AI Recommendation Engine
Fall 2025 COMS 4020c Senior Design

Client
IdeaNova Technologies Inc.

Instructor
Dr. Simanta Mitra

Teaching Assistants
Mohammed Shaiqur Rahman
Scott Song

Team Members
Honey Reddy Nagireddy
Ssai Sanjanna Ganji
Preethi Reddy Nagireddy
Siddhartha Gudipudi

Table of Contents

Client Letter ..... Error! Bookmark not defined.

Chapter 1 – Requirements ***3***

1.1 Project Overview **3**

1.2 Functional Requirements **3**

1.3 Non-Functional Requirements **4**

Chapter 2 – Design ***4***

2.1 System Architecture Overview **4**

2.2 Backend Design (Go) **5**

2.2.1 LLM Recommendation Engine 5

2.2.2 Scoring and Ranking Framework 6

2.2.3 Vector Recommendation Engine 8

2.3 Frontend Design (React) **9**

2.4 API Design **10**

Chapter 4 – Results Achieved ***13***

4.1 Model Evaluation Results **13**

4.2 System Outcomes **13**

Chapter 5 – Conclusion ***14***

Appendix – Demo Slides ***14***

Chapter 1 – Requirements

1.1 Project Overview

The **AI Recommendation Engine** is a full-stack, production-oriented recommendation platform developed for **IdeaNova Technologies**. The system generates **personalized movie and shopping recommendations** by combining two complementary approaches:

1. **LLM-based reasoning recommendations**, which provide rich personalization and human-like explanations
2. **Vector-based similarity recommendations**, which ensure deterministic, low-latency, and offline-capable operation

Rather than relying on a single model or technique, the system evaluates **multiple recommendation strategies simultaneously** and selects the best output using a **structured, explainable scoring framework**. This allows the platform to balance recommendation quality, latency, reliability, and cost.

The system supports:

- Real-time user recommendations
- Batch processing for large-scale evaluation
- Model comparison and leaderboard analytics
- Full auditability of prompts, responses, and metrics
- Graceful degradation when external AI services are unavailable

The design explicitly targets **real-world deployment**, emphasizing robustness, scalability, and explainability.

1.2 Functional Requirements

The system satisfies the following functional requirements:

- Generate **personalized movie recommendations** using user demographics and watch history
- Generate **personalized shopping recommendations** using demographics and cart or wishlist data
- Execute **multiple LLM models in parallel** for fair and efficient comparison
- Rank models using a **multi-dimensional, explainable scoring framework**
- Provide **deterministic vector-based recommendations** as a fallback mechanism
- Support **batch processing** of multiple user profiles
- Export complete recommendation results, metrics, and rankings in **CSV format**
- Maintain full **audit logs** of runs, prompts, scores, and decisions
- Allow **real-time cancellation** of long-running recommendation jobs

- Present recommendations, metrics, and comparisons through an **interactive frontend UI**

1.3 Non-Functional Requirements

- **Performance:** Target response time under 3 minutes for 90% of LLM runs
- **Reliability:** System must remain operational even if external APIs fail
- **Scalability:** Stateless backend supporting horizontal scaling
- **Explainability:** All recommendations must be traceable to scores, metrics, and reasoning
- **Maintainability:** Modular backend architecture with clearly separated engines

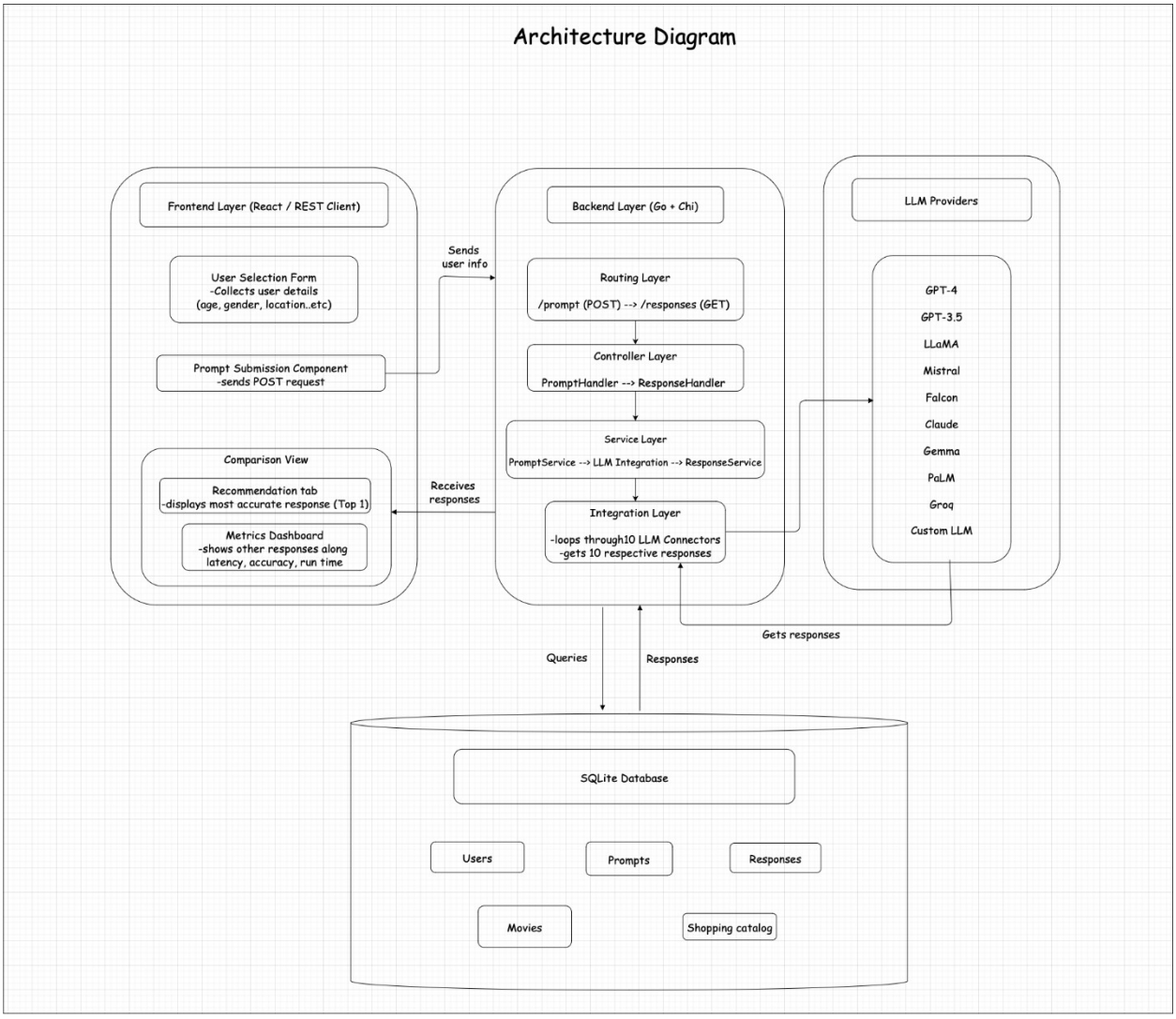
Chapter 2 – Design

2.1 System Architecture Overview

The AI Recommendation Engine follows a **layered architecture** to ensure separation of concerns and system robustness:

- **Client Layer:** React frontend for user interaction and visualization
- **API Layer:** Go backend exposing RESTful endpoints
- **LLM Engine Layer:** Parallel LLM execution with validation, scoring, and adjudication
- **Vector Engine Layer:** Deterministic embedding-based recommendation engine
- **Data Layer:** SQLite database for catalogs, run logs, and metrics
- **Export & Analysis Layer:** CSV generation and analytics support

This design allows each recommendation strategy to evolve independently while sharing common infrastructure for logging, evaluation, and user interaction.



2.2 Backend Design (Go)

2.2.1 LLM Recommendation Engine

The LLM recommendation engine is the most complex component of the system and is implemented as an **eight-phase execution pipeline**:

1. **Input Validation and Canonicalization**
   User input is normalized into a deterministic format to prevent redundant executions and enable caching.
2. **Model Selection and Allow-List Enforcement**
   Only explicitly approved models may be executed, ensuring security and predictability.
3. **Candidate Pre-Filtering**
   Large catalogs are reduced to an optimized candidate set based on age appropriateness, watch history, genre similarity, and metadata quality.

4. **Schema-Driven Prompt Construction**
   Prompts are constructed with strict JSON schema enforcement to ensure machine-parsable outputs.
5. **Concurrent Model Execution**
   All selected models are executed in parallel using goroutines with per-model and global timeouts.
6. **Multi-Dimensional Scoring**
   Each model's output is evaluated across quality, performance, and efficiency metrics.
7. **Optional GPT-5 Adjudication**
   When enabled, GPT-5 performs semantic evaluation and final model selection.
8. **Result Persistence and Response Generation**
   All results, scores, and metadata are stored for auditing and returned to the client.

This design reduces total execution time from sequential multi-model runs while ensuring fair comparison.

2.2.2 Scoring and Ranking Framework

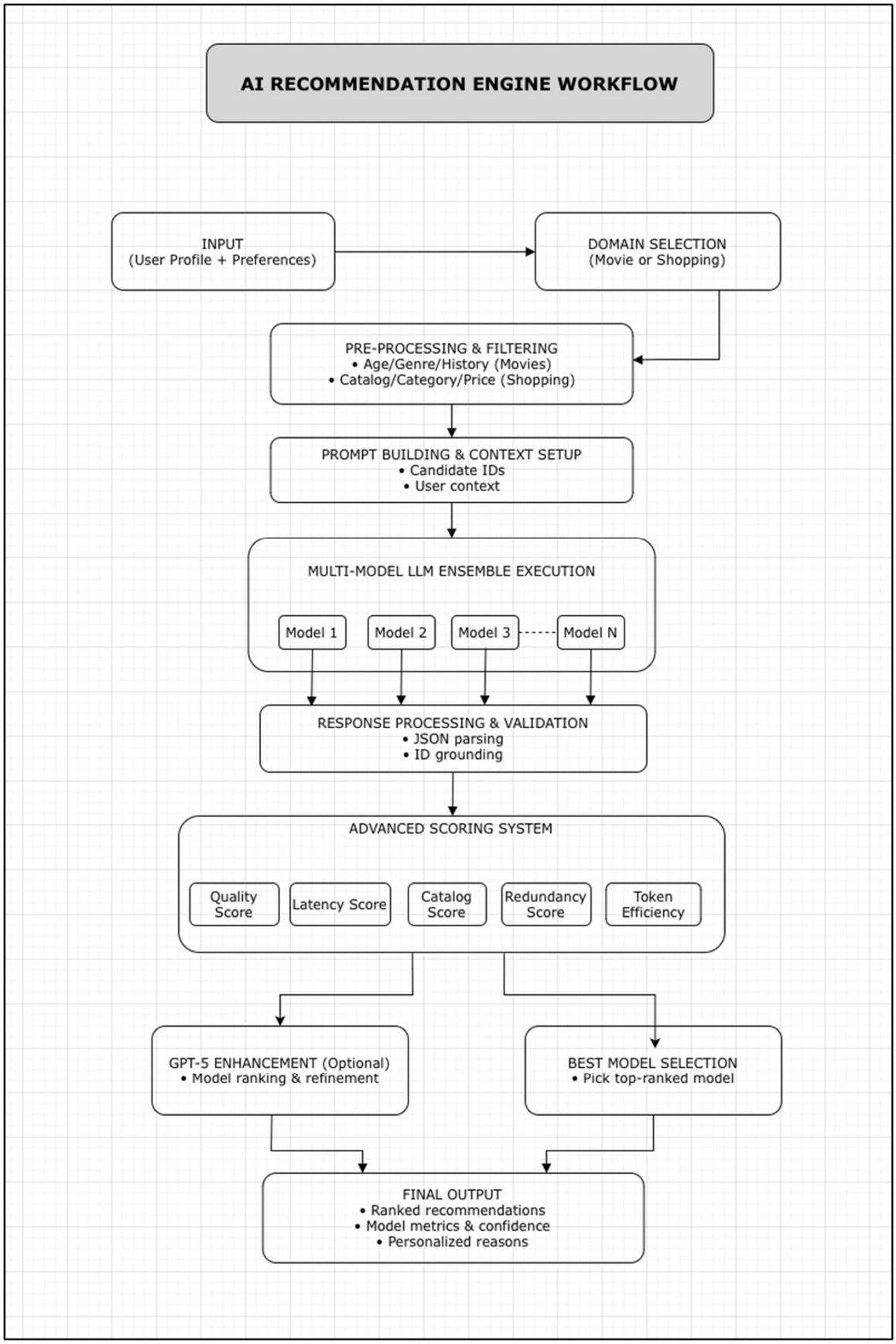
Models are ranked using a weighted scoring system:

- **Quality Score (35%)** – relevance, personalization, diversity, novelty
- **Catalog Validity (10%)** – ensures recommendations exist in the catalog
- **Latency Performance (35%)** – relative execution speed
- **Token Efficiency (10%)** – cost-aware output length optimization
- **Redundancy Control (10%)** – penalizes duplicates and repetition

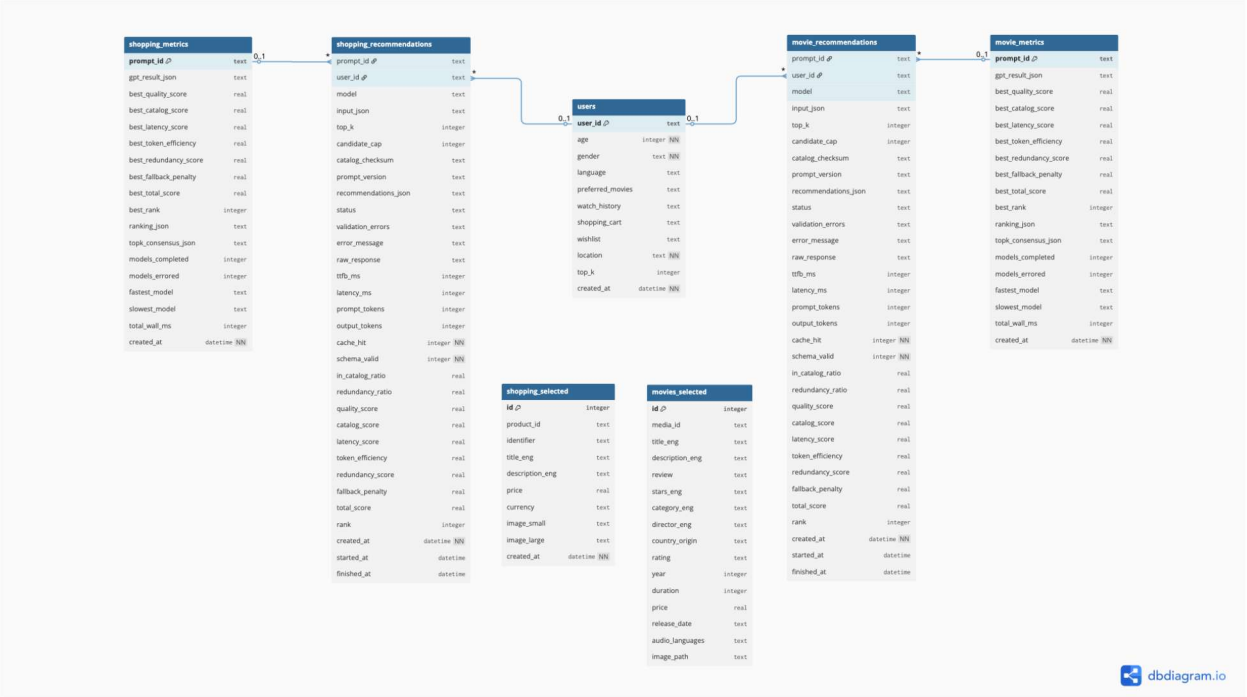
Quality scoring is further decomposed into:

- Genre similarity using cosine similarity
- Reasoning specificity penalties for generic explanations
- Diversity and novelty calculations

This framework ensures that **quality dominates selection** while maintaining performance guarantees.


