CineMatch: A Hybrid LLM and Relational Approach for Personalized Movie Recommendation

Abdulrahman Alamoudi
Computer Science and Applications
Virginia Tech
aramoudi@vt.edu

Gary Young
Computer Science and Applications
Virginia Tech
garyyoung@vt.edu

Jack Lyons
Computer Science and Applications
Virginia Tech
jackl25@vt.edu

Yazeed Alharthi
Computer Science and Applications
Virginia Tech
yazeena@vt.edu

Yordanos Tessema
Computer Science and Applications
Virginia Tech
yordanost@vt.edu

***Abstract*—CineMatch is a hybrid movie recommendation system where each component handles the part of the problem it is suited for. XGBoost takes care of all candidate scoring and ranking. A Large Language Model handles three specific tasks: enriching the movie database through a PostgreSQL trigger pipeline, converting natural language queries into SQL, and writing short explanations for why each movie was recommended. PostgreSQL manages all structured storage, filtering, and aggregation. Apache AGE handles graph traversal for related-title discovery inside the same PostgreSQL cluster. A Python service owns the ML pipeline. User preference data is collected at signup through a short onboarding conversation so the system can make personalized recommendations from the very first session.**

***Index Terms*—Recommendation Systems, Large Language Models, XGBoost, Text-to-SQL, PostgreSQL, Apache AGE, Graph Databases, RAG, Trigger-Based Enrichment, Personalization**

I. Introduction

A. Motivation

Most recommendation systems work the same way: watch what users do, accumulate history, and slowly build up a taste model. This is fine when there is enough data, but it runs into two problems that do not get enough attention.

The first is cold-start. When someone creates an account, there is nothing to go on, so the system recommends whatever is popular. Some platforms add a preference quiz at signup, but those answers rarely reach the actual ranking model. The user gets generic results until watch history builds up.

The second is mood. Even users with years of history do not always want the same kind of movie they usually watch. A model trained on behavioral history can tell you what a person usually likes but not what they feel like watching right now. Mood shifts between sessions and a system that cannot account for it keeps getting things partially wrong.

CineMatch collects structured preference data at signup so recommendations are personalized from day one. Mood is treated as a real feature in the user profile and gets updated as users leave feedback.

B. Problem Statement

There are three main problems this project addresses. The first is the gap between how users express what they want and what a database and ML model can work with. A search like "something like Interstellar but less confusing" has real signal in it: sci-fi genre, preference for linear narrative, similar runtime. The system needs to extract that and turn it into a SQL query and scored features reliably. A rough similarity match is not enough.

The second is enriching the movie database without doing it by hand. The corpus has around 100,000 movies. Going through each one to assign mood labels and thematic keywords is not realistic. The system needs a pipeline that handles this automatically when a new movie is inserted.

The third is explainability. A ranked list with no context is hard to trust and hard to give feedback on. Recommendations should come with a reason.

C. Significance

The usual framing in recommendation research is that the database is just storage and the model does the real work. CineMatch questions that. The database here is not passive. It gets enriched automatically by the LLM through a trigger pipeline and handles filtering, aggregation, and graph traversal directly. The LLM is not the recommendation engine. It handles the parts involving natural language and nothing else. XGBoost does the ranking using structured features the database and onboarding flow have already prepared. This makes the system easier to inspect since any recommendation traces back to specific feature values, a specific SQL query, and a specific logged LLM output.

II. Related Work

A few platforms are worth examining to understand where CineMatch fits.

Letterboxd [1] is probably the best social film discovery tool available. Users log what they watch, write reviews, and follow each other. The community lists are genuinely useful. The problem is there is no personalized recommendation engine behind it. What you find depends entirely on who you follow. There is no ranking algorithm, no cold-start handling, and no preference model.

Rotten Tomatoes [2] aggregates critic and audience scores and is useful as a quality signal, but it does not try to model individual taste at all. Every user sees the same scores for every film.

IMDb [3] has the most complete structured movie data that exists, with cast, crew, and ratings at a depth nothing else matches. Its "More Like This" feature uses shallow genre overlap with no underlying preference model. It is a reference database, not a recommendation system.

TMDB [4] has a well-documented REST API that updates daily and covers a wide range of international cinema. It is the main external data source for CineMatch.

None of these platforms solve cold-start through onboarding that connects to a ranking model. None run LLM-based semantic enrichment as a live part of the data pipeline. None use graph traversal to find thematic connections between films. CineMatch puts all three together.

