VECTOR RECOMMENDATION ENGINE

Table of content

1. Executive Summary ---------- 2
2. Role in the System ---------- 2
3. Key Features ---------- 2
  3.1 Dual Embedding System ---------- 2
  3.2 Shared Pipeline for Movies and Shopping ---------- 3
  3.3 Deterministic Scoring ---------- 3
  3.4 Graceful Degradation ---------- 3
Files Overview (What Each File Does) ---------- 3
4. Architecture Overview ---------- 4
  4.1 User Input Received ---------- 5
  4.2 Request Handling (Movies or Shopping) ---------- 5
  4.3 Embedding Layer Initialization ---------- 5
  4.4 User Vector Construction ---------- 5
  4.5 Candidate Scoring ---------- 5
  4.6 Post-Processing Logic ---------- 6
  4.7 Top-K Recommendation Output ---------- 6
5. Core Components ---------- 6
  5.1 OpenAI Embedder ---------- 6
  5.2 VectorGo TF-IDF Embedder ---------- 6
  5.3 Catalog Loading ---------- 7
  5.4 User Vector Construction ---------- 7
6. Candidate Scoring ---------- 7
  6.1 Similarity Components ---------- 7
  6.2 Boosting Signals ---------- 8
  6.3 Filtering Rules ---------- 8
  6.4 Genre Diversity Control (Movies Only) ---------- 8
7. Endpoints ---------- 8
8. Troubleshooting ---------- 9
  8.1 Movies Vector Engine (Watchlist-based) ---------- 9
  8.2 Shopping Vector Engine (Cart-based) ---------- 10
  8.3 Running the Server with OpenAI (Environment Variables) ---------- 11
9. Future Work ---------- 12
10. Things tried ---------- 12

1. Executive Summary:

The Vector Recommendation Engine is a deterministic, embedding-based system that generates personalized movie and shopping recommendations without relying on large language models. It supports two domains:

- Movies Vector Engine: /api/v1/recommendations/vector
- Shopping Vector Engine: /api/v1/shopping/recommendations/vector

The engine embeds catalog items, user demographics, and user behavior (watch history or shopping cart) using either OpenAI embeddings or a local TF-IDF embedder. It produces fast, stable, mathematically explainable rankings and always remains operational even without external APIs.

Key goals:

- Provide low-latency recommendations
- Offer full offline capability
- Produce consistent results independent of LLM variability
- Serve as the fallback backbone of the entire recommendation system

2. Role in the System:

The vector engine is responsible for:

- Embedding items and user signals into a shared vector space
- Computing similarity scores between users and catalog items
- Ranking items using closeness of the user and catalog vectors
- Providing recommendations even if LLM services fail
- Uses OpenAi based embedding if the KEY and internet exists.
- Uses the fall back TF-IDF search if KEY or internet isn't available

It is the reliable core of the recommendation system.

3. Key Features

3.1 Dual Embedding System

- OpenAI Embeddings (primary): High-quality semantic vectors (1536-dim).
- VectorGo TF-IDF (fallback): Local, deterministic, fast embedding with no external dependencies.

3.2 Shared Pipeline for Movies and Shopping

- Both domains use the same vector ranking logic.
- Movies use genre mixing and MMR; shopping does not.

3.3 Deterministic Scoring

- All ranking operations use cosine similarity and well-defined blending weights.

