Context-Aware FinCommerce Pipeline (End-to-End Report)

January 22, 2026

Overview

This report documents the full end-to-end Context-Aware FinCommerce pipeline implemented in this workspace. It covers:

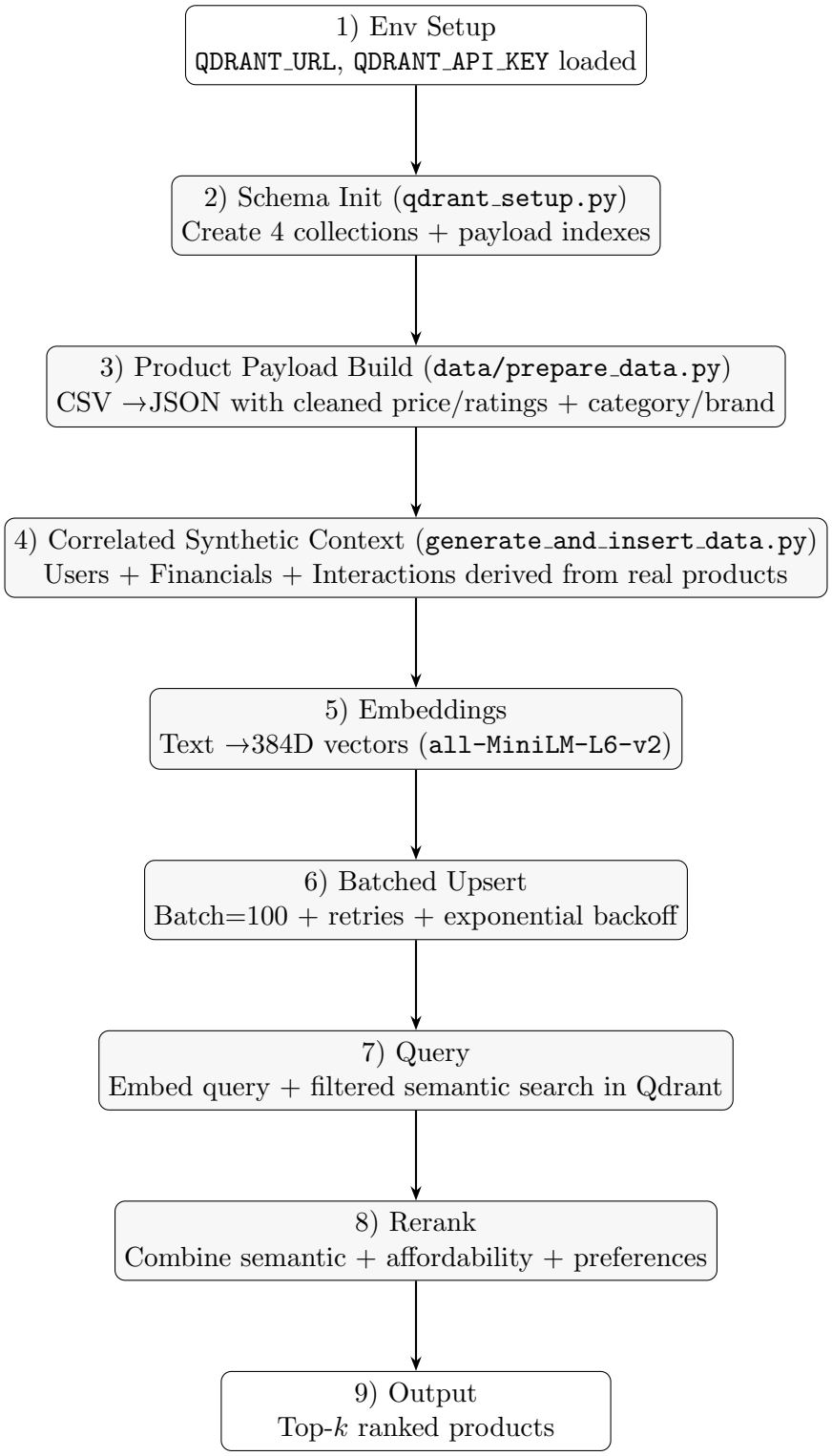
- Data preparation from Amazon CSV into JSON payloads.
- Qdrant collection schema creation (4 collections + payload indexes).
- Synthetic but *correlated* user profiles, financial context, and interaction history.
- Embedding generation and batched upserts to Qdrant Cloud with retry/backoff.
- Semantic search and a personalized reranker (affordability + preferences).

Code map (scripts).

- `data/prepare_data.py`: CSV →JSON with semantic category assignment + brand extraction.
- `qdrant_setup.py`: creates/recreates collections and payload indexes.
- `generate_and_insert_data.py`: embeds and inserts products, users, financials, interactions.
- `search_pipeline.py`: query embedding, Qdrant search, rerank, and output.

Environment variables. The pipeline expects `QDRANT_URL` and `QDRANT_API_KEY` to be set (e.g., via `.env`).