III. Proposed Approach

The design comes down to three ideas: collect real preference data at signup so the system works from session one; automatically enrich the movie database using an LLM trigger pipeline; and use XGBoost for all ranking so the LLM never touches the scoring path.

A. Interactive Onboarding

At signup, a short questionnaire builds a structured preference profile before the user has watched anything. It collects genre preferences with free-text notes, preferred runtime, decade range, language openness, and content tolerance stored as numeric values. The most useful part is the mood anchor: the user names three films they liked and three they did not. The LLM reads all six and extracts attributes like pacing, emotional tone, and narrative complexity, stored as a JSONB column. People are much better at naming films they have opinions about than describing those opinions in the abstract. The extracted attributes feed into XGBoost from the very first request.

B. LLM-Triggered Database Enrichment

Every time a new row gets inserted into the `movies` table, a PostgreSQL trigger fires and queues an enrichment job. The job sends the movie's title, genre list, and plot summary to Gemini Flash, which returns semantic tags covering mood, pacing, narrative complexity, and thematic keywords, each with a confidence score between 0 and 1. Those tags get written back into `movie_tags` with `source = 'llm'`. There is no batch job to schedule. Enrichment happens as a side effect of insertion.

The same pipeline builds the graph in Apache AGE. After tagging, the LLM scores thematic similarity between the new movie and existing movies that share genre or keyword overlap. An edge is created only when the similarity score clears a set threshold, which keeps the graph from just connecting everything in the same genre. Within minutes of being inserted, a movie is tagged and graph-connected.

C. Ranking, Feedback, and Architecture

When a user opens the home feed, the Go API loads their profile from PostgreSQL and calls the Python scoring service. That service filters out movies that fail hard constraints then runs XGBoost over the rest. The feature vector includes genre weights, mood profile attributes, the movie's assigned tags, average rating, runtime, release year, and cast-and-crew overlap with the user's stated preferences. The top 20 come back to the Go API. For the top 5, Gemini Flash generates a one-sentence explanation produced after ranking with no effect on the order. Feedback closes the loop: star ratings and not-interested signals update weights via exponential moving average, rejected titles are excluded permanently, and the model retrains weekly on accumulated data.

The system runs as four Docker containers via `docker-compose`: a **Go API server** for routing, JWT auth, and coordination; a **Python analytics service** (FastAPI) exposing `/score`, `/train`, `/enrich`, and `/rag`; **PostgreSQL 16 with Apache AGE and PostGIS 3** as the single source of truth; and a **React and TypeScript frontend** that only talks to the Go API.

IV. System Features

Personalized Home Feed. Twenty candidates ranked by XGBoost with one-sentence LLM explanations for the top five, generated after ranking and cached per session.

Conversational Movie Search. Users type queries in plain English. The Go API sends the query with the relevant schema to Gemini Flash, which returns a parameterized SQL `SELECT`. The query is checked against a whitelist before running: read-only, `LIMIT` required, no unsafe keywords. The generated SQL can be shown to the user as a transparency option. Filters include genre, MPAA rating, score range, year, runtime, language, director, actor, and mood tags.

Mood-Based Quick Pick. A mood selector offers six options: Feel-Good, Tense, Thought-Provoking, Funny, Romantic, and Epic. Picking one filters `movie_tags` and re-ranks the results against the user's XGBoost score. The mood tags were written by the trigger pipeline when each movie was inserted.

Related Movies via Graph. Each movie detail page has a "You might also like" section from Apache AGE traversal. Edges represent shared director, shared lead actor, shared genre, and LLM-scored thematic similarity. Going two or three hops out finds movies with multiple indirect connections,

a stronger signal than a flat genre match. AGE runs inside PostgreSQL so no separate infrastructure is needed.

City-Level Activity Map. A live map shows the most-watched movie per city and state. A background Go service generates simulated activity written to `activity_events` with a `city_state` field. A materialized view refreshes every 60 seconds over a rolling 10-minute window. The frontend renders this with ECharts using city and state GeoJSON.

Analytics Dashboards. ECharts charts backed by pre-computed PostgreSQL views cover genre trends over time, rating distribution by decade, top directors by recommendation frequency, a user activity heatmap, cold-start versus returning-user accuracy, mood tag popularity by rating, and language diversity. Free-text analytics questions get turned into SQL by the LLM and rendered as a chart.

