Case Study: Mini RAG (Retrieval-Augmented Generation) Pipeline for Contract Q&A

1. Business Context

Dataeaze manages thousands of legal contracts (MSAs, NDAs, SOWs, DPAs) with enterprise customers. Currently, legal and sales teams manually search through long PDFs and Word docs to answer questions such as:

- "What is the termination notice period in this contract?"
- "Does this agreement include a data processing addendum?"
- "What are the liability caps for indirect damages?"

Build a **small prototype RAG system** that can answer such questions by **retrieving relevant contract snippets and generating grounded answers**.

2. Assignment Goal (High-Level)

Build a **mini-RAG pipeline** that:

1. **Indexes a small set of contract-like documents** (provided as .txt / .md / .pdf or your own dummy data).
2. **Retrieves relevant passages** for a user's natural language question.
3. **Use an LLM (any open or hosted model)** to generate an answer grounded in the retrieved context.
4. Provides:
    a. A simple **CLI or minimal API/UI** for asking questions.
    b. A **short report** describing your design choices, limitations, and possible improvements.

Dataset (You Can Define a Simple Corpus)

- Use any **publicly available sample contracts** (e.g., standard NDAs, sample SaaS agreements), converted to text/Markdown, **or**
- Create **3–5 synthetic contract documents** in plain text that include:
    - Parties, effective date, term, termination clause.
    - Confidential clause.
    - Liability / indemnity.
    - Data protection / GDPR-related terms (if any).

Minimum: At least **3 documents**, each **3–5 pages worth of text**. Assume all data is **English-only**.

Functional Requirements

4.1. Ingestion & Preprocessing

- Read contract documents from a folder (e.g., `data/contracts/`).
- Split documents into **chunks** appropriate for retrieval (e.g., by section headings and/or fixed token/character windows).
- Preserve useful metadata where possible:
    - `contract_id`
    - `section title`
    - `chunk_id` / position

Be prepared to explain your **chunking strategy** and why you chose it.

4.2. Embeddings & Indexing

- Use any text-embedding model (e.g., OpenAI, Hugging Face, sentence-transformers).
- Compute embeddings for each chunk.
- Store embeddings in **any simple index**:
    - In-memory (e.g., FAISS, sentence-transformers built-in), or
    - Lightweight vector DB (e.g., Chroma, Pinecone, etc.).

The index should support:

- Given a **user question**, return top-k most similar chunks.
- Allow easy configuration of k (e.g., k=3 or k=5).

4.3. Retrieval-Augmented Answer Generation

- Choose **any LLM** you like (open source or API-based).
- Implement a function such as:

```
def answer_question(question: str) -> str:

“””
return:
{
"question": …,
```

"retrieved_chunks": [...],
"answer": ...,
"metadata": {...}
}

"""

Behavior:

1. Embed the question.
2. Retrieve the top-k relevant chunks.
3. Construct a prompt to the LLM that:
    a. Includes the retrieved chunks as **context**.
    b. Instructs the model to:
        i. Answer **only based on the provided context**.
        ii. Say "I don't know" or similar if the answer is not present.
4. Return the generated answer plus the retrieved context.

5. Interface

Option A: Command-Line Interface (CLI)

- Final answer
- List of retrieved snippets

Option B: Minimal API (FastAPI / Flask)

- Endpoint: `POST /ask`
- Request: `{"question": "..."}`
- Response: JSON with answer + retrieved snippets.

Option C: Simple Web UI