LLM-Based Recommender System Prototype

1. High-level idea of the application

This application is a prototype of an online retail storefront where:

1. Recommendations are generated using **embeddings** (vector representations of products), not handcrafted rules.
2. The user's **purchase history** is used to build a user profile in the same embedding space (a **user embedding**).
3. The system selects products that are **semantically similar** to the user's interests.
4. A separate **LLM model** generates **human-readable explanations** ("we recommend this because…") which are shown as tooltips in the UI.

The system is essentially **content-based / embedding-based** (using product descriptions), but grounded in **real user behavior** (which user bought which items – collaborative signal).

---

2. Data and offline preparation (Offline stage)

2.1. Purchase history: user_purchases.json

We use the **Online Retail** dataset (~400k transactions).

1. We clean the data:
    - remove rows without CustomerID or StockCode;
    - remove returns (invoice numbers like C...);
    - remove negative quantities.
2. For each CustomerID, we collect a list of **unique StockCode values** (product IDs).

This is saved as backend/data/user_purchases.json.
From a recommender-systems perspective, this is a **user–item matrix in compact form**: for each user we store which items they have purchased.

2.2. Product embeddings: product_embeddings.json

Next, we process all products:

1. From the CSV we extract (StockCode, Description) pairs.

2. We drop empty descriptions and duplicates.
3. For each product we call a **Cloud.ru embedding model** Qwen/Qwen3-Embedding-0.6B:
    - Input: the product description text;
    - Output: a vector of length **1024** (embedding dimension = 1024).
4. We store the results as:


```
{
  "85123A": {
    "product_id": "85123A",
    "description": "WHITE HANGING HEART T-LIGHT HOLDER",
    "embedding": [0.0123, -0.0456, "... 1024 values ..."]
  },
  "...": {
    "product_id": "...",
    "description": "...",
    "embedding": ["..."]
  }
}
```


This is backend/data/product_embeddings.json.

In recommender-system terms:

- We build a **content-based representation** for each product.
- Each embedding is a point in a **1024-dimensional semantic space**: products with similar meaning are located close to each other.

Important:

- **Embeddings are computed offline, once.**
- At runtime we only load these precomputed vectors → online recommendations are fast.

---

3. How recommendations are computed online (Online stage)

When a user opens the site and selects themselves in the dropdown, the frontend calls the API:

GET /api/users/{user_id}/recommendations?top_n=12

Inside the backend (FastAPI), the following happens.

3.1. Retrieve user's purchase history

We read from user_purchases.json:

items_bought = user_purchases[user_id] # list of product_ids

In parallel:

- We fetch the **descriptions** of these products from product_embeddings.json.
- We display them in the **"Previous purchases"** section.
- We also use them as context for the LLM explanations.

3.2. Build the user embedding

This is the key step in the recommendation logic.

1. For each product the user has bought, we take its embedding vector:

v1, v2, v3, ... (each is length 1024)

2. We compute the **average**:

user_vec = mean(v1, v2, v3, ...)

3. We **normalize** it (divide by its norm) to make cosine similarity easier to compute.

Intuitively:

The user embedding is the "center of mass" of the products they have bought. It is a compact numerical representation of the user's **tastes** in the same space where product embeddings live.

3.3. Compare the user to all products

We have:

- A matrix of all product embeddings:
  E with shape num_items × 1024;
- The user vector: u of length 1024.

We compute **cosine similarity** as:

similarities = E @ u

(Since both the rows of E and the vector u are normalized,
the dot product equals cosine similarity.)

For each product we get:

sim(item_i, user) → value in [-1, 1]

- close to 1 → item fits the user's interests very well;
- close to 0 → neutral / weak relation;
- < 0 → "anti-interest" — the vector points in an opposite direction.

3.4. Rank & Filter: from candidates to final recommendations

The algorithm then:

1. **Sorts** all products by similarity in descending order.
2. **Filters out** products that the user has already purchased
   (we don't want to recommend the same product again).
3. **Takes the top-N items** (e.g., 12 products).

We end up with a list like:

```
[
  {"product_id": "...", "description": "...", "score": 0.87},
  {"product_id": "...", "description": "...", "score": 0.83},
  ...
]
```

From a theoretical perspective:

- This is **content-based ranking** using embeddings.
- The similarity function is **cosine similarity**.
- It is effectively a **single-stage pipeline**:
  candidate generation and ranking happen in one vector-space step
  (no separate multi-stage candidate–ranking architecture).

---

4. LLM-based explanations (Explainability)

To avoid a “black box” feeling, we add **explanations** on top of the recommendations.

4.1. Separate chat LLM: openai/gpt-oss-120b

For textual explanations we use a **different model** from Cloud.ru:

- openai/gpt-oss-120b via client.chat.completions.create.

4.2. Prompt design

For **each recommended product** we build a prompt that includes:

- a short list of **descriptions of products the user previously bought**;
- the **description of the recommended product**;
- an instruction, for example:

“Explain in one short sentence in English why it makes sense to recommend this product to this user.
Do not mention technical terms like ‘algorithm’ or ‘model’.”

The LLM returns a short explanation, for example:

“This product matches your previous purchases of home decor items and fits the same aesthetic style.”

We then:

- attach this text to the recommendation JSON as explanation,
- show it in the frontend as a **tooltip on hover** over the product card.

---

5. Frontend: what the user sees

On the page /retail_shop:

1. There is a **dropdown of users** (IDs of users who have a purchase history).
2. When the user is selected:
    - A **“Previous purchases”** block on the top/left shows what they have bought.
    - A **“Recommendations”** block on the bottom/right shows:
        - recommended product cards,
        - an LLM-generated explanation as a tooltip when you hover over each card.

This allows us to demonstrate:

- how **different users get different recommendations** based on their historical purchases,
- and the specific scenarios discussed in the lectures: **personalized recommendations + explainability using LLMs**.