Explicit Feedback Loop. Star ratings and not-interested signals update genre and mood weights through exponential moving average. Rejected titles are permanently removed from the candidate set between retraining cycles.

V. Datasets

The corpus is built from four sources merged on TMDB ID. The **TMDB Movies Dataset** [6] is the main source with about 100,000 titles from a collection of over one million daily-updated entries. The **IMDb Non-Commercial Datasets** [7] add crew data and audience ratings across 1.6 million titles. The **Netflix Reviews Dataset** [8] seeds the activity simulation and provides sentiment signals for feature engineering. The **Kaggle Movies Dataset** [5] verifies the ingestion pipeline is working correctly. Poster images and daily updates come from the TMDB [4] and OMDB [9] APIs.

VI. Checkpoint I

In this project checkpoint, we revised and clarified the features of our project based on feedback, including the schema, tech stack, and LLM scope. We have also redefined and reallocated group member roles in accordance with our expected project timeline to ensure an even distribution of work across our remaining time. With these tasks completed, we have finalized the design of our application and the schema of our database, and have begun building our backend.

LLM Guardrails. One of our concerns when enabling an LLM to dynamically construct SQL queries is the possibility to perform destructive actions on our database. As indicated by our conversational movie search feature, we intend for the LLM to parse plain text into a parameterized SELECT statement, and as such we wish to whitelist certain behavior that could be possible from user input. To ensure the integrity of our database, we are imposing limitations on the types of SQL statements the LLM is able to perform by checking the queries it constructs for specific keywords. We intend to implement a filter that checks all LLM generated queries and blocks potentially harmful ones from being executed on our database. This filter will be implemented in our backend to integrate with the LLM such that we can provide sample queries during testing. All statements capable of altering the

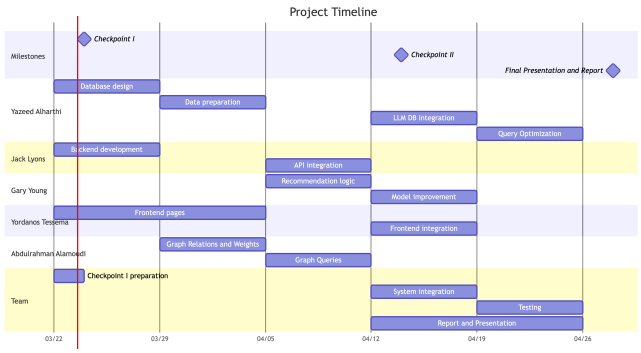

Fig. 1. Project Timeline

rows in our movie dataset will be forbidden by our filter, this includes INSERT, UPDATE, DELETE, and TRUNCATE. Additionally, we want to prevent any statements that could alter our database schema, including CREATE, ALTER, and DROP. The primary goal is to only allow for queries formatted as a SELECT statement and return an error for all others.

Users and Spatial Relations. The design of our database schema includes data for user locations as another means to provide accurate movie recommendations. As part of the initial user questionnaire intended to collect data for recommendations, the user can choose to provide location data. If provided, users will be able to toggle recommendations based on movie data from users in their area, improving accuracy by limiting preferences to what is popular in the area. This feature is intended to be simple to implement, using our same recommendation model on a subset of the full user dataset based on location data collected from the user during our initial questionnaire. We also intend for this feature to interact with our city-level activity map such that the user can visualize where their recommendations are derived.

Project Roles and Timeline. Since our project proposal, we have redefined each member's role in the project and have constructed a timeline of when each member should expect to complete each task they're assigned (See Fig. 1). This timeline is expected to be a guide, so the exact dates and members assigned to a specific task are subject to change if the amount of time and resources required is different from what we're projecting. We expect all members to contribute to system testing and integration to ensure the functionality of all components. All members will also contribute to the final report and presentation so we can accurately discuss all parts of the system.

Current Tasks and Milestones We have reached several goals at this stage of our project, and our project timeline indicates the tasks that are currently in progress. We completed the design for our database schema for both our users and our movie datasets, and we have a system to pull the relevant datasets from Kaggle. We have set up the required tools for our backend, including running Postgres in Docker. Our current tasks are developing our python scoring service and

questionnaire data insertion in the backend and developing the dashboard for our React frontend. Our reports on individual team member contributions reflect the changes in task allocation and our timeline.

Yazeed Alharthi

Yazeed is tasked with database design, data preparation, LLM DB integration, and query optimization. These tasks focus on determining our database schema and managing how the LLM interacts with it. For this checkpoint, he has set up Postgres, Docker, and Go API for our project.

