Table of Contents

1. Introduction

Purpose. MoviePsychic recommends films based on user ratings and (future) personality features. The goal is to deliver fast, high-quality recommendations with a modern, dynamic UI.

Scope. A web application composed of a React frontend, a Django backend, an in-process Python recommendation engine, a PostgreSQL database, and a Redis cache.

Vision & Constraints. Per the SRS: frontend = React, backend = Django (Python), database = PostgreSQL, dataset = MovieLens (for training/recs), metadata & artwork via TMDB API (for search/details).

2. Problem Solving & Decomposition

We apply divide-and-conquer to decompose MoviePsychic into independent modules that align with event-driven and object-oriented design principles.

Subproblems / Modules:

• Frontend UI (React) — dynamic Netflix-style interface with live updates.

• API Layer (Django) — REST endpoints for auth, search, rating, and recommendations.

• Recommendation Engine (Python) — in-process component that builds user vectors and computes top-N.

• Data Layer (PostgreSQL) — users, movies, ratings; indexed for performance.

• Caching (Redis) — caches per-user recommendations to meet latency targets.

• External Integration (TMDB API) — enriches movie metadata and images.

Inputs/Outputs.

• Inputs: user ratings (1–5 stars), search queries, (future) personality responses.

• Outputs: top-10 movie recommendations with explanation tags (e.g., 'similar to your likes', 'liked by similar users').

3. High-Level System Architecture

The architecture emphasizes performance and user experience: React handles a dynamic UI and live refresh of recommendations; Django exposes REST endpoints; the Python engine computes recommendations using a MovieLens-derived table; Redis accelerates repeated requests; TMDB supplies images and metadata.

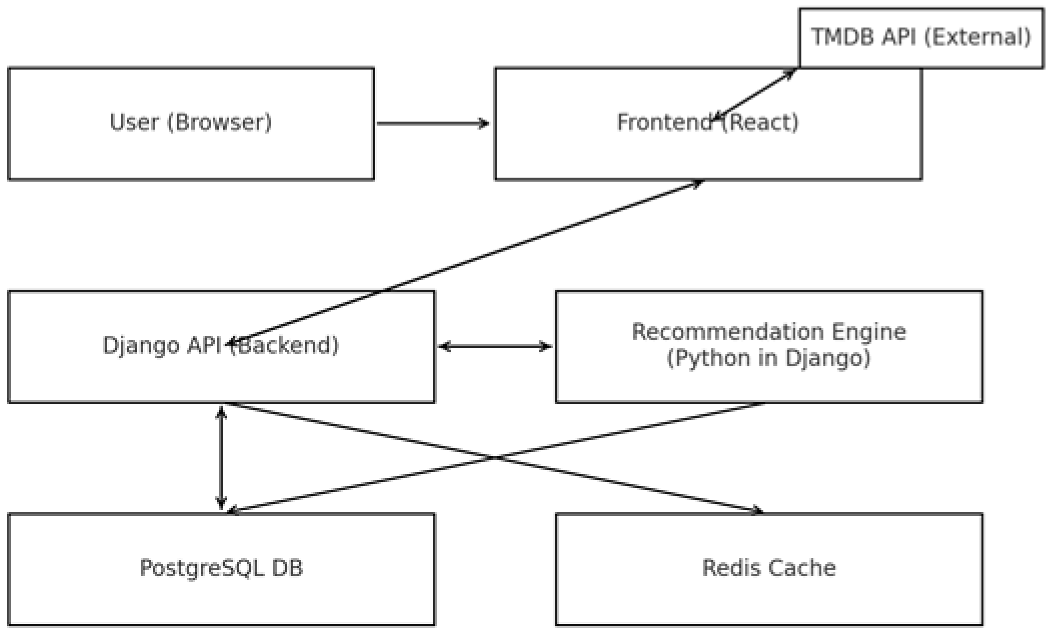

Figure 1. System Architecture Diagram.

4. Component Specifications & Interfaces

4.1 Frontend (React)

Responsibilities: Modern Netflix/HiAnime-style interface with live updates; debounced rating calls; optimistic UI for instant feedback.

Key Views: Home (recommendations), Search, Movie Detail, Profile (ratings).

APIs Used:

• POST /api/ratings {user_id, movie_id, stars}

• GET /api/recommendations?user_id=…

• GET /api/search?query=…

• GET /api/movies/{id}

4.2 Backend API (Django)

Responsibilities: Authentication, input validation, request orchestration, caching, and rate limiting for external API calls.

Endpoints (examples):

• POST /api/ratings — persist rating, invalidate cache, return ACK.

• GET /api/recommendations — fetch from Redis or compute via engine.

