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",...]}`

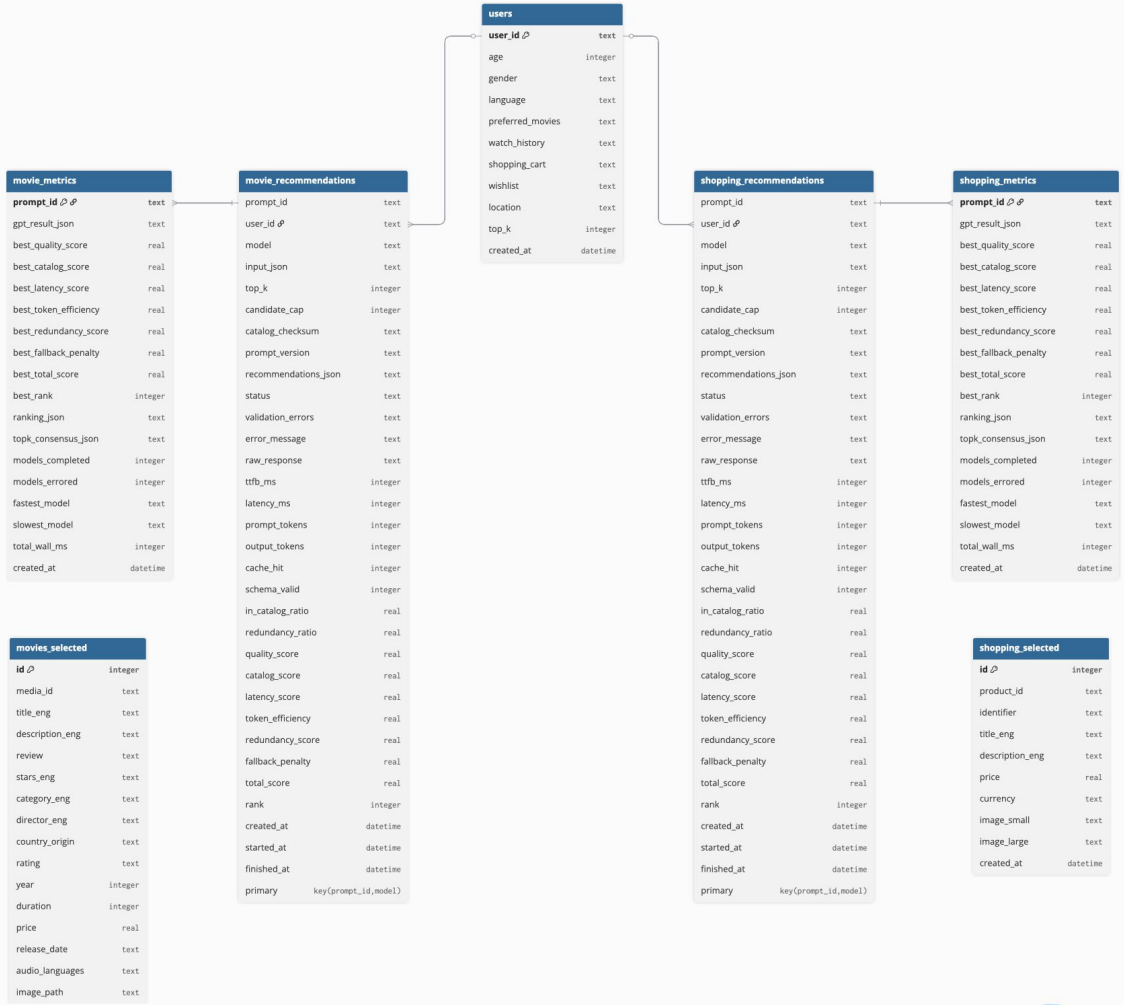
userJSON, _ := json.Marshal(payload)

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.