Jack Lyons

Jack is tasked with backend development and API integration, focusing on the underlying logic and user features of the application. For this checkpoint, Jack primarily worked on expanding and formatting the report for this checkpoint to reflect the new changes to project goals, work allocation, and milestones. He is currently developing the questionnaire to collect initial user data and create entries in our user database.

Gary Young

Gary is tasked with recommendation logic and model improvement, focusing on designing and improving our core recommendation model. Thus, his plan is to consider parameters on recommendations for others. He also plans on supporting the team if there are problems. To simulate real user behavior, he plans to create dozens of virtual users with different locations and movie-watching activity within a limited area. The goal is to provide a sense of a more realistic user experience.

Yordanos Tessema

Yordanos' responsibility is to implement the frontend pages and integration, which entails a great deal of UX/UI work related to our application and our data. She works on the full visual stack of CineMatch, including the implementation of our React/TS frontend, such as our mood selector, our conversational search input, our movie pages, and our explicit feedback features.

Regarding data visualization, she integrates ECharts to render our analytics dashboards based on pre-computed PostgreSQL views, such as our genre trends, rating distributions, and our city-level activity map. She also works on ensuring our frontend remains responsive, as well as coordinating integration testing with our backend team.

Abdulrahman Alamoudi

Abdulrahman is responsible for designing the graph relationships, weights, and queries. His work focuses on implementing a graph structure to map movie interests and support our recommendation algorithm. He will prepare the data as a graph and build queries based on users' watched movies to find similar movies, as well as identify what other people in the same area are watching using PostGIS and graph technology.

Database Schema Overview

The schema splits into two groups. On the movie side, `movies` holds core metadata including title, year, runtime, language, ratings, and plot summary. Genres link through `movie_genres` and semantic tags from the LLM pipeline live in `movie_tags` with a float confidence score and a source field. Cast and crew go into `movie_crew` with a role column. On the user side, `user_preferences` stores genre weights and tolerance thresholds as JSONB, and `user_mood_profile` holds the onboarding-extracted mood attributes. Watch history, explicit feedback, and every recommendation made by the system each have their own table, forming the training data for XGBoost and the evaluation record for tracking model improvement over time.

Task Assignment

TABLE I
TASK ASSIGNMENT BY MEMBER

Task	Member
PostgreSQL schema design, migrations, and query optimization	Yazeed Alharthi
Data preparation and corpus ingestion pipeline	Yazeed Alharthi
LLM–database integration and query interaction	Yazeed Alharthi
Backend development and API integration	Jack Lyons
User questionnaire flow and backend data insertion	Jack Lyons
Recommendation logic and model improvement	Gary Young
Virtual user simulation for location-based movie activity	Gary Young
React/TypeScript frontend pages and UI integration	Yordanos Tessema
ECharts dashboards and city-level activity map integration	Yordanos Tessema
Responsive frontend design and UX/UI implementation	Yordanos Tessema
Apache AGE graph relationships, weights, and traversal queries	Abdulrahman Alamoudi
Spatial recommendation logic and local activity analysis using PostGIS	Abdulrahman Alamoudi
Integration, testing, final report, and presentation	All members

VII. CHECKPOINT II

A. Progress Report

Abdulrahman Alamoudi — Database Seeding, Graph Layer, Deployment

Data Pipeline and Database Seeding: We decided to go with the TMDB dataset because it covers much of the essential information needed to display movie data, including the poster URL, and it also references each movie with an IMDb ID for extra information such as the IMDb rating. To build this, I needed to merge both datasets. Instead of using regular data processing tools such as Python and pandas, I decided to use SQL for cleaning and ingesting the data, and that turned out to be the right decision because it made the process much

more performant. By mounting the TSV and CSV files into the Postgres Docker container, I was able to use the COPY statement to load the raw data into staging tables first, then manipulate those staging tables and join them to form the actual tables.

I first loaded the raw TMDB and IMDb files into staging tables inside Postgres. Then I cleaned and joined the staged data to insert only valid movies into the movies table. After that, I populated the related information for those movies, including directors, writers, genres, and the relationship tables movie_crew and movie_genres. In this way, the raw source files were transformed into clean, structured, and application-ready tables.

The final result contains millions of records, and some tables are hundreds of megabytes in size, so the ingestion process still takes a few minutes. Because of that, I created a database dump for my colleagues so they can restore the fully prepared database in seconds instead of rebuilding it every time. This is especially useful for testing, since if anything goes wrong, they can quickly restore the data and continue working.

