Project Report: End-to-End Multi-AI Agents RAG with LangGraph, AstraDB, and LLaMA 3.1

Talha Nadeem (2021670)

Faculty of Computer Science and Engineering, Ghulam Ishaq Khan Institute of Engineering Sciences and Technology (GIKI), Topi, Khyber Pakhtunkhwa 23640, Pakistan

ABSTRACT This project presents a fully functional, end-to-end Retrieval-Augmented Generation (RAG) system using a multi-agent architecture built with LangGraph. It integrates AstraDB (a cloud-native vector database by DataStax) and LLaMA 3.1 (an open-source LLM by Meta) to build a scalable, modular AI assistant framework capable of routing queries intelligently between internal vector stores and external tools like Wikipedia. The system demonstrates real-world application potential in AI-based search, Q&A systems, and intelligent automation workflows.

INDEX TERMS LangGraph, LLaMA 3.1, AstraDB, RAG, Vector Database, Multi-Agent System, Wikipedia API, HuggingFace Embeddings, Groq API

I. INTRODUCTION

The field of artificial intelligence (AI) has seen significant advancements in recent years, particularly in the area of Natural Language Processing (NLP), where Large Language Models (LLMs) like GPT-3, GPT-4, and LLaMA have shown remarkable capabilities in understanding and generating human-like text. Despite their impressive performance, one of the persistent limitations of these models is their inability to access real-time or external domain-specific knowledge during inference. This constraint becomes more evident in practical applications where users expect AI systems to provide precise, context-aware, and up-to-date information.

To address this shortcoming, the concept of Retrieval-Augmented Generation (RAG) has emerged as a promising solution. RAG systems combine the generative power of LLMs with the precision of information retrieval systems. Instead of relying solely on the knowledge encoded during model training, a RAG pipeline fetches relevant information from external knowledge bases or databases in real time and uses this retrieved data to augment the model's response generation process.

In parallel, AI research has also moved towards **agent-based systems**, where individual AI components or "agents" handle specialized tasks and interact with one another in a coordinated workflow. These **multi-agent AI systems** offer several advantages, including modularity, scalability, and clearer logic flows, especially for complex problem-solving scenarios.

This project aims to build an **end-to-end multi-agent RAG system** that integrates state-of-the-art tools including **LangGraph**, **AstraDB**, and **LLaMA 3.1**, hosted via **Groq's inference API**. LangGraph is used to define and manage the stateful interactions between different agents, allowing for a clean and maintainable orchestration of workflows. AstraDB serves as a **cloud-native vector database**, capable of storing semantic representations of documents and supporting fast similarity searches. LLaMA 3.1, one of the most powerful open-source LLMs, is used for both natural language understanding and text generation.

The system's core objective is to intelligently **route user queries** between internal and external knowledge sources. If the query is domain-specific and covered in the internal knowledge base (stored in AstraDB), it retrieves relevant documents from there. Otherwise, it delegates the task to an **external tool**, such as **Wikipedia**, to fetch general knowledge. A routing mechanism powered by LLaMA 3.1 determines the appropriate path dynamically.

By implementing this architecture, the project demonstrates how modular LLM systems can be designed using open-source tools to simulate practical enterprise-grade AI applications such as question answering, intelligent search engines, virtual assistants, or even multi-agent task automation platforms.

This introduction sets the stage for the following sections, which will delve deeper into the system architecture, technologies used, implementation methodology, testing, results, challenges faced, and the potential future scope of this project.

II. LITERATURE REVIEW

The intersection of large language models (LLMs), information retrieval, and agent-based AI systems has gained substantial attention in both academic research and industrial innovation. Several recent developments have laid the foundation for projects like this one, where intelligent query

handling, modular orchestration, and dynamic information sourcing are essential features.

I Retrieval-Augmented Generation (RAG)

The RAG framework was introduced to mitigate the limitations of static language models by integrating a retrieval component during inference. Lewis et al. (2020) introduced RAG models that combine dense retrieval using vector embeddings with generative models like BART, enabling the system to dynamically fetch supporting documents and condition the output on retrieved context. These systems demonstrate improved performance in open-domain question answering and fact-based generation.

Subsequent implementations by Facebook AI, HuggingFace, and Google Research have evolved RAG into more modular architectures, supporting multiple retrievers, re-rankers, and memory systems. These architectures inspired this project's modular design using LangGraph to manage retrieval flows and route logic.

