Comprehensive Documentation: RAG-Based PDF Question Answering System

Project Report
Prepared by: Haile Tassew Belay
Date: October 11, 2025

Executive Summary

This document presents a **comprehensive technical exposition** of the RAG-Based PDF Question Answering System. The system is designed to provide **enterprise-level document interaction** capabilities by enabling users to query PDF files using natural language. Leveraging the **Retrieval-Augmented Generation (RAG)** paradigm, the system integrates **state-of-the-art Large Language Models (LLMs)** with vector-based retrieval to produce accurate, contextually relevant answers.

The document includes:

- System architecture and workflow
- Detailed explanation of the RAG pipeline and each component
- Backend and frontend design and implementation
- Project folder structure and environment configuration
- Challenges, limitations, and mitigations
- Potential future enhancements

This documentation targets **technical stakeholders, system architects, and enterprise engineers**, providing them with a **complete blueprint** for understanding, maintaining, and extending the system.

---

1. Introduction

In today's data-driven enterprises, **efficient extraction and retrieval of information from unstructured documents** is a critical requirement. PDFs are ubiquitous in business, legal, academic, and research environments, but manually querying them is time-consuming and error-prone.

This project addresses this challenge by implementing a **RAG-based system** that integrates:

- PDF text extraction

- Semantic text chunking
- Embedding generation
- Vector-based retrieval
- LLM-based answer synthesis

This system allows users to **upload a PDF and pose natural language questions**, producing answers that are **grounded in the content of the document**. The architecture is modular, enabling future expansion to include additional document types, persistent storage, and multi-language support.

---

2. Objectives

The main objectives of this project are:

1. Develop a **modular, scalable backend** capable of processing PDF documents and supporting RAG workflows.
2. Integrate a **Large Language Model (LLM)**, specifically **Google Gemini Flash**, for answer generation.
3. Implement an **efficient vector-based retrieval system** using FAISS.
4. Provide a **user-friendly frontend** interface for PDF uploads and question answering.
5. Ensure **cost-effectiveness and accessibility** by relying primarily on **free-tier services and open-source tools**.
6. Document the system comprehensively for **enterprise adoption, replication, and further research**.

---

3. Retrieval-Augmented Generation (RAG) Paradigm

3.1 Overview

RAG is a hybrid AI paradigm combining **retrieval of relevant data from external sources** with **generation of natural language responses** using an LLM. Unlike traditional LLMs that rely solely on pre-trained knowledge, RAG ensures that responses are:

- **Contextually grounded** in real-time data
- **Accurate** by minimizing hallucinations
- **Flexible** for different data sources

3.2 Core Components

Component	Description	Importance
Retrieval	Identify semantically relevant information from external sources (PDF in this project) using embeddings and vector search.	Ensures contextually grounded responses.
Generation	LLM produces natural language answers using retrieved context and user query.	Delivers coherent and readable answers.

3.3 Workflow

1. **Document Ingestion:** User uploads a PDF through the frontend interface.
2. **Text Extraction:** Raw text is extracted using **PyPDF2 (`pypdf`)**.
3. **Text Chunking:** Text is split into **semantically coherent chunks** (~500 characters with 50-character overlap).
4. **Embedding Generation:** Chunks are encoded into vector embeddings using **sentence-transformers/all-MiniLM-L6-v2**.
5. **Vector Storage:** Embeddings are stored in an **in-memory FAISS index** for similarity search.
6. **Query Embedding:** User's question is transformed into a vector using the same embedding model.
7. **Retrieval:** FAISS returns the **top-k most similar chunks**.
8. **Prompt Construction:** Retrieved chunks are formatted with the user query for LLM input.
9. **Answer Generation: Google Gemini Flash** generates a context-aware response.
10. **Response Delivery:** Answer is returned to the frontend for user display.

---

4. System Architecture

The system adopts a **client-server architecture**, ensuring separation of concerns and modularity.

4.1 Frontend Layer

- **Technology:** React.js, TypeScript, Tailwind CSS, built with Vite
- **Responsibilities:**
  - File upload interface
  - Query input interface
  - Display answers to the user
  - Communicate with backend via REST API

4.2 Backend Layer

- **Technology:** FastAPI (Python)
- **Responsibilities:**
  - Accept uploaded PDFs
  - Extract text from PDFs
  - Chunk and embed text
  - Store embeddings in FAISS
  - Process user queries
  - Retrieve relevant text chunks
  - Call LLM for final answer generation

4.3 LLM Integration

- **Model:** Google Gemini Flash (`gemini-flash-latest`)
- **API:** `google-generativeai` Python SDK
- **Prompt Engineering:**
  - Contextual grounding
  - Restriction from external knowledge
  - Fallback message for missing information

4.4 Vector Database Layer

- **Library:** FAISS (`IndexFlatIP`)
- **Responsibilities:** Efficient nearest-neighbor retrieval of embeddings
- **Normalization:** Embeddings are L2-normalized for cosine similarity