***Graph Building**:* To build the graph layer, I used Apache AGE on top of PostgreSQL so I could keep the relational tables and also create graph nodes and edges from the same database. Based on the graph plan, I modeled four main node types: `User`, `Movie`, `Genre`, and `Person`. Then I created the graph relationships such as `WATCHED`, `RATED`, `IN_GENRE`, `ACTED_IN`, and `DIRECTED`. This design allows us to support recommendation queries such as popular unseen movies, same-genre recommendations, similar-user recommendations, and "because you watched" style results.

Since the number of original movies was more than 1.4 million, I reduced it to more than 400K movies before graph construction. However, this number is still very large for building a graph with many relationships, as shown in Figure 2. Because of that, the graph build process took more than one day, and the graph shaping process is still ongoing. To solve this issue, we are considering reducing the dataset further, for example by keeping only movies from the last five years, only movies with rating 7 or above, or only movies from specific genres.

As a next step, I plan to add a faker graph to generate synthetic users, watch history, and rating relationships. This will help us test the recommendation queries even when the real user-related tables are still empty or limited. It will also make it easier to evaluate the graph logic, measure query performance, and compare different graph approaches. In addition, if Apache AGE continues to be slow for this scale, we may replace it with a more performant graph solution, such as a database-based option like Neo4j or a library-based solution like NetworkX.

***Setup Deployment Environment**:* I used Oracle Cloud Always Free to provision a virtual machine with 4 OCPUs, 24 GB of RAM, and 200 GB of storage, which can be used to host both the testing and production deployments. This gives us a practical cloud environment for running the application without extra infrastructure cost. To expose the deployment to the public internet, we can use a Cloudflare Tunnel, which provides secure public access without needing to open inbound ports directly on the VM.

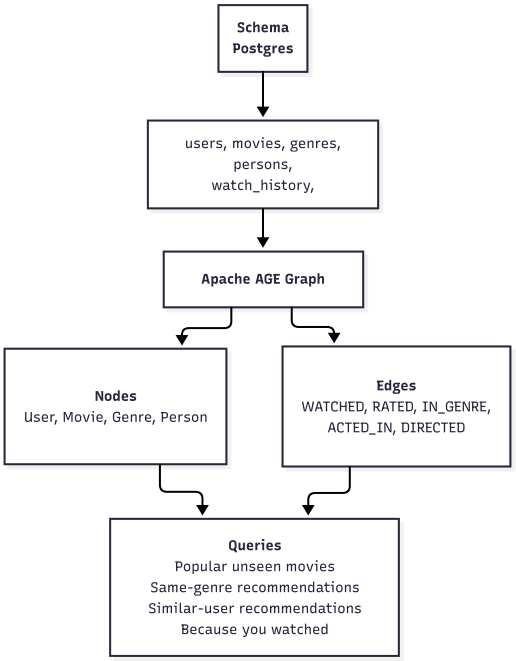

Fig. 2. Planned graph model and query structure for the recommendation system.

Jack Lyons - On-boarding and User API Endpoints

For this checkpoint, I focused on developing endpoints and features for users in the backend. The structure of the initial questionnaire and user database queries follow the specifications set by our proposal and the schema of our database.

User Questionnaire

Upon creating an account, the user is prompted to provide basic preference information. Firstly, the user is asked to provide 3 movies that they have watched and enjoyed. The API will construct a query in SQL that selects by movie title and fetches relevant metadata to display the movie and other information for the user to confirm. I'm planning to develop an endpoint to insert this data into our graph layer such that a `WATCHED` relationship is created. Other information including genre preference is compiled to create a user profile entry.

Backend User API

Once the questionnaire is completed, the API will insert the user profile into our user database through a user endpoint. Access to the user database is parameterized by a unique `user-id` field. User accounts are queried following successful login. I'm working to develop secure authentication for users to access their accounts.

Gary Young

So far, Gary has added PostGIS support on GitHub to allow geospatial indexing and distance-based queries, added a user locations table, some indexes, and added location to the user feedback table. For simplicity, there are plans to store latitude and longitude with the city and state. Latitude and longitude would allow features like clustering, distance, and radius search to exist if set up correctly.