• GET /api/search — proxy to TMDB or local index.

Non-Functional: All endpoints aim for <500ms under nominal load; JWT/session auth; input validation on server-side.

User Authentication and Security:

The backend handles user registration, login, and password management through Django's authentication system.

Passwords are securely hashed using Argon2 or Bcrypt before being stored in the PostgreSQL database.

Sessions or JWT tokens maintain secure logins, and server-side validation prevents unauthorized access to user-specific data.

This design ensures security, maintainability, and compliance with modern web standards.

4.3 Recommendation Engine (Python in Django)

Responsibilities: Build/update user vectors; compute top-N recommendations; explain recommendations.

Algorithm (initial): Content + collaborative hybrid (simple):

• Use ratings matrix (MovieLens) for user-based or item-based similarity (cosine).

• Filter candidates to popular/top-X% for latency (per SRS).

• (Future) Incorporate personality weights to adjust similarity scores.

Interfaces:

• compute_top_n(user_id, n=10) -> [MovieID] + reasons

4.4 Data Layer (PostgreSQL)

Tables:

• users(id, username, password_hash, created_at)

• movies(id, title, year, genres, tmdb_id, popularity, director)

• ratings(user_id, movie_id, stars, rated_at)

Indexes: ratings(user_id), ratings(movie_id), movies(popularity).

Performance Targets (from SRS): single movie fetch <100ms; user rating history (≤1000 ratings) <1s.

4.5 Caching Layer (Redis)

Responsibilities: Cache per-user recommendations; invalidate on new rating; optional caching of TMDB lookups.

Keys: recs:{user_id} -> list of movie IDs (TTL: e.g., 5–15 minutes).

4.6 External Integration (TMDB API)

Use: search, posters, and metadata. Rate limiting with backoff; cache responses in Redis.

5. Sequence Diagram — Live Update on Rating

Flow: user submits a rating; frontend posts to API; API saves rating and invalidates cache; frontend requests recommendations; engine computes or returns cached results; UI updates immediately.

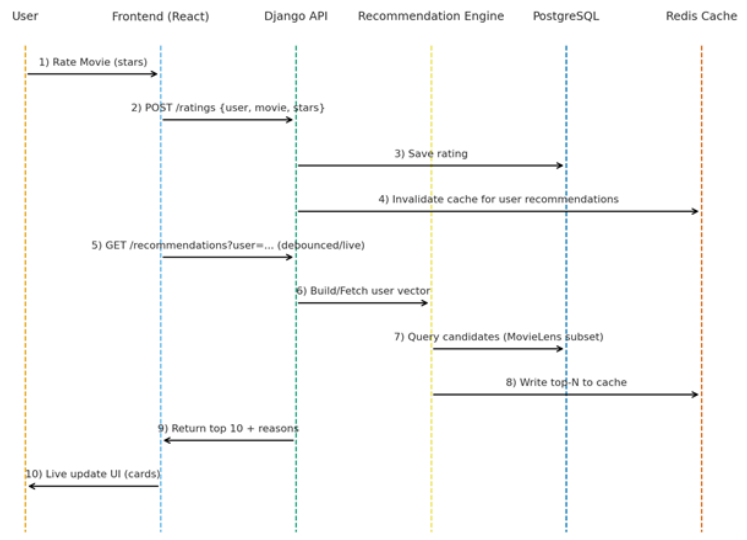
User
Frontend (React)
Django API
Recommendation Engine
PostgreSQL
Redis Cache
1) Rate Movie (stars)
2) POST /ratings {user, movie, stars}
3) Save rating
4) Invalidate cache for user recommendations
5) GET /recommendations?user=... (debounced/live)
6) Build/Fetch user vector
7) Query candidates (MovieLens subset)
8) Write top-N to cache
9) Return top 10 + reasons
10) Live update UI (cards)

Figure 2. Sequence Diagram for rating → live recommendations.

6. Classes and Encapsulation

The design enforces low coupling and high cohesion: the UI never queries the DB directly; backend exposes a stable REST API; the recommendation engine is encapsulated behind a function/class; data access is via ORM layer with repository-style services

Class	Description	Key Attributes	Key Methods
Movie	Represents a film in the database and provides access to metadata.	title, year, genre, director, tmdb_id	get_metadata(), get_avg_rating()
User	Represents a registered user, including preferences and rated movies.	username, favorite_genres, favorite_directors, rated_movies	rate_movie(), get_recommendations()

Searcher	Handles movie searches based on title, director, or genre parameters.	query, filters	search_movies(), sort_results()
Recommender	Extends Searcher to generate Top-N recommendations using algorithms and comparison logic.	user_id, algorithm, recommendation_list	generate_top_n(), explain_result()
Comparator	Compares user movie/rating sets to determine similarity scores.	user_a, user_b, similarity_score	compare_users(), compute_similarity()

6.1 Movie

The movie class shall contain relevant data associated with a movie in our database needed to perform algorithmic operations by other classes. Relevant data to said algorithmic operations will thus be able to be accessed by those classes. This data will include title, release year, genre, director, and ID.