---

4.5 Data Flow Diagram (Textual)

```
User Uploads PDF --> Frontend --> Backend
Backend: PDF Text Extraction --> Text Chunking --> Embedding Generation -->
FAISS Storage
User Submits Query --> Backend
Backend: Query Embedding --> FAISS Retrieval --> Prompt Construction -->
Gemini LLM --> Response
Frontend: Display Answer
```

---

5. Core Components – Detailed Analysis

5.1 PDF Text Extraction

- **Library:** `pypdf`
- **Strengths:** Lightweight, Python-native, straightforward integration
- **Limitations:** Cannot process scanned documents (OCR required)
- **Alternatives Considered:** `PyMuPDF`, `pdfplumber`, OCR via Tesseract

5.2 Text Chunking

- **Methodology:** Sentence-aware chunking
- **Chunk Size:** ~500 characters
- **Overlap:** 50 characters
- **Rationale:** Preserves context for better retrieval accuracy

Alternatives Considered: Fixed-size splitting, LangChain recursive splitter, semantic chunking

---

5.3 Embedding Generation

- **Model:** `sentence-transformers/all-MiniLM-L6-v2`
- **Vector Size:** 384
- **Advantages:** Fast, CPU-efficient, free, robust semantic representation

Alternatives Considered: OpenAI embeddings, Gemini embeddings (not publicly available), instructor models

---

5.4 Vector Storage & Indexing

- **Library:** FAISS, `IndexFlatIP`
- **Mechanism:** L2-normalized embeddings stored in-memory
- **Advantages:** Fast retrieval, minimal setup, cost-free
- **Limitations:** Volatile storage (no persistence)
- **Alternatives Considered:** Chroma, Pinecone, Weaviate, Qdrant, Annoy

---

5.5 Retrieval Process

- User query is embedded
- FAISS searches for **top-k closest chunks**
- Contextual text concatenated for LLM prompt
- Index validation prevents runtime errors

5.6 Answer Generation

- **LLM:** Google Gemini Flash
- **Prompt Engineering:**
    - Enforce context usage
    - Restrict external knowledge
    - Standard fallback response

Advantages: High accuracy, free-tier accessible, fast response

---

6. Project Folder Structure

```
pdf_rag_project/
│
├─ backend/
│   ├─ main.py                  # FastAPI app
│   ├─ pdf_processor.py         # PDF extraction & chunking
│   ├─ rag_engine.py            # RAG query logic
│   ├─ gemini_llm.py            # Gemini API wrapper
│   ├─ utils.py                 # Chunking, text preprocessing, etc.
│   ├─ requirements.txt         # Backend dependencies
│   └─ .env                     # API keys
│
├─ frontend/
│   ├─ public/
│   │   └─ index.html           # Main HTML
│   ├─ src/
│   │   ├─ App.tsx              # Main React component
│   │   └─ api.ts               # API communication
│   ├─ package.json             # Frontend dependencies
│   └─ .env                     # Backend URL
│
└─ README.md                    # Project overview
```

---

7. Environment Setup

Backend

```
cd backend
python -m venv venv
# Activate
# Windows: venv\Scripts\activate
# Linux/Mac: source venv/bin/activate
pip install -r requirements.txt
```

- `.env` example:

```
GEMINI_API_KEY=your_free_gemini_api_key_here
```

Frontend

```
cd frontend
npm install
npm start
```

- `.env` example:

```
REACT_APP_BACKEND_URL=http://localhost:8000
```

---

8. Challenges, Risks, and Mitigations

Challenge	Mitigation
PDF with images	OCR pipeline planned for future integration
Memory limitation (FAISS in-memory)	Switch to Chroma/Qdrant for persistent storage
LLM hallucinations	Strict prompt engineering and fallback response
API rate limits	Batch embedding generation and caching strategies
Multi-language PDFs	Future enhancement with multilingual embeddings

---

9. Future Enhancements

1. **OCR Support:** Integrate Tesseract to handle scanned PDFs.
2. **Persistent Vector Storage:** Use Chroma or Weaviate.
3. **Multilingual Support:** Extend embeddings and LLM to multiple languages.
4. **Advanced UI:** Chat-style interface with conversation memory.
5. **Security Enhancements:** Authentication, authorization, encrypted document storage.
6. **Distributed Backend:** Scale to support large PDF datasets and concurrent users.

---

10. Conclusion

This RAG-Based PDF Question Answering System demonstrates a **professional, scalable, and modular design**. By integrating:

- **PDF ingestion**
- **Semantic chunking**
- **Vector embeddings**
- **FAISS retrieval**
- **LLM-based answer generation**

the system provides **accurate, contextually grounded answers** to user queries. It is **easily extendable** for enterprise deployment, multi-document support, and integration with internal knowledge bases.

Optional Visuals to Include

1. **RAG Workflow Diagram** – showing PDF → Chunking → Embeddings → FAISS → LLM → Answer.
2. **System Architecture Diagram** – Frontend/Backend/Vector DB/LLM layers.
3. **Sequence Diagram** – User query to answer delivery.