More specifically, clustering based on geographic regions using PostGIS would allow the system to identify localized preference trends and improve cold-start recommendations. This spatial layer will further enhance the city-level activity map by enabling dynamic heatmaps and radius-based filtering of trending content. Synthetic users plan on being generated with randomized geospatial distributions to simulate realistic population density patterns and evaluate spatial recommendation stability under varying user density conditions.

Yazeed Alharthi — Database, Data Pipeline, and API Layer

Database Schema Design: The schema is organized into three layers. Three custom PostgreSQL enums enforce type safety across all tables: `tag_source` (`'llm'` or `'manual'`), `crew_role` (`'director'`, `'actor'`, `'writer'`, `'producer'`), and `mood_type` (the six supported moods).

a) Movie Layer.: `movies` is keyed by `tmdb_id` and carries all core metadata (title, release date, runtime, language, ratings, poster path, etc.) plus an `enriched` boolean that tracks LLM pipeline status and an `imdb_id` used for the IMDb merge during seeding. Genres use a normalized junction table: `genres` holds canonical names, `movie_genres` maps movies to genres with a composite primary key and a reverse index on `(genre_id, movie_id)` for bidirectional lookups. Semantic tags live in `movie_tags` as a key-value design (`tag_key`, `tag_value`, float `confidence`, `source` enum) indexed on `(movie_id, tag_key)`. This was chosen over wide columns because the tag vocabulary is expected to grow as the LLM prompt evolves. Cast and crew are stored across `persons` (keyed by IMDb `nconst`) and `movie_crew`, which links movies to persons with a role enum, optional character name, and billing order. A unique constraint on `(movie_id, person_id, role)` prevents duplicates, and a composite index on `(movie_id, role, ordering)` supports sorted crew retrieval.

b) User Layer.: `users` is keyed by UUID with unique username and email constraints. `user_preferences` stores genre weights and content tolerance as JSONB alongside scalar runtime and decade range fields. `user_mood_profile` holds the onboarding output: integer arrays of liked/disliked movie IDs and a JSONB `attributes` column with LLM-extracted mood dimensions. All three cascade on user deletion. `refresh_tokens` stores hashed JWT refresh tokens with expiration, revocation, and client metadata.

c) Behavioral and Infrastructure Layer.: `watch_history`, `user_feedback` (unique per user-movie pair), and `recommendations` (with score, explanation, and session UUID) form the training and evaluation corpus for XGBoost. `activity_events` links users and movies to a `city_state` field for the live activity map, indexed on timestamp and city for the rolling-window materialized view. A trigger function `queue_enrichment()` fires on every movie insert and writes to `enrichment_queue`, making LLM tagging a side effect of insertion with no external scheduler. The `pg_trgm` extension is enabled at schema creation for trigram GIN indexes on title fields.

Data Ingestion Pipeline: The corpus is built from four external sources merged on TMDB ID and IMDb ID. A Go-based downloader (`internal/dd`) defines each source as a struct with name, URL, format, auth method, and a zip flag. Kaggle sources (TMDB movies, the Kaggle movies cross-check set, and Netflix reviews) use API key authentication. IMDb sources (title basics, title ratings, title crew, title principals, and name basics) are public gzipped TSV files pulled directly from `datasets.imdbws.com`. On download, each archive is extracted into a per-source directory under `/data/raw/`.

Seeding runs as a single SQL transaction inside a `docker exec` call to the database container. The script creates temporary staging tables (all columns as text for flexible casting), bulk-loads each file via `\copy`, then performs the merge. The principals TSV required special handling: it is loaded as raw single-column lines and then split on tab characters in SQL because its character field contains embedded quotes that break standard CSV parsing.

Movies are inserted from staging with an inner join to `staging_ratings` so that only titles with a valid IMDb rating enter the corpus. The adult-content filter runs at insert time: rows where the `adult` column is true or where the title or overview matches a regex pattern for explicit keywords are excluded before the `INSERT` executes. This prevents explicit content from reaching the database rather than cleaning it up after the fact. Genre mapping and crew inserts reference only movies that passed the filter, preventing foreign-key violations from orphaned references.

Persons are inserted from `staging_names` with a CTE that unions all person IDs referenced by directors, writers, and principal entries, so only persons actually needed by the crew data are loaded. Directors and writers come from IMDb `title.crew.tsv`, which stores comma-separated `nconst` lists per title. Actors come from `title.principals.tsv` filtered to `category IN ('actor', 'actress', 'self')`. Producers are loaded from the same principals file filtered to `category = 'producer'`. After actor insertion, an update pass enriches existing director and writer rows with job description text from the principals file where available.

