An Agentic AI Approach to Retrieval-Augmented Generation for PDF Document Querying

Aditya Bhanudas Biradar
Department of Artificial Intelligence and Data Science
Vidyavardhini's College of Engineering and Technology
Vasai, Mumbai, India
adityabbiradar@gmail.com

Omkar Arun Bhikle
Department of Artificial Intelligence and Data Science
Vidyavardhini's College of Engineering and Technology
Vasai, Mumbai, India
omkar.213179112@vcet.edu.in

Manasvi Nayan Todkar
Department of Artificial Intelligence and Data Science
Vidyavardhini's College of Engineering and Technology
Vasai, Mumbai, India
manasvi.s221597209@vcet.edu.in

Prerna Sanjay Mhatre
Department of Artificial Intelligence and Data Science
Vidyavardhini's College of Engineering and Technology
Vasai, Mumbai, India
prerna.s221517205@vcet.edu.in

***Abstract*—In this paper, we present a Retrieval-Augmented Generation (RAG) system we developed for querying PDF documents, distinguished by its implementation using an agentic AI architecture. Departing from monolithic designs, our system delegates tasks to specialized, autonomous agents for PDF processing, vector database management, and query generation. These agents collaborate within structured workflows orchestrated by the LangGraph framework, ensuring modularity and robustness. Our system integrates standard RAG components, including Sentence Transformers for semantic embeddings and ChromaDB for efficient vector retrieval, with an external LLM accessed via the OpenRouter API. We detail our agentic design, its components, operational workflows, and evaluation considerations, demonstrating the effectiveness of this multi-agent approach for complex information retrieval tasks.**

I. Introduction

Large Language Models (LLMs) excel at text generation but are limited by knowledge cutoffs and the potential for "hallucination" [1]. Retrieval-Augmented Generation (RAG) mitigates these issues by retrieving relevant external information before generation. Feeding this retrieved "context" to the LLM alongside the user's query grounds the response in facts, significantly improving accuracy [1], [2]. Recent approaches such as Dense Passage Retrieval [3] and retrieval techniques that scale to trillions of tokens [4] further validate the effectiveness of dense retrieval in open-domain question answering. Moreover, advances in pre-training, notably BERT [5], have revolutionized language understanding by providing robust contextual embeddings that underpin modern RAG systems.

Our approach extends these ideas by adopting an **agentic AI architecture**, where specialized agents manage distinct tasks. This concept is inspired by foundational work in multiagent systems [6] and enables modular, scalable designs. Here, we describe a Python RAG system designed for querying PDF documents through dedicated ingestion and query pipelines.

II. System Requirements

Building our RAG system required specific Python libraries and access to an external LLM service.

A. Core Python Environment

- **Python:** We built the system using Python, requiring version 3.9 or higher (Python 3.9).

B. Key Libraries

Our system's functionality relies on the following libraries, as specified in our pyproject.toml file:

- **Workflow Orchestration:**
  - langgraph: We use this library to define and execute the multi-step ingestion and query processes as stateful graphs, promoting modularity and simplifying the management of complex flows.
- **Vector Database & Embeddings:**
  - ChromaDB: We selected this open-source vector database, optimized for AI applications, to store text chunks and their vector embeddings, enabling efficient similarity searches crucial for retrieval.
  - Sentence Transformers: This library provides the all-MiniLM-L6-v2 embedding model (specified in config.py) we use to convert text chunks into semantic vector representations.
- **PDF Processing & Text Handling:**
  - unstructured[pdf]: We utilize this library for robust PDF parsing and text extraction.
  - langchain-text-splitters: This provides the RecursiveCharacterTextSplitter algorithm for intelligently dividing texts into smaller, overlapping chunks suitable for embedding.
  - python-magic: We use this library to identify file types, ensuring only valid PDFs are processed during ingestion.

- **LLM Interaction & Data Validation:**
  - requests: We use the requests library to interact directly with the OpenRouter API for LLM communication.
  - pydantic-ai: This library, along with Pydantic, is used for defining structured data models (e.g., QueryInput, QueryResult) and ensuring data validation throughout the workflows.
  - tenacity: We incorporated this library to implement retry mechanisms for API calls, enhancing system robustness.
- **Configuration & Environment:**
  - python-dotenv: Used via load_dotenv() to load environment variables (like API keys) from the .env file.

C. External Services and Configuration

- **LLM Provider (OpenRouter):** Our system relies on the OpenRouter service (https://openrouter.ai/) to access the generative capabilities of the Mistral Small model (mistralai/mistral-small-3.1-24b-instruct:free). Access requires an API key provided via the OPENROUTER_API_KEY environment variable.
- **Configuration File (config.py):** This file centralizes key settings: the embedding model (all-MiniLM-L6-v2), the LLM model identifier, the local path for ChromaDB storage (./chroma_data), and API key retrieval logic.

III. Literature Survey and Core Concepts

Our design and implementation are grounded in key concepts from Natural Language Processing (NLP) and Information Retrieval (IR).

