LLM-Powered Knowledge Assistant

Mini Retrieval-Augmented Generation (RAG) System

(Github Repo: https://github.com/TechBaton/LLM-Powered-Knowledge-Assistant-Mini-RAG-System)

1. Problem Statement

Large volumes of unstructured documents, such as PDFs, are difficult to search and query efficiently using traditional keyword-based systems. Users often need context-aware answers rather than simple text matches, especially when working with technical, academic, or organisational documents.

The problem addressed in this project is:
"How can we design a system that allows users to upload their own documents and ask natural-language questions, while ensuring answers are grounded in the provided content and remain explainable and reliable?"

2. Proposed Solution

To solve this problem, we implement a Retrieval-Augmented Generation (RAG) based knowledge assistant using open-source models.

The system allows users to:

- Upload PDF documents through a user interface
- Convert documents into vector embeddings
- Retrieve the most relevant document chunks for a given query
- Generate answers strictly grounded in retrieved content
- Provide transparency through retrieved context and query history

No proprietary APIs (e.g., OpenAI) are used.

3. System Architecture Pipeline

The system consists of 4 major components:

1. Document Ingestion Pipeline
2. Vector Database (FAISS)
3. Retrieval-Augmented Generation Engine
4. Streamlit-Based User Interface

High-Level Workflow:

1. User uploads PDF documents via the UI
2. Documents are split into overlapping text chunks
3. Each chunk is converted into a dense vector embedding
4. Vectors are stored in a FAISS index
5. User submits a question
6. Relevant chunks are retrieved based on semantic similarity
7. A language model generates an answer using the retrieved context
8. The system displays the answer, retrieved context, source documents, and query history

4. Document Ingestion and Indexing

4.1 PDF Loading

PDF documents are loaded using a document loader that extracts text page-by-page.

4.2 Text Chunking

Documents are split using a recursive character text splitter with:

- Chunk size: 500 characters
- Overlap: 50 characters

This ensures context preservation across chunks and better retrieval quality.

4.3 Embedding Generation

Each chunk is converted into a dense vector using Sentence-Transformers (all-MiniLM-L6-v2). This model provides:

- Fast inference
- Good semantic representation
- CPU compatibility

4.4 Vector Storage

All embeddings are stored locally using FAISS, enabling efficient similarity search.

5. Retrieval Augmented Generation (RAG)

5.1 Retrieval Phase

For each user query, the system retrieves the top-K most similar document chunks from FAISS. Then, similarity is computed using vector distance.

5.2 Generation Phase

The retrieved chunks are passed to an open-source language model, '*google/flan-t5-base*'. The model generates an answer conditioned on retrieved context, ensuring responses remain document-grounded.

6. Suggested Questions Feature

To improve usability, the system automatically generates suggested questions based on uploaded documents.

How it works:

- Randomly samples document chunks
- Uses the language model to generate relevant questions
- Displays questions as clickable UI elements

Reload Questions

A "Reload Questions" button allows users to regenerate a fresh set of document-grounded questions without rebuilding the index.

This enables diverse exploration, reduced cold-start friction, and controlled LLM usage.

7. Query History and Session Memory

The system maintains session-level query history using Streamlit session state.

For each query, it stores the question that the user has asked, the generated answer and the source documents. This allows users to review past interactions, perform qualitative evaluation, and demonstrate system behaviour during demos.

History resets safely when the session is cleared.

8. Performance Optimisations

Several optimisations are implemented:

- Model caching using *st.cache_resource*
- Embedding caching
- FAISS index reuse
- CPU-only inference for portability
- Avoidance of unnecessary recomputation

These ensure smooth performance even on modest hardware.

9. Is this the most optimal solution?

Short Answer: Conceptually yes, operationally no (yet).

The chosen RAG approach is architecturally optimal for this class of problems. However, the current implementation represents a simplified prototype.

10. Industry-Grade Solutions

In production systems, companies typically use:

a) Advanced Retrieval

- Hybrid search (BM25 + dense vectors)
- Cross-encoder re-ranking
- Metadata-aware filtering

b) Scalable Infrastructure

- Vector databases such as Pinecone, Weaviate, or Milvus
- Distributed storage and indexing

c) Robust LLM Integration

- Larger instruction-tuned models
- Prompt templates with safety guards
- Retrieval confidence gating (hallucination control)

d) Monitoring & Governance

- Logging retrieved contexts
- Answer validation
- Feedback loops

11. Comparison: Prototype vs Industry System

Aspect	**This Project**	**Industry Systems**
Retrieval	Dense vectors (FAISS)	Hybrid + re-ranking

Scale	Small-medium corpus	Millions of documents
Hallucination Control	Design-level	Enforced at runtime
UI	Notebook-based	Web/API
Cost	Free	Paid infrastructure

Despite these differences, the core architectural principle remains the same.

12. Limitations

- No persistent long-term memory across sessions
- Answer quality depends on document quality
- Not optimized for extremely large document corpora

13. Future Enhancements

- Hybrid retrieval (BM25 + dense vectors)
- Cross-encoder re-ranking
- Automated evaluation pipelines
- Feedback-driven answer refinement
- Access control and multi-user support