Pipeline Diagram (What Happens End-to-End)



Collections, Vectors, and Payloads

The pipeline stores data in 4 Qdrant collections. Three collections use 384D text embeddings. Financial context uses 256D vectors (a numeric/placeholder representation).

Collection	Vector Dim	Distance	Key Payload Fields
`products_multimodal`	384	Cosine	product_id, name, category, brand, price, in_stock, region
`user_profiles`	384	Cosine	user_id, name, location, risk_tolerance, preferred_categories
`financial_contexts`	256	Cosine	user_id, available_balance, credit_limit, current_debt, eligibl
`interaction_memory`	384	Cosine	user_id, query, clicked_product_id, purchased

Schema Creation (`qdrant_setup.py`)

The schema script recreates collections and then creates payload indexes. Indexes matter because they make filters (e.g., `price`, `in_stock`) efficient and reliable.

Why payload indexes? Search is a combination of vector similarity and structured filters. Payload indexes allow Qdrant to efficiently apply constraints like:

- `price <= max_price`
- `in_stock = true`
- `category` / `brand` matching

Step 1: Data Preparation (CSV →JSON)

Input

The primary product source is an Amazon export CSV (e.g., `data/All Electronics.csv`) with fields such as: `name`, `image`, `ratings`, `no_of_ratings`, `discount_price`, `actual_price`.

Cleaning and normalization (`data/prepare_data.py`)

Key transformations:

- Price parsing: remove currency symbols/commas and convert to numeric.
- Rating count parsing: `"113,956"` →`113956`.
- De-duplication: drop repeated products by normalized name.
- Fill additional fields used downstream: `region`, `in_stock`.

Semantic category assignment

Because the CSV's top-level category is often generic, categories are inferred from the product title using embeddings.

Let $e \in \mathbb{R}^{384}$ be the normalized embedding of the product name, and let $c_k \in \mathbb{R}^{384}$ be the normalized embedding of category description k. The chosen category is:

$$\hat{k} = \arg\max_k \; e^\top c_k$$

Brand extraction

Brand is extracted from the product title using lightweight heuristics (e.g., first meaningful token) to support preference matching.

Step 2: Synthetic Context Generation (Correlated, Not Random)

Why correlation matters

If user profiles and interactions are generated independently from products, personalization becomes meaningless. This pipeline derives user preferences and financial context from the actual product distribution so that:

- Users tend to prefer categories/brands that exist in the catalog.
- Interactions (click/purchase) are biased toward preferred categories.
- Financial context (balances/limits) is scaled relative to observed product prices.

Generated entities (`generate_and_insert_data.py`)

- **Products**: loaded from JSON, embedded, inserted.
- **Users**: preferences sampled from product categories/brands.
- **Financial contexts**: derived per user using price statistics from preferred categories.
- **Interactions**: queries and clicks generated, biased by user preferences.

Step 3: Embeddings (Text $\rightarrow$Vectors)

The system uses SentenceTransformers `all-MiniLM-L6-v2`:

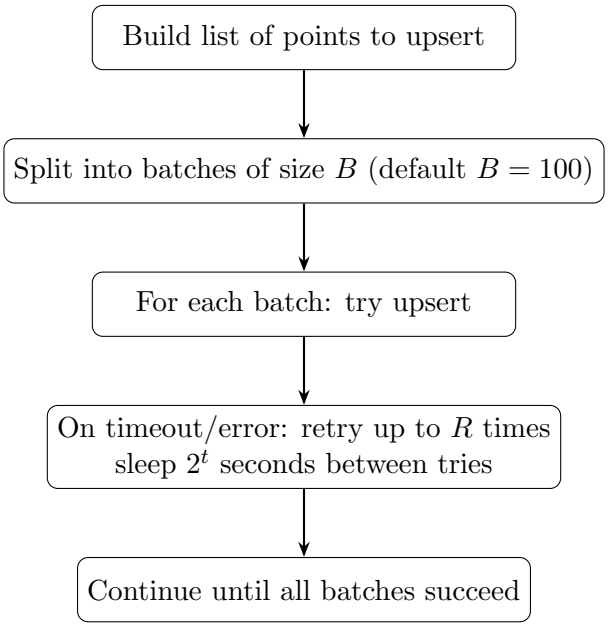
- 384-dimensional embeddings for product text, user profile text, and interaction text.
- GPU acceleration when available (CUDA).

Step 4: Batched Upserts + Retry/Backoff (Qdrant Cloud Stability)

Large single upserts can time out on hosted Qdrant. The solution used across products/users/financials/interactions is:

- Batch size $B = 100$ points.
- Up to $R = 3$ retries per batch.
- Exponential backoff between retries (e.g., 1s, 2s, 4s).

Batching diagram



Step 5: Search Pipeline (Semantic Search + Personalized Rerank)

Semantic search (`search_pipeline.py`)

Given a query string, the pipeline:

1. Embeds the query to a 384D vector.
2. Queries `products_multimodal` by vector similarity.
3. Applies structured filters when enabled (e.g., max price, in stock).

