Project Report: Intelligent RAG System with Calculator Tools

System Architecture Documentation

December 1, 2025

Abstract

This document provides a comprehensive technical overview of the ”Chat with PDF” system developed using Python, LangChain, and Google Gemini. It details the architectural workflow, the principles of Retrieval-Augmented Generation (RAG), the specific role of Hugging Face for local embeddings, and the mechanics of FAISS vector storage. The report explains how these components synergize to provide accurate, grounded answers to user queries while mitigating hallucinations and API limits.

Contents

1 Introduction

The project is an intelligent question-answering system designed to bridge the gap between static documents (PDFs) and dynamic computational capabilities. Traditional Large Language Models (LLMs) possess vast general knowledge but lack access to private user data and often struggle with precise mathematical operations.

This system solves these two problems by implementing an **Agentic RAG** architecture. It allows users to upload a PDF document and ask natural language questions. The system intelligently decides whether to retrieve information from the document, perform a mathematical calculation, or both, ensuring high accuracy and context awareness.

2 Detailed System Workflow

The workflow is architected to handle data privacy, cost-efficiency, and logical reasoning. It is divided into two distinct phases: **Ingestion** and **Inference**.

2.1 Phase 1: Document Ingestion (The ”Learning” Phase)

When a user uploads a PDF via the Streamlit interface, the system transforms the unstructured file into a structured mathematical representation.

2.1.1 1. Loading

The system uses `PyPDFLoader` to read the raw binary stream of the PDF file. It extracts text page by page, stripping away formatting like bolding or italics, leaving only the raw textual content.

2.1.2 2. Recursive Chunking (The Splitting Strategy)

LLMs have a ”context window” limit (the maximum amount of text they can process at once). Feeding an entire book is inefficient and often impossible. We use the `RecursiveCharacterTextSplitter` to solve this.

How it works: The splitter does not simply cut the text every 1000 characters. Instead, it employs a ”hierarchy of separators” to keep semantically related text together. It attempts to split the text in this specific order:

1. **Paragraphs (`\n\n`):** It first tries to split by double newlines to keep paragraphs intact.
2. **Sentences (`. `):** If a paragraph is still too large (over 1000 chars), it splits by periods to keep sentences intact.
3. **Words (` `):** If a sentence is somehow too large, it splits by spaces.

The Importance of Overlap: We configure a `chunk_overlap` of 200 characters. This is crucial for context preservation.

- *Scenario:* A sentence might start at the very end of ”Chunk A” and finish at the start of ”Chunk B”.
- *Without Overlap:* Neither chunk would contain the complete thought, meaning the embedding would fail to capture the sentence’s meaning.
- *With Overlap:* The end of Chunk A is repeated as the start of Chunk B, ensuring that at least one chunk contains the complete sentence and its full semantic meaning.

2.1.3 3. Local Embedding Generation

Each chunk is passed through the local Hugging Face model (`all-MiniLM-L6-v2`) to generate a 384-dimensional vector. This ensures no API quotas are consumed during the heavy reading process.

2.1.4 4. Vector Indexing

These vectors are stored in RAM using the FAISS library, creating a high-speed search index.

2.2 Phase 2: Query Inference (The Agentic Loop)

When a user types a query (e.g., ”What is the tax rate in the document and how much is 500 times that rate?”), the system enters the inference loop:

1. **Intent Analysis (ReAct Pattern):** The LLM (Google Gemini) acts as an autonomous agent following the **ReAct** (Reason + Act) pattern. It does not just answer; it ”thinks.”
2. **Step 1: Information Retrieval**
    - *Thought:* ”I need to find the tax rate first.”
    - *Action:* Call `pdf_search_tool("tax rate")`.
    - *Observation:* The vector store returns the specific text chunk: ”The corporate tax rate is set at 10% for FY2024.”
3. **Step 2: Computational Logic**
    - *Thought:* ”I have the rate (0.10). Now I need to calculate 500 * 0.10.”
    - *Action:* Call `calculator("500 * 0.10")`.
    - *Observation:* The tool returns ”50.0”.
4. **Step 3: Final Synthesis**
    - The Agent combines the retrieved context and the calculated result to form a natural language response: ”The tax rate mentioned is 10%. 500 times this rate is 50.”