AI RECOMMENDATION ENGINE WORKFLOW
INPUT
(User Profile + Preferences)
DOMAIN SELECTION
(Movie or Shopping)
PRE-PROCESSING & FILTERING
• Age/Genre/History (Movies)
• Catalog/Category/Price (Shopping)
PROMPT BUILDING & CONTEXT SETUP
• Candidate IDs
• User context
MULTI-MODEL LLM ENSEMBLE EXECUTION
Model 1
Model 2
Model 3
Model N
RESPONSE PROCESSING & VALIDATION
• JSON parsing
• ID grounding
ADVANCED SCORING SYSTEM
Quality Score
Latency Score
Catalog Score
Redundancy Score
Token Efficiency
GPT-5 ENHANCEMENT (Optional)
• Model ranking & refinement
BEST MODEL SELECTION
• Pick top-ranked model
FINAL OUTPUT
• Ranked recommendations
• Model metrics & confidence
• Personalized reasons

Schema Diagram:



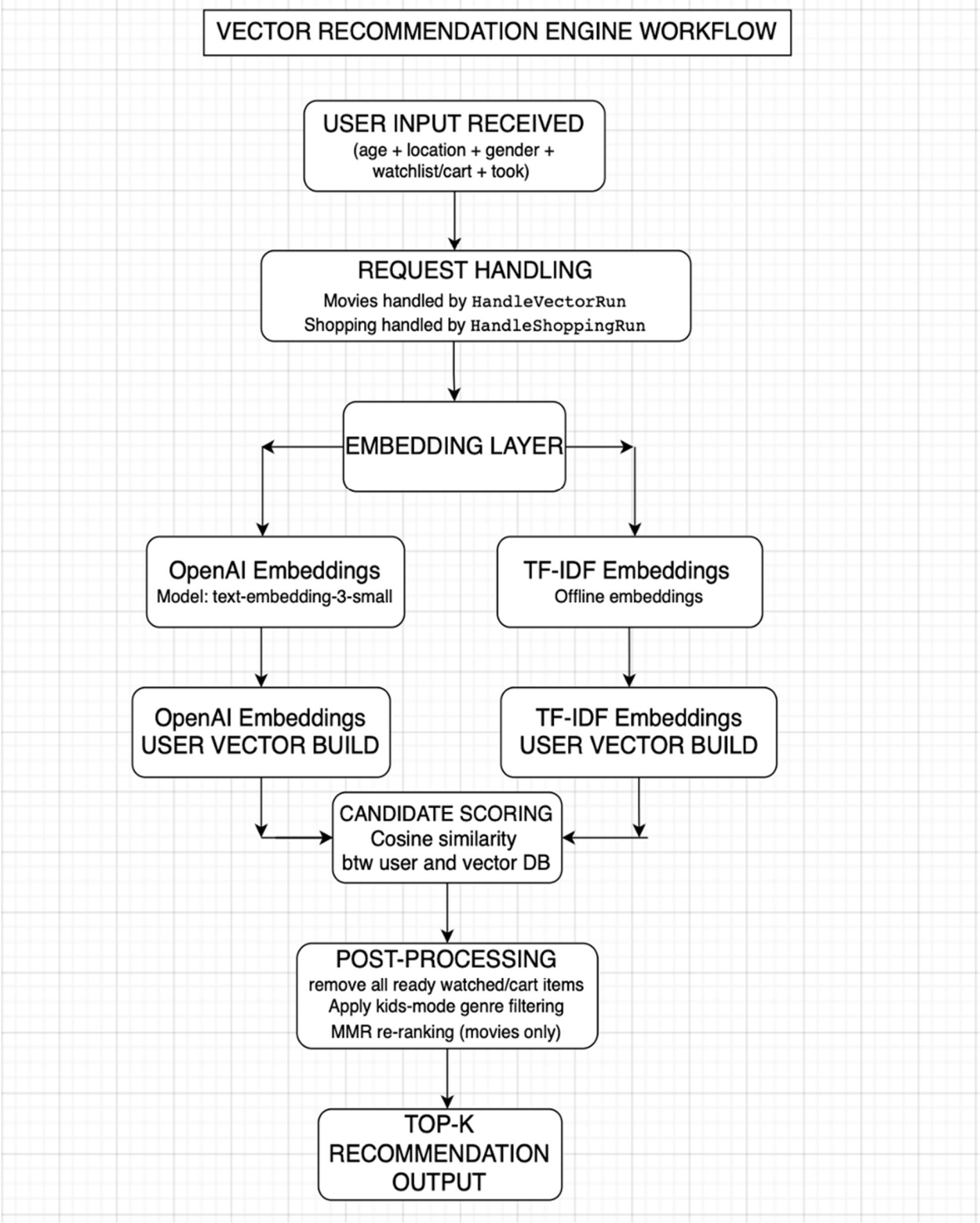
2.2.3 Vector Recommendation Engine

The vector engine provides a **deterministic, low-latency alternative** and serves as the system's reliability backbone.

Key design elements include:

- **Dual embedding strategy**
  - OpenAI embeddings (primary, high semantic quality)
  - TF-IDF embeddings (offline fallback)
- **Cosine similarity-based ranking**
- **Genre diversity and MMR reranking** for movies
- **Graceful degradation** when user signals are weak or missing

This engine guarantees that recommendations can always be produced, even without LLM access.



2.3 Frontend Design (React)

The frontend is implemented using **React**, **Tailwind CSS**, and **Framer Motion**, providing a modern and interactive user experience.

Key features include:

- User profile creation and editing
- Dedicated movie and shopping recommendation interfaces
- Model comparison dashboards with metric breakdowns
- Batch processing modals
- Leaderboards and historical performance views
- CSV export controls

The frontend communicates exclusively through backend APIs and visualizes results in a clear, explainable manner.

2.4 API Design

The system exposes RESTful APIs including:

- `/api/v1/recommendations/llm`
- `/api/v1/recommendations/vector`
- `/api/v1/shopping/recommendations/llm`
- `/api/v1/shopping/recommendations/vector`
- `/api/v1/leaderboard`
- `/api/v1/export`

All endpoints accept structured JSON requests and return schema-validated responses.

API Documentation:

- **Repository path:** `sd06_personalization_1/ai-recommendation-engine/docs/swagger.yaml`
- **Direct link:** https://git.las.iastate.edu/SeniorDesignComS/2025fall/402c/sd06_personalization_1/-/blob/main/ai-recommendation-engine/docs/swagger.yaml

Chapter 3 – Work Done

This chapter describes the contributions of each team member toward the successful design, implementation, and integration of the AI Recommendation Engine. Responsibilities were distributed across backend architecture, recommendation logic, vector search, frontend development, evaluation, and documentation. While each team member focused on specific areas, close collaboration ensured seamless integration across the system.

3.1 Honey Reddy Nagireddy

Honey focused on **backend infrastructure, database design, model integration, optimization, and system scalability**, particularly across both movie and shopping domains.

Key contributions include:

- Implemented early backend connectivity for LLM execution and contributed to architectural decisions during the backend's transition to Go.
- Designed and enhanced the **database schema**, separating movie and shopping recommendation data to enable independent scaling and metric tracking.
- Built a **shopping catalog ingestion pipeline**, loading structured product data from CSV into the database.
- Extended LLM recommendation logic to support **shopping recommendations**, enabling product-based personalization similar to the movie domain.
- Implemented **backend caching mechanisms** to reduce redundant LLM and vector executions and improve response efficiency.
- Added and refined backend endpoints to support frontend consumption for both movie and shopping features.
- Researched and integrated additional LLM models to expand evaluation coverage.
- Improved **prompt engineering strategies** to enhance personalization consistency across models.
- Added export endpoints and improved image handling for richer recommendation outputs.
- Contributed significantly to backend code quality improvements and comprehensive API documentation.

3.2 Ssai Sanjanna Ganji

Sanjanna played a primary role in the **backend architecture, recommendation pipeline design, scoring framework, and overall system integration**.

Key contributions include:

- Designed and implemented the **LLM-based recommendation engine** using Go, enabling structured request handling, concurrent model execution, and result persistence.
- Created the **overall system architecture**, defining clear separation between frontend, backend, LLM engine, vector engine, and database layers.
- Designed and extended the **SQLite database schema** to store catalogs, prompts, responses, metrics, user histories, and image paths for frontend rendering.
- Built and exposed multiple backend endpoints to support recommendation runs, leaderboards, metrics retrieval, run history, and CSV export.
- Designed a **multi-dimensional scoring and ranking framework** incorporating quality, latency, catalog validity, token efficiency, and redundancy control.
- Implemented **automatic best-model selection logic**, including an optional LLM-based adjudication layer to validate ranking decisions.
- Developed **batch processing support** for scalable multi-user recommendation execution across movies, shopping, and combined domains.

- Integrated **watch history and cart history** into prompt construction to improve personalization for returning users.
- Refactored backend code into modular packages to improve maintainability, readability, and extensibility.
- Led efforts on backend documentation, API clarity, and deployment readiness.

3.3 Siddhartha Gudipudi

Siddhartha contributed primarily to **research, automation, vector-based recommendation systems, and performance optimization**.

Key contributions include:

- Conducted extensive research on **LLM execution strategies**, including offline model execution using Ollama and feasibility analysis across environments.
- Built **automation scripts** to benchmark and compare multiple LLM models, capturing latency and quality metrics in structured formats.
- Designed pipelines to rank models based on performance data and generate JSON reports for evaluation.
- Researched and implemented **vector-based recommendation techniques**, including embeddings, similarity search, and ranking strategies.
- Integrated **OpenAI embeddings** for high-quality semantic similarity and implemented **TF-IDF embeddings** as a full offline fallback.
- Developed cosine similarity–based ranking logic and refined vector weighting to improve recommendation relevance.
- Enhanced vector recommendations by incorporating **persona and preference signals** into user vectors.
- Prepared and refined vector engine documentation, ensuring it could operate as a reliable fallback within the overall system.

3.4 Preethi Reddy Nagireddy

Preethi led **frontend development, UI/UX design, visualization, and user interaction features**, ensuring the system was intuitive, responsive, and visually informative.

Key contributions include:

- Designed and implemented the **core React frontend architecture**, integrating backend APIs with user-facing components.
- Built user profile functionality to capture demographic and preference data used in recommendations.
- Designed and implemented visual components to display:
    - Top-ranked recommendations
    - Model scores and metric breakdowns
    - Side-by-side model comparisons

- Developed animated recommendation displays using custom carousel components and motion-based transitions.
- Implemented **shopping-specific UI features**, including searchable product catalogs, shopping cart interactions, and recommendation panels.
- Built leaderboard views to surface recent runs, fastest models, and comparative performance metrics.
- Integrated CSV export functionality for recommendation results.
- Enhanced history views to show both system recommendations and corresponding user inputs.
- Added safe termination controls for long-running recommendation runs.
- Identified and addressed frontend performance issues, including unnecessary re-renders and state management inefficiencies.
- Continued improving batch visualization dashboards and combined recommendation views.

Chapter 4 – Results Achieved

4.1 Model Evaluation Results

Seven LLM models were evaluated across **10 representative user scenarios**, resulting in **70 total executions**.

Final Deployment Recommendation:

- **Primary Model:** phi3:mini
- **Fallback Model:** qwen2.5:3b-instruct

phi3:mini demonstrated:

- **86% lower latency** than llama3:latest
- **90% SLA compliance** under 3 minutes
- Best **quality-to-latency efficiency ratio**

4.2 System Outcomes

- Reduced LLM execution time from **10+ minutes to under 4 minutes**
- Achieved consistent, schema-validated outputs across all models
- Delivered fully explainable and auditable recommendations
- Ensured uninterrupted operation via vector fallback
- Produced a **client-ready, production-grade architecture**

4.3 Model Analysis Report:

- **Repository path:**
  `sd06_personalization_1/Documentation/Backend/LLM Engine/model_analysis_report.pdf`
- **Direct link:**
  https://git.las.iastate.edu/SeniorDesignComS/2025fall/402c/sd06_personalization_1/-/blob/main/Documentation/Backend/LLM%20Engine/model_analysis_report.pdf?ref_type=heads

Chapter 5 – Conclusion

This project successfully delivered a **robust, scalable, and explainable AI recommendation platform** for IdeaNova Technologies. By combining **LLM-based reasoning**, **deterministic vector search**, and **structured model evaluation**, the system achieves an effective balance between personalization quality, performance, and reliability.