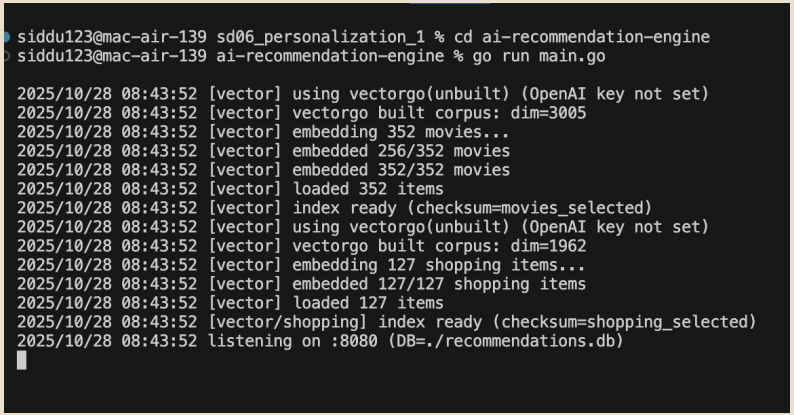
```
siddu123@MacBook-Air ai-recommendation-engine % go run main.go

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.

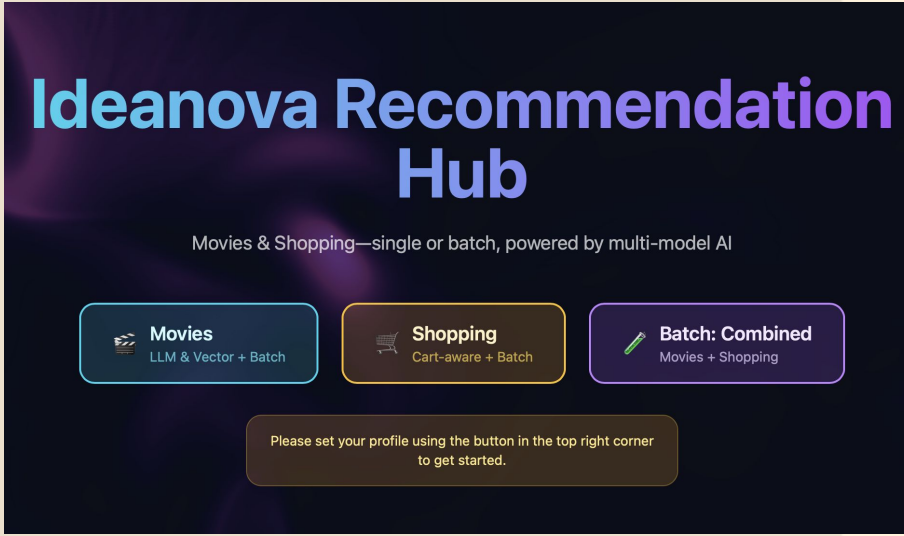

```
        ],
        "reason": "Genres: Crime, Drama | Fits your top genres: drama, action | Strong match to your watchlist tastes
| Great pick for crime fans. | Genre mix enforced"
```

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.

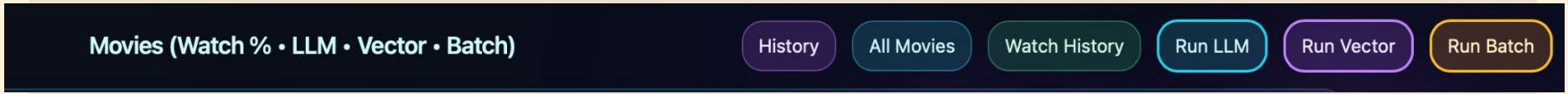
Frontend - Design and Structure