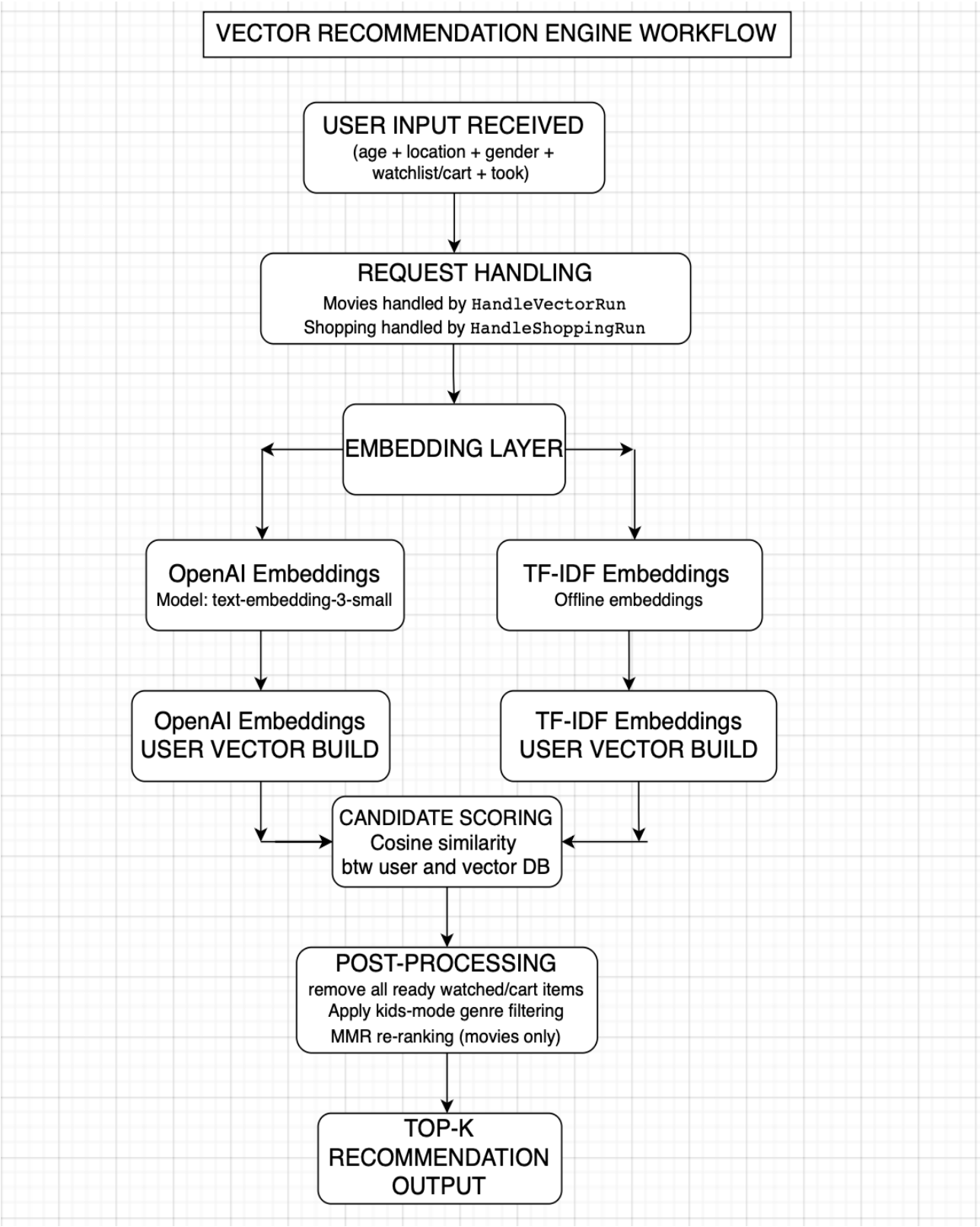
3.4 Graceful Degradation

- If OpenAI fails → switch to TF-IDF.
- If the watchlist is missing → rely on demographics.
- If both are missing → still produce stable recommendations(random).