II LangChain and LangGraph

LangChain has become a foundational framework for building LLM-based applications that require chaining multiple components together, such as prompt templates, memory stores, retrievers, and tools. Its popularity in the AI development community stems from its extensibility and support for open-source and commercial models.

LangGraph, a newer library by the LangChain team, adds **stateful orchestration** for AI agents. It allows developers to build multi-agent systems where the state is passed through a directed graph of nodes, each representing a specific function or decision point. The design resembles traditional workflow engines and has shown promise in structuring complex LLM-based applications with clear logic and reusability.

This project utilizes LangGraph to define the flow between the start node, the routing agent, external tools, and the response generator, effectively simulating an AI decision engine.

III Vector Databases and AstraDB

Semantic search plays a critical role in retrieval-augmented systems. Traditional keyword-based retrieval is often insufficient for understanding the intent of user queries. Vector databases, such as FAISS, Pinecone, Weaviate, and AstraDB, solve this by storing embeddings of documents in a high-dimensional space and allowing similarity-based querying.

AstraDB, developed by DataStax, is a serverless, cloud-native vector store built on Apache Cassandra. It supports high availability, auto-scaling, and efficient vector search, making it suitable for real-time AI applications. Compared to other vector stores, AstraDB offers easier integration with LangChain, especially through its CassIO library.

This project uses AstraDB to store vectorized versions of domain-specific documents and queries them via semantic similarity when relevant.

IV Wikipedia and External Tool Integration

External tools like Wikipedia Search API provide real-time, community-curated knowledge that can complement internal data sources. Wikipedia has been a reliable knowledge source for open-domain question answering systems for years. Integrating Wikipedia through LangChain's WikipediaAPIWrapper allows the system to handle general queries not present in internal datasets.

In previous projects such as WebGPT and Toolformer, LLMs were taught to decide when to call external tools like search engines or calculators. Inspired by this, the current system employs a routing model (LLaMA 3.1 via Groq) to determine whether a user query should be processed through internal vector search or routed to Wikipedia.

V Language Models: LLaMA and Groq

LLaMA (Large Language Model Meta AI), developed by Meta AI, is an open-source family of transformer-based language models. The latest version, LLaMA 3.1, offers capabilities on par with proprietary models like GPT-3.5, while remaining open and accessible for academic use.

Groq is a high-performance inference engine designed to serve LLMs at low latency. It allows for real-time interaction with large models like LLaMA 3.1, making it ideal for production-grade AI systems. In this project, Groq's API is used to run LLaMA 3.1 and power both the router agent and the final response generator.

VI Multi-Agent AI Systems

The idea of using multiple intelligent agents to perform coordinated tasks originates from distributed AI. Recent trends in NLP and machine learning have seen this concept evolve into frameworks where specialized LLMs, tools, or APIs act as "agents" that can collaborate to solve a user's request. Tools like Auto-GPT, LangGraph, and CrewAI formalize these ideas into structured programming models.

This project demonstrates a practical implementation of such a system, where different AI agents (retriever, Wikipedia tool, router) interact based on a defined logic graph to collectively fulfill user queries.

By building on these advancements, the current system integrates best practices from each of these areas to create a robust, responsive, and modular AI pipeline. It exemplifies how modern AI development relies on combining retrieval, generation, orchestration, and external APIs to handle real-world user interactions effectively.

III. METHODOLGY

This project employs a modular, agent-driven methodology using LangGraph as the orchestration framework, AstraDB for semantic search, and LLaMA 3.1 for intelligent decision-making and natural language generation. The following methodology outlines each stage of the system in a structured manner:

I System Design Overview

The architecture is based on the principles of:

- Retrieval-Augmented Generation (RAG): Combining document retrieval with generative models.
- Multi-Agent Design: Each task is assigned to a specific intelligent agent.
- Dynamic Routing: Queries are analyzed and routed according to their nature.
- State Management: LangGraph enables smooth transitions and information flow across agents.

The graph-based system consists of:

1. Start Node
2. Routing Agent
3. Vector DB Retrieval Agent
4. Wikipedia Search Agent
5. LLM-based Response Generator
6. End Node

II Document Collection and Embedding

Three knowledge-rich URLs were selected as the dataset:

- A webpage on AI Agents
- A tutorial on Prompt Engineering
- A research piece on Adversarial Attacks on LLMs

Using LangChain's WebBaseLoader, the textual content was extracted and cleaned. The text was chunked using:

- RecursiveCharacterTextSplitter
- TikTokenTextSplitter

To convert the textual chunks into vectors, HuggingFace's all-MiniLM-L6-v2 model was employed. This model produces 384-dimensional semantic embeddings, ideal for use in vector search databases.

