LLM & RAG Challenge (Part 2)

In this part you build a small LLM-powered question-answering layer on top of the **validated orders** from Part 1.

Focus on:

- Reusing the **accepted / valid** data from Part 1.
- A **very simple in-memory retrieval index** (Python lists + NumPy, no DB).
- A **single pydantic-ai agent** with one retrieval tool.
- Answering **3 core business questions** via RAG.

Assume you have about **4 hours**. Prioritize correctness and clarity over polish or extra features.

0. Constraints & Setup

- Use **Python 3** and **pydantic-ai**.
- Use an **OpenAI-compatible chat API** for the LLM:
    - Configure via environment variables (no hard-coded secrets), e.g.:
        - `OPENAI_API_KEY`
        - `OPENAI_BASE_URL` (so we can point it to a free endpoint)
        - `OPENAI_MODEL_NAME`
- For retrieval, use a **simple in-memory index**:
    - It's enough to store:
        - A list of document texts.
        - Corresponding vectors (e.g. from an embedding model).
        - Metadata (e.g. `order_id`).
    - You can use **NumPy** for cosine similarity.
    - No external vector DB required.

You may use additional small libraries if needed, but keep it simple.

1. Reuse Validated Orders from Part 1

1. Start from the **accepted / valid** records produced in Part 1.
2. Load these records in Part 2 in any reasonable way, for example:
    - A saved JSON/NDJSON file with accepted rows from Part 1, or
    - A Python module/function that returns a list of validated Pydantic models.
3. Document briefly (in a comment or small README) how Part 2 loads the validated orders.

Do **not** re-implement custom parsing of the original `orders_sample.ndjson`; reuse the Part 1 models / output.

2. Build Simple Text Documents and In-Memory Index

We want to convert each accepted order into a small text "document" and build a minimal in-memory index.

1. **Document text per order**
    - For each accepted order, build a concise textual representation including at least:
        - `order_id`
        - `customer_email`
        - `status`
        - `shipping.country_code` and `shipping.city`
        - `quantity`, `unit_price`
        - `coupon_code` and `tags` (if present)
    - Example style (you can design your own):

      Show Text Code
2. **Embeddings & vectors (simple)**
    - Use any OpenAI-compatible **embedding model** to turn each document text into a vector.
    - Store:
        - `embeddings: np.ndarray` of shape (n_orders, dim)
        - `meta: List[dict]` with at least `order_id` and `text` per document.
3. **In-memory similarity search**

- Implement a small function to:
  - Embed a **query string**.
  - Compute cosine similarity to all document vectors.
  - Return top-K most similar documents (K can be fixed, e.g. 5).

  Example (you don't have to follow exactly):

  Show Python Code

No need for any external database; keep everything in memory.

3. Define a pydantic-ai Agent with a Retrieval Tool

Create a single pydantic-ai agent that answers questions about the orders using the in-memory index.

1. **Output schema**
   - Define a Pydantic model for the agent's structured output, e.g.:

     Show Python Code
   - The `answer` is natural language.
   - `used_order_ids` are the order IDs you used as evidence (from retrieved docs).
2. **Retrieval tool**
   - Expose your search function as a **pydantic-ai tool** that:
     - Takes a query string.
     - Returns a list of retrieved documents, each with at least:
       - `order_id`
       - `text`
   - The agent should call this tool when answering questions.
3. **Agent configuration**
   - Use `OPENAI_MODEL_NAME`, `OPENAI_API_KEY`, `OPENAI_BASE_URL` from environment.
   - Add a system / developer prompt instructing the agent to:
     - Use only the retrieved order documents as factual evidence.
     - Not invent order IDs or values not present in the retrieved documents.

        - Always fill in `used_order_ids` based on the retrieved documents it relied on.

Example usage (shape, not exact code):

Show Python Code

4. Answer Core Business Questions with RAG

Implement a small script or notebook to run the agent for the following **3 questions** and print the results.

1. **Per-order explanation**

   Question example:

   > "Explain what is special or noteworthy about order `ORD-0003` from a data quality or business perspective."

   The agent should:

   - Retrieve `ORD-0003` and possibly similar orders.
   - Explain issues/patterns such as:
     - Quantity or unit price values.
     - Status vs amounts, tags, coupon usage, etc.
   - Include `ORD-0003` in `used_order_ids`.

2. **Coupon and tag usage**

   Question:

   > "Describe patterns of how coupon codes and tags like `vip` or `promo` are used across the accepted orders."

   The agent should:

   - Retrieve orders containing coupons and/or tags.
   - Provide a short summary of patterns.
   - Include relevant `order_id`s in `used_order_ids`.

3. **Suspicious / edge cases**

   Question:

   > "Based on the accepted orders, identify a few orders that look suspicious or edge-cases from a business rules perspective and explain why."

The agent should:

- Retrieve orders with unusual combinations (e.g. zero/negative prices or quantities, unexpected statuses, odd priorities).
- Explain briefly why they are suspicious.
- List those orders in `used_order_ids`.

You may hard-code these 3 questions in your script or accept them from stdin.

5. Deliverables

Provide:

1. **Code**
   - How you load the **validated orders** from Part 1.
   - Code that:
     - Builds document texts.
     - Computes embeddings and builds the in-memory index.
   - pydantic-ai agent definition, including:
     - Output schema.
     - Retrieval tool.
   - A simple entry point (script or notebook) that:
     - Builds the index.
     - Runs the 3 questions above.
     - Prints `answer` and `used_order_ids` for each.
2. **Configuration**
   - Brief instructions or example `.env` (no secrets) showing expected environment vars:
     - `OPENAI_API_KEY`
     - `OPENAI_BASE_URL`
     - `OPENAI_MODEL_NAME`
3. **Short notes (can be comments or minimal README)**
   - How Part 2 expects to receive/load the accepted orders from Part 1.
   - Rough description of your document format and index.

    - Any limitations or shortcuts you took due to time (e.g. fixed K, simple prompt, simple error handling).

Keep everything as small and focused as possible so it fits into about **4 hours** of work.

For the presentation:

- bring your code in a running environment, to present the code running.
- We can provide an API Endpoint/Key/Model during the presentation.
- For preparation use a freely available endpoint or local model.
  In case of questions, let us know