Adaptive Framework Support through Self-Reflective Retrieval-Augmented Generation System

CSE 495/496 - Second Presentation

Eren Torlak

210104004090

Project Supervisor:
Dr. Salih Sarp

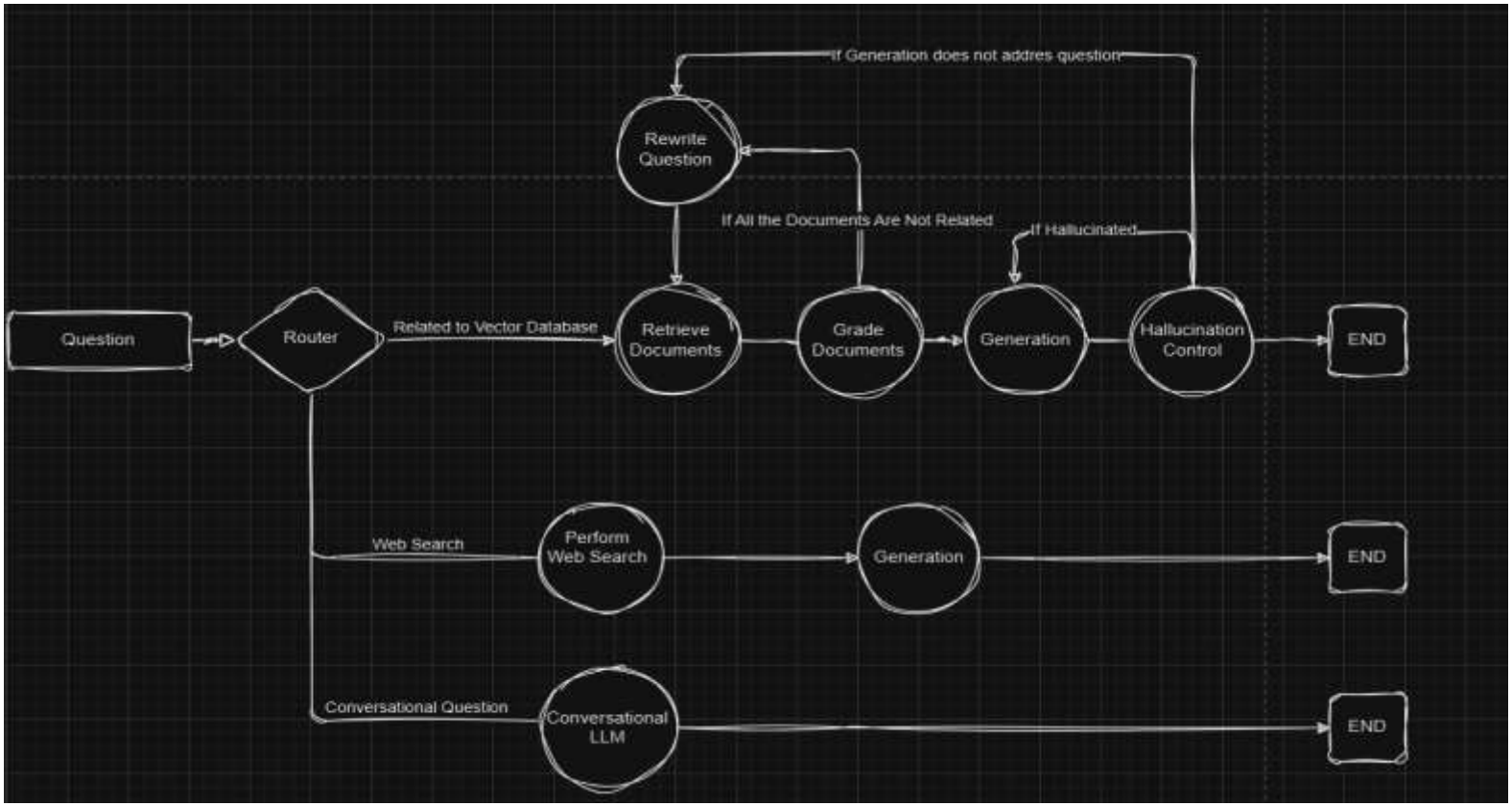
10.12.2024

Contents

- Project Description
- Chunking Problem
- How I Solved Chunking Problem
- Routing Mechanism
- Example
- Monitoring
- Project Timeline
- References

