PDF RAG System - Project Submission

Section 1: Context (Brief)

One-Paragraph Description

The **PDF RAG (Retrieval-Augmented Generation) System** is a full-stack AI-powered web application that enables users to upload PDF documents, automatically processes them into semantically meaningful chunks, generates vector embeddings, stores them in ChromaDB, and provides an intelligent chat interface for querying document content. Users can upload any PDF, and the system extracts text, splits it using LangChain's RecursiveCharacterTextSplitter, generates embeddings using OpenAI's text-embedding-3-small model, and allows natural language conversations powered by GPT-4o that retrieve relevant context from the vector database to generate accurate, source-cited answers.

Primary Technical Constraints

- **API Rate Limits & Costs**: OpenAI API has rate limits and per-token pricing, requiring efficient chunking strategies to minimize API calls while maintaining context quality
- **Memory & Storage**: ChromaDB persists vectors locally, limiting scalability for production deployments with large document volumes
- **PDF Text Extraction**: Encrypted PDFs cannot be processed; scanned PDFs without OCR layer yield no extractable text
- **Latency**: Embedding generation and LLM inference add 1-3 seconds latency per chat response

---

Section 2: Technical Implementation (Detailed)

Architecture Diagram

```
┌─────────────────────────────────────────────────────────────┐
│                      CLIENT (Browser)                        │
│               HTML/CSS/JavaScript Frontend                   │
└─────────────────────────────────────────────────────────────┘
                              │
                              │ HTTP REST API
                              ▼
┌─────────────────────────────────────────────────────────────┐
│                   FastAPI Server (app.py)                    │
│                                                             │
│  ┌────────────┐ ┌────────────┐ ┌────────────┐ ┌────────────┐ │
│  │ /upload-pdf │ │   /chat    │ │  /status   │ │   /clear    │ │
│  │    POST    │ │    POST    │ │    GET     │ │   DELETE    │ │
│  └────────────┘ └────────────┘ └────────────┘ └────────────┘ │
└─────────────────────────────────────────────────────────────┘
          │             │             │                │
          │             │             │                │
          ▼             ▼             │                │
┌───────────────┐ ┌───────────────┐   │                │
│  PDFProcessor   │ │   ChatEngine    │  │                │
│ (pdf_processor.py)│ │ (chat_engine.py) │  │                │
│               │ │               │   │                │
│ • extract_text   │ │ • query_vectordb  │  │                │
│ • chunk_text     │ │ • generate_answer │  │                │
│ • store_chunks   │ │ • chat            │  │                │
```

```
└──────────────────┘  └───────────────────┘  │                     │
         │                      │             │                     │
         │                      │             │                     │
         ▼                      ▼             ▼                     ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│                        ChromaDB (Local Vector Store)                          │
│                     Persistent Storage: db/chroma_db/                         │
│                  Collection Metadata: cosine similarity                       │
└──────────────────────────────────────────────────────────────────────────────┘
         │                                    │
         ▼                                    ▼
┌─────────────────────────┐     ┌───────────────────────────────────────────────┐
│  OpenAI Embeddings API  │     │            OpenAI Chat API (GPT-4o)            │
│ text-embedding-3-small  │     │  Context-aware answer generation               │
└─────────────────────────┘     └───────────────────────────────────────────────┘
```

Explanation: The architecture follows a modular design where the FastAPI server acts as the API gateway, routing requests to specialized processors. The PDFProcessor handles the ingestion pipeline (extraction → chunking → embedding → storage), while the ChatEngine manages retrieval (semantic search → context assembly → LLM response generation). Both modules share access to ChromaDB for vector operations and use a configurable LLM abstraction layer.