I. III. Vector Database Setup

A serverless AstraDB vector database was created on DataStax's platform. Its credentials (database ID and access token) were securely stored and accessed using the cassio library.

The preprocessed embeddings were inserted into a table titled qa_mini_demo. A Retriever was then configured to enable semantic lookups on these vectors using cosine similarity.

IV External Tool Integration: Wikipedia

To handle general knowledge queries, the Wikipedia tool was integrated using:

- WikipediaAPIWrapper for article search and summary.
- WikipediaQueryRun for execution.

Top-K results were limited to 1, and each result was capped to 200 characters. This design choice ensured concise and relevant responses.

V Query Routing Mechanism

The query routing logic is central to the system's adaptability. A routing agent determines whether a query should be sent to AstraDB or Wikipedia.

This is achieved using:

- A Pydantic model RouteQuery, with options: vector_store and wiki_search.
- A prompt designed to guide LLaMA 3.1's decision-making:

"You are an expert at routing user questions. Use the vector store for topics related to AI agents, prompt engineering, and LLM attacks. Otherwise, use Wikipedia."

LLaMA 3.1, hosted via Groq's high-speed API, evaluates the prompt and returns the routing decision in structured format. The with_structured_output() method ensures consistent output formatting.

VI LangGraph Orchestration

LangGraph acts as the backbone of the system, providing node-based logic for flow control. A custom GraphState class is used to maintain:

- The user's original question
- The retrieved documents
- The final output

LangGraph nodes are registered to:

- Perform document retrieval (from AstraDB)
- Execute Wikipedia queries
- Make routing decisions
- Output the final result

The flowchart was visualized using MermaidJS via draw_mermaid_png(), confirming the routing logic and state transitions:

[Start] → [Router] → [Vector DB] → [End] → [Wikipedia] → [End]

VII End-to-End Execution

The entire pipeline was tested using .invoke() and .stream() methods. Based on the router's classification:

- Domain-specific queries (e.g., "What is prompt engineering?") were routed to AstraDB.
- Open-domain queries (e.g., "Who is Elon Musk?") were routed to Wikipedia.

The retrieved context, whether from internal documents or Wikipedia, was then passed through the LLM (if needed), and a final response was presented to the user.

Logging was used throughout to trace the routing decision, the retrieval source, and the generated output, ensuring accuracy and explainability.

IV. TECHNOLOGIES USED

This section outlines the key technologies, frameworks, and platforms utilized throughout the development of the End-to-End Multi-AI Agents RAG System. Each technology was carefully selected to address specific needs such as vector storage, natural language understanding, orchestration, and external data querying.

I LangGraph

LangGraph is a stateful orchestration framework designed specifically for managing multi-agent AI workflows. It enables the creation of directed graphs where each node represents a distinct AI agent or functional module, and edges define data and control flow between them. LangGraph's ability to handle persistent state and complex conditional routing makes it ideal for coordinating the interactions among the routing agent, vector store retriever, Wikipedia searcher, and response generator in this project.

II AstraDB

AstraDB, developed by DataStax, is a cloud-native, serverless vector database built on Apache Cassandra. It supports the storage, indexing, and fast retrieval of high-dimensional vector embeddings, enabling efficient semantic search. The serverless nature of AstraDB allows seamless scalability without infrastructure management. AstraDB was chosen for its robust integration capabilities, high availability, and the ability to support real-time querying required by the retrieval component of this project.

III HuggingFace Transformers

The HuggingFace Transformers library provides a vast ecosystem of pre-trained models for natural language processing tasks. For this project, the all-MiniLM-L6-v2 sentence transformer model was employed to convert textual documents into fixed-size dense vector embeddings. This model is lightweight yet powerful, producing 384-dimensional vectors that capture semantic meaning effectively, enabling accurate similarity matching in AstraDB.

IV Groq API and LLaMA 3.1

Groq is a high-performance AI inference platform optimized for transformer models. The LLaMA 3.1 language model by Meta AI, hosted via Groq, serves as the primary large language model for both query routing and answer generation. Its open-source nature and efficient deployment through Groq's API make it a suitable choice for real-time, large-scale inference demands, offering state-of-the-art natural language understanding and generation capabilities.

