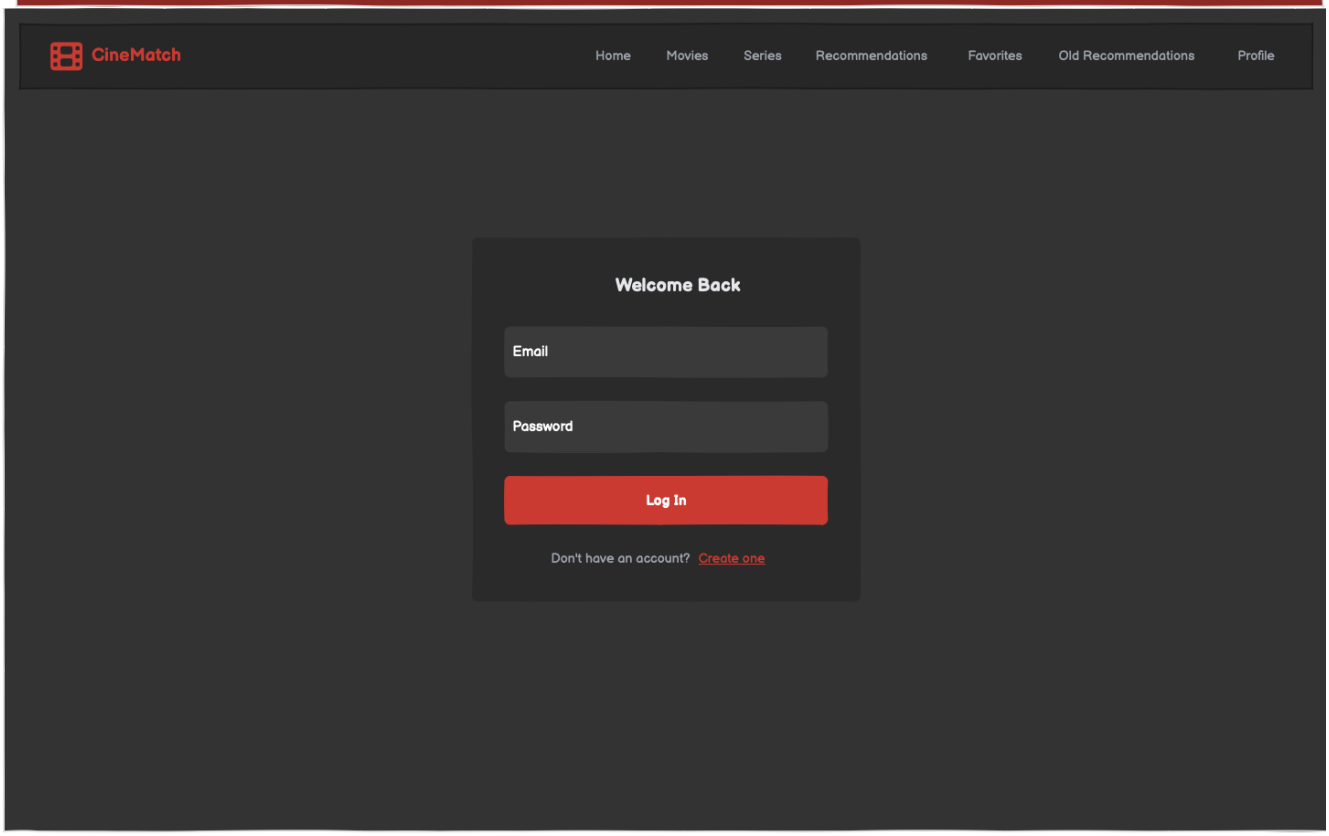


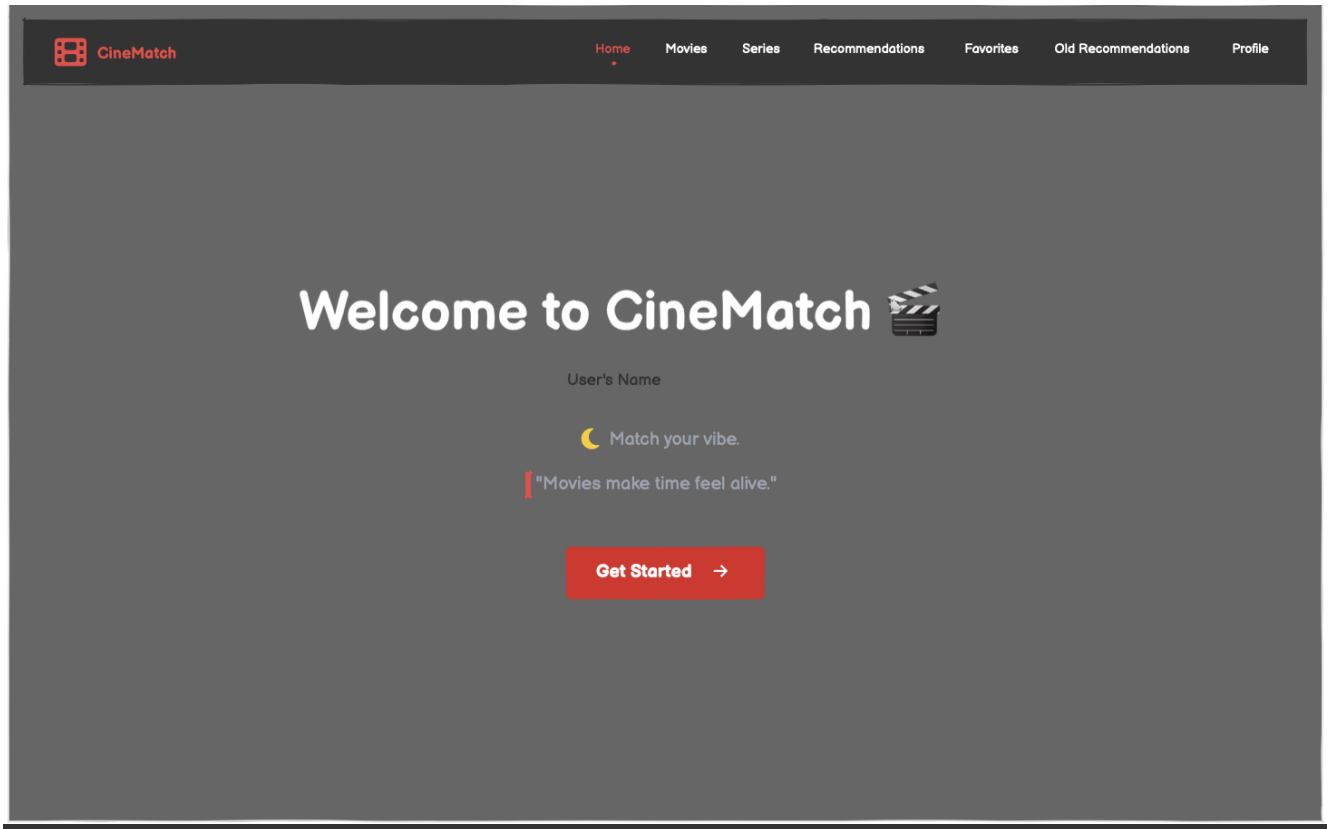
6.3.4 Sequence Diagram