Files Overview (What Each File Does)

- ***embedder_openai.go*** – Sends batch requests to OpenAI Embeddings API, normalizes vectors, and maps batch indices back to items. Activated only when API key + internet are available.
- ***vector_go_embedder.go*** – Implements the offline TF-IDF embedder. Builds vocabulary, computes TF-IDF weights, and normalizes vectors. Used automatically as fallback when OpenAI cannot run.
- ***engine.go*** – Main orchestration layer. Loads catalog, embeds items, constructs user vectors, computes similarity scores, applies genre rules/MMR, and returns Top-K results.
- ***shopping.go*** – Converts shopping cart input into the unified RunInput format and uses the same scoring pipeline (without MMR/genre mixing).
- ***helpers.go*** – Contains utilities for cosine similarity, normalization, tokenization, MMR reranking, franchise de-clumping, and text cleanup.
- ***llm_preferences.go*** – Generates a short LLM-based preference sentence that improves personalization. Embedded and blended with the main profile vector when available.

4. Architecture Overview



4.1. User Input Received

- Inputs include age, gender, location, watchlist/cart, and topK.
- This request enters the server through the routing layer.
- File: main.go / routes.go

4.2. Request Handling (Movies or Shopping)

- Movies are processed by HandleVectorRun.
- Shopping requests are processed by HandleShoppingRun.
- Both functions convert incoming JSON into the internal RunInput structure.
- Files: engine.go, shopping.go

4.3. Embedding Layer Initialization

Determines embedding mode automatically:

- If OPENAI_API_KEY exists → use OpenAI Embedder
- File: embedder_openai.go
- Otherwise → use TF-IDF fallback embedder
- File: vector_go_embedder.go

Embeds the following components:

- Catalog item text
- Watchlist or cart items
- Demographic sentence
- Optional LLM preference sentence
- File: llm_preferences.go

4.4. User Vector Construction

- Builds a weighted average of the watchlist/cart embeddings.
- Creates demographic embedding and blends it.
- Optionally adds the LLM preference vector.
- Produces a normalized “user profile vector” used for scoring.
- File: engine.go

4.5. Candidate Scoring

For every catalog item, calculates:

- Cosine similarity to the user profile vector
- Cosine similarity to demographic vector
- Token overlap boosts
- Genre preference boosts
- Recency boost (movies only)
- Uses fixed blending weights (a_watch, a_demo) for consistency.
- File: engine.go

4.6. Post-Processing Logic

- Filters invalid or previously watched items.
- Applies kids-mode genre restrictions when needed.
- Removes franchise clustering (de-clumping).
- Applies genre diversity rules and MMR re-ranking (movies only).
- Files: helpers.go, engine.go

4.7. Top-K Recommendation Output

- Scores are sorted and the top K ranked items are returned.
- This forms the final vector-based recommendations sent to the client.
- File: engine.go

5. Core Components

5.1 OpenAI Embedder

- Sends batch embedding requests.
- Normalizes vectors.
- Maps OpenAI batch indices back to original items.
- Becomes active when the API key exists.

5.2 VectorGo TF-IDF Embedder

Builds corpus vocabulary from all catalog texts.

Computes term frequency and IDF weights.

Produces consistent and explainable embeddings.

Becomes active when OpenAI keys are missing or during fallback(no internet).

5.3 Catalog Loading

Movies Loader:

- Loads movies_selected or movies_all.
- Extracts title, description, genre, director, stars, and images.
- Merges text using concatText.
- Embeds catalog in batches of 64.
- Applies fallback to TF-IDF if needed.

Shopping Loader:

- Loads shopping_selected.
- Converts identifiers to genre lists.
- Stores small/large image URLs.
- Embeds items exactly like the movie engine.

5.4 User Vector Construction

Demographic Vector:

- Embeds a sentence such as: "gender: female location: US age: 30"

Watchlist / Cart Vector:

- For movies: weighted by watched percent.
- For shopping: every cart item is treated as 100%.

LLM Preference Vector (optional):

- A single summarizing sentence describing user taste.
- Embedded and blended with the profile vector if dimensions match.

6. Candidate Scoring

6.1 Similarity Components

Each item receives:

- Similarity to watch/profile vector

- Similarity to demographic vector
  Weighted by alphaWatch and alphaDemo, depending on watchlist size.