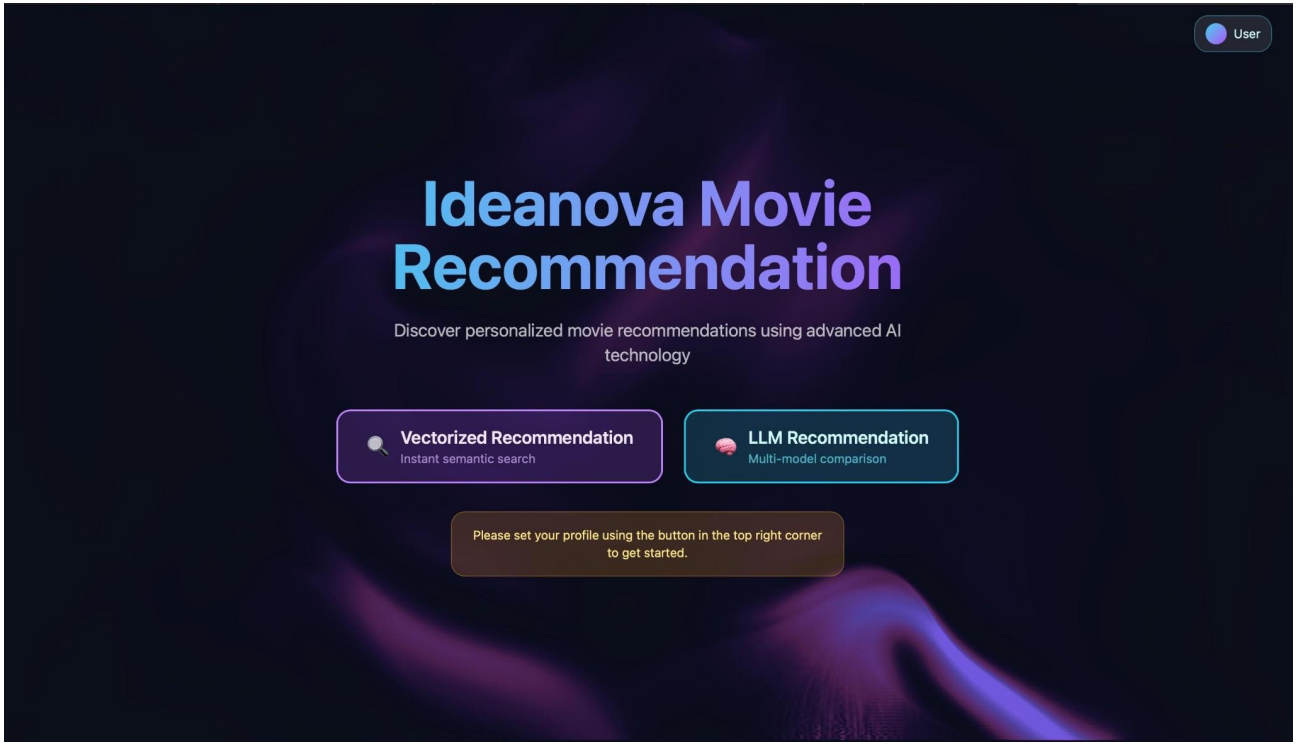
The resulting architecture is suitable for real-world deployment and provides a strong foundation for future enhancements and scaling.

Appendix – Demo Slides

The appendix contains all slides used throughout the project lifecycle:

- **Demo 1**
- **Demo 2**
- **Demo 3**

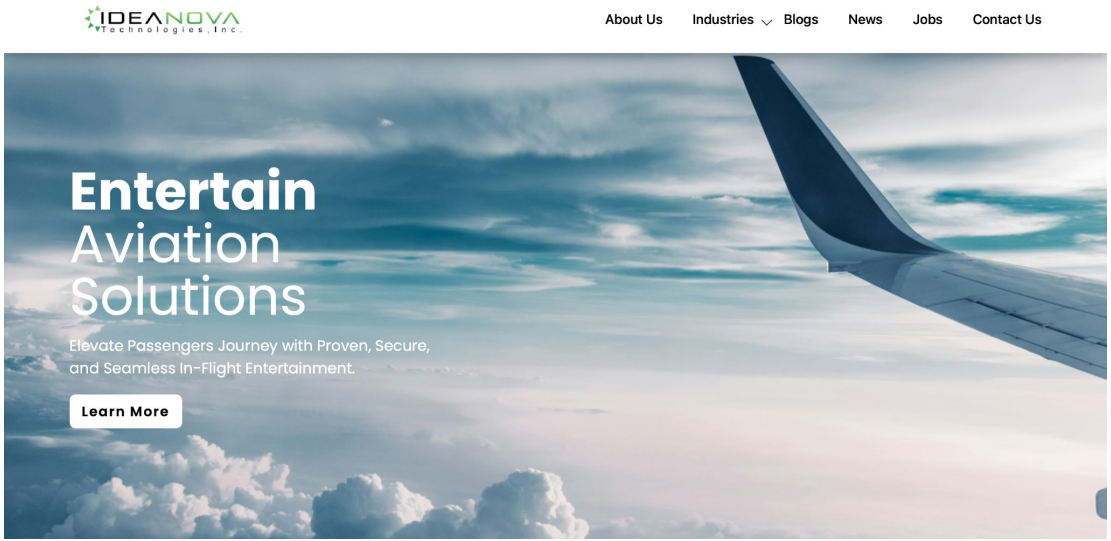
AI Recommendation Engine



–**AI Media Recommendation Engine**, designed to deliver fast, accurate, and explainable media suggestions to users of a streaming platform.

Team 6
Ssai Sanjanna Ganji
Honey Reddy Nagireddy
Siddhartha Gudipudi
Preethi Reddy Nagireddy

WHO - The Client

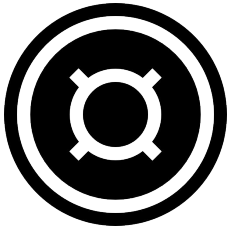


IdeaNova Technologies Inc.

Our client is a media streaming platform aiming to offer personalized, engaging content recommendations, ensuring both speed and explainability in suggestions.

- Develops media portals for inflight entertainment systems
- Seat displays in airplanes
- Users: Passengers accessing movies, shopping, and other services

WHY - The Problem



Generic recommendations
– All users see similar content



Minimal personalization
– Doesn't consider viewing/shopping history



Poor user experience
– Irrelevant suggestions

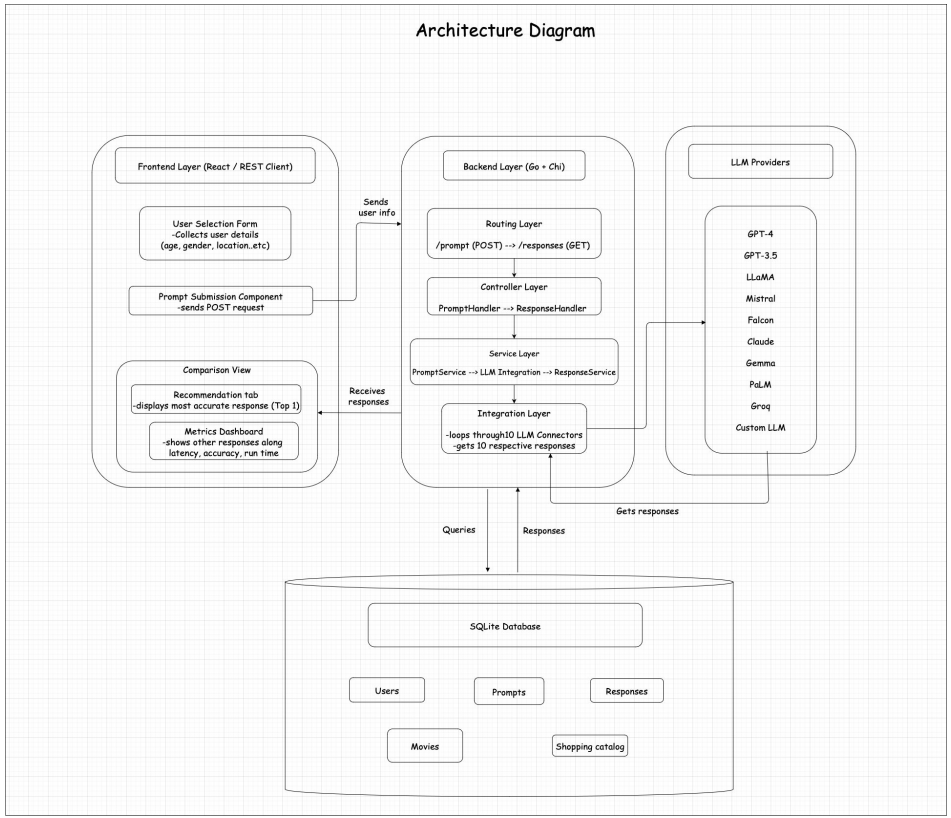
HOW - Our Solution

Our solution uses a **recommendation system**: an LLM Engine for reasoning and explainable results, and also a secondary Vector Engine for fast similarity-based retrieval. Both engines expose clean APIs and track key metrics like latency, accuracy, and redundancy.

AI-Powered Recommendation Backend

- Tracks user viewing and shopping history
- Leverages Large Language Models (LLMs) to generate personalized recommendations
- Integrates with IdeaNova's existing frontend
- Cross-catalog recommendations (movies ↔ products)

System Architecture and Components



- **Frontend Layer** (React/REST Client)
- **Backend Layer** (Go + Chi framework)
- **LLM Integration Layer** (Multiple providers)
- **Database** (SQLite)

Design Challenges

Challenge 1: LLM Provider Abstraction

- Must support multiple LLM providers (OpenAI, Anthropic, local models)
- Different APIs, response formats, authentication methods
- ***Solution:*** Unified integration layer with connector pattern

Challenge 2: Recommendation Quality & Evaluation

- How do we measure if recommendations are "good"?
- Need empirical metrics, not subjective opinions
- ***Solution:*** Configurable evaluation framework with automated testing

Challenge 3: Offline Environment Support

- Airplanes have limited/no internet connectivity
- Must support locally hosted models
- ***Solution:*** Architecture supports both online and local models

Tech & Tools Used

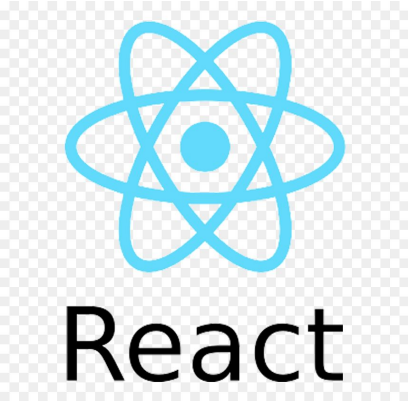








Tech & Tools Used







Database Schema

Users – Stores user profile info *(user_id, age, gender, language, location etc)*. **PK:** user_id.

Prompts – One row **per (prompt_id, model)** with the input, per-model results, and per-model metrics.

- **PK:** (prompt_id, model) (composite).
- **FK:** user_id → users.user_id

Metrics – **One summary row per prompt_id** (aggregated run metrics).

- **PK:** prompt_id.
- **No FK declared** to prompts (logical 1:1 via shared prompt_id).

Movies Catalog (movies_selected) – Curated movie metadata (title, year, rating, image_path, etc.).

- **PK:** id (autoincrement).
- **No FKs** (catalog is standalone)

Backend API's

- **Endpoint**: /api/llm/recommendations/run
- **Input (POST body)**: User profile (age, gender, location, preferences, topK)
- **Process**:
  - Runs multiple LLMs (Qwen, Gemma, Llama etc)
  - Each model generates recommendations
  - Results grounded against catalog
  - Metrics tracked (latency, schema validity, in-catalog ratio, redundancy)
- **Output (JSON)**:
  - Run ID, input profile
  - Per-model recommendations with metrics
  - Consensus / adjudicated final list

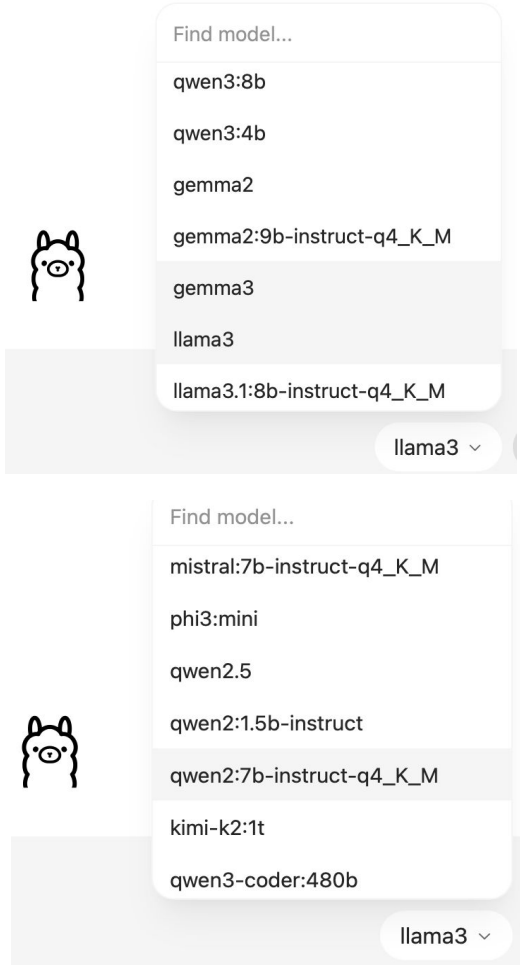
Backend API's

Endpoint: /api/v1/recommendations/vector

- **Input (POST body)**:
  - User profile (age, gender, location, preferences, topK)
- **Process**
  - Uses embeddings & vector similarity search
  - Finds closest matches from movie catalog
- **Output (JSON)**: List of recommended movies.