- 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



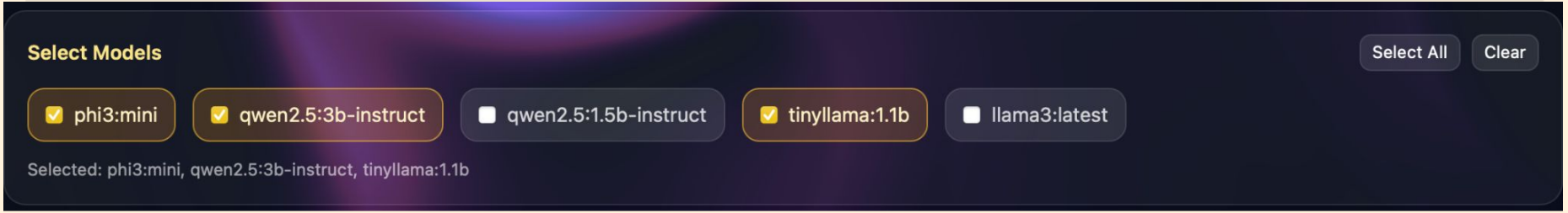
Movies Interface

- 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

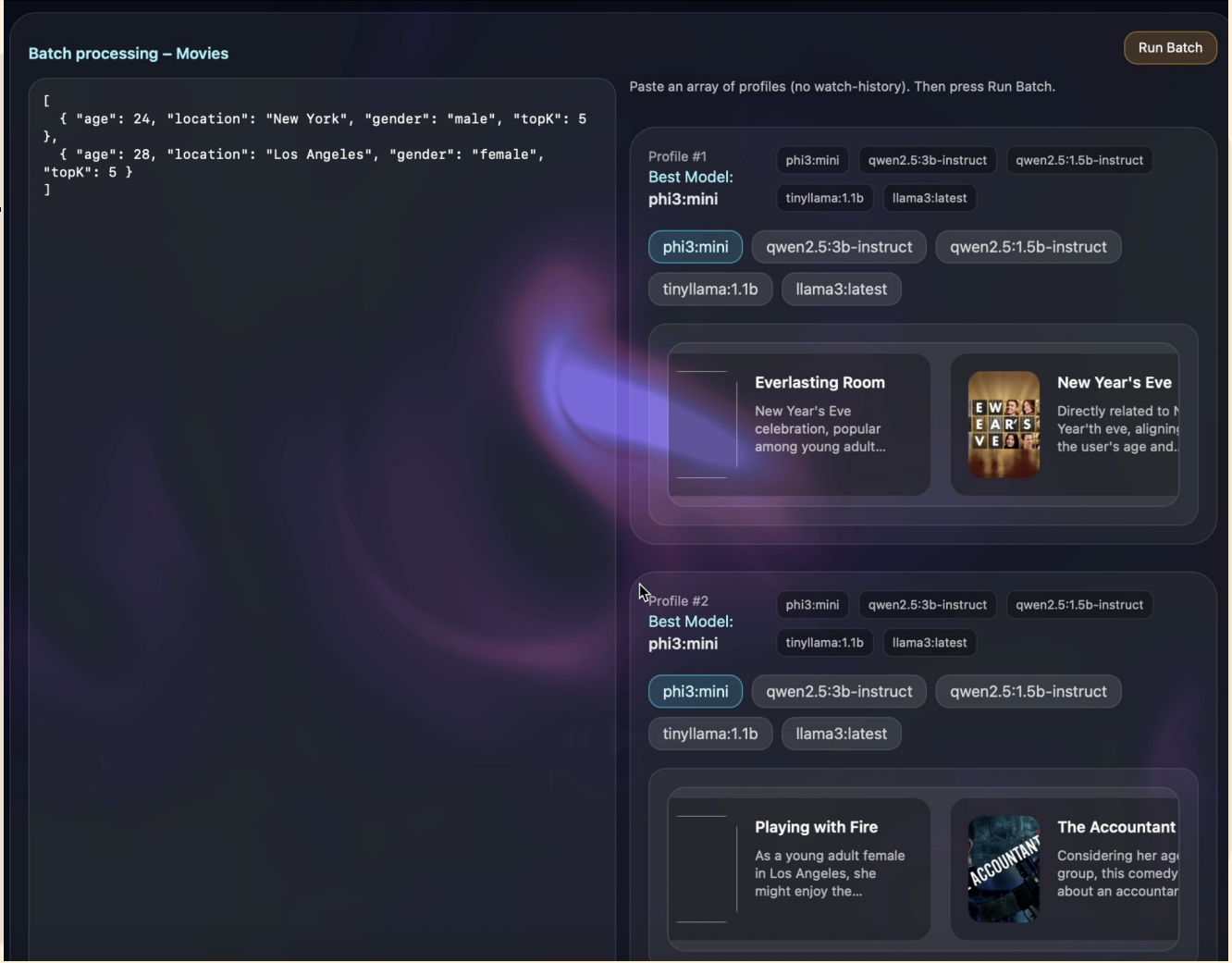
- Combined Batch results showing raw data - add refine parsing and formatting in frontend
- Earlier JSON parsing bug → fixed by input sanitization and parsing
- Different API formats → handled with normalizers
