High-Level Design (HLD) Document

Retrieval-Augmented Generation (RAG) System for PDF Q&A

Author: Vasala Harinadha

Date: August 2025

Table of Contents

1. Introduction

Purpose:

This High-Level Design (HLD) document describes the architecture, components, workflows, and design decisions for a Retrieval-Augmented Generation (RAG) system that ingests PDF documents (native and scanned), builds a local vector index, and answers natural language questions with citations that highlight evidence in source PDFs.

Scope:

- Local, open-source stack deployable on a developer workstation or on-prem server.
- Supports native PDFs and scanned images (OCR).
- Produces answers with verifiable citations and highlighted text in source pages.
- Includes an optional Streamlit UI for demo/interaction.

Audience:

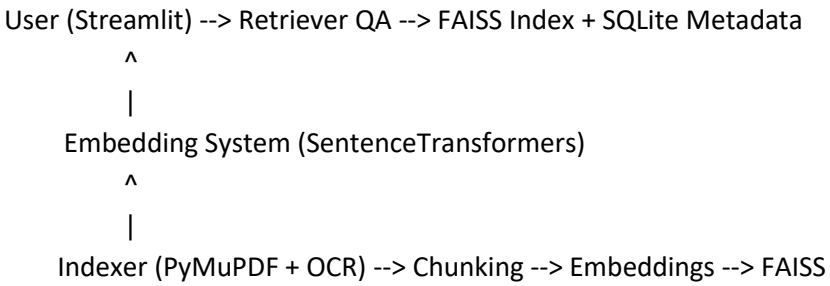
Technical leads, architects, and engineers who will implement or review the system.

2. System Overview

High-Level Description:

The system converts PDF content into searchable vector embeddings and stores them in a FAISS index, while storing chunk metadata in SQLite. A retriever maps user queries to top-k chunks, which are then passed to a local LLM (e.g., Flan-T5) to generate answers. The frontend displays answers along with source citations and highlighted evidence on the original PDF pages.

Architecture Diagram (textual):



3. Core Components

3.1 Frontend (Streamlit UI)

Responsibilities:
- Allow users to upload PDFs or select pre-indexed PDFs.
- Accept natural language queries and display answers.
- Present a ranked list of retrieved chunks with source, page, and score.
- Render PDF pages and visually highlight exact text spans used as evidence.
- Allow feedback (thumbs up/down, flag incorrect citation).

Key UI Components:
- File upload panel
- Document selector
- Query input box
- Answer panel with citations
- PDF viewer with highlights
- Retrieval details (score, chunk text, page)

Implementation notes:
- Use Streamlit for rapid prototyping.
- Use PyMuPDF (fitz) to render pages as images and draw highlight rectangles.
- Keep LLM calls asynchronous on the server side to avoid blocking UI (use background threads or asyncio).

3.2 Ingestion & Indexing

Responsibilities:
- Accept PDF files (native and scanned) and produce text content with page-level locations.
- Perform OCR for scanned pages using Tesseract (or PaddleOCR) and preserve coordinates for highlighting.
- Split page text into overlapping chunks using configurable chunk_size & overlap.
- Attach metadata: document_id, page_number, chunk_id, char_start, char_end, original_text.

Chunking strategy:
- Default: 1000-character chunks with 200-character overlap (configurable).
- For semantic boundaries, optionally use sentence/paragraph-aware splitting via spaCy or NLTK.
- Store mapping from chunk to (page, bbox) where possible so that highlighting can map to exact coordinates.

OCR and coordinates:
- When using OCR, record word bounding boxes so highlights can render exact spans.
- For native PDFs, use PyMuPDF text extraction with bbox info (page.get_text('blocks') or 'words').

3.3 Embedding System

Responsibilities:

- Convert chunks of text to dense vectors using Sentence Transformers.
- Normalize vectors (L2 normalize) for cosine-similarity search in FAISS.
- Provide batch embedding API and caching to avoid recomputing embeddings for unchanged documents.