3 Deep Dive: Embeddings

3.1 What is an Embedding?

An embedding is a numerical representation of a piece of information (text, image, or audio) in a high-dimensional vector space. In simple terms, it converts words into a list of floating-point numbers (a vector) where the position and direction of the vector capture the *meaning* of the text.

For example, in a simplified 2D space:

- `King`: [0.9, 0.8]
- `Queen`: [0.91, 0.79]
- `Apple`: [0.1, 0.2]

Notice how ”King” and ”Queen” have very similar numbers, while ”Apple” is far away.

3.2 Why Do We Need Embeddings?

Computers operate on logic and mathematics, not language.

1. **Beyond Keywords:** Traditional search (Ctrl+F) relies on exact keyword matching. If you search for ”canine”, keyword search will not find a document that only contains the word ”dog”.
2. **Semantic Understanding:** Embeddings solve this. The vector for ”canine” and the vector for ”dog” are mathematically very close. Therefore, a vector search for ”canine” will successfully retrieve the ”dog” document, even if the words share no common letters.

3.3 How Are We Doing It Here?

In this project, we utilize the `sentence-transformers` library from Hugging Face.

1. **Model Selection:** We use `all-MiniLM-L6-v2`. This is a pre-trained model, meaning it has already ”read” billions of sentences and learned the relationships between words.
2. **Process:**
   - We pass the text chunks (created in the splitting phase) into this model.
   - The model outputs a fixed-size vector of 384 dimensions for each chunk.
3. **Storage:** These 384 numbers are what FAISS stores in memory. When the user asks a question, their question is also converted into 384 numbers, and FAISS looks for the closest match.

4 Deep Dive: Vector Storage

4.1 What is Vector Storage?

Vector Storage (often implemented as a Vector Database) is a specialized system optimized for storing and retrieving high-dimensional vectors. Unlike a traditional SQL database that stores rows of text and numbers, a vector store is designed to handle geometric coordinate data (embeddings).

4.2 Why Do We Need It?

Standard databases are excellent at exact matches (e.g., `WHERE id = 5`), but they fail at ”similarity” searches.

1. **The Scale Problem:** If you have 10,000 PDF pages, checking which page is ”closest” to your query would require comparing your query against all 10,000 pages one by one (Linear Time, $O(N)$). This is too slow for real-time chat.
2. **Efficient Indexing:** Vector Stores use advanced mathematical algorithms (like HNSW or Inverted Files) to organize the data. This allows them to eliminate 90% of the wrong answers instantly and find the closest match in milliseconds, even with millions of documents.

4.3 How Are We Doing It Here?

We utilize **FAISS (Facebook AI Similarity Search)**.

1. **In-Memory Indexing:** We use the `FAISS` class from LangChain. When `create_vector_store` is called, FAISS takes all the 384-dimensional vectors generated by Hugging Face and builds an index in the computer’s RAM.
2. **Euclidean Distance (L2):** FAISS is configured to calculate the ”L2 Distance” (straight-line distance) between vectors.
3. **Retrieval Process:**
    - The user’s query (”What is the profit?”) is converted into a vector.
    - FAISS scans the index to find the 5 vectors with the smallest distance to the query vector.
    - It returns the original text chunks associated with those winning vectors.

Visualizing Vector Space

The concept of ”Apple” is numerically closer to ”Fruit” than to ”Car”.

[Query: ”Fruit”] $\xrightarrow{\text{Distance Check}}$ *[Doc: ”Apple”]* (Distance: 0.1)
[Query: ”Fruit”] $\xrightarrow{\text{Distance Check}}$ *[Doc: ”Toyota”]* (Distance: 0.9)

Figure 1: Conceptual representation of semantic distance in FAISS.