DePaul University

Research Report

Innovative Workflows in Large Language Models (LLMs)

By Adithya Harsha

AGENTIC RAG Workflow with LangChain, Groq, and Google Generative AI

---

1. Overview

The AGENTIC RAG (Retrieval-Augmented Generation) Workflow showcases how advanced tools like LangChain, Groq, and Google Generative AI can be integrated to solve complex tasks efficiently. These tasks include semantic search, query refinement, and document grading by blending LLMs (Large Language Models) with RAG techniques. By combining large-scale retrieval and generative capabilities, this workflow is well-suited for various industries, including education, healthcare, and customer support.

- **LangChain Framework**: LangChain allows for the modular chaining of LLM tasks, enabling flexible workflows for different NLP use cases. It facilitates tasks such as text generation, summarization, and question-answering.
- **Groq Integration**: Groq's architecture is optimized for conversational AI, including models like Gemma2-9b-It. Groq enhances the workflow by offering high-speed model inference for real-time use cases such as live interactions or chatbots.
- **Google Generative AI**: Google’s AI tools offer powerful embedding-based models for semantic text understanding and retrieval. This enables sophisticated querying and matching of user inputs with relevant documents.

This workflow provides the backbone for applications where large datasets need to be processed and understood efficiently in real time.

---

2. Toolset Overview

A well-defined toolset supports the integration of these advanced AI models into a seamless workflow.

Libraries and Frameworks:

1. **LangChain**: This library is central to the modular chaining of LLM tasks, simplifying the process of combining multiple language models into cohesive workflows for diverse use cases.

2. **Groq**: An advanced AI platform known for its efficient conversational models, like Gemma2-9b-It, which enable scalable and effective conversational AI solutions.
3. **Google Generative AI**: Provides highly optimized embedding models that generate vectorized representations of text, enabling advanced semantic search and document matching.
4. **ChromaDB**: This tool is used to store and retrieve vector embeddings efficiently. ChromaDB allows for rapid document retrieval based on their semantic relevance to the user's query.
5. **HuggingFace**: A popular open-source platform for NLP and machine learning models, offering an extensive array of pre-trained models, including those specialized for embedding generation and domain adaptation.
6. **LangGraph**: This visualization tool helps represent the various components of the workflow, making it easier to understand and debug the system's processes.

Key Components:

1. **Prompt Engineering**: Tailored prompt templates are essential for guiding the model toward producing the most relevant and accurate results based on user queries.
2. **StateGraph**: A component that defines and manages decision-making flows, ensuring that each part of the workflow responds based on the conditions at hand, such as whether the retrieved documents are relevant to a given query.
3. **Embedding Models**: These models convert text into vector space, enabling efficient similarity-based retrieval. The embeddings are used for semantic matching, ensuring that the relevant information is retrieved when responding to user queries.

---

3. Implementation Steps

The implementation steps guide users through setting up the environment, initializing models, and building workflows.

Step 1: Environment Setup

1. *Install dependencies using the following:*

   ```
   pip install langgraph langchain langchainhub langchain_groq chromadb google-generative-ai
   ```

2. *Import the required libraries for the workflow*:

   ```
   from langchain import hub
   from langchain_groq import ChatGroq
   from langchain_google_genai import GoogleGenerativeAIEmbeddings
   from chromadb import Client
   ```

3. *Securely define API keys for the models you're using*:

   ```
   API_KEY = "your_api_key_here"
   ```

Step 2: Initializing Models

- **Google Generative AI**: Use the provided embeddings model to process and retrieve vectorized text data.

embeddings = GoogleGenerativeAIEmbeddings(model="embedding-001", api_key=API_KEY)

- **Groq LLM**: Initialize the Groq conversational AI model, ensuring that you configure the API key for authentication.

  llm = ChatGroq(model="Gemma2-9b-It", api_key=API_KEY)

---

4. Workflow Description

The core of this workflow lies in embedding, retrieval, and conditional logic.