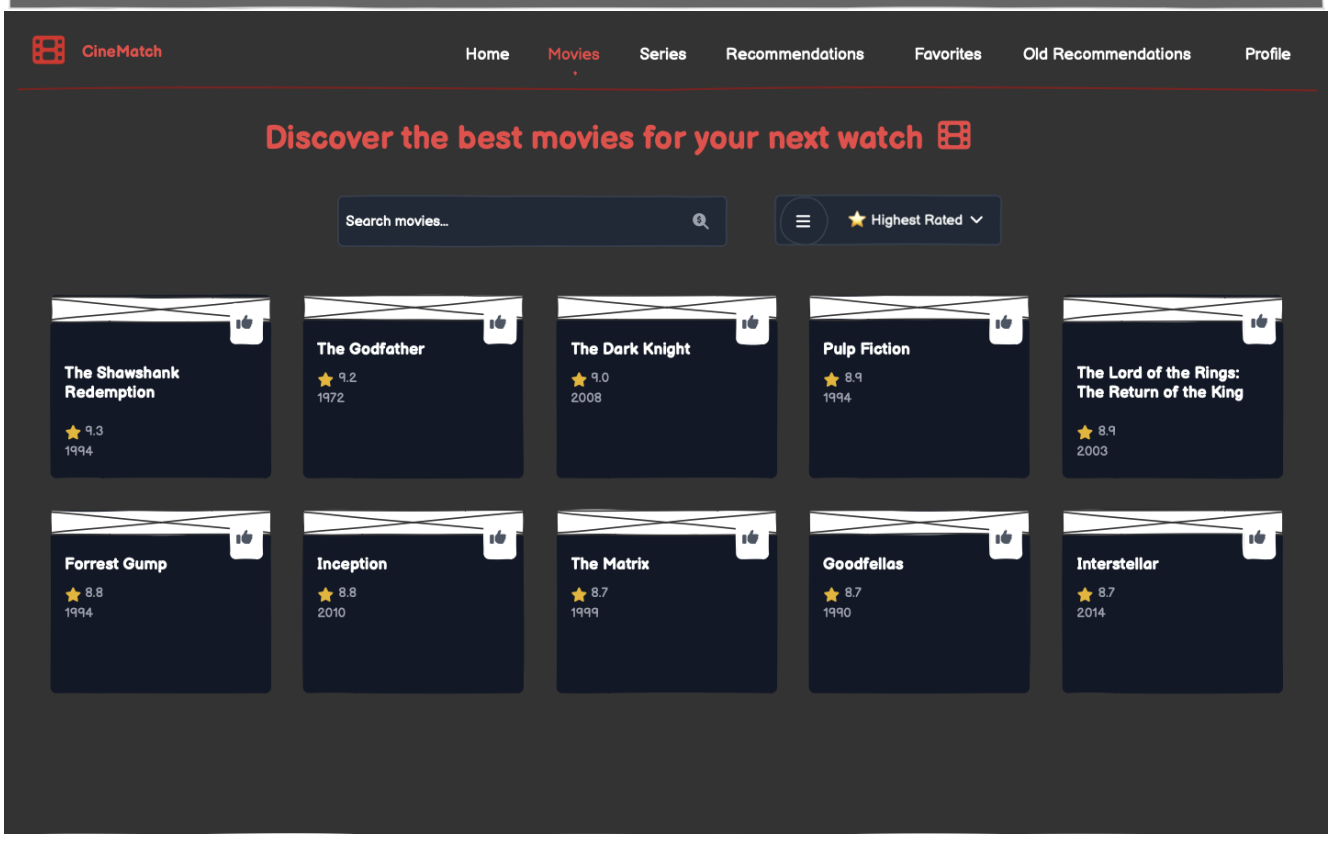


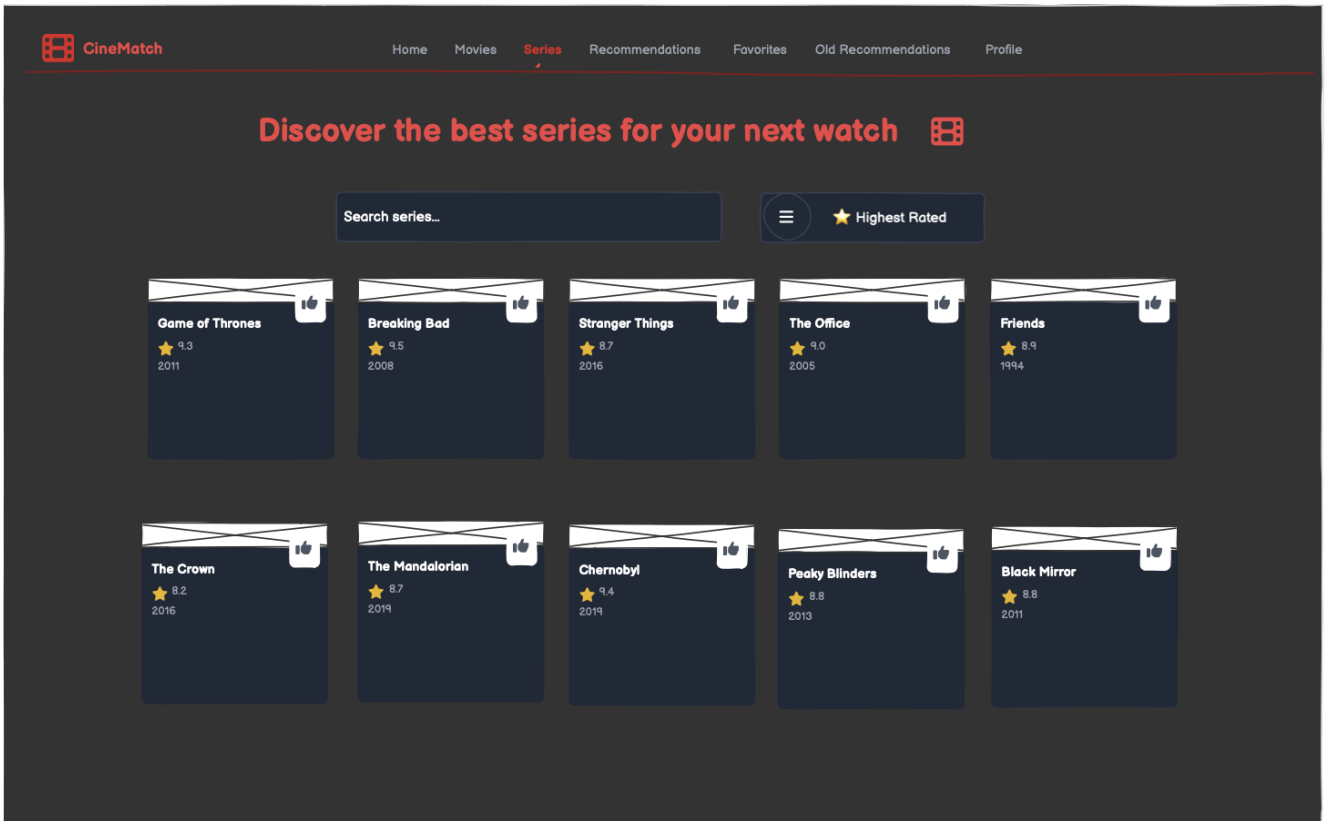
6.3.4. User Interface—Navigational Paths and Screen