A. Retrieval-Augmented Generation (RAG)

As introduced earlier, RAG [1] enhances LLMs by dynamically fetching relevant information before generation. In addition to the approach detailed in [1], retrieval-augmented pre-training methods such as REALM [2] and dense retrieval techniques exemplified by Dense Passage Retrieval [3] further advance the integration of external knowledge. Moreover, recent work by Borgeaud et al. [4] shows that retrieving from trillions of tokens can significantly boost model performance, reducing hallucinations and increasing factual accuracy.

B. Dense Vector Embeddings

To retrieve information based on semantic meaning, we use dense vector embeddings.

- **Sentence Transformers:** Models like Sentence-BERT [7] excel at creating semantic embeddings. We use all-MiniLM-L6-v2 from the Sentence Transformers library, chosen for its balance of efficiency and performance. Our LocalEmbedder class in tools/embeddings.py wraps this model.

C. Vector Databases

Efficiently searching high-dimensional vectors requires specialized databases.

- **ChromaDB:** We utilize ChromaDB (https://www.trychroma.com/), an open-source vector database. It stores text chunks and embeddings, using indexing techniques like HNSW for fast nearest neighbor searches. Our DatabaseAgent in agents/db_agent.py manages all interactions with ChromaDB.

D. Large Language Models (LLMs)

The generator component of our RAG system is an LLM.

- **Mistral Models & OpenRouter:** We use the mistralai/mistral-small-3.1-24b-instruct:free model via OpenRouter, a platform providing a unified API for various LLMs. This interaction logic is encapsulated in tools/llm_tools.py.

E. Workflow Orchestration and Agentic AI

We adopted an **agentic AI** approach, delegating tasks to specialized agents orchestrated within workflows. This modular design is inspired by foundational work in multiagent systems [6] and supports scalable and robust system architectures.

- **LangGraph:** We use LangGraph (https://github.com/langchain-ai/langgraph) to build our stateful, multi-agent application. It defines workflows as graphs, facilitating a modular, robust, and debuggable agentic architecture. Both our IngestionWorkflow and QueryWorkflow utilize LangGraph for agent coordination.

IV. System Architecture: An Agentic Approach

Our RAG system employs a modular, agentic architecture centered around two primary workflows managed by the RAGSystem class, with specialized agents executing tasks within these LangGraph-orchestrated workflows.

A. High-Level Overview

We conceptualize the system as follows:

1) **Ingestion Pipeline:** Takes a PDF file and user identifier, processes the PDF, and stores its content as embedded chunks in a user-specific ChromaDB collection.
2) **Query Pipeline:** Takes a user question and identifier, retrieves relevant chunks from the user's ChromaDB collection, uses these chunks and the question to prompt the LLM, and returns the generated answer.

B. Core Components

Our system comprises these key modules:

- **Main Interface (main.py):** The RAGSystem class provides ingest() and query() methods.
- **Workflows (workflows)** (using LangGraph):
  - IngestionWorkflow: State graph coordinating PDF processing via PDFAgent and DatabaseAgent.
  - QueryWorkflow: State graph for query handling, involving DatabaseAgent and QueryAgent.