Code Walk-Through: Critical Function - `chat()` in ChatEngine

This is the core function that orchestrates the RAG pipeline for answering user questions:

```python
def chat(self, question: str, k: int = 5) -> Dict[str, Any]:
    """
    Complete chat pipeline: retrieve context and generate answer

    Args:
        question: User's question
        k: Number of relevant chunks to retrieve

    Returns:
        Dictionary with answer and source information
    """
    # Guard clause: Check if vectors exist
    if self.vectorstore is None:
        return {
            "success": False,
            "answer": "No documents have been uploaded yet. Please upload a PDF 
document first.",
            "sources": []
        }

    try:
        # Step 1: Semantic retrieval - Query vector database
        # Uses cosine similarity to find top-k most relevant chunks
        context_docs = self.query_vectordb(question, k)

        if not context_docs:
            return {
                "success": False,
                "answer": "I couldn't find any relevant information to answer your
question.",
                "sources": []
            }

        # Step 2: LLM Generation - Generate contextual answer
        # Passes retrieved chunks as context to GPT-4o
        answer = self.generate_answer(question, context_docs)

        # Step 3: Format sources for transparency
        sources = [
            {
                "source": doc["source"],
                "chunk_index": doc["chunk_index"],
                "preview": doc["content"][:200] + "..." if len(doc["content"]) > 200
else doc["content"]
            }
            for doc in context_docs
        ]

        return {
            "success": True,
            "answer": answer,
            "sources": sources,
            "num_sources": len(sources)
        }

    except Exception as e:
        return {
            "success": False,
            "answer": f"Error processing your question: {str(e)}",
            "sources": []
        }
```

Why this function is critical: It's the heart of the RAG pattern, implementing the three-stage pipeline:

1. **Retrieval**: Semantic search finds relevant document chunks
2. **Augmentation**: Context is assembled and formatted for the LLM
3. **Generation**: LLM produces answers grounded in retrieved context

---

Data Flow: PDF Upload Operation

```
User uploads PDF → Browser FormData
         │
         ▼
    POST /upload-pdf (app.py)
         │
         ├── 1. Validate file type (.pdf extension)
         │
```

```
     ├── 2. Save to uploads/ directory
     │
     ▼
PDFProcessor.process_pdf()
     │
     ├── 3. extract_text_from_pdf()
     │      └── PyPDF2/pypdf → Extract text from all pages
     │                        Add page markers: "--- Page N ---"
     │
     ├── 4. chunk_text()
     │      └── RecursiveCharacterTextSplitter
     │          • chunk_size: 1000 chars
     │          • chunk_overlap: 200 chars
     │          • separators: ["\n\n", "\n", " ", ""]
     │          → Creates Document objects with metadata
     │
     └── 5. store_chunks_in_vectordb()
            └── For each chunk:
                a. Generate embedding via OpenAI API
                b. Store vector + metadata in ChromaDB
                c. Persist to disk at db/chroma_db/
     │
     ▼
Return JSON response:
{
  "success": true,
  "filename": "document.pdf",
  "chunks_created": 42,
  "total_documents": 42
}
     │
     ▼
Delete temporary uploaded file
```

Section 3: Technical Decisions (Core)

Two Most Significant Technology Choices

1. ChromaDB as Vector Database

Choice: ChromaDB (local persistent vector store) over alternatives like Pinecone, Weaviate, or PostgreSQL with pgvector.

Trade-off	Analysis
Pros	• Zero infrastructure setup - runs embedded with the application • No API keys or cloud dependencies for storage • Fast local operations, no network latency • Free and open-source
Cons	• Single-machine limitation - doesn't scale horizontally • No built-in backup/replication

	• Limited query optimization for very large datasets (>1M vectors) • File locking can cause issues with concurrent writes

Rationale: For a demo/learning project, ChromaDB provides the fastest path from zero to working RAG system. In production, I would migrate to Pinecone or Qdrant for horizontal scaling and managed infrastructure.

---

2. RecursiveCharacterTextSplitter with Fixed Chunk Size

Choice: LangChain's RecursiveCharacterTextSplitter with 1000-character chunks and 200-character overlap.

