Architectural Blueprint of a Modular, Hybrid RAG System

A Technical Guide to Designing a Personal RAG System for Document Interrogation

by Nikhil Sharma

June 11, 2025

Abstract

Abstract: Large Language Models (LLMs) have demonstrated remarkable capabilities in natural language understanding and generation, yet they are fundamentally limited by their static, pre-trained knowledge. Retrieval-Augmented Generation (RAG) addresses this by grounding LLMs in external, verifiable knowledge bases. This documentation presents a detailed architectural blueprint for an advanced, local-first RAG system designed for querying private documents. The system features a modular design with components for data processing, hybrid indexing (FAISS and BM25), retrieval fusion, and local LLM generation via Ollama. Key innovations include sentence-aware chunking with overlap, weighted hybrid retrieval (70% dense, 30% sparse), and comprehensive metadata tracking for source attribution. The implementation demonstrates how to achieve robust performance while maintaining privacy and extensibility, serving as a template for production-ready RAG applications.

Contents

1 Introduction

Large Language Models (LLMs) such as GPT, Llama, and Mistral represent a paradigm shift in artificial intelligence, capable of generating fluent, human-like text. However, their utility is constrained by two inherent limitations:

- **Knowledge Cutoff:** An LLM's knowledge is frozen at its training date, making it unaware of newer information.
- **Hallucination:** When faced with unfamiliar queries, LLMs may generate plausible but incorrect information.

Retrieval-Augmented Generation (RAG) [1] has emerged as the leading solution, combining information retrieval with LLM generation. Our system implements several key advancements:

- **Hybrid Retrieval:** Combines dense semantic search (FAISS) with sparse keyword matching (BM25)
- **Metadata Preservation:** Tracks source documents, page numbers, and character offsets
- **Local-First Design:** Uses Ollama for private LLM inference and local embedding models
- **Modular Architecture:** Separates components for easy maintenance and extension

2 System Architecture Overview

The system follows a clear pipeline with offline indexing and online query phases:

2.1 Core Components

The implementation consists of six main Python modules:

Table 1: System Modules and Responsibilities

Module	Functionality
config.py	Centralized configuration with dataclass
data_structures.py	Pydantic models for chunks and results
data_processing.py	PDF loading and smart chunking
indexing.py	Hybrid index creation and management
retrieval.py	Hybrid search and result fusion
generation.py	Prompt engineering and LLM interaction
pipeline.py	Main orchestration class
main.py	CLI interface and entry point

3 Implementation Details

3.1 Data Processing

The PDFProcessor class handles document ingestion with several key features:

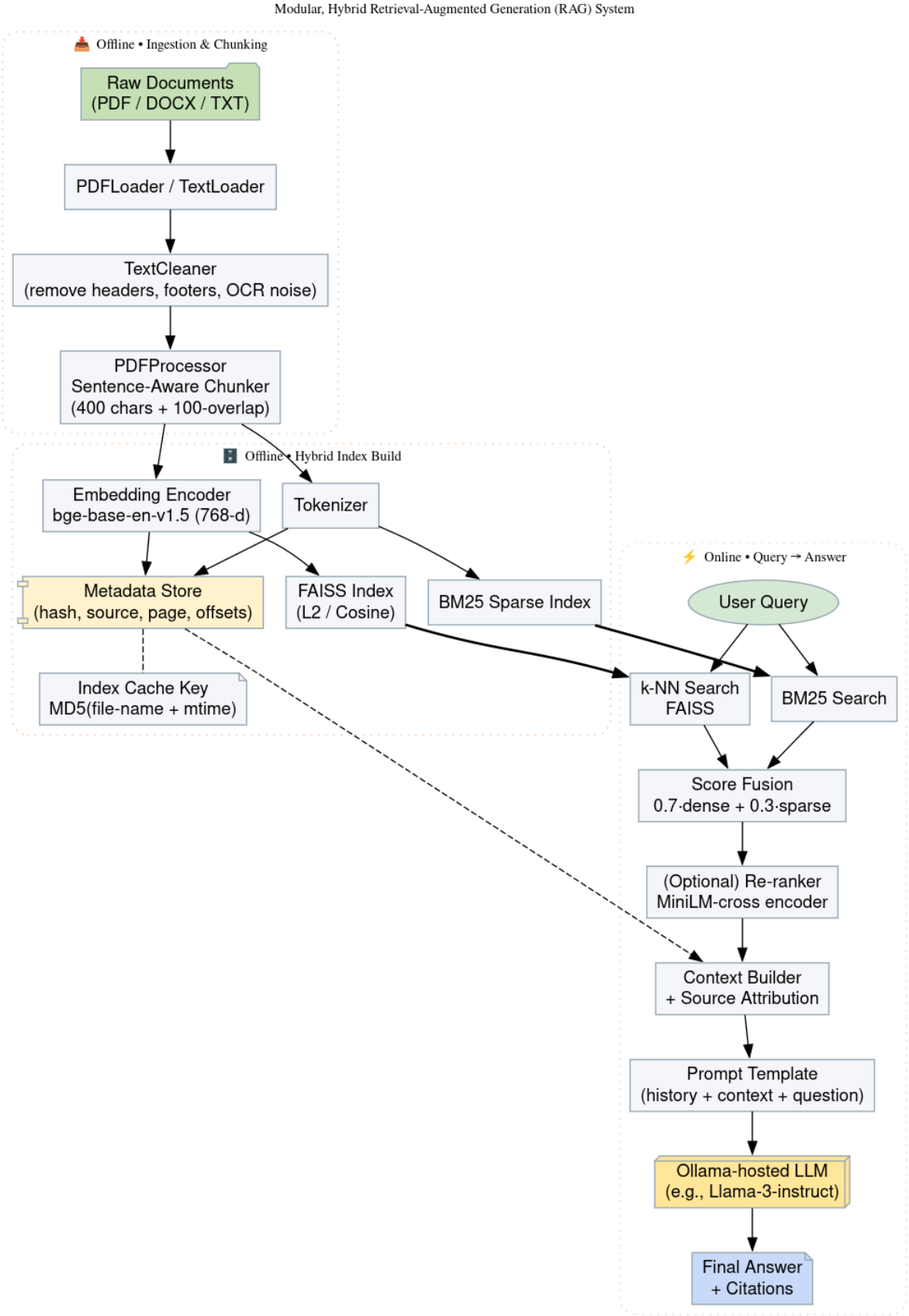

Figure 1: System architecture showing the complete workflow from document ingestion to answer generation

```python
def _smart_chunk_text(self, text: str, source_file: str,
                      page_metadata: List[Dict]) -> List[ChunkMetadata]:
    """Splits text into chunks based on sentences and overlap."""
    sentences = sent_tokenize(text)
    chunks = []
    current_chunk = ""
    current_start_char = 0
    chunk_id = 0

    for sentence in sentences:
        if len(current_chunk) + len(sentence) > config.CHUNK_SIZE and
    current_chunk:
            chunks.append(self._create_chunk_metadata(...))
            overlap_text = self._get_overlap_text(current_chunk)
            current_start_char += len(current_chunk) - len(overlap_text)
            current_chunk = overlap_text + " " + sentence
            chunk_id += 1
        else:
            current_chunk += " " + sentence if current_chunk else sentence
```

Listing 1: Key chunking logic from data_processing.py

Key parameters (configurable in config.py):

- CHUNK_SIZE = 400 characters
- CHUNK_OVERLAP = 100 characters
- MIN_RELEVANCE_SCORE = 0.3

3.2 Hybrid Indexing

The VectorStore class manages both dense and sparse indexes:

Table 2: Indexing Configuration

Index Type	Model/Library	Parameters
Dense (FAISS)	BAAI/bge-base-en-v1.5	768-dim embeddings
Sparse (BM25)	rank_bm25	Default k1=1.5, b=0.75

3.3 Retrieval Strategy

The HybridRetriever implements weighted score fusion:

$$S_{\text{combined}} = 0.7 \times S_{\text{dense}} + 0.3 \times S_{\text{sparse\_norm}} \tag{1}$$

```python
def _combine_and_rerank(self, dense: List[RetrievalResult],
                        sparse: List[RetrievalResult]) -> List[RetrievalResult]:
    # Normalize BM25 scores to [0,1]
    max_sparse = max(r.sparse_score for r in sparse) if sparse else 1.0
    for r in sparse:
        r.sparse_score /= max_sparse

    # Combine with dense scores
    for r in dense + sparse:
        r.combined_score = 0.7 * r.dense_score + 0.3 * r.sparse_score
```

Listing 2: Score fusion in retrieval.py

3.4 Generation Pipeline

The EnhancedGenerator handles context formatting and LLM interaction:

- Prepares context with source attribution
- Manages conversation history
- Implements retry logic for Ollama API calls

```python
def _build_prompt(self, question: str, context: str) -> str:
    history = self.conversation.get_history_prompt() if self.conversation else
    ""
    return f"""{history}Based on the following context, answer the question.
    Cite sources using [Source: file, Page: number].

    CONTEXT:
    {context}

    QUESTION:
    {question}

    ANSWER:"""
```

Listing 3: Prompt construction in generation.py

4 Performance Considerations

4.1 Caching Strategy

The system implements a smart caching mechanism:

- Indexes are cached based on file content hashes
- Avoids reprocessing unchanged documents
- Cache invalidation through force_rebuild flag

```python
def _get_cache_key(self, pdf_files: List[str]) -> str:
    """Generates cache key from file names and modification times"""
    info = "".join(f"{f}{os.path.getmtime(f)}" for f in sorted(pdf_files))
    return hashlib.md5(info.encode()).hexdigest()
```

Listing 4: Caching logic in indexing.py

4.2 Resource Optimization

Key optimizations include:

- FAISS for efficient vector search
- L2-normalized embeddings for cosine similarity
- NLTK for lightweight text processing
- Configurable batch sizes for embedding generation

5 Conclusion and Future Work

The implemented system demonstrates how to build a production-ready RAG application with hybrid retrieval capabilities. Key strengths include:

- Privacy-preserving local execution
- Verifiable source attribution
- Balanced retrieval strategy
- Modular, maintainable design

Future enhancements could include:

- Cross-encoder reranking for precision window retrieval
- Automated evaluation framework
- Support for additional file formats

References

[1] P. Lewis, et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, 2020.

[2] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," in *Foundations and Trends® in Information Retrieval*, 2009.

[3] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," in *IEEE Transactions on Big Data*, 2019.

If you find this repository and guide useful, please remember to give feedback on the repo and cite this project in your work.