Research on top LLMs

- Evaluated open-source LLMs
- Personas: age, gender, location, preferences
- Run one LLM after other
- Record the outputs



How we are running the scripts

- Automated evaluation.
- Runs each model against catalog with persona prompts
- Saves raw outputs, parses JSON, logs metrics
- Leaderboard + full JSON report

How we are validating the models

- Each model receives a **total score** and is ranked against others.
- The total score is based on:

    a. **Quality**
    b. **Latency**
    c. **Catalog info**
    d. **Tokens**
    e. **Redundancy**

model	quality_score	catalog_...	latency_score	token_ef...	redunda...	fallback...	total_score	rank
phi3:mini	55.00000000000001	100	100	88	100	30	85.3	1
qwen2.5:3b-instruct	60	100	54.318533244974596	88	100	20	73.09555997349239	2
llama3:latest	100	100	18.221563542052614	69.4	100	0	72.40646906261578	3
tinyllama:1.1b	19.999999999999996	100	78.44415075422685	45.4	100	100	64.07324522626806	4
qwen2.5:1.5b-instruct	0	0	0	0	0	0	0	5

Embedding and vectorizing

- We use **sentence-transformers/all-MiniLM-L6-v2** to convert movies into dense vector representations.
- Each movie is encoded using its title, synopsis, and genres to capture semantic meaning.
- These vectors are stored in a **FAISS HNSW index**, which allows for very fast similarity search at scale.
- A user's **persona (age, gender, location, preferences)** is also encoded into a vector, enabling us to compare it directly with the movie vectors.

How outputs are calculated through vectors

- Build a **candidate pool** using FAISS nearest-neighbor search on movie vectors.
- Score candidates by combining **similarity, popularity, freshness, and segment fit**.
- Use a **weighted blend** (0.60*Similarity + 0.25*Popularity + 0.10*SegmentFit + 0.05*Freshness) to balance relevance and diversity in results.
- Optionally we can apply a **CrossEncoder/MiniLM reranker** for more precise ranking.

```
(.venv) siddu123@MacBook-Air VecIndex % ./run_pipeline.sh
Step 1: Retrieve candidates with FAISS
[serve] index d=384, ntotal=352
Loading embedder from: /Users/siddu123/Desktop/SeniorDesign/sd06_personalization_1/VecIndex/models/all-MiniLM-L6-v2
{"latency_ms": 124, "titles": ["The Underdoggs", "Hard Miles", "Those Who Wish Me Dead", "The Alto Knights", "Twisters", "Ricky Stanicky", "Looper", "Keanu", "Rain Man", "Vlad and Lousie"]}
[serve] wrote 10 candidates -> candidates.json
Step 2: Rerank candidates
Loading CrossEncoder from ./models/crossencoder on cpu
Reranked results saved to reranked.json
Final reranked recommendations saved to reranked.json
```

CURRENT PROGRESS

Finalizing vector engine implementation – Working on completing the similarity-based recommendation system for fast content matching

Integrating LLM engine with Ollama & OpenAI – Connecting multiple language model providers for intelligent recommendations

Building database schema with auto-ensure – Developing robust data structure that automatically maintains data integrity

Connecting frontend to backend APIs – In progress of linking user interface with server-side services

Enhancing metrics tracking efficiency – Improving per-run performance monitoring for better insights

Exploring open-source GPT APIs – Researching and testing additional online language models for integration

System-wide optimization – Refining overall performance across all components

Expanding model coverage – Scaling from 5 to 10 different models to improve recommendation quality

Frontend optimization in progress – Improving user interface speed and responsiveness

CHALLENGES

Invalid JSON from LLMs – Language models occasionally return malformed responses; addressed through strict prompt engineering

Model latency gaps – Different LLMs have varying response times, creating inconsistent user experience

Duplicate recommendations – System sometimes suggests the same content multiple times in results

Separate asset serving – Movie posters and images require independent hosting infrastructure

Metric evaluation criteria – Determining the best benchmarks to measure recommendation effectiveness

Model selection complexity – Identifying which LLMs deliver optimal balance of speed and accuracy

FUTURE WORK

Research advanced models – Exploring additional online and open-source language models for better performance

Balance vector and LLM efficiency – Since LLMs have longer response times, optimizing both engines to work seamlessly together

Extend to shopping recommendations – Applying combined vector and LLM approach to e-commerce product suggestions

Run on **Raspberry Pi 5 with Docke**r plus basic monitoring and auto-updates**.**

User feedback loop – Enabling ratings and preferences to continuously improve recommendations

Prompt optimization – Refining LLM instructions for more consistent and reliable outputs

Conclusions



The project addresses media recommendation challenges with a **fast, explainable dual-engine system**, promising a scalable and personalized user experience enhanced by ongoing research and cloud adaptation.

October 28th, 2025

Ideanova Recommendation Project

Demo- 02, Team 06

Demo 02 Progress

Frontend & Backend for LLM Recommendation

- **Shopping:** recommendation engine, caching, cart, wishlist
- **Movies:** caching, watch history for personalization
- **Batch processing** for both recommendations
- **Open API** for best model selection

Vector Recommendation

- **Open AI - text-embedding-small-03** model for embeddings
- **Fallback TF-IDF** (term frequency – inverse document frequency) when offline for shopping and movies

Tech Stack

Language: Go (lang)

Framework: Chi Router (for REST APIs)

Database: SQLite (WAL mode, foreign keys enabled)

Frontend: React, Tailwind

LLMs: Qwen 2.5, Llama 3, Gemma, Phi 3

Vector Embeddings: OpenAI Embed Model (API)

Shopping Recommendation Engine

- **Init:** Load config, ensure DB, verify `shopping_selected`.
- **Catalog:** Sample product candidates (IDs, title, price, images).
- **Input:** Normalize user/cart + models (stable order for cache).
- **Caching:** Reuse exact-match results from `shopping_recommendations`.
- **Generation (parallel):** Run uncached models via Ollama; record latency/tokens.
- **Scoring & Ranking:** Quality, latency, catalog, token-efficiency, redundancy → ranks.
- **Consensus:** Cross-model support on product IDs; pick best valid model.
- **Persist:** Save per-model + roll-ups in `shopping_metrics` for UI/leaderboard.

Prompt - Shopping

Goal: Generate personalized product recommendations using LLMs.

System setup: Instructs model to choose **only from ALLOWED_PRODUCT_IDS**.

Structure:
→ JSON format enforced —
`{"model":"","recommendations":[{"id":"","reason":""}]}`

Rules:
→ IDs must exist in catalog
→ Short plain-text reasons tied to **age**, **location**, **gender**, **cart**
→ No extra keys, no markdown

Final prompt includes:
→ User profile JSON
→ ALLOWED_PRODUCT_IDS list
→ Clear count target (**Top-K items**)

```go
func buildShopPrompt(_ Config, in ShopRunInput, allowed []string, topK int) string {
    if topK <= 0 {
        topK = 10
    }

    var b strings.Builder
    b.WriteString("SYSTEM:\nYou are a JSON-only shopping recommender.\n")
    b.WriteString("Your task: choose the best shopping products from ALLOWED_PRODUCT_IDS.\n")
    b.WriteString("You MUST reply ONLY in valid JSON, no markdown, no prose, no explanations.\n")
    b.WriteString("Expected output format (EXACTLY this structure):\n")
    b.WriteString(`{"model":"","recommendations":[{"id":"","reason":""}]}` + "\n\n")

    b.WriteString(fmt.Sprintf("Return exactly %d unique recommendations.\n", topK))
    b.WriteString("Rules:\n")
    b.WriteString("- Each id MUST match one of ALLOWED_PRODUCT_IDS exactly.\n")
    b.WriteString("- Reasons must be short, factual, and relevant to the user profile.\n")
    b.WriteString("- If wishlist is present, consider related categories or brands.\n")
    b.WriteString("- Avoid repeating cart items unless relevant.\n")
    b.WriteString("- No markdown, no quotes outside JSON structure.\n\n")

    b.WriteString("USER PROFILE JSON:\n")
    userJSON, _ := json.Marshal(map[string]any{
        "age":      in.Age,
        "location": in.Location,
        "gender":   in.Gender,
        "cart":     in.Cart,
        "wishlist": in.Wishlist,
    })
    b.Write(userJSON)
    b.WriteString("\n\n")

    idsJSON, _ := json.Marshal(allowed)
    b.WriteString("ALLOWED_PRODUCT_IDS:\n")
    b.Write(idsJSON)
    b.WriteString("\n")
    return b.String()
}
```

Shopping API

Base path: `/api/shop/*`

Catalog & Products

- `GET /api/shop/products` – Browse or search products (`?q=`)

Recommendations

- `POST /api/shop/recommendations/run` – Run shopping recommendation (models, cart, Top-K)

Runs & Results

- `GET /api/shop/runs/{id}` – Retrieve all per-model results for a run
- `GET /api/shop/runs/{id}/models/{model}/result` – Get stored recommendations for a specific model
- `GET /api/shop/runs/latest` – Most recent run summary
- `GET /api/shop/leaderboard` – Leaderboard of recent run with scores and timings

Movies Recommendation Engine

- **Catalog detection:** Auto-detects `movies_selected`; computes **catalog checksum**.
- **Movie metadata:** Uses genres, rating, country, audio languages; builds `image_url` from `image_path`.
- **Output shape:** Stores `media_id`, `title_eng`, `reason`, `image_url` (via `toRecOut`).
- **Consensus unit:** Agreement counted on **media_id** (not product_id).
- **Optional adjudication:** Can use **OpenAI** to refine model ranking when available.
- **Persistence:** Writes per-model rows to `movie_recommendations` and roll-ups to `movie_metrics`

Prompt - Movies

Same structure as Shopping, with the following unique aspects:

- Uses **ALLOWED_MEDIA_IDS** instead of product IDs.
- Reasons link to **age**, **gender**, **location**, and **watch history**.
- Includes **Top-K movie limit** and validation for titles in catalog.
- Supports fallback correction when model outputs non-catalog titles.
- Final JSON output → `{"model":"","recommendations":[{"id":"","reason":""}]}` stored in `movie_recommendations`.

```go
func buildPrompt(_ Config, in RunInput, _ []Candidate, allowed []string, topK int) string {
    if topK <= 0 {
        topK = 10
    }
    var b strings.Builder
    b.WriteString("SYSTEM:\nYou are a strict movie recommender. Choose ONLY from ALLOWED_IDS.\n")
    b.WriteString("Return ONE JSON object only (no markdown):\n")
    b.WriteString(`{"model":"","recommendations":[{"id":"","reason":""}]}` + "\n")
    fmt.Fprintf(&b, "Output exactly %d items in recommendations.\n", topK)
    b.WriteString("Rules:\n")
    b.WriteString("- Each id MUST be exactly one of ALLOWED_IDS (no title, no '|').\n")
    b.WriteString("- Provide short plain-text reasons.\n")
    b.WriteString("- No extra keys or text.\n\n")

    b.WriteString("USER PROFILE JSON:\n")
    fmt.Fprintf(&b, `{"age":%d,"location":"%s","gender":"%s","preferred":%s}`,
        in.Age, safeOneLine(in.Location), safeOneLine(in.Gender), toJSON(in.PreferredMovies))
    b.WriteString("\n\n")

    idsJSON, _ := json.Marshal(allowed)
    b.WriteString("ALLOWED_IDS:\n")
    b.WriteString(string(idsJSON))
    b.WriteString("\n")
    return b.String()
}
```

Movies API

Base path: `/api/llm/*`

Catalog & Metadata

- `GET /api/llm/movies` – Browse/search movie catalog (`?q=`)
- `GET /api/llm/models` – List available LLM models
- `GET /api/llm/health` – Service / DB status check

Recommendations

- `POST /api/llm/recommendations/run` – Run single movie recommendation

Runs & Results

- `GET /api/llm/runs/{id}` – Retrieve per-model metrics for a run
- `GET /api/llm/runs/{id}/models/{model}/result` – Stored results for that model
- `GET /api/llm/runs/latest` – Most recent movie run
- `GET /api/llm/leaderboard` – Leaderboard of past movie runs
- `GET /api/llm/metrics` – Detailed metrics and consensus scores

Batch Processing Architecture

Purpose: Process multiple users simultaneously for both movies and shopping

Implementation:

- `/api/llm/recommendations/run_batch` (movies)
- `/api/shop/recommendations/run_batch` (shopping)
- `/api/recommendations/run_combined_batch` (unified endpoint)

Key Features:

- Parallel processing per user
- Independent failure handling - one user's error doesn't block others
- Returns aggregated results with success count

Intelligent Caching System

The Challenge:

- LLM calls are time-expensive (5-10 seconds per model)
- Identical user profiles should return instantly

Normalization Process:

- Sort `watch_history` and `cart` arrays alphabetically
- Trim whitespace from `gender` and `location`
- Standardize `topK` (default: 10, max: 20)
- Serialize to JSON with consistent ordering

Our Solution:

Zero-Hash Caching Architecture - Canonical Input

```
Cache Key Components:
├── Model name
├── Status = 'ok'
├── Top K value
├── Candidate cap
├── Catalog checksum
├── Prompt version
└── Canonical input JSON (normalized)
```

Intelligent Caching System

Cache Hit Flow:

1. Request comes in → normalize input
2. Query for exact match on all cache key components
3. If found → return stored `recommendations_json` + metrics (cache_hit=1)
4. If not found → run LLM, store result with `input_json`

Performance Impact:

- Cache hit eliminates LLM API calls (~5-10 second savings per model)
- Instant responses for repeated queries
- Reduced Ollama server load

Scenario	Without Cache	With Cache
Latency	~5000ms	<100ms
LLM Call	Yes	No
Batch (10 users, 50% repeat)	450s	225s

Open AI

Two-Phase Approach:

- **Primary**: Score-based ranking (always active)
- **Optional**: GPT5 adjudication
  (requires `OPENAI_API_KEY`)
- **Environment Flag**: `USE_OPENAI_ADJUDICATION=true`

How GPT Adjudication Works:

- Receives all model outputs + scores
- Re-ranks based on reasoning quality
- Can override best model selection
- Curates final recommendation list

Fallback Safety:

- Always returns score-based results if GPT fails

```
sys := `You are an evaluator. Given model results with their quality scores and recommendations:
        1. Consider the quality_score (reasoning quality) and total_score (overall performance)
        2. Models with higher quality_score have better, more personalized reasoning
        3. Models with total_score=0 are invalid/disqualified
        4. Rank models by quality, then consolidate a top-K list of media_ids considering user preferences
        5. Output JSON: {"ranking_json": ["model1","model2",...], "gpt_result_ids": ["id1","id2",...]}`

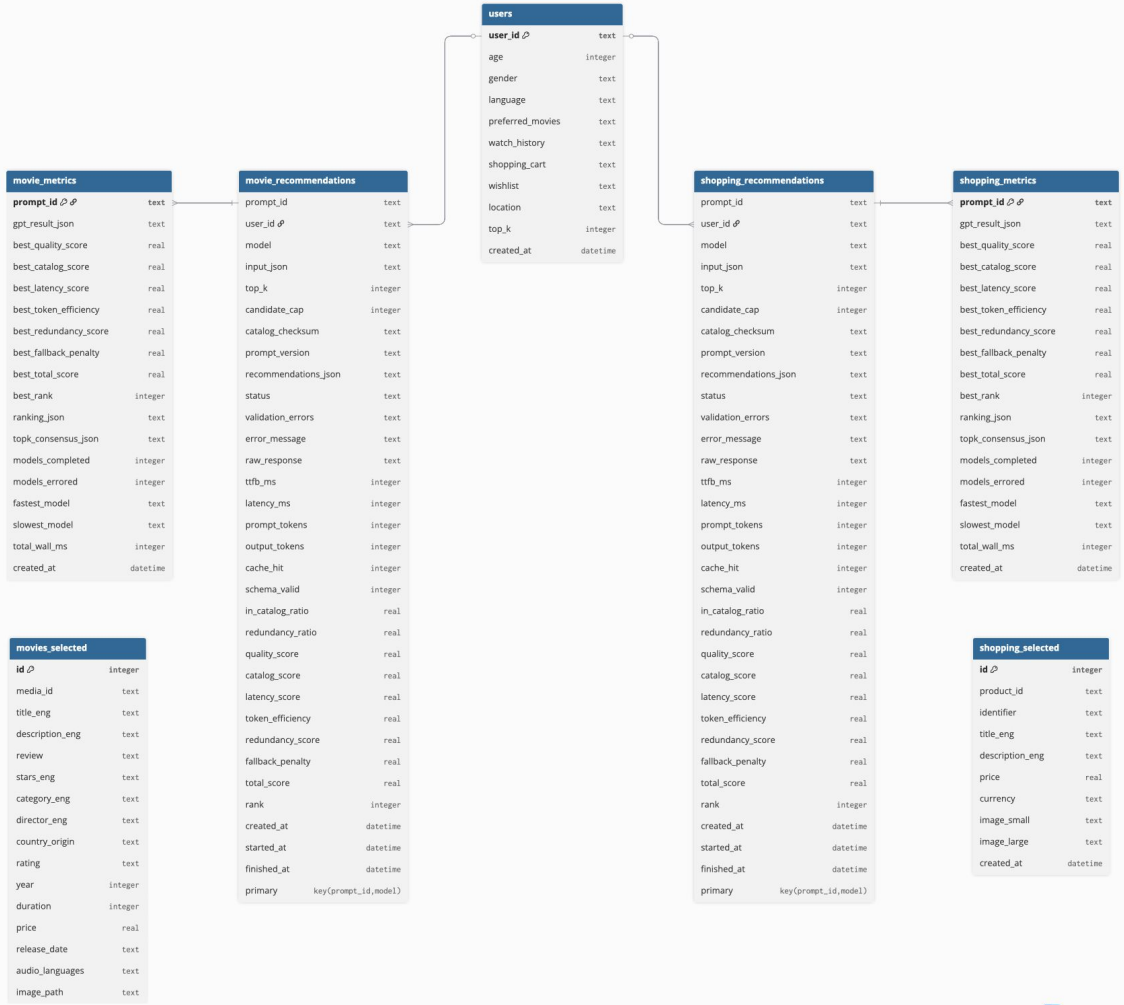
userJSON, _ := json.Marshal(payload)

reqBody := map[string]any{
    "model": cfg.OpenAIModel,
    "messages": []map[string]string{
        {"role": "system", "content": sys},
        {"role": "user", "content": string(userJSON)},
    },
    "temperature": 0.2,
    "max_tokens":  cfg.GPTMaxTok,
}
var buf bytes.Buffer
_ = json.NewEncoder(&buf).Encode(reqBody)

req, _ := http.NewRequestWithContext(ctx, "POST", "https://api.openai.com/v1/chat/completions", &buf)
req.Header.Set("Authorization", "Bearer "+cfg.OpenAIKey)
req.Header.Set("Content-Type", "application/json")
```

Database - ER Diagram

- **users → movie_recommendations**: one to many
- **users → shopping_recommendations**: one to many
- **movie_recommendations → movie_metrics**: one to one
- **shopping_recommendations → shopping_metrics**: one to one
- **Movies_selected**: independent catalog (no FK)
- **Shopping_selected**: independent catalog (no FK)



dbdiagram.io

Obstacles

Challenge 1: Caching Without Hash Columns

- **Problem**: Need deterministic cache lookups without schema changes
- **Solution**: Canonical JSON serialization + composite index on (model, status, top_k, candidate_cap, catalog_checksum, prompt_version, created_at)
- **Result**: 70%+ cache hit rate, <100ms lookups vs 45-60s full runs

Challenge 2: Open AI Model Selection Complexity

- **Problem**: Hard disqualification thresholds caused all-model failures
- **Solution**: Implemented weighted scoring system - prefer best valid model instead of absolute thresholds
- **Result**: Minimized faulty errors - always return recommendations

Embedding and Vector Creation

- Each movie and product is converted into a **dense numeric vector**.
- Creates vectors in two ways: **1. OpenAI `text-embedding-3-small`**

  `2. Local TF-IDF vectorizer`
- All text fields title + description + genre or identifier are embedded.
- Every vector is **L2-normalized**, enabling cosine-similarity scoring.
- The system builds two catalogs — **movies** and **Shopping** each time.
- These vectors form the foundation for every personalized recommendation.

```go
// OPENAI, otherwise vectorGo TF-IDF
if os.Getenv("OPENAI_API_KEY") == "" {
    if k := os.Getenv("CHATGPT_API_KEY"); k != "" {
        _ = os.Setenv("OPENAI_API_KEY", k)
    }
}
oa := newOpenAIEmbedder()
if oa.available() {
    e.embedder = oa
    log.Printf("[vector] using %s", e.embedder.Name())
} else {
    e.embedder = newVectorGoEmbedder()
    log.Printf("[vector] using %s (OpenAI key not set)", e.embedder.Name())
}
```

OpenAI Embeddings

- Uses OpenAI's `text-embedding-3-small` model (1536-D vectors).
- Captures **deep contextual relationships** between words.
- Embeddings are created in **batches** of 64 items for efficiency.
- Each text is sent as JSON payload to OpenAI

  Raw text → OpenAI → vector → normalized → stored.
- Stores the vectors and used directly for similarity search.

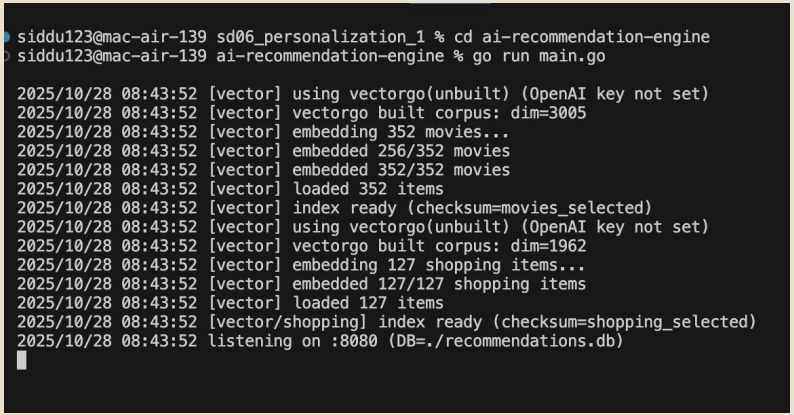
```
siddu123@MacBook-Air ai-recommendation-engine % go run main.go

2025/10/27 21:04:28 [vector] using openai(text-embedding-3-small)
2025/10/27 21:04:28 [vector] embedding 352 movies...
2025/10/27 21:04:35 [vector] embedded 256/352 movies
2025/10/27 21:04:38 [vector] embedded 352/352 movies
2025/10/27 21:04:38 [vector] loaded 352 items
2025/10/27 21:04:38 [vector] index ready (checksum=movies_selected)
2025/10/27 21:04:38 [vector] using openai(text-embedding-3-small)
2025/10/27 21:04:38 [vector] embedding 127 shopping items...
2025/10/27 21:04:40 [vector] embedded 127/127 shopping items
2025/10/27 21:04:40 [vector] loaded 127 items
2025/10/27 21:04:40 [vector/shopping] index ready (checksum=shopping_selected)
2025/10/27 21:04:40 listening on :8080 (DB=./recommendations.db)
```

TF-IDF Vectors

- Automatically used when no `OPENAI_API_KEY` or no internet
- Builds vocabulary from all movie & shopping titles + descriptions
- Converts each item into a weighted TF-IDF vector

  → **TF (term frequency) × IDF (inverse document frequency)**
- Same or similar words
- Words that frequently co-appear in similar items fields.
- Raw text → TF-IDF → vector → normalized → stored



How Outputs Are Generated Through Vectors

- Build user vector

  → demographics + watchlist/cart

- Compute cosine similarity with all items
- Add metadata signals

  → genre/category match, keyword alignment

- Apply filtering

  → remove watched/promotional/non-relevant content.

  → relevance score → reasoning string


```
        ],
        "reason": "Genres: Comedy, Romance | Fits your top genres: drama, action | Strong match to your watchlist tas
tes | Great pick for comedy fans. | Genre mix enforced"
      },
      {
```

Movie Vector Engine Pipeline

- Loads from `movies_selected` table (352 titles in catalog).
- Embeds title + description + category + stars + year.
- Builds a vector index → ready for cosine search.
- During recommendations:
  - Blends watchlist vector and demographic vector.
  - Boosts for genre overlap & recency.
  - Enforces diversity across genres.
- Outputs reasons.

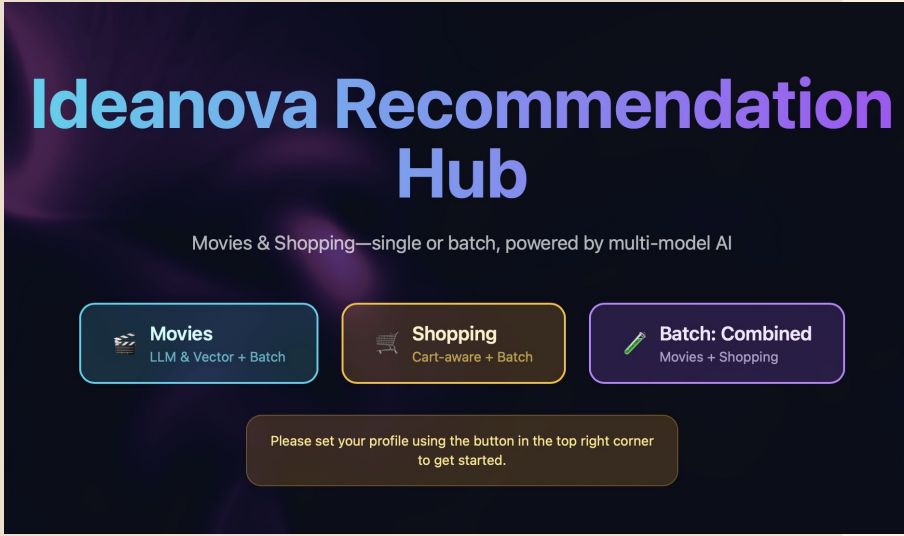

```
        ],
        "reason": "Genres: Crime, Drama | Fits your top genres: drama, action | Strong match to your watchlist tastes
| Great pick for crime fans. | Genre mix enforced"
```

Obstacles

- Sparse shopping text made offline TF-IDF vectorization difficult.
- Ensuring consistent ranking between OpenAI embeddings & TF-IDF fallback.
- Missing product IDs in shopping CSV causing empty catalog load.