6.2 User

Similar to movie, the user class shall contain relevant data associated with a user. The user class will have any pertinent information that needs to be accessed like username, and lists of ~5 favorite genres, directors, and favorite movies.

6.3 Searcher

This class works to utilize an algorithm that accesses our database of movies and performs a time-efficient search operation based on parameters the user inputs. These parameters can include any qualities a movie has, such as by director, title, etc. in ascending or descending order.

Search(String keyword, String director…)

The search algorithm will take given inputs from the end user and utilize the search operation from TMDB API to get the results in array format. This array will then be interpreted by the frontend and listed in a grid format with the movie posters as thumbnails.

6.4 Recommender

This class extends Searcher. It will modify the search algorithm to serve a slightly different purpose; that being to search for the five movies that best fit the parameters given by the recommendation algorithm that will also be a part of this class. The recommendation algorithm will utilize the Comparator class and the lists from the User class to form parameters that then are fed into the Search method to find five films to recommend

Recommend()

This method will modify the Search method to not take in user-inputted parameters, but instead data from the user class and data from the comparator class to return an array of five films that best meet the criteria, again displayed in the same grid format. Currently, the specifics of the Recommend method are still a work-in-progress and will require active testing and feedback to fine-tune.

6.5 Comparator

This class will be used for the comparison of two users' movie and rating sets. This comparison algorithm will be called on by the Recommender as part of the recommendation process.

Compare(int UserID, int ComparisonID)

This method will be called when the Recommender tries to recommend movies. It will take in two users, compare them to get some quantifiable value which; if high enough, will take into consideration movies enjoyed by the user with the ComparisonID as part of the Recommend algorithm.

7. Maintenance & Evolution Considerations

As a professionally executed course project, we prioritize clarity and testability over scale-out.

Future changes: add personality features, richer explainability, and social comparison (user-to-user). The engine can be extracted to a microservice if needed.

This design prioritizes clarity, modularity, and testability for long-term maintenance.

Future evolution plans include integrating personality-based weighting, expanding API endpoints, and implementing automated testing with CI/CD pipelines via GitHub Actions.

Each module (Frontend, Backend, Recommendation Engine) can be updated independently due to low coupling and well-defined interfaces.

8. Data Design

We combine MovieLens (for ratings/model input) with TMDB IDs for imagery/metadata. The ER model links users ↔ ratings ↔ movies; movies store a tmdb_id to join with external data.

9. Quality Attributes (Emphasis on Performance & UX)

Performance:

• Recommendation response ≤3s; DB queries indexed; Redis cache for hot paths.

• API endpoints aim for ≤500ms; batch TMDB calls when possible.

User Experience:

• Netflix/HiAnime-style dynamic UI; live refresh on rating; hover cards with reasons.

Security: strong password hashing, session/JWT; server-side validation.

Reliability: fail gracefully on API errors; show cached or partial results.

10. Glossary

• TMDB — The Movie Database, an external API for movie metadata and images.

• MovieLens — Public dataset of movie ratings used for recommendation research.

• Top-N — Returning the top N items ranked by predicted relevance (N=10).

• TTL — Time-to-live for cache entries.

11. References

• MoviePsychic SRS (Brandon Addison, Matthew Rodriguez, Jonah Schwab).

• The Movie Database (TMDB) API docs.

• MovieLens Dataset (GroupLens Research).

• Course lecture: 'Dimensions of Software Design' (event-driven, modular design).

Appendix A: Data Source Design Update

MoviePsychic now uses MovieLens as the main dataset for all recommendation logic and queries, with TMDB serving as a backup source for metadata and posters. This provides both reliability and performance.

Primary Dataset: MovieLens (Static, Local)

• Stored in PostgreSQL for quick, deterministic queries.
• Used by the recommendation engine to calculate similarities and generate Top-N results.
• Enables consistent, reproducible testing and results across environments.

Secondary Source: TMDB API (Backup)

• Used for visual assets (posters, backdrops) and to fill in missing metadata.
• Called only when a user searches for a film not found in the local dataset.
• Results cached in Redis to reduce latency and API call volume.

Rationale

This hybrid approach ensures fast, reliable recommendations for demos and grading, while still providing professional-quality movie information and visuals. The static MovieLens dataset ensures reproducible results, while TMDB enhances user experience.