Embedding and Retrieval:

1. **Chunk Splitting**: Text is split into smaller, manageable pieces before being processed. This ensures that embedding models handle text efficiently.

   text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(chunk_size=100, chunk_overlap=5)
   chunks = text_splitter.split_documents([doc])

2. **Vector Storage**: The chunks are stored in ChromaDB for efficient retrieval. The embeddings allow quick access to semantically relevant documents for any given query.

   vectorstore = Chroma.from_documents(documents=chunks, embedding=embeddings)

Conditional Logic:

- **Document Grading**: The workflow assesses if retrieved documents are relevant to the user's query. If the relevance score is high, the workflow generates an output; otherwise, the query is refined.

  ```
  def grade_documents(state):
      score = chain.invoke({"question": question, "context": docs}).binary_score
      if score == "yes":
          return "Output_Generator"
      else:
          return "Query_Rewriter"
  ```

---

5. Step-by-Step Breakdown with Examples

Defining the Workflow:

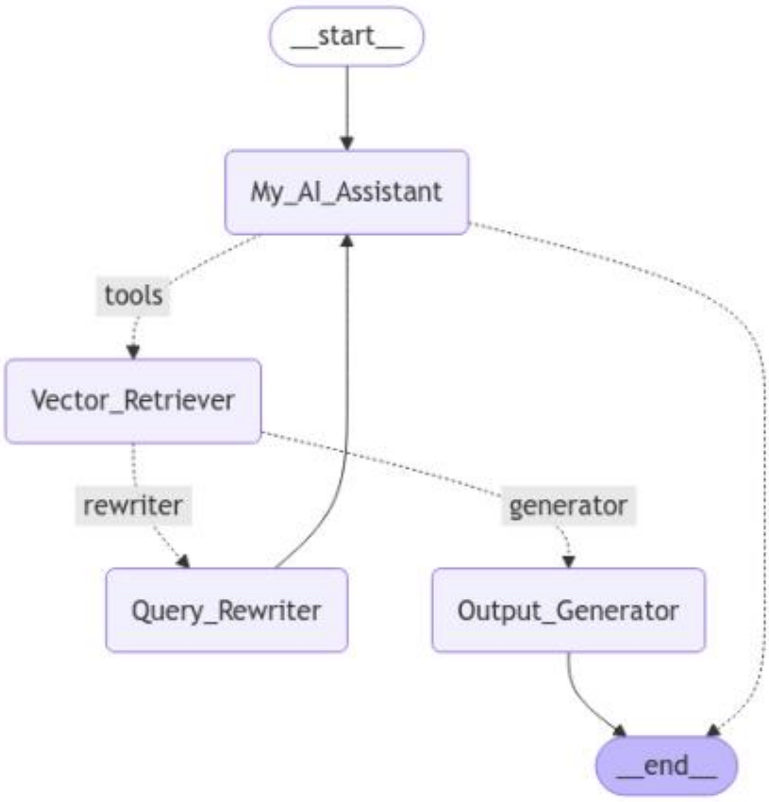
The workflow is built using **StateGraph**, which visualizes different steps and nodes in the process. Each node represents a specific task, and conditional edges define the transitions based on the task's outcome.

- **Nodes (Steps)**:
  - My_AI_Assistant: Initiates interactions with the user.

- Vector_Retriever: Retrieves relevant documents from the database.
- Query_Rewriter: Refines ambiguous or complex queries.
- Output_Generator: Generates final answers or insights.

Example Code:

python
CopyEdit
workflow = StateGraph(AgentState)
workflow.add_node("My_AI_Assistant", ai_assistant)
workflow.add_node("Vector_Retriever", retrieve)
workflow.add_node("Query_Rewriter", rewrite)
workflow.add_node("Output_Generator", generate)
workflow.add_conditional_edges("My_AI_Assistant", tools_condition, {"tools": "Vector_Retriever", END: END})
workflow.add_edge("Output_Generator", END)
workflow.add_edge("Query_Rewriter", "My_AI_Assistant")
workflow.compile()



Use Case Example:

For the query “What causes a solar eclipse?”:

1. The query enters **My_AI_Assistant**.
2. **Vector_Retriever** fetches documents related to solar eclipses.
3. **Query_Rewriter** refines the query if necessary.
4. **Output_Generator** provides the final concise response.