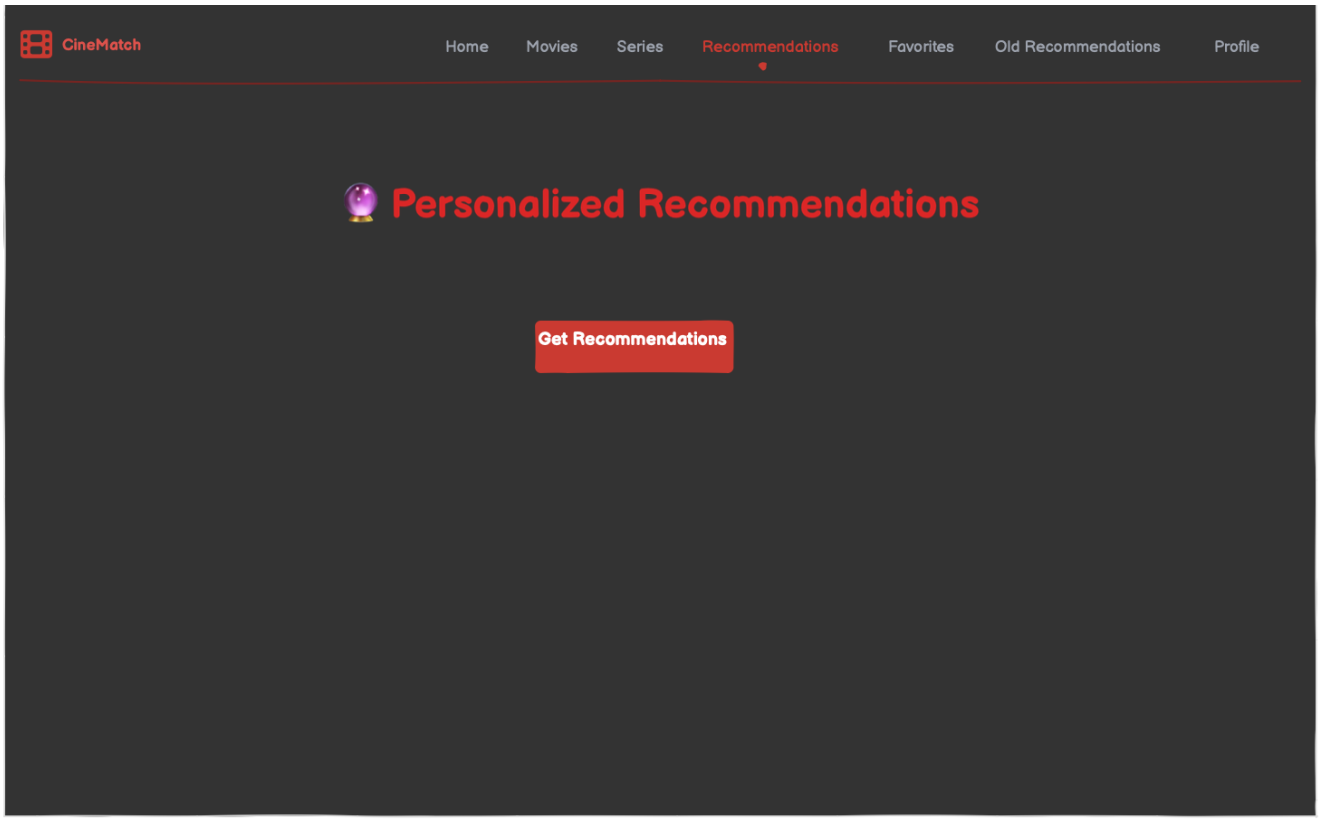


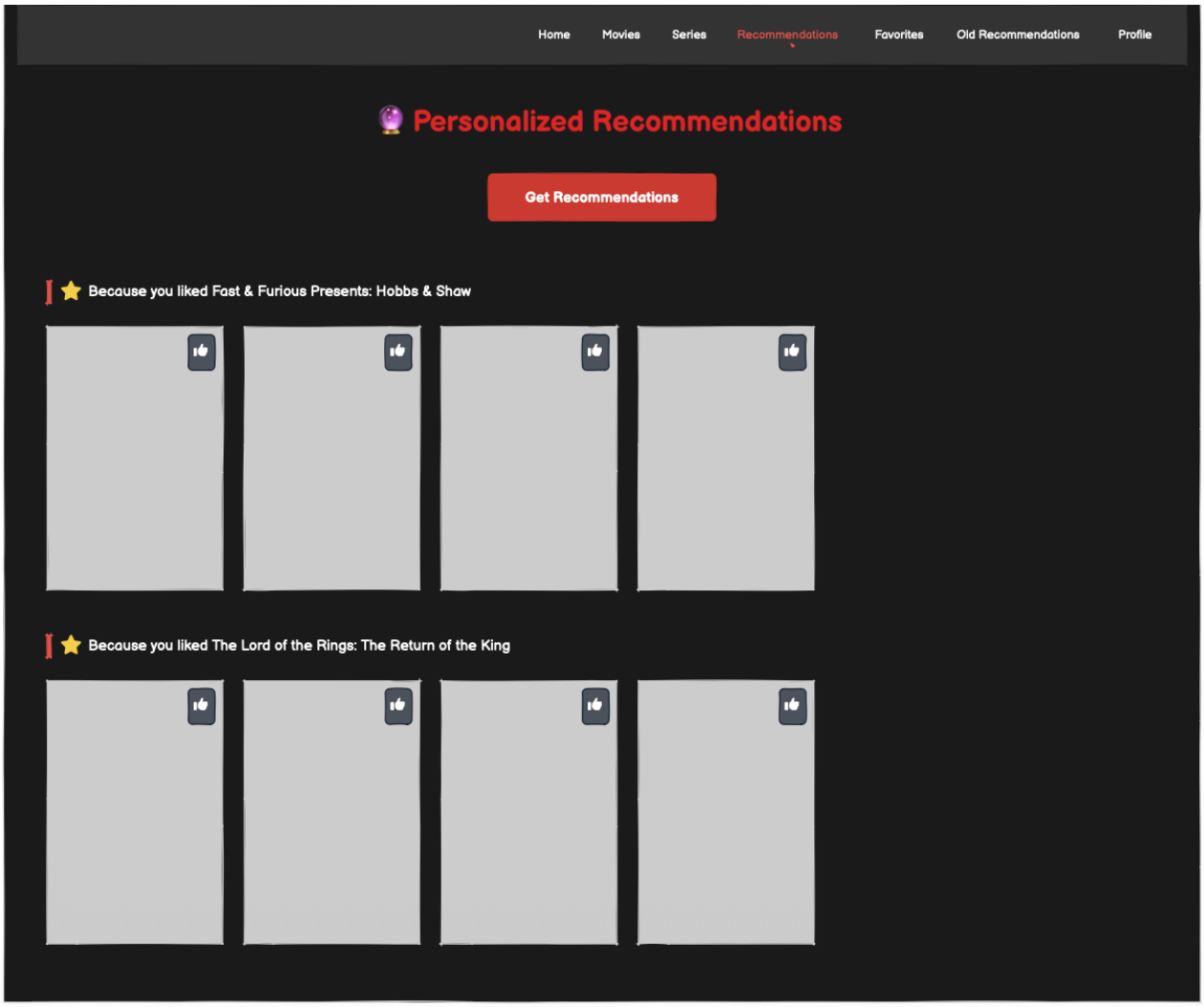


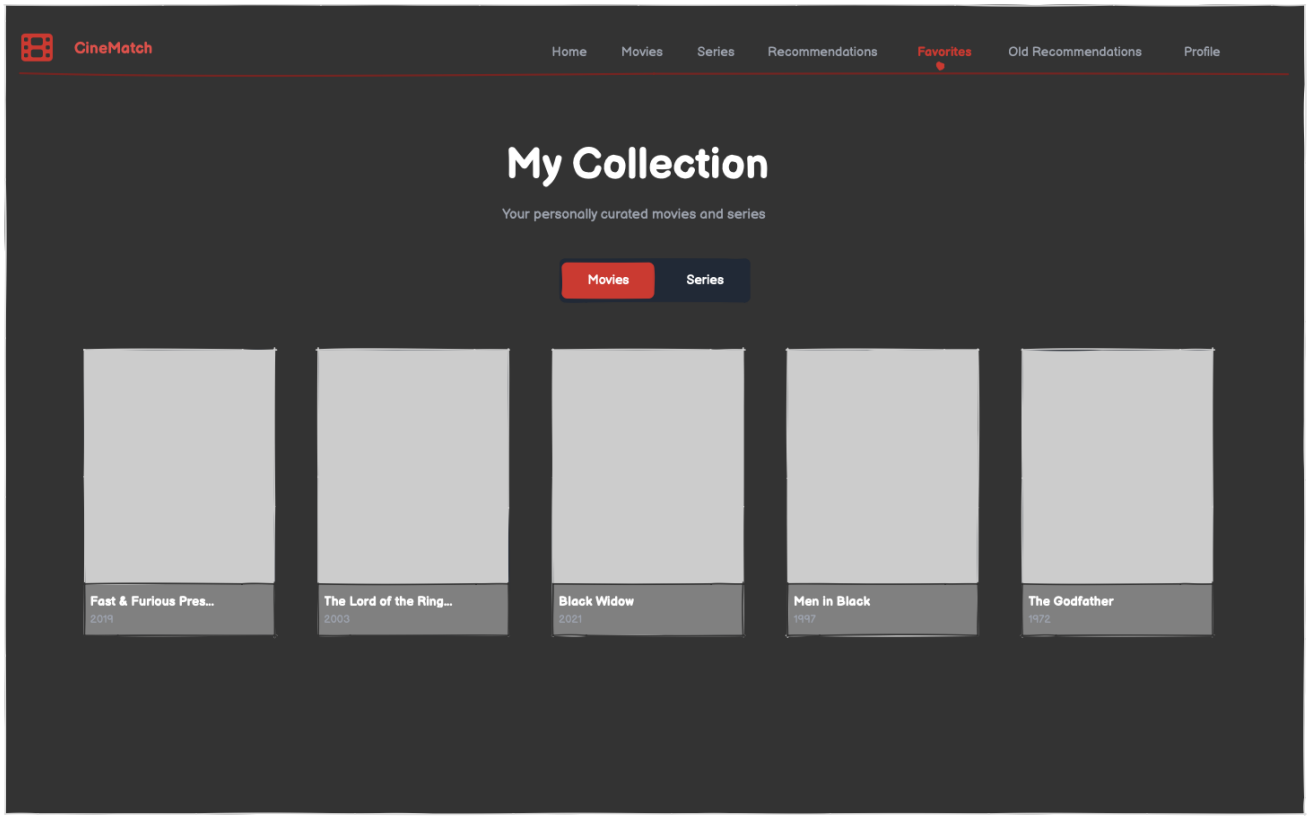

CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
Welcome to CineMatch
User's Name
Match your vibe.
"Movies make time feel alive."
Get Started


CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
Discover the best movies for your next watch
Search movies...
Highest Rated
The Shawshank Redemption
9.3
1994
The Godfather
9.2
1972
The Dark Knight
9.0
2008
Pulp Fiction
8.9
1994
The Lord of the Rings: The Return of the King
8.9
2003
Forrest Gump
8.8
1994
Inception
8.8
2010
The Matrix
8.7
1999
Goodfellas
8.7
1990
Interstellar
8.7
2014


CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
Discover the best series for your next watch