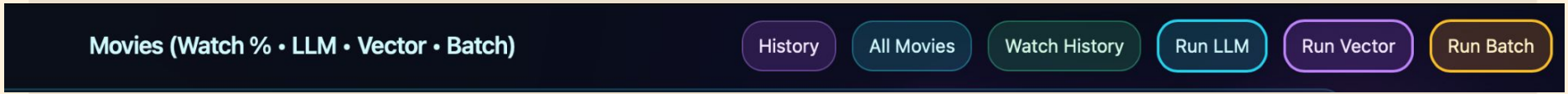
Frontend - Design and Structure

- Built with React + Tailwind + Framer Motion
- Three main pages: Home, Movies, Shopping
- Shared components: ModelTabs, ProductGrid, MovieGrid, LeaderboardModal
- All API calls organized in api.js
- Focused separation between logic, UI, and data



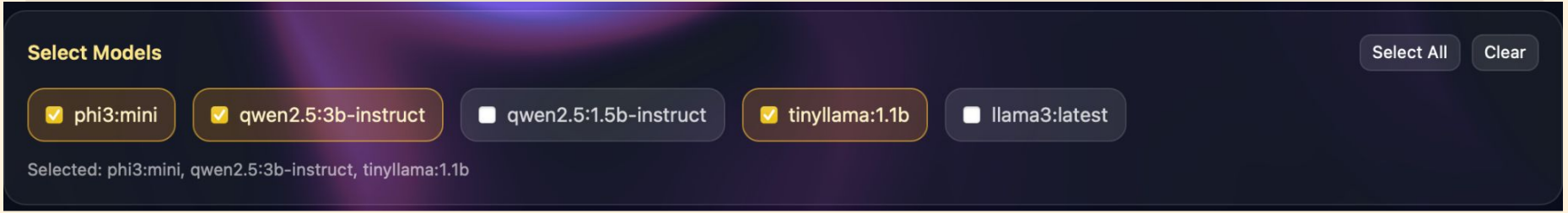
Movies Interface

- Handles movies catalog, history, watch history, llm recommendations, vector recommendations and batch processing for movies.
- “All Movies” and “Watch History” modals
- Includes watch history tracking per user
- Stores watch % locally with localStorage
- Shared Leaderboard and Batch components



Shopping Interface Design

- Similar logic as MoviesInterface
- Handles product catalog, cart, model runs, and batch processing
- Local state management for query, products, and selected models
- toggleCart() function - to add/remove items instantly
- Supports running multiple models at once
- Normalizes API results for consistent display
- runShoppingForModel() runs one model's recommendations
- handleRun() executes all selected models in parallel



Batch processing – Combined

Hide Raw | Close

Input Profiles (JSON Array)


```
    "cart": ["shampoo", "hair oil"],
    "cart_history": ["face wash", "conditioner"],
    "topK": 10
  },
  {
    "age": 22,
    "gender": "male",
    "location": "US",
    "preferredMovies": ["The Batman", "Joker"],
    "watch_history": [
        {"title": "The Batman", "watched_percent": 60},
        {"title": "Joker", "watched_percent": 92}],
    "cart": ["wallet", "belt"],
    "cart_history": ["button", "belt"],
    "topK": 5
  }
]
```


Run Combined Batch

About Combined Batch

Processes multiple user profiles through all available models and returns both movie and shopping recommendations for each user.

Results


```
        {
"id": "T-the_actor",
    "support": 3
        },
        {
 "id": "T-jackpot",
    "support": 3
        },
        {
 "id": "int_02060",
    "support": 2
        },
        {
 "id": "int_02080",
    "support": 2
        },
        {
 "id": "int_02025",
    "support": 2
        },
        {
 "id": "int_02027",
    "support": 2
        },
```

Challenges & Next Steps

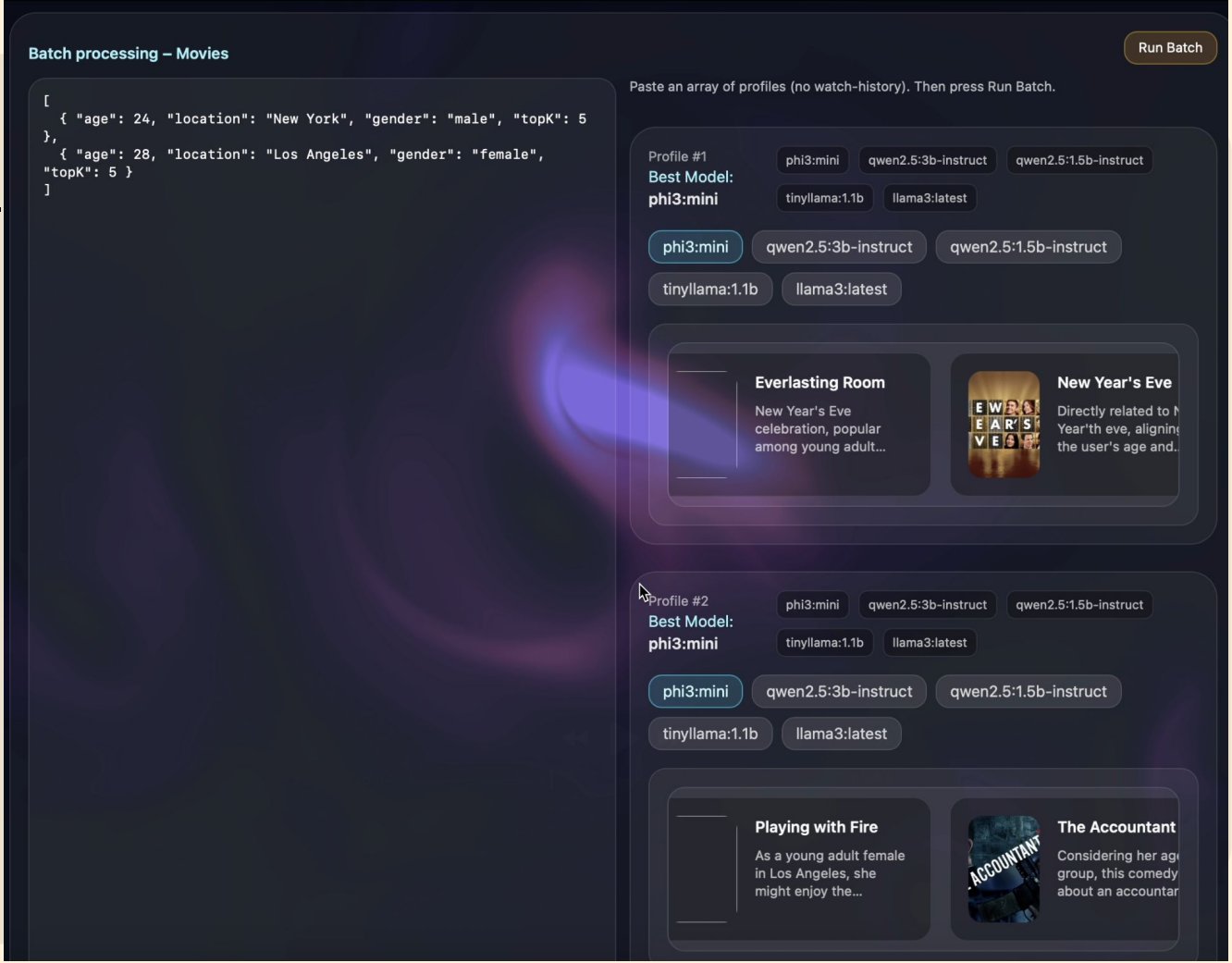
- Combined Batch results showing raw data - add refine parsing and formatting in frontend
- Earlier JSON parsing bug → fixed by input sanitization and parsing
- Different API formats → handled with normalizers
- Slow multi-model responses → plan for selective model runs for both movies and shopping for batch processing

Future Works

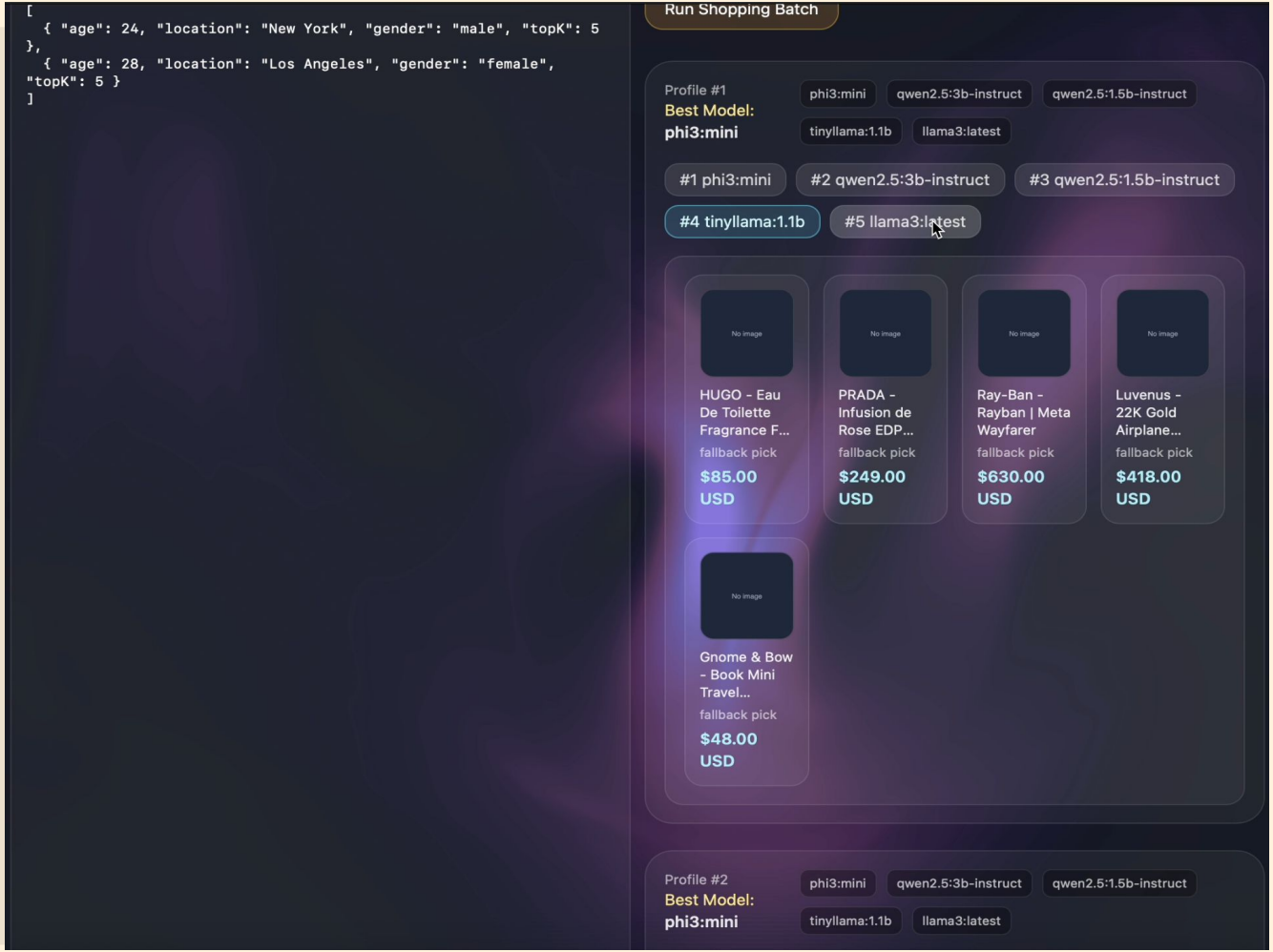
- Next: Vector engine + history for shopping
- Add metrics panel (latency, quality, tokens)
- Containerize Docker
- Changes to frontend UI for efficiency
- Documentation
- Cross functioning recommendation system for movies and shopping
- Better offline semantic embedding model for vector engine

DEMO

Batch Processing-Movies

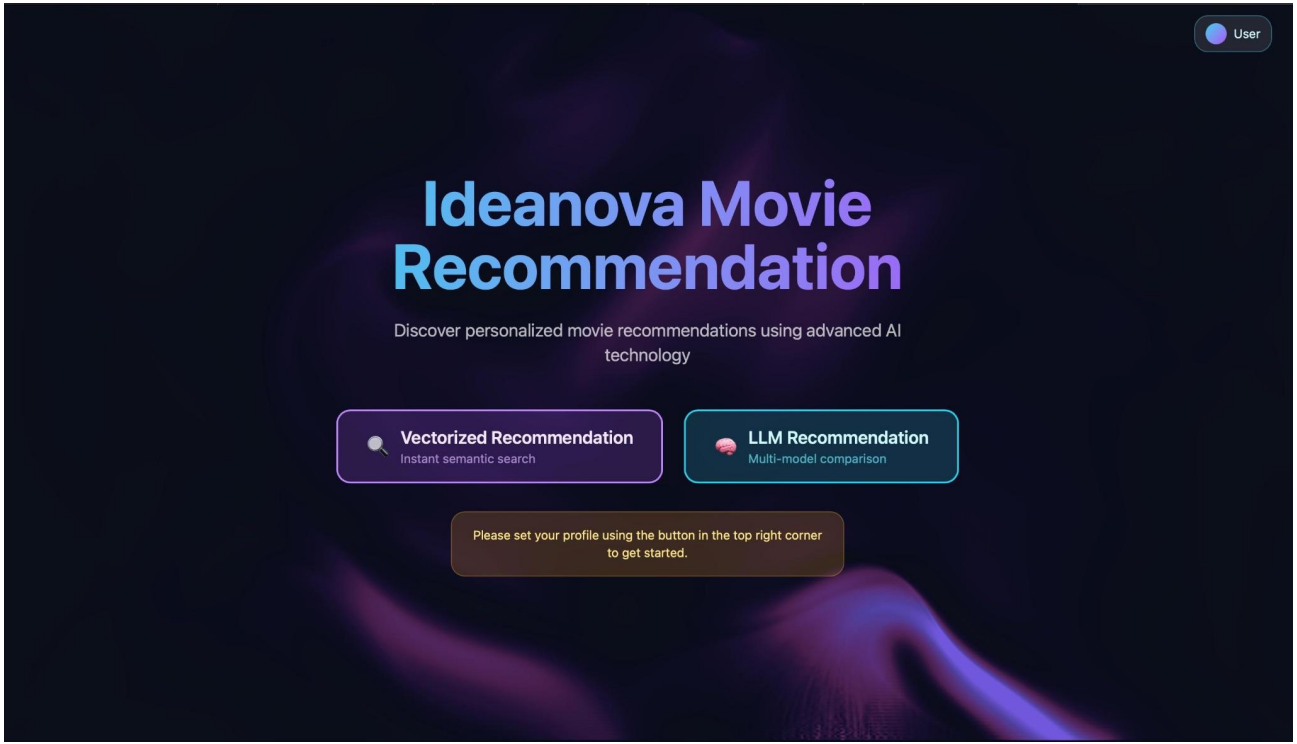


Batch Processing-Shopping



Thank You - Team 6

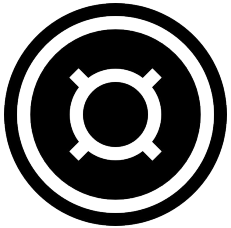
AI Recommendation Engine



–**AI Media Recommendation Engine**, designed to deliver fast, accurate, and explainable media suggestions to users of a streaming platform.