Movie API Endpoints and Repository Layer: The Go API exposes five movie endpoints through chi router:

- GET /api/movies — paginated movie list with optional text search, controlled by limit, offset, and q query parameters.
- GET /api/movies/search — dedicated search endpoint using the same underlying query with text matching.
- GET /api/movies/:id — single movie detail by TMDB ID, returning metadata with genres and mood tags.
- GET /api/movies/:id/crew — full crew list for a movie, ordered by role priority (director, writer, producer, actor) then billing position.
- GET /api/movies/:id/related — related movies ranked by shared genre count and popularity.

The repository layer (MovieRepository interface) defines four methods: ListMovies, GetMovieByID, GetMovieCrewByID, and GetRelatedMovies. All queries are written as raw SQL with parameterized inputs. The list query uses a CTE pattern: a base_movies CTE selects only the IDs that match the filter and pagination constraints, then the main query joins the full detail (genres, mood tags, director name, cast names) only for those rows. Each associated dimension is fetched through a LEFT JOIN LATERAL subquery that aggregates into a PostgreSQL array, keeping the result as a single flat row per movie. This avoids the N+1 problem entirely.

The handler layer validates inputs, enforces parameter bounds (limit capped at 200, offset at 1,000,000), and wraps all responses in a consistent envelope: {success, data, error}.

Query Performance Optimization: The initial list query was completing in approximately 68 seconds per request. The query was not a simple SELECT with LIMIT; it joined genres, mood tags, director, and cast for every returned row. With 100,000+ movies and no indexes on the join columns, the planner was performing sequential scans across multiple tables for each lateral join.

Two changes brought the response time under one second. First, the query was restructured so that the CTE narrows down to the target movie IDs using only the movies table with a simple sort and limit, then the expensive lateral joins execute only for those selected rows. Second, a dedicated index migration (schema-02-indexes.sql) was added to cover the critical access patterns: a composite index on movie_crew(movie_id, role, ordering) for crew lookups, a composite on movie_tags(movie_id, tag_key) for tag retrieval, a reverse index on movie_genres(genre_id, movie_id) for genre-based filtering, and trigram GIN indexes on lower(title) and lower(original_title) for fuzzy text search. The index migration uses CREATE INDEX IF NOT EXISTS so it can be applied idempotently via the migrator CLI.

Content Moderation: An audit of the movie corpus found 43,193 rows with explicit content, all in the title and overview text columns. No matches were found in tag or genre tables. The initial cleanup used a regex-based delete targeting ten keyword patterns. This query is preserved in migration/clean.sql for re-use.

More importantly, the same filter logic was added to the seed script's INSERT statement so that explicit content is excluded at ingestion time. The seed query checks both the TMDB adult flag and the keyword regex before a row reaches the movies table. Genre and crew inserts join back to movies rather than staging, which ensures no foreign key references point to filtered-out rows.

The regex approach has known limitations: it can produce false positives on legitimate titles containing matched keywords, and it misses explicit content that uses different wording. For production use, a classifier-based approach would be more precise, but the keyword filter is sufficient for the current corpus and runs at zero runtime cost since it executes only during seeding.

Cast Data Fix: Movie detail pages initially showed directors but actor lists were empty. The root cause was that the seeder only imported crew data from IMDb title.crew.tsv, which contains only director and writer nconst lists. Actor data lives in IMDb title.principals.tsv, which was not included in the downloader sources or the seed flow.

The fix added imdb-title-principals as a new source in the downloader, created a staging_principals table in the seed script, loaded the principals file with a raw-line import and tab-split approach, expanded the persons insert CTE to union principal person IDs alongside director and writer IDs, and inserted actor and producer rows into movie_crew with the appropriate role enum values. Character names are extracted from the principals characters column when available. The database can now be fully rebuilt from scratch with complete cast and crew data.

Yordanos Tessema - Frontend Development

Project Setup and Architecture

Configured React + TypeScript + Vite frontend project with proper folder structure according to the teams agreement. Installed react-router-dom and implemented client-side routing. Created proper ui/src/ folder structure with sub-folders for api/, components/, pages/, routes/, and types/.

Type System (types/movie.ts)

Created TypeScript interfaces used throughout the whole application like Movie, User, UserOnboardingData, and AnalyticsData. Mapped enum values to PostgreSQL schema (mood_type, tag_source, crew_role), including the MOODS constant with both display labels and database values, and the GENRES list matching the team's genres table.