Search series...
Highest Rated
Game of Thrones
9.3
2011
Breaking Bad
9.5
2008
Stranger Things
8.7
2016
The Office
9.0
2005
Friends
8.9
1994
The Crown
8.2
2016
The Mandalorian
8.7
2019
Chernobyl
9.4
2019
Peaky Blinders
8.8
2013
Black Mirror
8.8
2011


CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
Personalized Recommendations
Get Recommendations


Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
Personalized Recommendations
Get Recommendations
Because you liked Fast & Furious Presents: Hobbs & Shaw
Because you liked The Lord of the Rings: The Return of the King


CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
My Collection
Your personally curated movies and series
Movies
Series
Fast & Furious Pres...
2019
The Lord of the Ring...
2003
Black Widow
2021
Men in Black
1997
The Godfather
1972

The Lord of the Rings:
The Return of the King

8.492 12/17/2003

Overview

As armies mass for a final battle that will decide the fate of the world--and powerful, ancient forces of Light and Dark compete to determine the outcome--one member of the Fellowship of the Ring is revealed as the noble heir to the throne of the Kings of Men. Yet, the sole hope for triumph over evil lies with a brave hobbit, Frodo, who, accompanied by his loyal friend Sam and the hideous, wretched Gollum, ventures deep into the very dark heart of Mordor on his seemingly impossible quest to destroy the Ring of Power.

Favorite Saved

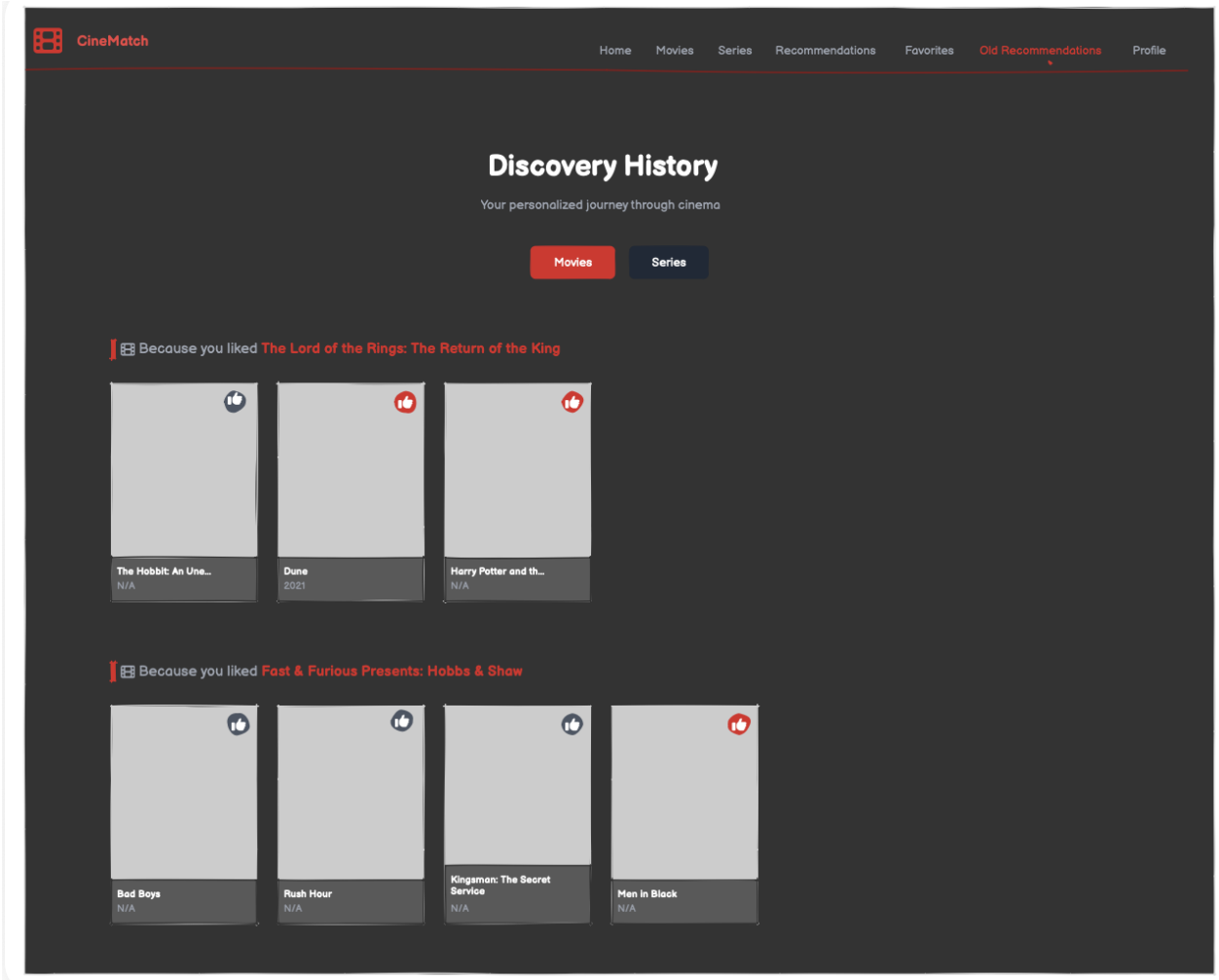
The Lord of the Rings:
The Return of the King