6.2 Boosting Signals

- Title token overlap with watchlist.
- Genre preference weights derived from the user's watched items.
- Recency boost for younger users.

6.3 Filtering Rules

Movies:

- Remove previously watched titles.
- Remove non-movie categories.
- Restrict to family/kids genres for younger users(12- age).
- Apply franchise de-clumping.

Shopping:

- No genre mixing or MMR.
- A simple top-K sort by score.

6.4 Genre Diversity Control (Movies Only)

- Genre-mix selection: prevents too many titles from one genre.
- MMR reranking: balances relevance and diversity.

7. Endpoints

Movies Vector API:
POST /api/v1/recommendations/vector
Input:
{
"age": 25,
"gender": "male",
"location": "USA",
"watchlist": [...],

"topK": 10
}

Shopping Vector API: POST /api/v1/shopping/recommendations/vector
Input:
{
"age": 22,
"gender": "female",
"location": "India",
"cart": [...],
"topK": 10
}

Shopping requests are internally converted to RunInput.

8. Troubleshooting(Problems I ran Into):

Problem: OpenAI errors or missing vectors
Cause: No API key, rate limiting, network issues, Export KEY before running main through terminal.
Fix: Set OPENAI_API_KEY or rely on TF-IDF fallback (already automatic)

Problem: Empty recommendations
Cause: Watchlist not matching catalog, empty embeddings
Fix: Ensure titles match catalog exactly, add more inputs

Problem: Relevance feels off
Cause: Weak watchlist signal
Fix: LLM preference vector improves profile quality(already automatic)

Problem: How to run without frontend: using curl

8.1 Movies Vector Engine (Watchlist-based)

Command:
curl -s http://localhost:8080/api/v1/recommendations/vector
-H "Content-Type: application/json"
--data @req.json | jq .

Example req.json file:

```
{
"age": 30,
"gender": "female",
"location": "US",
"topK": 10,
"watchlist": [
{ "title": "The Big Sick", "percent": 100 },
{ "title": "You're Cordially Invited", "percent": 95 },
{ "title": "All Summers End", "percent": 90 },
{ "title": "Companion", "percent": 85 },
{ "title": "Solis", "percent": 80 },
{ "title": "Future World", "percent": 75 }
]
}
```

8.2. Shopping Vector Engine (Cart-based)

Command:

```
curl -s http://localhost:8080/api/v1/shopping/recommendations/vector
-H "Content-Type: application/json"
--data @shop_req.json | jq .
```

Example shop_req.json file:

```
{
"age": 28,
"gender": "male",
"location": "US",
"cart": ["wireless earbuds", "usb c charger", "iphone case"],
"topK": 10
}
```

Notes:

- jq is used only for pretty-printing; output works without it.
- Both files (req.json and shop_req.json) must be in the same directory where you run curl.
- Make sure the backend server is running on port 8080.

8.3 Running the Server with OpenAI (Setting Environment Variables)

To run the Vector Recommendation Engine using OpenAI embeddings, the following environment variables must be set **before starting the server**:

export OPENAI_API_KEY="your-key-here"

export OPENAI_EMBED_MODEL="text-embedding-3-small"

export DB_PATH="./recommendations.db"

export ADDR=":8080"

Example :

cd ai-recommendation-engine

export OPENAI_API_KEY="sk-proj-xxxx"

export OPENAI_EMBED_MODEL="text-embedding-3-small"

export DB_PATH="./recommendations.db"

Then start the server: go run main.go

Example Output:
2025/12/08 10:28:05 [vector] using openai(text-embedding-3-small)

2025/12/08 10:28:05 [vector] embedding 290 movies...