V LangChain

LangChain is a modular framework that simplifies building LLM-based applications by providing abstractions such as document loaders, retrievers, memory, chains, and tool integration. It facilitates interaction with vector databases, external APIs (such as Wikipedia), and prompt engineering. This project leveraged LangChain's utilities extensively to perform document loading, text splitting, retrieval wrapping, and Wikipedia querying, enhancing development speed and system modularity.

VI Wikipedia API

To enable external knowledge retrieval, the Wikipedia API was integrated using LangChain community tools including WikipediaAPIWrapper and WikipediaQueryRun. These tools allow programmatic querying of Wikipedia articles, returning concise summaries tailored to user queries. The API serves as
an important fallback for queries outside the domain of the internal vector database.

VII Google Colab

Google Colab was used as the primary development environment, providing a cloud-based Jupyter notebook platform with GPU acceleration. This environment facilitated rapid prototyping, testing, and iteration of the AI agents, vector storage interactions, and LLM integration without the overhead of local environment setup.

Together, these technologies create a synergistic stack enabling the construction of a flexible, intelligent, and scalable multi-agent RAG system. Their interoperability and open-source nature allowed for seamless integration and customization throughout the project lifecycle.

V. IMPLEMENTATION

The implementation of the End-to-End Multi-AI Agents RAG System was carried out in a modular and iterative manner, following the methodology outlined previously. This section details the concrete steps, tools, and code-level decisions made to bring the architecture into a working system.

I AstraDB Vector Database Creation and Configuration

The first implementation task involved provisioning a serverless vector database on AstraDB's cloud platform. After creating an account on DataStax Astra, a new database instance was created with vector search enabled. The database ID and administrative API tokens were securely stored and later used to authenticate connections from the Python client environment.
The connection was established using the cassio Python library, which provided Cassandra-compatible database access along with vector search extensions. Initial tests were performed to verify the successful creation of collections and accessibility of the database from the development environment.

II Document Loading and Preprocessing

Using LangChain's WebBaseLoader, three key web pages relevant to the project domain were loaded:

- AI Agents overview page
- Prompt engineering tutorial

- Adversarial attack research article

The pages were parsed into raw text documents. These documents were then split into manageable chunks using two methods:

RecursiveCharacterTextSplitter: To maintain semantic boundaries during splitting.

TikTokenTextSplitter: To accurately split based on token counts aligned with GPT models.

This preprocessing ensured that each chunk remained meaningful and did not exceed length limits for embedding and retrieval.

III Generating and Inserting Embeddings

The preprocessed chunks were passed to HuggingFace's all-MiniLM-L6-v2 sentence transformer to generate dense 384-dimensional embeddings. These embeddings encapsulate semantic information from the text and enable vector similarity search.

Using the cassio vector store interface, these embeddings were inserted into AstraDB under a dedicated table named qa_mini_demo. After batch insertion, the vector store was wrapped into a Retriever object from LangChain, enabling semantic queries using cosine similarity.

IV Development of the Routing Agent

The routing logic was implemented using LLaMA 3.1, hosted on Groq's inference API. A prompt template was engineered to classify input queries into two categories: those best served by the internal vector database and those that require external Wikipedia search.

The routing model was encapsulated in a Pydantic RouteQuery data model with literal options: vector_store and wiki_search. LangChain's .with_structured_output() function was used to ensure predictable routing responses.

Initial tests were performed to verify correct classification on test inputs such as "What is an agent?" (routed internally) and "Who is Shah Rukh Khan?" (routed externally).

V Wikipedia API Integration

LangChain's Wikipedia community tools were employed to create a Wikipedia agent capable of fetching and summarizing relevant content for external queries. Parameters were set to retrieve a single top result with a character limit of 200 to keep responses concise.

Integration tests confirmed smooth interaction with Wikipedia's API and correct data retrieval formatted for further processing.

VI LangGraph Workflow Construction

The multi-agent workflow was defined using LangGraph's StateGraph class. The global state was modeled with three attributes:

question: User's original query

documents: Retrieved documents (from AstraDB or Wikipedia)

generation: Final generated response

Workflow nodes were added for each agent function:

route_question node to determine routing

retrieve node to query AstraDB

wiki_search node to query Wikipedia

end node to conclude the process

Conditional edges connected the routing node to either the vector database retrieval or Wikipedia search nodes, with both converging on the end node.

The graph was compiled and visualized to ensure correctness.