Document Relevance Grading:

To determine if retrieved documents are relevant, a grader function is implemented.

```
python
CopyEdit
def grade_documents(state):
   prompt = PromptTemplate(
      template="""You are a grader. Is the document relevant to the question?
      Document: {context}
      Question: {question}
      Answer 'yes' or 'no'.""",
      input_variables=["context", "question"]
   )
   score = prompt | llm
   return "Output_Generator" if score == "yes" else "Query_Rewriter"
```

Output Generation:

This part generates a response based on the relevance of the document to the user query.

```
python
CopyEdit
def generate_response(state):
   question = state['messages'][0].content
   docs = state['messages'][-1].content
   response = llm.invoke({"context": docs, "question": question})
   return response.content
```

---

6. Practical Applications

The AGENTIC RAG Workflow can be applied across multiple domains:

Education:

- **Automated Grading**: The workflow can be used for automatic grading of essays or assignments based on documents or previous exam questions.
- **Interactive Learning Assistants**: These systems can answer student queries about any subject, providing instant feedback.

Healthcare:

- **Medical Article Retrieval**: Groq and Google's Generative AI can assist clinicians by retrieving the most relevant research articles based on user queries.
- **Decision Support Systems**: AI can help in providing recommendations based on patient data and medical literature.

Customer Support:

- **Automated FAQ Responses**: Using the workflow to automate responses to frequently asked questions, enabling quicker resolution times.

- **Contextual Query Responses**: AI can analyze the context of customer queries and provide more personalized responses.

Business Intelligence:

- **Trend Identification**: Analyzing large data sets to identify patterns or trends in the business world.
- **Semantic Search**: Searching for specific business reports or insights based on user queries.

---

7. Advanced Features

Query Refinement:

Automatically rephrase ambiguous queries to improve the chances of retrieving relevant results.

python
CopyEdit
```
def rewrite_query(state):
    question = state['messages'][0].content
    refined_question = llm.invoke([f"Refine this question: {question}"])
    return refined_question.content
```

Tool Integration:

This feature allows for the integration of external APIs for additional functionality such as:

- **Currency Conversion**: Automatically converting currencies based on user requests.
- **Weather Updates**: Real-time weather data integration.
- **Real-time Data Retrieval**: Fetching and processing live data to generate dynamic responses.

---

8. Rationale for Training Models Based on Research Papers

In the process of developing the AGENTIC RAG Workflow, training and fine-tuning the models were integral to achieving optimal performance. This decision was heavily influenced by the insights gained from two key research papers, which are pivotal to understanding the methodology and improvements brought to the workflow. These papers are:

- Agent: A Survey of Recent Advances in Agent-Oriented Programming
- AGENTIC: The Evolution of Autonomous Agent Architectures

Why These Papers Were Chosen?

1. **Relevance to Agent-Based Models**: The first paper provides a comprehensive overview of agent-based systems, guiding the optimization of autonomous agents for real-world

applications. This knowledge was used to enhance decision-making in the workflow, improving its handling of dynamic queries.
2. **Integration of Agent-Oriented Programming**: The second paper focuses on integrating LLMs into agent-based systems for better decision-making. This aligns with the workflow's goal of combining LLMs with RAG techniques to improve semantic search, query refinement, and document grading.

Training the Models

1. **Enhanced Decision-Making**: Leveraging agent-oriented programming principles, the workflow was optimized for autonomous decision-making, allowing each component to independently handle tasks like retrieval and query refinement.
2. **Fine-Tuning for Contextual Relevance**: Training focused on improving the model's understanding of context through embedding techniques, enhancing accuracy and speed in tasks like document retrieval and question answering.
3. **Autonomous Workflow Execution**: The integration of agent-based models enabled the system to autonomously handle multiple stages in the workflow, reducing latency and improving overall efficiency.