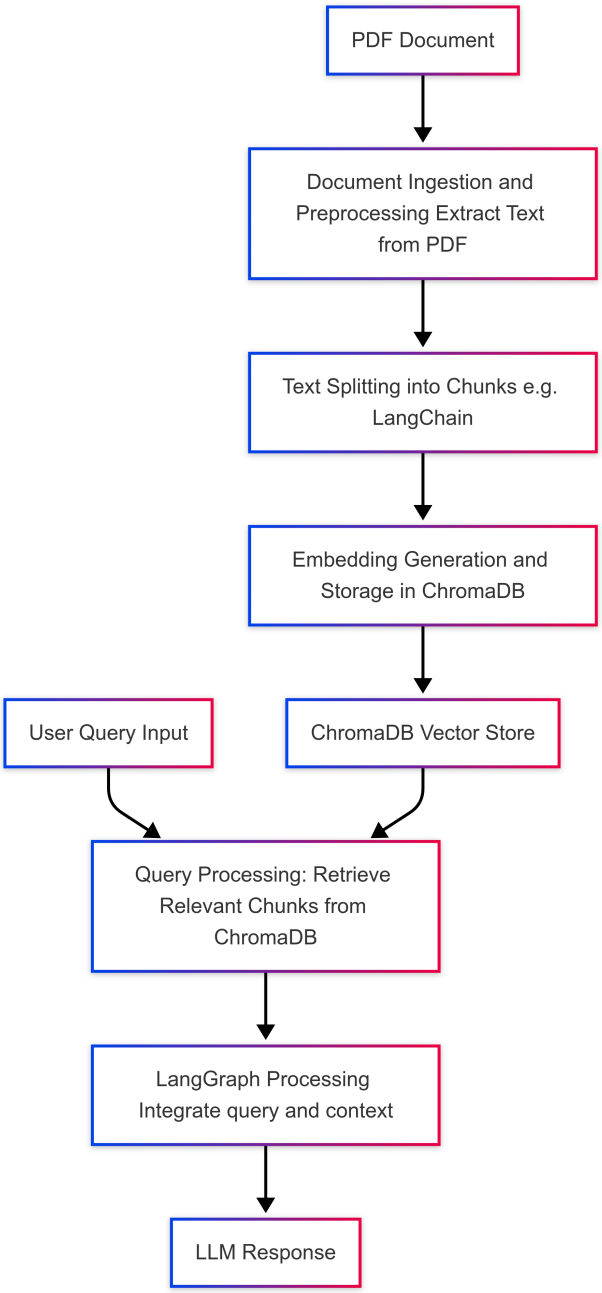

Fig. 1. Agentic AI Architecture Diagram

- **Agents (agents)** (Specialized task handlers):
  - PDFAgent: Handles PDF validation, text extraction, and chunking (pdf_tools.py).
  - DatabaseAgent: Manages all ChromaDB interactions via ChromaManager.
  - QueryAgent: Constructs the LLM prompt, interacts with the LLM service (llm_tools.py), and formats the QueryResult.
- **Tools (tools)** (Core utilities):
  - embeddings.py: Provides the LocalEmbedder.
  - llm_tools.py: Handles LLM API calls (openrouter_request()) with retries.
  - pdf_tools.py: Contains PDF processing functions.
- **Data Models (schema)** (Pydantic): Defines structures (QueryInput, QueryResult, PDFContent).
- **Vector Store (chroma_data)** (ChromaDB): Persistent storage.
- **Configuration (config.py, .env):** Manages settings and keys.

C. Agent Coordination in Workflows

We use LangGraph workflows to coordinate the agents:

- **Ingestion:** The IngestionWorkflow directs the flow: validation → PDFAgent (processing) → DatabaseAgent (storage). Each step updates the shared state.
- **Querying:** The QueryWorkflow manages the sequence: parsing → DatabaseAgent (retrieval) → QueryAgent (prompting/LLM call) → validation. Conditional logic handles potential retries.

V. Implementation Details of Agentic Components

Our key implementation choices highlight the agentic nature:

- **PDF Agent (PDFAgent):** This agent encapsulates PDF handling. It uses python-magic for validation, PyPDF2 for text extraction, and RecursiveCharacterTextSplitter for chunking with overlap.
- **Database Agent (DatabaseAgent):** This agent manages vector storage. It uses the all-MiniLM-L6-v2 model via LocalEmbedder, performs batch embedding, and leverages ChromaDB with cosine similarity and user-specific collections, using the upsert() operation.
- **Query Agent (QueryAgent):** This agent focuses on generation. It takes the retrieved context (top k=3 chunks) and the user query to construct a prompt instructing the LLM to answer *only* from the context, mitigating hallucination. It calls the LLM via the resilient openrouter_request() function (using tenacity).
- **Workflow Orchestration (LangGraph):** We use LangGraph to enable the agentic design by managing state and transitions between agent actions (nodes), providing modularity and facilitating debugging.

VI. Evaluation Considerations

Evaluating our agentic RAG system involves assessing both individual agent performance and overall pipeline effectiveness.

A. Illustrative Example

Our main.py script demonstrates functionality by ingesting the "Attention Is All You Need" paper [8] and querying "what is my gender". The expected faithful response, due to our QueryAgent's prompt engineering, is "I don't know", confirming the system's ability to avoid hallucination when context is insufficient.

B. Metrics for Agentic RAG

We apply standard RAG metrics, adapted to assess agent contributions:

- **Retrieval Quality (DatabaseAgent):** Context Relevance metrics (Precision, Recall, MRR, NDCG) evaluate chunk retrieval accuracy.
- **Generation Quality (QueryAgent + LLM):** Answer Faithfulness and Answer Relevance are crucial, often requiring human judgment or specialized models.
- **End-to-End Performance:** Overall QA metrics like Exact Match (EM) and F1-Score assess final output quality against ground truth.

Our current validate node provides a point for integrating automated checks.

VII. Conclusion

In this paper, we demonstrated a practical implementation of querying PDF documents using an agentic AI architecture. By leveraging specialized agents coordinated via LangGraph, combined with standard RAG components like ChromaDB, Sentence Transformers, and an external LLM, we have built a modular and robust solution.

Our agentic approach offers clear benefits in modularity and maintainability. Key strengths include the use of established RAG techniques and resilience through retries. Future enhancements will focus on more sophisticated validation and confidence scoring, improved source attribution, advanced chunking/retrieval strategies, and implementing a comprehensive evaluation framework.

Overall, our system represents a well-structured foundation for an agentic RAG application, showcasing how distinct agents can collaborate effectively within an orchestrated workflow to achieve complex NLP tasks.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[2] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, "Realm: Retrieval-augmented language model pre-training," in *International Conference on Machine Learning*, 2020.

[3] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[4] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, T. Rocktäschel, W. Czarnecki, S. Gehrmann, D. Weissenborn *et al.*, "Improving language models by retrieving from trillions of tokens," *arXiv preprint arXiv:2111.07891*, 2022.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[6] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2008.

[7] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2019.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.