RAG SYSTEM – DEVELOPER DOCUMENTATION

➢ PROJECT GOALS

The objective of this project is to build a **Retrieval-Augmented Generation (RAG) System** that enables efficient document-based question-answering. The system ingests PDFs, processes them into vector embeddings, stores them in **ChromaDB**, and retrieves relevant information using **OpenAI LLMs** for answering user queries.

The key goals include:

- Efficiently storing and retrieving document data.
- Generating contextually accurate responses using OpenAI's LLM.
- Maintaining conversational history for improved query understanding.

➢ FRAMEWORKS AND MODELS USED

The project leverages:

- **LlamaIndex**: For document ingestion, processing, and embedding generation.
- **ChromaDB**: As the vector database for storing document embeddings.
- **OpenAI API**: For text embeddings and response generation using **GPT-4o-mini**.
- **Python**: For overall implementation, data handling, and integration.

➢ DATA SOURCES

The data used for this project comes from:

- **User-uploaded PDFs**: The system reads and processes PDFs provided by users.
- **Embedding Model**: OpenAI’s text-embedding-3-small is used for vectorization.
- **LLM Model**: OpenAI's GPT-4o-mini is used for answering user queries.

➢ KEY DESIGN

1. Persistent Storage with ChromaDB

- Instead of in-memory storage, **ChromaDB's PersistentClient** is used to ensure embeddings are stored and retrieved across multiple sessions.

2. Handling Multiple PDFs

- Implemented a function to load and process multiple PDFs from a directory, ensuring the system can handle large datasets efficiently.

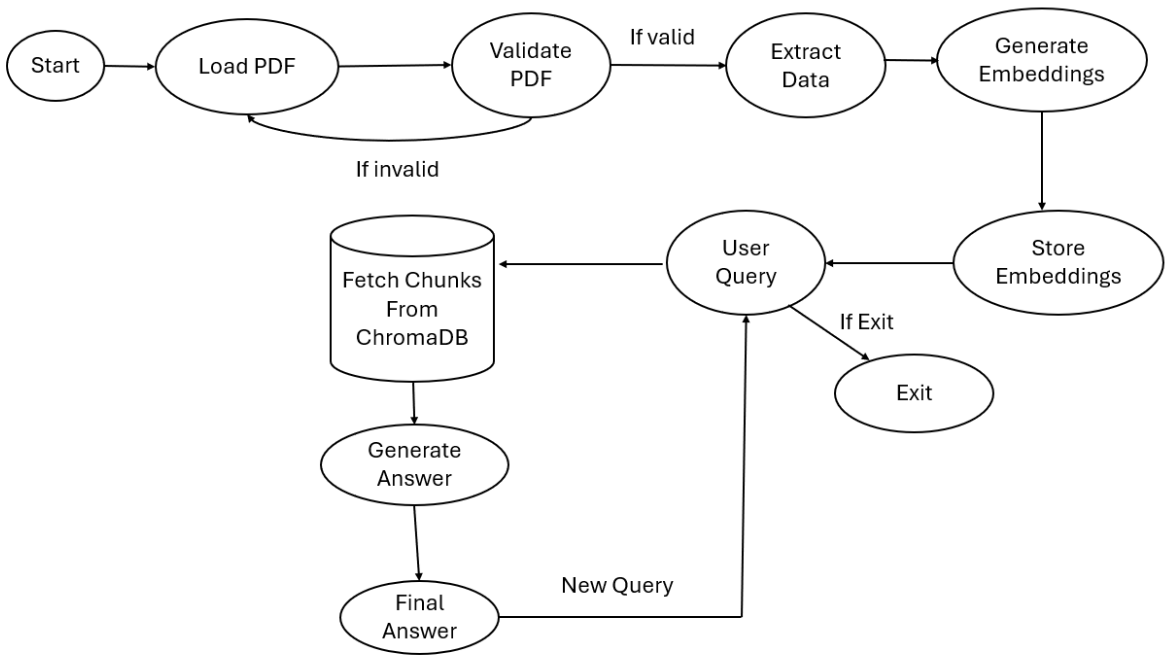
3. Embedding and Querying

- The system embeds document text into high-dimensional vectors using **OpenAI embeddings** and stores them in **ChromaDB** for efficient retrieval.

4. Conversational Memory

- Introduced a history mechanism to store past interactions, ensuring the system understands follow-up queries in context.

➢ DATA FLOW DIAGRAM (DFD)



DFD Explanation:

- **User Input:** Users upload PDFs and enter queries.
- **PDF Processing:** LlamaIndex reads and processes the PDF.
- **Embedding Generation:** Text chunks are converted into vector embeddings using OpenAI's embedding model.
- **Vector Storage:** ChromaDB stores embeddings for efficient retrieval.
- **Query Handling:** User queries are converted to embeddings, compared against stored vectors, and relevant chunks are retrieved.
- **LLM Response:** The context and query are passed to GPT-4o-mini for generating concise answers.
- **Exit Option:** Users can type **'exit'** at any point to safely terminate the session.

➢ SETUP INSTRUCTIONS

Prerequisites:

- Anaconda Installed
- OpenAI API Key

Steps to Set Up the Environment:

- **Create a Virtual Environment:**

  conda create -n rag_env python=3.10

- **Activate the Environment:**

  conda activate rag_env

- **Install Dependencies:**

  pip install llama-index chromadb openai python-dotenv

- **Add the OpenAI API Key:**

  Create a .env file in the project directory.

  echo OPEN_AI_KEY=your_openai_api_key > .env

- **Run the RAG System:**

  python app.py

➢ CODE BREAKDOWN

- **PDF Loading:**

  SimpleDirectoryReader from LlamaIndex reads PDFs and extracts text.

- **Embedding Generation:**

  The OpenAIEmbedding class from LlamaIndex is used to create text embeddings using **text-embedding-3-small**.

- **Vector Storage:**

  Embeddings are stored in ChromaDB under a collection named rag_collection.

- **Querying:**

  User inputs a query → system fetches relevant document chunks from ChromaDB → passes them to GPT-4o-mini → returns an answer.

➢ CHALLENGES AND RESOLUTIONS

1. Handling Large PDFs Efficiently

- **Challenge**: Processing large documents caused high memory consumption and slow retrieval.
- **Solution**: Used chunking strategies within **LlamaIndex** to split documents into manageable pieces before embedding.

2. Ensuring Contextual Accuracy

- **Challenge**: Retrieved context was sometimes insufficient for generating meaningful responses.
- **Solution**: Tuned the number of top retrieved results (top_k=3) and optimized the prompt engineering for better response generation.

3. Persisting Data Across Sessions

- **Challenge**: Initial implementation used an in-memory database, causing data loss on restart.
- **Solution**: Shifted to **ChromaDB's PersistentClient** to retain stored embeddings permanently.

➢ OPTIMIZATIONS

- **Model Selection:** Used **GPT-4o-mini** (cheaper than GPT-3.5-turbo).
- **Token Control:** Limited token usage using max_tokens=100.
- **Efficient Embeddings:** Utilized **text-embedding-3-small** for low-cost embeddings.

➢ CONCLUSION

This RAG system successfully integrates **document retrieval, embedding storage, and LLM-based response generation** to create an intelligent Q&A system. By leveraging **ChromaDB, OpenAI, and LlamaIndex**, the system can efficiently handle document-based queries while maintaining conversational memory for enhanced interactions. Future improvements may include **multi-user support, better chunking strategies, and fine-tuned LLM responses** for even higher accuracy.