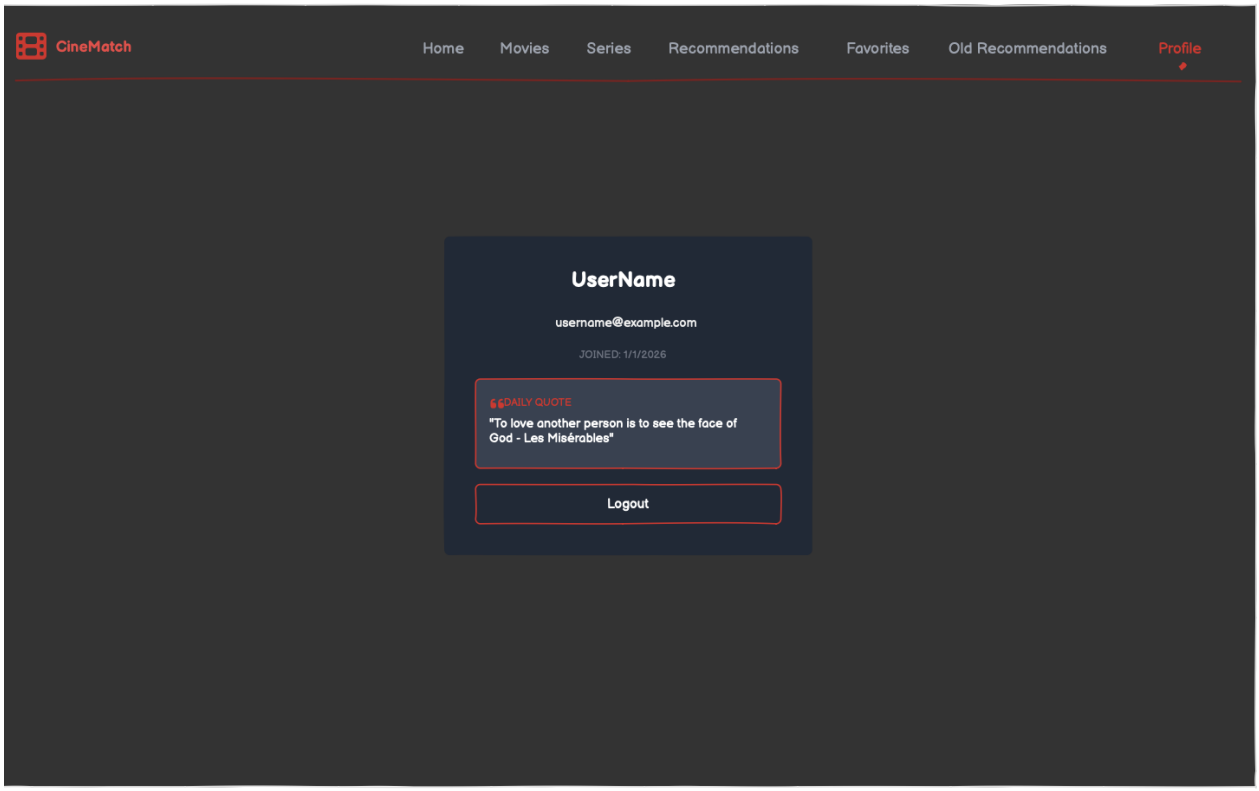
8.492 12/17/2003

Overview

As armies mass for a final battle that will decide the fate of the world--and powerful, ancient forces of Light and Dark compete to determine the outcome--one member of the Fellowship of the Ring is revealed as the noble heir to the throne of the Kings of Men. Yet, the sole hope for triumph over evil lies with a brave hobbit, Frodo, who, accompanied by his loyal friend Sam and the hideous, wretched Gollum, ventures deep into the very dark heart of Mordor on his seemingly impossible quest to destroy the Ring of Power.

Remove from Favs


CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
Discovery History
Your personalized journey through cinema
Movies
Series
Because you liked The Lord of the Rings: The Return of the King
The Hobbit: An Une...
N/A
Dune
2021
Harry Potter and th...
N/A
Because you liked Fast & Furious Presents: Hobbs & Shaw
Bad Boys
N/A
Rush Hour
N/A
Kingsman: The Secret Service
N/A
Men in Black
N/A


CineMatch
Home
Movies
Series
Recommendations
Favorites
Old Recommendations
Profile
UserName
username@example.com
JOINED: 1/1/2026
DAILY QUOTE
"To love another person is to see the face of God - Les Misérables"
Logout

7. Other Analysis Elements

1.6. Risks and Alternatives

Table

Risk ID	Risk Description	Likelihood	Effect on Project	Plan (B-Plan)
R1	Gemini returns irrelevant or low-quality recommendations.	Medium	High – Reduces user satisfaction and system value.	Refine prompts, add basic filtering rules (e.g., by rating), or temporarily fallback to TMDB-based similarity (genres, categories).
R2	Problems connecting the AI system or TMDB synchronization problems	Medium	Medium – Slows development and testing.	Implement robust error handling, mock services for local testing, and clear configuration management for API keys.
R3	Project scope grows beyond the planned time (feature creep).	High	Medium – Risk of unfinished or unstable features.	Prioritize core flows (auth, favorites, recommendations, old recommendations) and postpone advanced features (social sharing, advanced filters).
R4	External API quota or rate limits are exceeded during testing/demo.	Medium	Medium – Recommendations may fail temporarily.	Cache responses where possible, limit the number of calls per request, and prepare fallback demo data.
R5	Database schema changes late in the project cause migration issues.	Low–Med	Medium – Risk of inconsistent data or delays.	Design schema carefully early on, use migration scripts (e.g., Flyway/Liquibase), and document changes.