- Slow multi-model responses → plan for selective model runs for both movies and shopping for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

Future Works

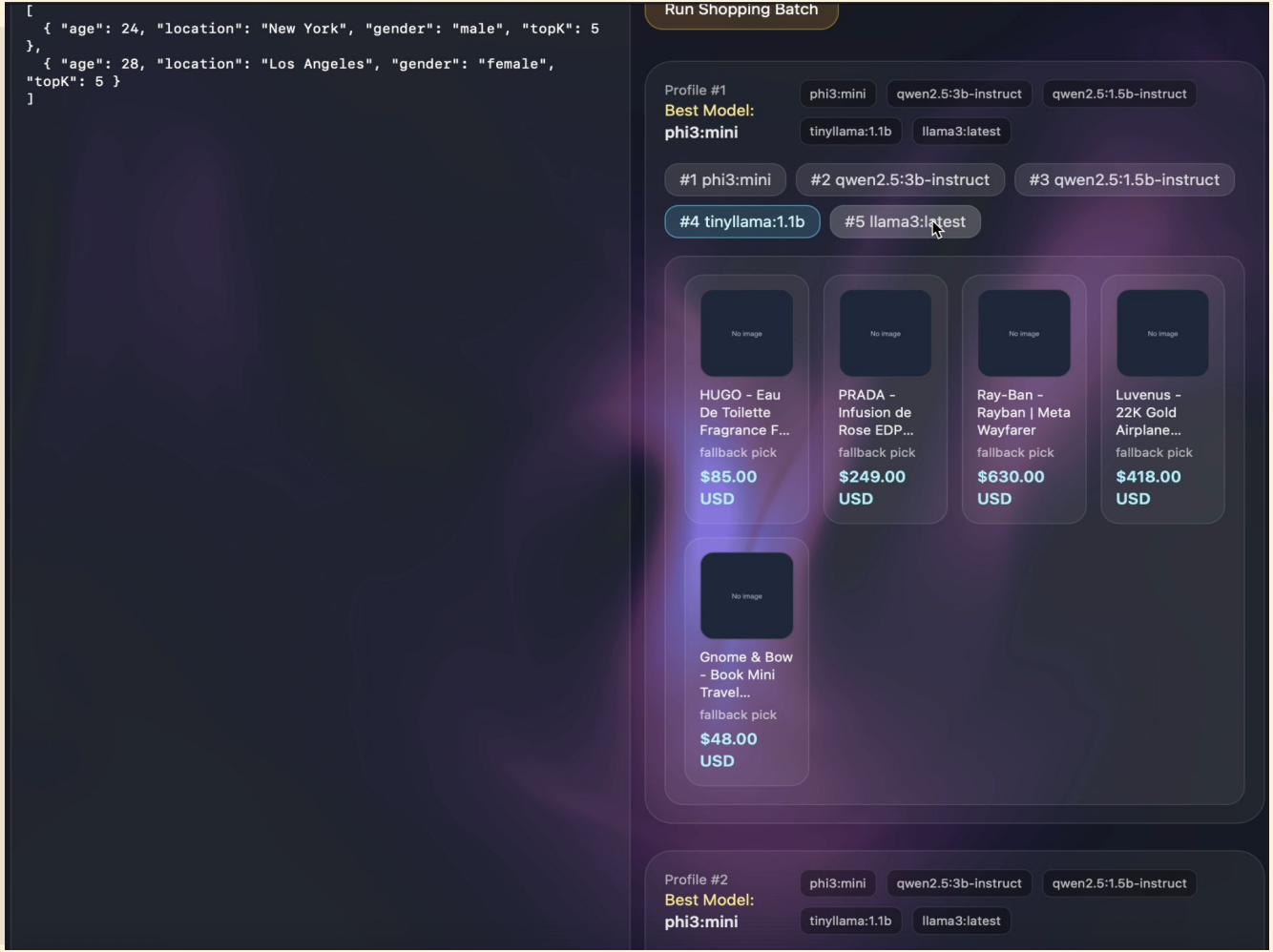
- Next: Vector engine + history for shopping
- Add metrics panel (latency, quality, tokens)
- Containerize Docker
- Changes to frontend UI for efficiency
- Documentation
- Cross functioning recommendation system for movies and shopping
- Better offline semantic embedding model for vector engine

DEMO

Batch Processing-Movies



Batch Processing-Shopping



Thank You - Team 6