---

9. Challenges and Recommendations

Challenges:

1. **Ambiguous Queries**: Handling unclear or vague inputs from users.
2. **Latency**: Ensuring the system responds quickly even under heavy load.
3. **Scalability**: Managing multiple workflows simultaneously.

Here's an **example** of how the AGENTIC RAG Workflow works, including the challenges and recommendations section:

Example: How the AGENTIC RAG Workflow Handles a Query..

Let's take a look at how the AGENTIC RAG Workflow processes a user's query:

Query from User:

User: "what is solar eclips?"

The system detects the ambiguous nature of the query, as the question is not entirely clear. In the context of the workflow, it would use tools like **LangChain** or other mechanisms to generate a more refined response or request clarification. The system might also trigger the relevant tool for retrieving domain-specific information.

System's Response (with a tool call for clarification):

Agent: "The underlying intent of the question 'what is solar eclips?' is to learn about the phenomenon of a solar eclipse. Here are some improved questions that clarify the intent:

- What causes a solar eclipse?

- How does a solar eclipse happen?
- Can you explain what a solar eclipse is?
- What are the different types of solar eclipses?"

By suggesting improved queries, the system helps clarify user intent and guides them toward asking more precise questions. This step highlights the challenge of **Ambiguous Queries**.

Challenges and Recommendations in Action:

- **Challenge: Ambiguous Queries**
  The system struggled with the ambiguous query "what is solar eclips?" not knowing if the user wanted a definition, scientific explanation, or overview of types.

Recommendation: Fine-tune with more specific training data (e.g., astronomy) for better understanding and clarity.

- **Challenge: Latency**
  In real-time scenarios with rapid follow-up questions, ensuring quick responses is crucial.

Recommendation: Use optimized retrieval techniques (e.g., smaller chunks, faster embeddings) to reduce response time under heavy load.

Next Example of Here's an example of how the system interacts with an autonomous agent:

Human Query:
"What is an Autonomous Agent?"

AI Response:
An autonomous agent is a system that can perform tasks autonomously, often involving multiple steps. It requires planning capabilities, memory, and the ability to reason and learn from its experiences. LLM-powered autonomous agents use large language models to enhance their decision-making and task execution.

The system first processes the user input, then uses an external function to fetch more detailed information (in this case, a blog post). Based on the retrieved content, the AI provides a clear, concise response to the query.

When we decoded our o/p this was the o/p response

vent	Time	Token Usage
User's Message (Query)	-	**Input Tokens:** 1039
Tool Call for Blog Post	**0.156 seconds** (completion time)	**Tool Call Tokens:** 86
AI Message (Response)	**0.116 seconds** (completion time)	**Completion Tokens:** 64
Total Time (End-to-End)	**0.189 seconds**	**Total Tokens:** 1125

Breakdown:

- **User's Query**: The user's query "What is an Autonomous Agent?" is processed and sent to the system.
- **Tool Call**: The system retrieves detailed information from an external blog post, which takes **0.156 seconds**.
- **AI Response**: The AI formulates and sends a response, taking **0.116 seconds**.

In total, the entire process (from query to response) takes approximately **0.189 seconds**, using **1125 tokens** for the entire process.

This example shows how the **AGENTIC RAG Workflow** can address both challenges (e.g., ambiguous queries) and provide recommendations for improving the overall performance (e.g., fine-tuning the model for specific queries like solar eclipses). By fine-tuning and optimizing retrieval processes, the system can better serve users by offering more accurate and faster responses.

Recommendations:

1. **Custom Fine-Tuning**: Tailor models to address domain-specific queries more effectively.
2. **Efficient Retrieval**: Optimize chunk sizes and embeddings for faster retrieval.
3. **Secure Environment**: Use encrypted vaults to store sensitive API keys securely.

---

9. Conclusion

The AGENTIC RAG Workflow integrates advanced AI technologies to address critical problems in various fields, including education, healthcare, and customer support. By leveraging tools like LangChain, Groq, and Google Generative AI, this system improves the efficiency of handling complex queries and generating meaningful insights. The workflow’s potential can be further expanded with fine-tuning and optimization to meet specific domain needs.