1.7. Project Plan

The project is broken down into several work packages (WPs) to ensure structured and manageable development.

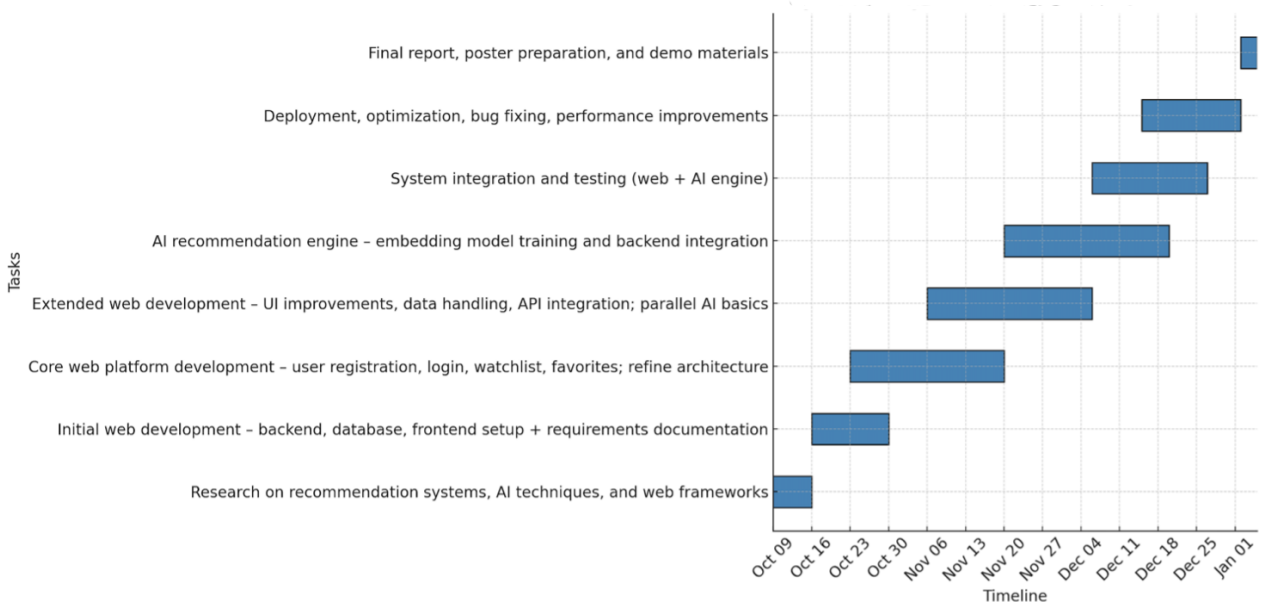


Table 3. List of work packages

WP#	Work package title	Members involved
WP1	Project Management & Requirements Analysis	Tülin Eylül Erdoğan, Ihsan Eren Erben
WP2	System Architecture & Database Design	Tülin Eylül Erdoğan, Ihsan Eren Erben
WP3	Backend Development (API)	Tülin Eylül Erdoğan, Ihsan Eren Erben
WP4	Frontend Development (UI)	Tülin Eylül Erdoğan, Ihsan Eren Erben
WP5	AI Recommendation Engine Development	Tülin Eylül Erdoğan, Ihsan Eren Erben
WP6	Integration, Testing & Deployment	Tülin Eylül Erdoğan, Ihsan Eren Erben

Table 4. Work Package 1

WP 1: Project Management & Requirements Analysis			
Start date: *<09.10.2025>* **End date:** *<16.10.2025>*			
Leader:	*İhsan*	**Members involved:**	*Eylül, İhsan*
Objectives: To establish a clear project foundation by defining the scope, objectives, and detailed requirements. This phase focuses on planning the entire project lifecycle and producing the core Requirements Analysis Document (RAD).			
Tasks: **Task 1.1** Conduct initial project planning and define the development timeline. **Task 1.2** Define and document the system's purpose, scope, and objectives (Chapter 1).			
Task 1.3 Identify and describe all functional requirements, use cases, and scenarios (Chapter 3.2). **Task 1.4** Specify all non-functional requirements, including performance, usability, and security (Chapter 3.3). **Task 1.5** Analyze and document potential project risks and contingency plans (Chapter 4.1).			
Deliverables **D2.1:** System Architecture Document. **D2.2:** Finalized Database Schema. **D2.3:** API Specification Document.			

Table 5. Work Package 2

WP 2: System Architecture & Database Design			
Start date: *<16.10.2025>* **End date:** *<30.10.2025>*			
Leader:	*Eylül*	**Members involved:**	*Eylül, İhsan*
Objectives: To define the overall system architecture, select the technology stack, and design a scalable and efficient database schema that supports all functional requirements.			
Tasks: **Task 2.1** Define the three-tier architecture (Frontend, Backend, Database). **Task 2.2** Create the Entity-Relationship (ER) diagram for the database. **Task 2.3** Normalize the schema and define tables, columns, and relationships. **Task 2.4** Define the RESTful API endpoints and data transfer objects (DTOs).			
Deliverables **D2.1:** System Architecture Document. **D2.2:** Finalized Database Schema. **D2.3:** API Specification Document.			

Table 6. Work Package 3

WP 3: Backend Development (API)			
Start date: *<23.10.2025>* **End date:** *<20.11.2025>*			
Leader:	*Eylül*	**Members involved:**	*Eylül, İhsan*
Objectives: To develop the server-side application that handles all business logic, data processing, and communication between the user interface, the database, and the AI recommendation engine.			
Tasks: **Task 3.1** Set up the backend server environment and project structure. **Task 3.2** Implement the database connection and data models. **Task 3.3** Develop RESTful API endpoints for user authentication and management (UC1, UC8). **Task 3.4**Develop API endpoints for content interaction (browsing, viewing details, marking as watched, managing favorites) (UC2, UC3, UC4, UC7). **Task 3.5** Develop API endpoints for administrative content management (UC6). **Task 3.6**Implement the service layer to communicate with the AI engine for generating recommendations (UC5).			
Deliverables: **D3.1:** Source code for the backend API. **D3.2:** A fully functional and tested RESTful API. **D3.3:** API documentation (e.g., Swagger or Postman collection).			