API Layer (api/)

Implemented an API layer with mock and real files for a seamless transition from mock to real API implementation:

mockApi.ts — Complete mock API implementation with realistic delays, seed movies with meta information, and

analytics data. Allows the frontend app to function completely independently from the backend.

realApi.ts — Real API client implementation with JWT Bearer authentication, complete API paths mapped according to Go chi router, setAccessToken and clearAccessToken functions, and a 401 handler hook for token refresh.

mappers.ts — Maps the raw PostgreSQL/API field names (vote_avg, runtime_min, original_lang, tmdb_id) to the Movie object on the frontend, including building the TMDB image URL and mapping the mood enums to their display-friendly names.

Changing from mock to live data on any page only needs a one-line edit.

Components

Developed three reusable components that were used throughout all pages:

Navbar.tsx — Navigation bar with the CineMatch logo, links to pages with an active class when visited, inline search form for searching movies that redirects users to the search page, and a button for showing the user's initials as an avatar.

MovieCard.tsx — Movie card with its poster image, hover effect with genre tags and movie rating, a heart button for marking favorites with toggle state, a fallback image if no poster exists, and optional text explanation for the top five AI recommendations.

MoodSelector.tsx — Six moods with their corresponding emojis, each represented as horizontal pills, with toggle state to enable or disable individual moods.

Pages

OnboardingPage.tsx - This is an onboarding process consisting of four steps designed to gather everything necessary for tailoring recommendations during the initial login: creating an account including password verification, choosing preferred genres, providing anchors that represent films that the user likes and dislikes (the LLM pipeline can then identify the user's tastes from these anchors), and setting a default mood.

HomePage.tsx - This is the home page that features a cinematic hero section with a headline and statistics, the mood selector bar that filters the recommendation grid dynamically, the personalized recommendation grid that ranks the top 20 films using XGBoost and provides AI reasoning for the top 5, the trending this week horizontal scroller, and the top rated all time horizontal scroller.

SearchPage.tsx - This is the two-mode search page that features an AI natural language query input above (this gets sent to the Go API which invokes the Gemini Flash service and returns the parameterized SQL statement – this is a transparency feature). It also features a keyword filter bar beneath it, allowing users to select genres, moods, and decades.

MoviePage.tsx — Full movie detail page with a cinematic hero section using the movie poster as a blurred background, complete metadata display (rating, year, runtime, language, MPAA rating, mood tags), star rating feedback that updates the recommendation model, a not-interested exclusion button, cast and director cards, and a "You Might Also Like" section powered by Apache AGE graph traversal.

DashboardPage.tsx — Analytics dashboard with four stat cards, three bar charts (genre trends, mood tag popularity, average rating by decade), a city-level activity table showing the most-watched film per city backed by the PostgreSQL materialized view, and a natural language analytics query box that converts plain English questions to SQL via the LLM.

Routing (routes/AppRouter.tsx)

Configured client-side routing with routes for /, /search, /movie/:id, and /dashboard, with a catch-all redirect to home. The router sits behind the onboarding gate in App.tsx so unauthenticated users always hit the onboarding flow first.

Styling (App.css)

Designed and implemented a full custom dark cinema aesthetic with a gold accent colour palette (#e8c468), Playfair Display for display headings paired with DM Sans for body text, CSS custom properties for the full design token system, responsive grid layouts, animated hover states on movie cards, fade-in animations with staggered delays, a custom scrollbar, and mobile breakpoints for the navbar and movie detail layout.

APPENDIX

REFERENCES

[1] Letterboxd. https://letterboxd.com/
[2] Rotten Tomatoes. https://www.rottentomatoes.com/
[3] IMDb. https://www.imdb.com/
[4] The Movie Database API. https://developer.themoviedb.org/docs/getting-started
[5] R. Banik, "The Movies Dataset," Kaggle. https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset
[6] A. Saniczka, "TMDB Movies Dataset 2023," Kaggle. https://www.kaggle.com/datasets/asaniczka/tmdb-movies-dataset-2023-930k-movies
[7] IMDb Non-Commercial Datasets. https://datasets.imdbws.com/
[8] A. Kumarak, "Netflix Reviews," Kaggle. https://www.kaggle.com/datasets/ashishkumarak/netflix-reviews-playstore-daily-updated
[9] OMDB API. https://www.omdbapi.com/