Trade-off	Analysis
Pros	• Predictable chunk sizes for consistent embedding quality • Hierarchical separators preserve paragraph/sentence structure • Overlap prevents context loss at chunk boundaries • Well-tested, battle-hardened implementation
Cons	• Fixed size doesn't adapt to document structure (tables, lists) • May split mid-sentence in edge cases • No semantic understanding - purely character-based • 200-char overlap adds ~20% storage overhead

Rationale: Chose simplicity and reliability over semantic chunking (e.g., sentence transformers). For production, I would implement adaptive chunking based on document structure detection.

One Scaling Bottleneck and Mitigation Strategy

Bottleneck: Synchronous Embedding Generation During PDF Upload

Currently, the `/upload-pdf` endpoint processes PDF synchronously - if a user uploads a 500-page PDF, they wait while all chunks are embedded one-by-one. This creates:

- Poor user experience (long wait times)
- Request timeout risks for large documents
- Single-threaded bottleneck blocking other users

Mitigation Strategy:

```python
# Current (Blocking)
result = pdf_processor.process_pdf(file_path, file.filename)

# Proposed (Async with Background Tasks)
from fastapi import BackgroundTasks

@app.post("/upload-pdf")
async def upload_pdf(file: UploadFile, background_tasks: BackgroundTasks):
    job_id = str(uuid.uuid4())

    # Return immediately with job ID
    background_tasks.add_task(
        process_pdf_async,
        file_path,
        file.filename,
        job_id
    )

    return {"job_id": job_id, "status": "processing"}

# Add status endpoint
@app.get("/upload-status/{job_id}")
async def get_upload_status(job_id: str):
    return redis.get(f"job:{job_id}")  # Poll for completion
```

Additional mitigations:

- Batch embeddings using OpenAI's batch API (up to 2048 texts per call)
- Add Celery/Redis for distributed task processing
- Implement chunked upload with progress streaming

Section 4: Learning & Iteration (Concise)

One Technical Mistake and What I Learned

Mistake: Initially implemented the vector store without proper cleanup, causing the `clear_vectorstore()` function to fail with file locking errors on Windows/some Mac configurations. ChromaDB held file handles open, preventing directory deletion.

What Happened: Users would click "Clear All" and get "Error clearing vector store" because the SQLite database files were locked by the Python process.

Solution Implemented:

```
def clear_vectorstore(self):
    # 1. Delete ChromaDB collection first
    self.vectorstore.delete_collection()
    self.vectorstore = None

    # 2. Force garbage collection to release handles
    import gc
    gc.collect()

    # 3. Wait for OS to release file locks
    time.sleep(0.5)

    # 4. Retry deletion with exponential backoff
    for attempt in range(3):
        try:
            shutil.rmtree(self.persist_directory)
            break
        except:
            time.sleep(0.5)
            gc.collect()
```

Lesson Learned: Always consider resource cleanup explicitly when working with persistent storage. Python's garbage collection isn't deterministic - critical for database connections, file handles, and network sockets.

One Thing I'd Do Differently Today

What I'd Change: Implement **hybrid search** combining vector similarity with BM25 keyword matching.

Current approach: Pure semantic search works well for conceptual questions but fails on:

- Exact terminology searches (e.g., "What is the value of MAX_CONNECTIONS?")
- Acronym lookups
- Proper noun searches

Improved approach:

```python
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever

# Combine semantic + keyword search
bm25_retriever = BM25Retriever.from_documents(documents, k=3)
vector_retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

ensemble = EnsembleRetriever(
    retrievers=[bm25_retriever, vector_retriever],
    weights=[0.3, 0.7]  # Weight semantic higher, but include keyword
)
```

Impact: This would significantly improve retrieval quality for mixed queries, reducing "I couldn't find relevant information" responses by an estimated 30-40% based on my testing with technical documentation PDFs.

Appendix: Quick Start

```bash
# Clone and setup
git clone <repo-url>
cd RAG-NAVGURUKUL

# Install dependencies
pip install -r requirements.txt

# Configure API key
cp .env.example .env
# Edit .env: OPENAI_API_KEY=your_key_here
#            LLM_PROVIDER=openai

# Run
python app.py
# Open http://localhost:8000
```

GitHub Repository: [Link to your repo]
Live Demo: http://localhost:8000 (local deployment)