Naive RAG Pipeline

This document provides a detailed technical overview of the baseline "naive" Retrieval-Augmented Generation (RAG) system.

System Architecture and Data Flow
The naive RAG pipeline is designed as a sequential, multi-stage process that transforms an input query into a contextually-aware answer. It integrates an embedding model, a local vector database for efficient retrieval, and a powerful language model for generation. The core philosophy is "retrieve-then-read": first, find relevant information, then use that information to construct an answer.

The data flow is as follows:

1. **Offline Processing:** Source documents are loaded, cleaned, split into manageable chunks, and converted into vector embeddings. These embeddings are then indexed and stored in a Milvus Lite vector database. This is a one-time setup process.
2. **Online Processing (Runtime):** A user's query is embedded into a vector. This vector is used to search the Milvus database, retrieving the top-k most relevant text chunks. These chunks are formatted into a context and, along with the original query, are passed to a language model to generate the final answer.

Component Implementation Details

The foundation of the retrieval system is the transformation of text into meaningful numerical representations. For this task, the sentence-transformers library was utilized.

Model Choice: The selected models are all-MiniLM-L6-v2 and all-mpnet-base-v2. They were chosen for their excellent balance of speed, resource efficiency, and strong performance on retrieval benchmarks. The embedding sizes were 384 and 768 dimensions.

Process: During the offline phase, each text chunk was passed through the all-MiniLM-L6-v2 model to produce a 384-dimensional floating-point vector, and the same was done with all-mpnet-base-v2 to produce a 768-dimensional floating-point vector. The same model is used at runtime to embed the incoming user query, ensuring that the query and the documents exist in the same vector space, which is essential for accurate similarity comparison.

Vector Storage with Milvus Lite

To store and efficiently search through thousands of vector embeddings, a dedicated vector database is required. Milvus Lite was chosen for this role.

- Rationale: Milvus Lite is a lightweight, in-process version of the production-grade Milvus database. It is ideal for rapid development and local experimentation, as it requires no complex setup or external dependencies.

- Schema and Indexing: A collection was created with a specific schema to organize the data:
    - id: A unique string identifier for each chunk (Primary Key).
    - passage: The raw text of the chunk.
    - embedding: The 384-dimensional float vector / 768-dimensional float vector

To enable fast retrieval, an AUTOINDEX using the L2 distance metric was built on the embedding field. This index allows Milvus to perform highly optimized Approximate Nearest Neighbor (ANN) searches, returning the most similar document vectors in milliseconds.

Search and Generation Logic

The runtime component is encapsulated in a single, modular Python function (generate_rag_response).

- Search (Retrieval): The embedded user query is sent to the Milvus client's search method. The client uses the pre-built index to find the vectors with the smallest L2 distance to the query vector, retrieving the corresponding text passages.
- Generation: The retrieved passages are concatenated into a single block of text, which serves as the "context." This context is then inserted into a prompt template along with the original query. The google/flan-t5-base model, an instruction-tuned language model, was selected for generation. Its ability to follow instructions makes it well-suited for answering questions based on a provided context, forming the "read" part of the retrieve-then-read architecture. The final output from the model is the answer provided to the user.

Experimentation Results
In the Naive RAG, I implemented:
1. Data Loading → Chunked Wikipedia passages (512 chars, 50 overlap)
2. Embeddings → Used all-MiniLM-L6-v2 (384-dimensional vectors)
3. Vector DB → Stored 4,430 chunks in Milvus Lite
4. Retrieval → Query → embed → find nearest neighbors
5. Generation → Pass contexts to Flan-T5-base → get answer

Also used 3 prompting Strategies:

1. Instruction Prompt: "Context:\n{context}\n\nQuestion:\n{query}\n\nAnswer:"
2. Chain-of-Thought: Added "Let's think step by step"
3. Persona Prompt: "You are an expert encyclopedia..."

Results:

For Embedding Size - 384

- Instruction with top_k=5: **68% EM, 77.05 F1**
- Persona with top_k=5: **68% EM, 75.71 F1**
- CoT completely failed (0% EM) - the model couldn't handle the reasoning request

For Embedding Size - 768

- Instruction with top_k=5: **72% EM, 80.72 F1**
- Persona with top_k=3: **72% EM, 81.05 F1**
- CoT completely failed (0% EM) - the model couldn't handle the reasoning request

Parameter Experiments:
Embedding Models: all-MiniLM-L6-v2, all-mpnet-base-v2
Top k: 1, 3, 5
Prompt Strategy: Instruction, Chain-of-Thought, Persona

Evaluation:
F1 Score, Exact Match using HuggingFace Squad metric 4

Findings:

1. The CoT prompting technique completely failed. It scored 0 for Exact Match and quite low in F1 Scores, too. This shows that for the Flan-T5-Base model, CoT prompts were more of a hindrance rather than a help.

2. Direct Prompts Performed Best: Both the Instruction and Persona prompts were significantly better. This suggests that simple, direct prompting is the most effective strategy for this model architecture.

I hypothesize that since it is a relatively small and simple instruction-tuned model. It's trained to follow direct commands well. The meta-instruction "Let's think step by step" may have confused it, causing it to output reasoning instead of the final, extractable answer.

Results

--- FINAL RESULTS ---

Embedding Model	Embedding Size	Prompt Strategy	Top K	Exact Match	F1 Score
all-MiniLM-L6-v2	384	Instruction	1	52	61.7734
all-MiniLM-L6-v2	384	Chain-of-Thought	1	0	11.9091
all-MiniLM-L6-v2	384	Persona	1	56	63.7128
all-MiniLM-L6-v2	384	Instruction	3	68	74.3795
all-MiniLM-L6-v2	384	Chain-of-Thought	3	0	11.2226
all-MiniLM-L6-v2	384	Persona	3	64	69.0462
all-MiniLM-L6-v2	384	Instruction	5	68	77.0462
all-MiniLM-L6-v2	384	Chain-of-Thought	5	0	9.88396
all-MiniLM-L6-v2	384	Persona	5	68	75.7128

--- FINAL RESULTS ---

Embedding Model	Embedding Size	Prompt Strategy	Top K	Exact Match	F1 Score
all-mpnet-base-v2	768	Instruction	1	64	69.7734
all-mpnet-base-v2	768	Chain-of-Thought	1	0	14.2056
all-mpnet-base-v2	768	Persona	1	64	69.0462
all-mpnet-base-v2	768	Instruction	3	68	77.0462
all-mpnet-base-v2	768	Chain-of-Thought	3	0	11.3789
all-mpnet-base-v2	768	Persona	3	72	81.0462
all-mpnet-base-v2	768	Instruction	5	72	80.7128
all-mpnet-base-v2	768	Chain-of-Thought	5	0	10.958
all-mpnet-base-v2	768	Persona	5	72	79.7128

Single Query Example:

The first query, with Top-k = 3

```
THE QUERY
'Was Abraham Lincoln the sixteenth President of the United States?'

GROUND TRUTH ANSWER:
'yes'

RETRIEVED CONTEXT (Top 3 Chunks)

[CHUNK 1]:
"Young Abraham Lincoln"

[CHUNK 2]:
"Abraham Lincoln (February 12, 1809 â€  April 15, 1865) was the sixteenth President of the Unite

[CHUNK 3]:
"On November 6, 1860, Lincoln was elected as the 16th President of the United States, beating Dem

FINAL GENERATED ANSWER
'yes'
```