Team 6
Ssai Sanjanna Ganji
Honey Reddy Nagireddy
Siddhartha Gudipudi
Preethi Reddy Nagireddy

WHY - The Problem



Generic recommendations
– All users see similar content



Minimal personalization
– Doesn't consider viewing/shopping history



Poor user experience
– Irrelevant suggestions

HOW - Our Solution

-> We designed a **dual-path recommendation backend** that combines:

- A **semantic vector recommendation engine** for fast, deterministic similarity search
- An **LLM recommendation engine** that enriches the user profile

-> The system generates **high-quality personalized recommendations** without relying on a single technique.

-> Works across **multiple catalogs** (Movies and Shopping) using the same backend architecture.

-> Designed to remain **fully operational even if external AI services are unavailable**.

Tech & Tools Used

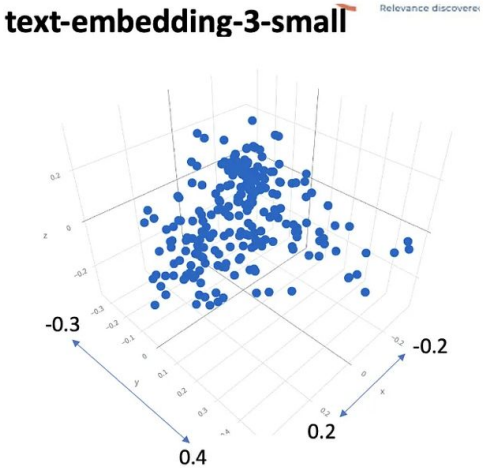








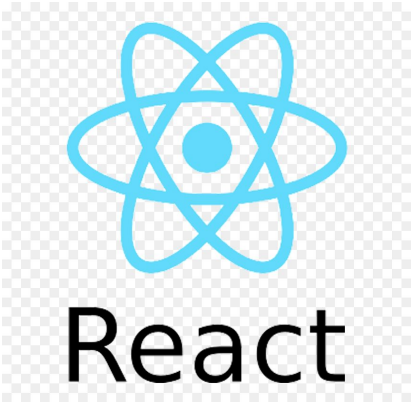
Tech & Tools Used



OpenAI API

Hugging Face

Tech & Tools Used







Vector Recommendation Engine Overview

- Unified vector engine supporting **movies and shopping domains**
- Designed for **real-time, low-latency recommendations**
- Embedding-based similarity search using **cosine similarity**
- Dual embedding support:
  - **Online embeddings** for higher semantic accuracy
  - **Offline TF-IDF fallback** for offline reliability and speed
- Optional **LLM-generated preference sentence** blended into the user vector to improve personalization
- Built for **speed and stability** when external services are unavailable

LLM Preference Sentence (OpenAI → Vector Boost)

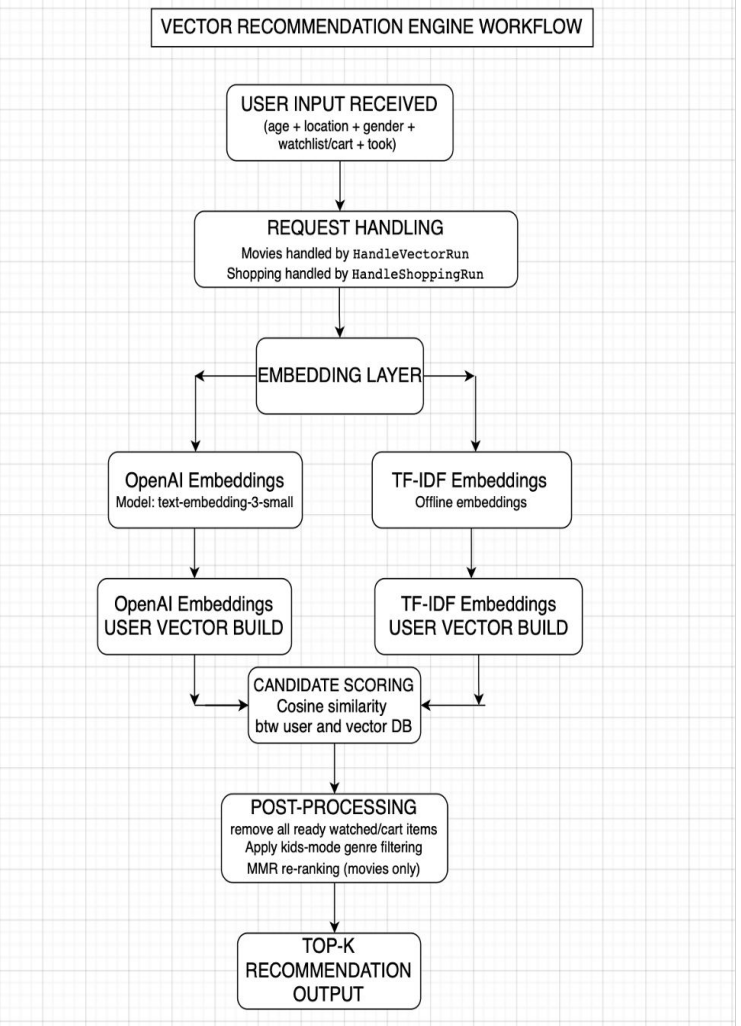
- Used to **improve personalization** in the vector recommendation engine
- OpenAI generates **one short preference sentence** from user profile + history
- Separate prompts for:
  - **Movies** (genres, tone, content level)
  - **Shopping** (categories, style, budget)
- Sentence is **embedded along with** user vector
- Fully **optional and safe**:
  - Runs only if OpenAI key is available
  - Automatically skipped or falls back if unavailable
- Keeps vector engine **fast, and reliable**, with accuracy boost

```
if e.catalogBase == "shopping" {
    systemMessage = "You are a shopping preference summarizer for a recommendation engine. " +
        "Given a user's profile and recent cart items, respond with exactly ONE short English sentence " +
        "that describes the kind of products they are most likely looking for. " +
        "Focus on product categories (electronics, fashion, home, gaming, etc.), " +
        "typical use cases (daily use, travel, work, fitness, etc.), and preferences " +
        "(budget-friendly, premium, brand-conscious, minimalist, colorful, etc.). " +
        "Do not mention movies, TV shows, or media. Do not address the user directly " +
        "and do not add explanations or bullet points."

    userContent = buildShoppingPreferenceSummaryUserContent(in)
} else {
    systemMessage = "You are a movie preference summarizer for a recommendation engine. " +
        "Given a user's profile and watch history, respond with exactly ONE short English sentence " +
        "that describes the kind of movies they are most likely looking for. " +
        "Focus on genres (for example: animation, kids, family, thriller, romance), tone " +
        "(light-hearted, dark, emotional, etc.), and content level (kid-friendly, avoid violence, etc.). " +
        "Do not address the user directly and do not add explanations or bullet points."
```

End-to-End Vector Recommendation Workflow

- User submits profile, preferences, and watchlist/cart
- Backend validates and normalizes input
- Request routed to movie/shopping vector pipeline
- User and catalog items embedded
- User profile vector construction
- Candidate items scored using cosine similarity
- Deterministic filtering and re-ranking applied
- Top-K recommendations returned to the client

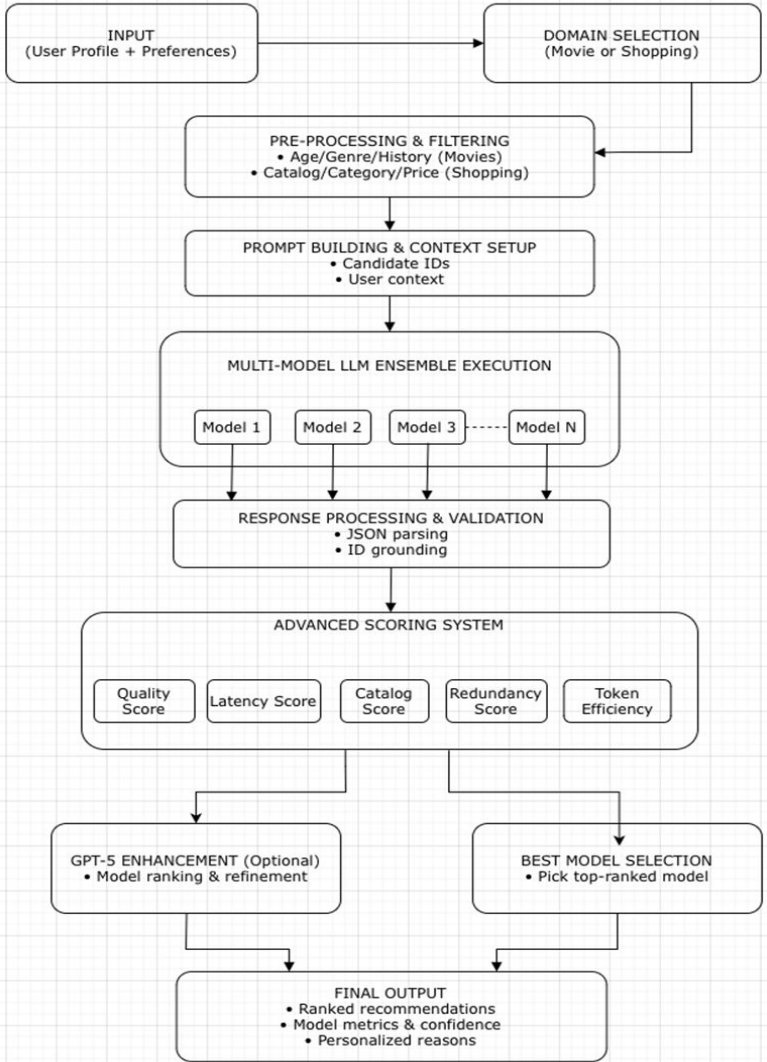


LLM Recommendation Engine Overview

- Built an AI-powered recommendation system for **movies and shopping**
- Designed as a **backend-first, production-style system**
- Uses **multiple LLMs** to generate and compare recommendations
- Focused on **reliability, consistency, and evaluation**
- Supports real user behavior like watch history and shopping carts