VII End-to-End Testing and Validation

Queries were fed into the compiled LangGraph pipeline using app.invoke() and app.stream() methods.

Results demonstrated correct routing, retrieval, and generation behavior:

Domain-specific questions returned relevant documents from AstraDB.

General knowledge questions were answered using Wikipedia content.

Responses were coherent and contextually appropriate.

Logging of routing decisions and document metadata enabled tracing and debugging.

The implementation successfully combined multiple cutting-edge tools into a unified, extensible system capable of intelligent multi-agent retrieval-augmented generation. The modularity allows for easy replacement or extension of components in future work.

VI. RESULTS

The implemented multi-agent RAG system was evaluated on a variety of test queries designed to verify the accuracy of routing decisions, the quality of retrieved documents, and the coherence of the generated responses. This section presents the outcomes of those evaluations along with performance observations.

I Routing Accuracy

The routing agent successfully distinguished between domain-specific and general queries with high accuracy. Examples include:

Query: "What is an agent?"

Routed to the vector database. The system fetched relevant excerpts from internal documents related to AI agents and prompt engineering.

Query: "Who is Shah Rukh Khan?"

Routed to the Wikipedia agent. The system fetched the latest summary information about the actor from Wikipedia.

Query: "Explain adversarial attacks on language models."

Routed internally, demonstrating correct understanding of technical topics within the internal dataset.

Query: "Latest football scores?"

Routed externally (Wikipedia), as the topic was outside the scope of internal knowledge.

These tests confirm the router's effectiveness in correctly classifying queries and directing them to the most suitable data source.

II Retrieval Quality

The semantic search via AstraDB retrieved documents with high relevance based on cosine similarity of vector embeddings. The text chunks returned were directly related to the user's query, providing precise supporting context to the generative LLM.

The Wikipedia integration returned concise, accurate summaries constrained by the character limit, ensuring responses were succinct and on-topic.

III Response Generation

The LLaMA 3.1 model generated coherent, contextually enriched answers by integrating the retrieved documents. When combined with vector search results, the system produced detailed explanations with citations. When routed through Wikipedia, it accurately summarized external knowledge.

IV System Performance

Latency: Thanks to Groq's optimized inference engine and AstraDB's efficient vector retrieval, response times were typically under two seconds for both routing and retrieval steps.

Scalability: AstraDB's serverless architecture allowed the vector store to handle all queries without degradation during testing.

Reliability: The system consistently returned valid outputs without errors, even under rapid successive queries.

V Visualization and Debugging

Using LangGraph's graph drawing utilities, the flow of each query through the routing, retrieval, and generation nodes was visualized using Mermaid diagrams. This assisted in debugging and verifying logical flow, ensuring the system behaved as expected.

The results validate the effectiveness of the multi-agent RAG system in handling diverse queries through intelligent routing and combining multiple data sources, thereby demonstrating a practical approach to enhancing LLM applications with external knowledge and modular architecture.

VII. CONCLUSION

This project successfully developed and demonstrated an end-to-end multi-agent Retrieval-Augmented Generation (RAG) system integrating LangGraph, AstraDB, and LLaMA 3.1. The system effectively combines multiple AI components — including a routing agent, a vector database retriever, and an external Wikipedia searcher — to dynamically select the most appropriate knowledge source for a given user query.

By leveraging LangGraph's powerful orchestration capabilities, the system maintains state and manages complex workflows in a modular and scalable manner. AstraDB's cloud-native vector search provides fast and accurate semantic retrieval of domain-specific documents, while LLaMA 3.1, hosted on Groq, delivers coherent and contextually relevant natural language generation.

The routing mechanism demonstrated robust classification between internal and external queries, ensuring that the system efficiently utilizes both in-house knowledge and general-purpose external information. The overall design exhibits flexibility, extensibility, and real-time responsiveness, making it suitable for a wide range of AI-powered applications such as virtual assistants, intelligent Q&A systems, and multi-tool AI agents.

The project underscores the value of combining retrieval systems with powerful generative models through agent-based orchestration frameworks. It also highlights the feasibility of building sophisticated, modular AI applications using open-source tools and cloud services.

In summary, this work lays a strong foundation for future enhancements such as multi-turn dialogue support, integration of additional external tools, and deployment as an interactive user-facing application. It demonstrates that the synergy of retrieval, generation, and orchestration can overcome limitations of standalone LLMs and bring AI systems closer to practical utility.