Model choices & tradeoffs:

- sentence-transformers/all-MiniLM-L6-v2: small, fast, decent quality (recommended for local).
- Larger models (e.g., paraphrase-MPNet) improve retrieval at higher compute cost.
- Optionally support quantized models (float16 or int8) to reduce memory use.

3.4 Vector Index (FAISS)

Responsibilities:

- Store normalized vectors and provide fast top-k similarity search.
- Persist index to disk and support incremental updates (add/remove vectors).
- Expose search parameters: nprobe, ef_search (if using HNSW or IVF), top_k.

Index strategy:

- For small to medium datasets (<100k chunks): use IndexFlatIP (exact inner-product on normalized vectors = cosine similarity).
- For larger datasets: use HNSW or IVF+PQ for memory & speed tradeoffs. Provide reindexing scripts.
- Save mapping from FAISS internal id -> SQLite row id for metadata retrieval.

3.5 Retriever & QA Module

Responsibilities:
- Encode user query into embedding and search FAISS for top-k chunks.
- Rerank retrieved chunks using cross-encoder (optional) or simple lexical BM25 score.
- Build a context window for the LLM: concatenate top-n chunks with sources and separators.
- Call local LLM (Flan-T5 or LLaMA variant) to generate final answer with citation markers.
- Extract/highlight the exact supporting spans to show in UI.

Answer construction pattern:
- Template: 'Question: <Q>\nContext: <chunk1> [source:doc,page] <chunk2> ...\nAnswer:'
- Force citation tokens by appending 'CITE: [doc:page:chunk_id]' when chunk text is used.
- Keep context size within model max tokens; use iterative prompting if needed.

3.6 Metadata Storage (SQLite)

Schema (proposed):
documents(doc_id PK, file_name, indexed_at, num_pages)
pages(page_id PK, doc_id FK, page_number)
chunks(chunk_id PK, page_id FK, char_start, char_end, text, bbox_json, embedding_id)
embeddings(embedding_id PK, faiss_id, vector_normed)
retrieval_logs(log_id PK, query_text, timestamp, top_k_json, user_feedback)

Notes:
- Store bbox_json as JSON list of word boxes for highlighting.
- Keep embedding vectors in FAISS only; store FAISS id in embeddings table.
- Use WAL mode for concurrency if UI and indexing run concurrently.

4. Workflow

4.1 Indexing Phase

Steps:

1. Upload/ingest PDF file.
2. For each page:
   - If native PDF: extract text blocks and word bboxes via PyMuPDF.
   - If scanned: run OCR (Tesseract/PaddleOCR) to get text + per-word boxes.
   - Normalize extracted text and map to coordinates.
   - Chunk page text into overlapping chunks and record char ranges and bbox mappings.
   - Batch embed chunks and add vectors to FAISS; record mapping in SQLite.

Failure handling:

- If OCR fails on a page, mark page as 'ocr_error' and continue. Provide admin retry tool.

4.2 Query Phase

Steps:

1. User enters query in UI.
2. System encodes query to vector and searches FAISS for top-k vectors.
3. Fetch chunk metadata from SQLite for the returned faiss ids.
4. Optionally rerank with a cross-encoder or lexical scoring.
5. Construct prompt/context and call local LLM to synthesize answer.
6. Return answer + list of cited chunks to UI.

4.3 Display Phase

UI Behavior:

- Show answer at top with inline citation markers.
- Below, present a ranked list of chunks (text snippet, doc, page, score).
- Render the PDF page images and draw rectangles around the exact word boxes that were used by the answer.
- Allow the user to click a citation to jump to that page and see the highlighted spans.

Highlight mapping:

- Use the chunk's char range -> word boxes mapping to compute bounding rectangles.
- For multi-line spans, draw multiple rectangles per line.

5. Key Features

- Efficient vector search (FAISS) with configurable index type.
- Evidence-backed answers with explicit citations to document/page/chunk.
- Visual highlighting of evidence on PDF pages (both native and OCR'd).
- Fully local/open-source stack (no external APIs required).
- Configurable chunking, embedding model, and retriever parameters.
  - Admin features: reindex, delete documents, view retrieval logs.