Table 7. Work Package 4

WP 4: Frontend Development (UI)			
Start date: *<23.10.2025>* **End date:** *<20.11.2025>*			
Leader:	*İhsan*	**Members involved:**	*Eylül, İhsan*
Objectives: To develop a responsive, intuitive, and interactive web interface based on the mock-ups and non-functional requirements. The frontend will communicate with the backend via the RESTful API.			
Tasks: **Task 4.1** Set up the React/Next.js development environment. **Task 4.2** Implement user registration and login components (UC1). **Task 4.3** Develop the main dashboard and content browsing pages (UC2). **Task 4.4** Create the content details page with "Mark as Watched" and "Add to Favorites" functionality (UC3, UC4, UC7). **Task 4.5** Implement the recommendations display page (UC5).			
Deliverables: **D3.1:** Source code for the backend API. **D3.2:** A fully functional and tested RESTful API. **D3.3:** API documentation (e.g., Swagger or Postman collection).			

Table 8. Work Package 5

WP 5: AI Recommendation Engine Development			
Start date: *<20.11.2025>* **End date:** *<06.12.2025>*			
Leader:	*Eylül*	**Members involved:**	*Eylül, İhsan*
Objectives: To research, develop, and train an embedding-based machine learning model capable of generating personalized content recommendations based on user watch history and preferences.			
Tasks: **Task 5.1** Research and select an appropriate embedding-based algorithm. **Task 5.2** Preprocess the selected dataset for training. **Task 5.3** Implement the model training and evaluation pipeline. **Task 5.4** Develop a service to expose the trained model via an internal API for the backend.			
Deliverables: **D5.1:** A trained and serialized machine learning model. **D5.2:** Source code for the AI subsystem. **D5.3:** An internal API for generating recommendations.			

Table 9. Work Package 6

WP 6: Integration, Testing & Deployment			
Start date: <*06.12.2025***>** **End date:** <*05.01.2025*>			
Leader:	*İhsan*	**Members involved:**	*Eylül, İhsan*
Objectives: To integrate all developed components (frontend, backend, AI engine), conduct comprehensive testing to ensure the system meets all requirements, and deploy the application to a live server environment.			
Tasks: **Task 6.1** Integrate the frontend UI with the backend API endpoints. **Task 6.2** Connect the backend service with the AI recommendation engine service. **Task 6.3** Perform end-to-end testing of all user scenarios and use cases. **Task 6.4**Conduct performance and usability testing to ensure non-functional requirements are met.			
Task 6.5 Resolve bugs and issues identified during the testing phase. **Task 6.6** Configure the production server environment and deploy the final application.			
Deliverables: **D6.1:** A fully integrated and operational web application. **D6.2:** A comprehensive test report summarizing test cases and results. **D6.3:** A deployed application accessible via a public URL. **D6.4:** Final project documentation and source code repository.			

8. Glossary

This section provides definitions for key technical and domain-specific terms used throughout the document to ensure clarity and common understanding.

- **API (Application Programming Interface):** A set of rules and endpoints that allow software components to communicate. In this project, the backend exposes a RESTful API, and it also consumes external APIs (Gemini, TMDB).

- **Backend:** The server-side logic of the application, implemented in Spring Boot, responsible for handling requests, accessing the database, and calling external services.

- **Collaborative Filtering:** A classical recommendation technique that uses the preferences of many users to make predictions for a specific user. Mentioned for comparison; not directly implemented in this project.

- **Content-Based Filtering:** A classical recommendation technique that uses item attributes (e.g., genres, actors) to recommend similar items. Also used only as a conceptual background.

- **Embedding:** A representation of items or users as numeric vectors in a low-dimensional space. While common in modern recommender systems, this project instead relies on **Gemini** as an external AI service instead of training its own embedding model.

- **Frontend:** The client-side of the application built with React, responsible for rendering UI and interacting with the backend.

- **JWT (JSON Web Token):** A token format for securely transmitting user identity between frontend and backend; used here for stateless authentication.

- **RESTful API:** An API style using HTTP methods (GET, POST, PUT, DELETE) and resource-based URLs, returning JSON responses.

- **Gemini API:** Google's generative AI service that provides similar movie/series titles based on favorite titles sent by the system.

- **TMDB:** The Movie Database, an external service providing movie/TV metadata used to enrich Gemini's text suggestions.

9. References

(1) Wikipedia, List of Collaborative Software, https://en.wikipedia.org/wiki/List_of_collaborative_software, Accessed on 22 October 2019.Wikipedia, Comparison of Project Management Software, https://en.wikipedia.org/wiki/Comparison_of_project_management_software, Accessed on 22 October 2019.

(2) CATME Smarter Teamwork, https://catme.org/, Accessed on 22 October 2019.

(3) TEAMMATES - Online Peer Feedback/Evaluation System for Student Team Projects, https://teammatesv4.appspot.com/, Accessed on 22 October 2019.

(4) Bruegge B. & Dutoit A.H. (2010). *Object-Oriented Software Engineering Using UML, Patterns, and Java*, 3rd ed. Prentice-Hall.

(5) Ricci, F., Rokach, L., & Shapira, B. (2011). *Introduction to Recommender Systems Handbook*. Springer.

(6) React Documentation. Retrieved from https://reactjs.org/

(7) PostgreSQL Official Documentation. Retrieved from https://www.postgresql.org/docs/

(8) The Movie Database (TMDB) . *TMDB API Documentation*

https://developer.themoviedb.org/

(9) Spring Boot. *Spring Boot Reference Documentation*

.https://spring.io/projects/spring-boot

(10) Google. *Gemini API Documentation*.

https://ai.google.dev/