2025/12/08 10:28:12 [vector] embedded 256/290 movies

2025/12/08 10:28:13 [vector] embedded 290/290 movies

2025/12/08 10:28:13 [vector] loaded 290 items

2025/12/08 10:28:13 [vector] index ready (checksum=movies_selected)

2025/12/08 10:28:13 [vector] using openai(text-embedding-3-small)

2025/12/08 10:28:13 [vector] embedding 127 shopping items...

2025/12/08 10:28:16 [vector] embedded 127/127 shopping items

2025/12/08 10:28:16 [vector] loaded 127 items

2025/12/08 10:28:16 [vector/shopping] index ready (checksum=shopping_selected)

2025/12/08 10:28:16 listening on :8080 (DB=./recommendations.db)

-> If you miss the export commands for OpenAI tokens, TF-IDF helps as backup and still gets you good recommendations

9. Future Work:

- A standalone C file using OpenMP has been included in the project directory as a reference for future optimization of large-scale vector search.
- This code is not part of the running system; it is provided only as an example showing how multi-threaded cosine similarity search can be reduce the latency
- Preliminary tests showed that the OpenMP version performed about more than 25% faster than a standard serial loop when searching across large embedding pools.
- If the catalog grows significantly, similar parallel techniques may be incorporated into the production vector engine to reduce latency.

10. Things tried

As part of the system's architectural evolution, multiple vector-based approaches were tested to balance accuracy, offline operability, latency, and cost. The vector engine became the core fallback system and a high-performance alternative to full LLM-driven recommendations. Below is a summary of the experimentation and reasoning behind the final vector architecture.

1. ***Evaluation of Different Embedding Models***
   Initial tests used the MiniLM-L6-v2 model locally. While accuracy was not great and also led two practical issues emerged:

- Running MiniLM inference on Raspberry Pi class devices will be difficult(from my research and exp) due to memory constraints.
- Even on laptops, creating large embedding files and running inference repeatedly added overhead.

  (But I still have that code in my branch and if want to test feel free to do so)

  (branch name: VectorAndIndexing, under the VecIndex folder.)

  Online OpenAI embeddings produced much higher-quality semantic vectors, were more stable, and improved overall ranking relevance.
  Final decision:

- Use OpenAI embeddings when internet access is available.
- Provide a pure offline fallback using TF-IDF, which is extremely fast, requires no external dependencies, and ensures the system remains fully functional without connectivity.

This dual-mode design gives the client both reliability and flexibility: online accuracy with offline safety.

2. ***Introduction of the LLM Preference Sentence***
   A short descriptive user-preference sentence was introduced and embedded into the vector pipeline.
   Observed benefits:
- Increased personalization accuracy.
- 2–3 recommendations in the Top-10 list consistently shifted closer to the user's actual viewing taste.
- Provided a soft semantic hint that strengthened the watchlist vector.

  Drawbacks:

- Additional token usage per request increases operational cost.
- The overall pipeline now requires one extra API call before computing similarity, which slightly increases latency .

  Final decision:

- Keep the preference sentence as an optional enhancement, enabled only when high accuracy is preferred over minimal cost.

3. ***Attempt to Generate Preference Sentence Offline (Ollama)***
   We attempted generating the same preference summary sentence from an offline model via Ollama.
   Outcome:
- The offline generation took approximately 15 seconds for each request.
- This defeated the purpose of using vector search, which is designed to return results in under 2 seconds.
- At that latency, we may as well use full LLM recommendation generation instead of vector search.

  Final decision:

- Do not generate the preference sentence through offline Ollama.
- Maintain the fast vector pipeline, with online preference generation only when required.

4. *Resulting Final Architecture for Vector Search*

- Online: OpenAI embeddings + optional preference sentence → highest accuracy.
- Offline: TF-IDF embedding engine → guaranteed sub-second runtime with no dependencies.
- Preference sentences are optional to control cost vs. quality.
- The system automatically falls back to TF-IDF if any embedding call fails.

This design ensures:

- Stability
- Low latency
- Offline availability
- Configurable accuracy levels
- Predictable cost management