6. Quality Considerations

Accuracy:
- Choose embedding model suited to domain. Evaluate recall/precision using a small test set.
- Use reranking (cross-encoder) to improve final precision for top results.

Latency:
- Batch embedding during indexing. Use IndexFlatIP for small datasets to minimize retrieval latency.
- Cache recent query embeddings and frequently accessed chunk text.

Scalability:
- For >100k chunks, migrate FAISS to HNSW or IVF+PQ and consider sharding.
- Consider moving metadata to Postgres for larger deployments.

Security & Privacy:
- Keep all data on-prem if required. Encrypt disk volumes and limit access.
- Sanitize uploaded PDFs for macros or embedded code (attack surface minimal but check).

7. Alternatives & Tradeoffs

Vector store options:
- FAISS (local, fast, flexible) — recommended for offline, single-host use.
- Chroma (developer-friendly), Weaviate (offers vector + semantic search features), Pinecone (managed).

Embedding models:
- all-MiniLM-L6-v2: fast & small (best default).
- paraphrase-mpnet-base-v2: better quality, larger size.

LLM choices:
- Flan-T5 (local, small-medium sizes): good for structured prompts.
- LLaMA / Vicuna (requires GPU): higher quality but heavier infra.
- Hosted models (OpenAI): best quality but external API dependency and costs.

Chunking:
- Fixed-length vs semantic: fixed-length simpler and predictable; semantic splitting improves coherence but is more complex.

8. Deployment & Dependencies

Core Dependencies:
- Python 3.10+
- Streamlit
- PyMuPDF (fitz)
- FAISS (cpu version) - or faiss-cpu via pip
- sentence-transformers
- transformers (for local LLM)
- sqlite3 (builtin)
- tesseract-ocr (for scanned PDFs)

Deployment Options:
- Local developer workstation (CPU or GPU for faster embedding/LLM).
- On-prem server with GPU for LLM acceleration.
- Cloud VM with GPU when scaled up (use disk snapshots to persist FAISS & DB).

Operational notes:
- Backup FAISS index files and SQLite DB regularly.
- Provide scripts for reindexing and index migration.
- Use systemd or supervisor to run indexing and retriever services.

9. Conclusion & Future Enhancements

Conclusion:

- The proposed RAG system provides a reproducible, local pipeline for PDF Q&A with citations and visual evidence highlighting.

Future work:

- Improve OCR accuracy with domain-specific models.
- Add semantic chunking and adaptive retrieval.
- Integrate user feedback for relevance tuning (reinforcement learning or simple heuristics).
- Add admin UI for reindexing and monitoring.
- Add automated evaluation harness (QA pairs, MRR/NDCG metrics).

10. Appendix

Example SQLite schema (SQL):

```
CREATE TABLE documents(doc_id TEXT PRIMARY KEY, file_name TEXT, indexed_at TIMESTAMP, num_pages INTEGER);
CREATE TABLE pages(page_id TEXT PRIMARY KEY, doc_id TEXT, page_number INTEGER, FOREIGN KEY(doc_id) REFERENCES documents(doc_id));
CREATE TABLE chunks(chunk_id TEXT PRIMARY KEY, page_id TEXT, char_start INTEGER, char_end INTEGER, text TEXT, bbox_json TEXT, FOREIGN KEY(page_id) REFERENCES pages(page_id));
CREATE TABLE embeddings(embedding_id INTEGER PRIMARY KEY AUTOINCREMENT, faiss_id INTEGER, created_at TIMESTAMP);
CREATE TABLE retrieval_logs(log_id INTEGER PRIMARY KEY AUTOINCREMENT, query_text TEXT, timestamp TIMESTAMP, top_k_json TEXT, feedback TEXT);
```

Sample prompt template:

```
Question: {query}
Context:
[DOC: {docname} | PAGE: {pagenum} | CHUNK: {chunk_id}]
{chunk_text}
---
Answer:
```

Chunking config recommendations:

- chunk_size: 1000 chars
- chunk_overlap: 200 chars
- top_k_retrieval: 10
- top_n_context: 3