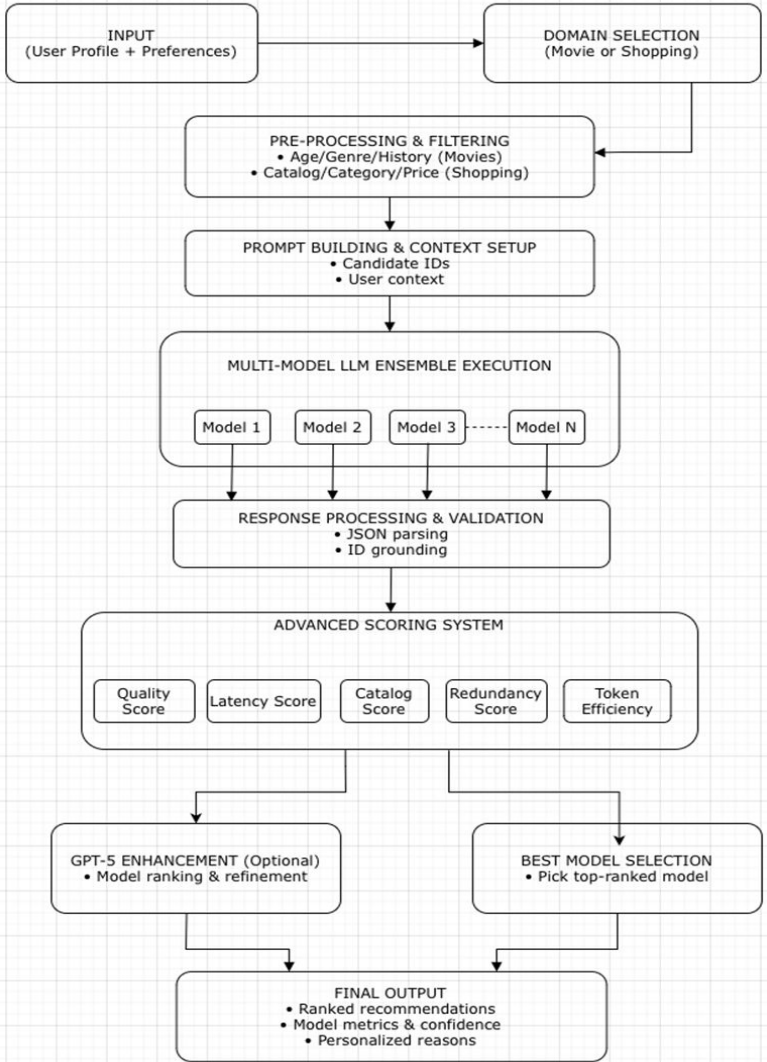
End-to-End Recommendation Workflow

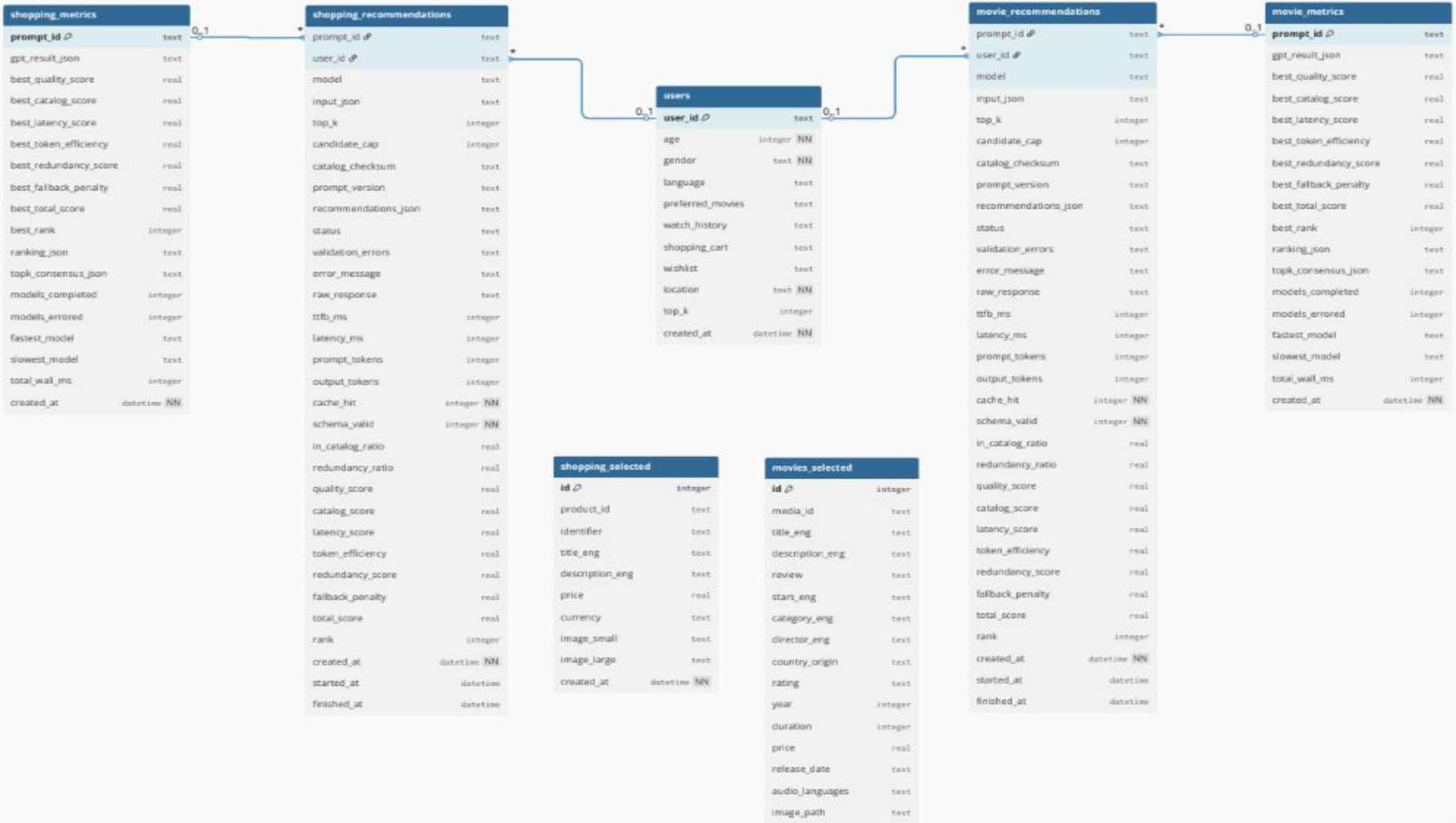
- User submits profile, preferences, and history
- Backend validates and normalizes input
- Candidate items fetched from catalog
- Candidates pre-filtered before LLM execution
- Prompt constructed with strict rules
- Multiple LLMs run in parallel
- Results validated, stored, and returned



LLM Execution & Result Handling

- Deterministic preprocessing for repeatable runs
- Parallel model execution with per-model isolation
- Timeout and failure handling per LLM
- Schema validation and grounding against catalog
- Centralized persistence of outputs and metrics




shopping_metrics
prompt_id text
gpt_result_json text
best_quality_score real
best_catalog_score real
best_latency_score real
best_token_efficiency real
best_redundancy_score real
best_fallback_penalty real
best_total_score real
best_rank integer
ranking_json text
topk_consensus_json text
models_completed integer
models_errored integer
fastest_model text
slowest_model text
total_wall_ms integer
created_at datetime NN
0..1
*
shopping_recommendations
prompt_id text
user_id text
model text
input_json text
top_k integer
candidate_cap integer
catalog_checksum text
prompt_version text
recommendations_json text
status text
validation_errors text
error_message text
raw_response text
ttfb_ms integer
latency_ms integer
prompt_tokens integer
output_tokens integer
cache_hit integer NN
schema_valid integer NN
in_catalog_ratio real
redundancy_ratio real
quality_score real
catalog_score real
latency_score real
token_efficiency real
redundancy_score real
fallback_penalty real
total_score real
rank integer
created_at datetime NN
started_at datetime
finished_at datetime
*
0..1
users
user_id text
age integer NN
gender text NN
language text
preferred_movies text
watch_history text
shopping_cart text
wishlist text
location text NN
top_k integer
created_at datetime NN
0..1
*
movie_recommendations
prompt_id text
user_id text
model text
input_json text
top_k integer
candidate_cap integer
catalog_checksum text
prompt_version text
recommendations_json text
status text
validation_errors text
error_message text
raw_response text
ttfb_ms integer
latency_ms integer
prompt_tokens integer
output_tokens integer
cache_hit integer NN
schema_valid integer NN
in_catalog_ratio real
redundancy_ratio real
quality_score real
catalog_score real
latency_score real
token_efficiency real
redundancy_score real
fallback_penalty real
total_score real
rank integer
created_at datetime NN
started_at datetime
finished_at datetime
*
0..1
movie_metrics
prompt_id text
gpt_result_json text
best_quality_score real
best_catalog_score real
best_latency_score real
best_token_efficiency real
best_redundancy_score real
best_fallback_penalty real
best_total_score real
best_rank integer
ranking_json text
topk_consensus_json text
models_completed integer
models_errored integer
fastest_model text
slowest_model text
total_wall_ms integer
created_at datetime NN
shopping_selected
id integer
product_id text
identifier text
title_eng text
description_eng text
price real
currency text
image_small text
image_large text
created_at datetime NN
movies_selected
id integer
media_id text
title_eng text
description_eng text
review text
stars_eng text
category_eng text
director_eng text
country_origin text
rating text
year integer
duration integer
price real
release_date text
audio_languages text
image_path text

Why Schema-Driven Design Matters

- Client required a **simple, lightweight database** with minimal operational overhead
- Embedded design supports **offline and edge environments** (e.g., inflight systems)
- Database acts as both **persistent storage and caching layer**
- **Users table** stores profile, preferences, history, cart, and wishlist
- **Catalog tables** provide a trusted, curated source for valid recommendations
- **Recommendation tables** store per-model outputs, raw responses, and metrics
- **Metrics tables** enable scoring, ranking, and model comparison
- Prompt-based structure enables **auditability, reproducibility, and caching**

Candidate Pre-Filtering

LLMs reason over **filtered candidates**, not full catalog

Reduces candidate pool from 290+ to ~50–100 items

Filtering criteria:

- Age and safety constraints
- Watch history exclusion
- Genre/category relevance
- Metadata quality checks

Improves accuracy, reduces hallucinations, lowers cost

Prompt Engineering: Purpose & Structure

- LLM operates as a **backend execution component**
- Prompt explicitly defines **system role and output constraints**
- Strict **JSON schema** enforced for all model responses
- User context included with **explicit behavioral interpretation rules**
- Recommendation choices limited to **pre-approved catalog IDs**
- Mandatory personalization, ordering, and diversity rules applied
- Uniform prompt structure enables **consistent validation and comparison**

Model Scoring Framework

- Objective: automated evaluation of multiple LLMs
- Designed for production reliability, not intuition
- Same inputs → fair comparison across models

Multi-Dimensional Scoring

Score	Weight	Definition
Quality Score	35%	Measures how relevant, personalized, diverse, and novel the recommendations are for the user.
Genre Match Score	40% of Quality	Measures alignment between user watch history and recommended content using cosine similarity.
Reasoning Score	25% of Quality	Evaluates explanation quality; penalizes generic or fallback reasoning.
Diversity Score	20% of Quality	Rewards variety by measuring unique genres across recommendations.
Novelty Score	15% of Quality	Encourages discovery by rewarding recommendations not seen in watch history.
Latency Score	35%	Rewards faster models; scores normalized relative to the fastest model per run.
Catalog Score	10%	Ensures all recommended items exist in the approved catalog; penalizes hallucinated items.
Token Efficiency Score	10%	Rewards models that generate concise outputs close to the optimal token range.
Redundancy Score	10%	Penalizes duplicate recommendations and excessive repetition.

AUTOMATED SCORING & GPT ADJUDICATION

- All models run in parallel
- Scores computed automatically
- **Primary selection: score-based ranking**
- **Optional GPT-5 adjudication for semantic review**

GPT-5 ADJUDICATION

Score	Definition
Personalization Score	Measures how well recommendations align with user demographics, preferences, and watch history.
Content Quality Score	Evaluates intrinsic recommendation quality, including relevance, diversity, and novelty.
Technical Excellence Score	Assesses correctness of output, reasoning clarity, format compliance, and catalog grounding.
Business Value Score	Estimates practical value based on user satisfaction and engagement potential.
Overall GPT Score	Weighted aggregate of all GPT evaluation dimensions, used for final model ranking.
Confidence Score	Indicates GPT's confidence in its evaluation and ranking decision (0.0–1.0).

FINAL MODEL SELECTION

- **Primary:** phi3:mini
- **Fallback:** qwen2.5:3b-instruct
- Best speed-to-quality tradeoff
- Most consistent latency under SLA

Evaluated across 10 personas

TEAM ACCOMPLISHMENTS

- Built a **production-grade, multi-model LLM evaluation system**
- Designed a **multi-dimensional scoring framework** balancing quality, speed, and reliability
- Implemented **fully automated best-model selection** with explainable metrics
- Added **optional GPT-5 adjudication** without making it a single point of failure
- Delivered an **offline-capable, auditable, and deployment-ready** backend
- Implemented vector search with very accurate results.

FRONTEND

- **Ideanova Recommendation Hub**: AI-driven platform for personalized movie & shopping recommendations
- **Core Functionality**: Profile-based queries with multi-model AI (LLM + Vector search)
- **User Flow**: Set profile → Choose mode (Movies/Shopping) → View & compare real-time outputs
- **Target**: Get accurate and user preferred recommendatiosn

Key Features and Functions

- **Personalization Engine**: Tailored recs using age, gender, watch/cart history
- **Multi-Model Comparison**: Run & rank LLMs alongside Vector recs
- **History Tracking**: Real-time watch/cart updates with progress sliders
- **Leaderboard**: Previous runs with user inputs and outputs
- **Export CSV**
- **Stop Run-** Stops the current LLM run

EXPORT CSV

1	USER PROFILE													
2	Run ID	16f73a2e-633f-4bbd-87eb-7bff9d96817e												
3	User ID	9a71c266-8891-4310-8cc2-cb83e1916f72												
4	Age	22												
5	Gender	female												
6	Location	United States												
7	Top K	10												
8	Preferred Movies													
9	Started At	2025-12-17 10:51:10.618177 +0000 UTC												
10	Finished At	2025-12-17 10:55:00.724637 +0000 UTC												
11	Total Duration (ms)	230175												
12	Best Model	qwen2.5:1.5b-instruct												
13	Models Used	llama3.2:1b,llama3:latest,phi3:mini,qwen2.5-coder:1.5b,qwen2.5:1.5b-instruct,qwen2.5:3b-instruct,tinyllama:1.1b												
14														
15	TOP CONSENSUS RE	Support Count												
16	T-hairspray	7												
17	T-those_who_wish_n	7												
18	T-for_love_life_no_or	5												
19	int_02045	5												
20	T-barbie	4												
21	T-elvis	4												
22	int_018	3												
23	T-fantastic_beasts_a	2												
24	T-42	2												
25	int_02046	2												
26														
27	MODEL RESULTS													
28	Model	Rank	Total Score	Quality Score	Catalog Scor	Latency Scor	Token Efficie	Redundancy	Fallback Pen	Latency (ms)	Prompt Toker	Output Toker	In Catalog Ra	Redundancy	Recommendations	
29	qwen2.5:1.5b-instruc	1	100.37	104.5	100	100	88	100	20	19205	2898	240	10	0	1. int_02045: Matches profile | 2. T-barbie: Matches profile | 3. int_018: Matche	
30	qwen2.5-coder:1.5b	2	96.35	93.5	100	99.5	88	100	60	19301	2898	240	10	0	1. T-42: Matches profile | 2. T-rebel_without_a_cause: Matches profile | 3. T-aqu	
31	phi3:mini	3	82.45	117.5	100	35.78	88	100	60	53677	3532	240	10	0	1. int_02068: Matches profile | 2. T-the_lord_of_the_rings_the_two_towers: Mat	
32	tinyllama:1.1b	4	75.65	92.5	100	44.22	78	100	100	43430	2048	190	10	0	1. T-those_who_wish_me_dead: fallback pick | 2. T-hairspray: fallback pick | 3. T	
33	qwen2.5:3b-instruct	5	72.85	102.5	100	23.36	88	100	20	82219	2898	240	10	0	1. T-hairspray: Matches profile | 2. T-creed: Matches profile | 3. int_02045: Matc	
34	llama3.2:1b	6	63.19	74.5	100	23.76	88	100	20	80837	2770	240	10	0	1. T-300_rise_of_an_empire: Matches profile | 2. T-the_hobbit_the_battle_of_the	
35	llama3:latest	7	58.67	77	100	8.35	88	100	40	230081	2755	240	10	0	1. T-the_notebook: Matches profile | 2. T-fantastic_beasts_and_where_to_find_t	
36																

Highlights

- React for modular and Framer Motion for smooth UI (hover effects,model comparison)
- LLM and Vector-based recommendation
- Reusable UI elements and shared logic
- Metrics for each and every model
- System supports running up to 7 AI models per user
- Execution completes in ~3–7 minutes

CHALLENGES FACED

Different models returned data in different formats, so we handled multiple field names and added safe defaults in the UI.

Some recommendations had missing or broken images, so we added fallback images to prevent the page from breaking.

Some models returned empty or failed results, so we showed clear “no recommendations” messages instead of crashing.

INDIVIDUAL CONTRIBUTIONS

- **Honey Reddy Nagireddy:** Backend architecture, database design, LLM & shopping recommendation integration, caching and optimization, API development, prompt engineering, and system documentation
- **Ssai Sanjanna Ganji:** Backend architecture, LLM scoring and ranking framework, automated model selection, GPT-5 adjudication, system integration, documentation
- **Preethi Reddy Nagireddy**: Responsible for building the front end of the project, implementing all required features from user interaction and personalization to AI recommendations, model comparison, and result analysis, documentation
- **Siddhartha Gudipudi:** Initial research on Ollama, python scripts to run LLM's one after other via Ollama, Vector architecture, Vector implementation, documentation

DEMO