User context retrieval

The pipeline retrieves the user profile and financial context using `user_id` and merges them into a single context object used for reranking.

Reranking Model

The reranker combines three signals:

- Semantic similarity score from Qdrant (higher is better).
- Affordability score based on user available balance vs product price.
- Preference score based on category/brand matches.

Core scoring formula

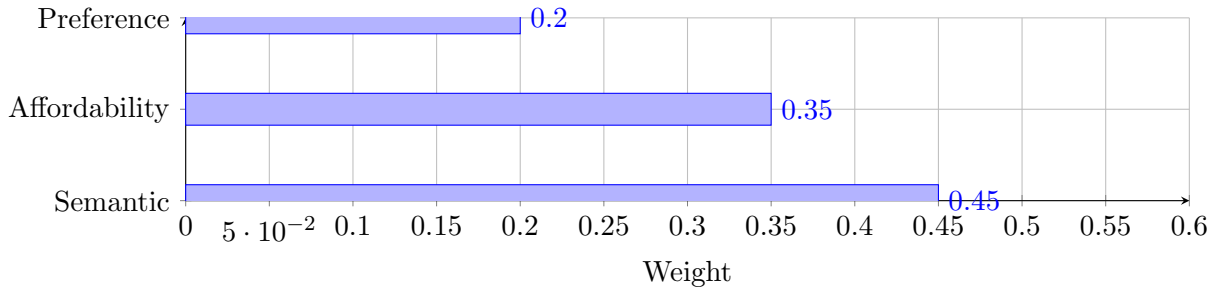
$$\text{final\_score} = 0.45\, s_{\text{semantic}} + 0.35\, s_{\text{afford}} + 0.20\, s_{\text{pref}}$$

$$s_{\text{afford}} = \begin{cases} 0, & \text{if available\_balance} \le 0 \\ \max\left(0, 1 - \frac{\text{price}}{\text{available\_balance}}\right), & \text{otherwise} \end{cases}$$

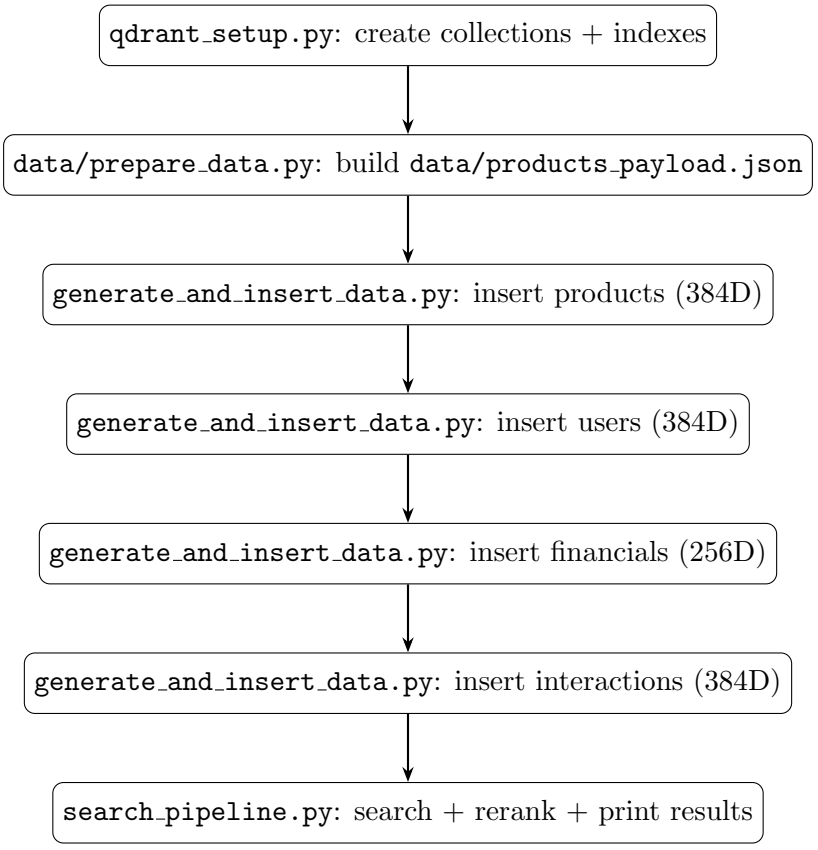
$$s_{\text{pref}} = \max\left(\mathbb{1}[\text{category} \in \text{preferred\_categories}],\ \mathbb{1}[\text{brand} \in \text{preferred\_brands}]\right)$$

Graphs for Better Understanding

Score weights (bar chart)



How the code fits together (script-level sequence)



Execution Checklist

1. Run `qdrant_setup.py` to reset schema.
2. Run `data/prepare_data.py` to generate `data/products_payload.json`.
3. Run `generate_and_insert_data.py` to upsert all collections (batched).
4. Run `search_pipeline.py` to validate retrieval and reranking.