Project Description

- The goal of this project is to integrate **Self-RAG** so that the model can **leverage** an **internal knowledge** base or perform **web searches** to gather **relevant information** and, through **self-evaluation**, generate more **reliable** and **accurate outputs** for **newly created** or **in-house** developed **frameworks**, such as software library documentation or APIs.



Chunking Problem

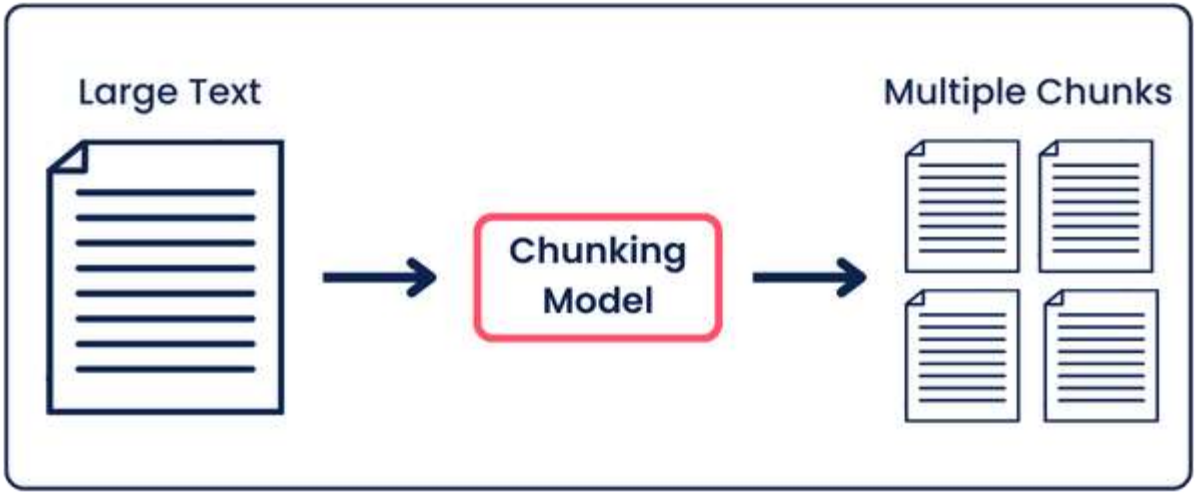
- When code and documentation are combined in a single document, chunking becomes challenging because:
- **Code** has a strict structure that can break if divided improperly.
- **Documentation** needs context, and breaking it into arbitrary chunks can result in incomplete ideas.

Large Chunk Problem

- When a query is made, large chunks may result in irrelevant or over-generalized results. Since the chunk covers a wide range of topics, the retrieved chunk might not directly answer the user's specific question.
- When large chunks are used indexing them in a vector database can be inefficient.

Small Chunk Problem

- **Small chunks** lose context, making it hard to understand code logic or complete ideas in documentation.
- **Example**: Breaking up a function or documentation into tiny pieces can disrupt meaning, leading to incomplete or inaccurate responses.



How I Solved Chunking Problem

For **each page** of code and documentation, I used an **LLM** to resolve the **chunking** problem by generating a **summary** and possible **query examples** for the content.

Processed the Entire Page: Instead of splitting content into arbitrary chunks, I used the LLM to process the **entire page.**

First, the **LLM** analyzed each page of code and documentation to understand the content. It then created a **5-7 sentence summary** and **5-7** possible **query examples** that users might ask, helping the assistant understand the context of **user queries.**

```python
class CreateSummary(BaseModel):
    """ Summary and possible queries for a document. """

    summary: str = Field(
        description="Summary of the document content in 5-7 sentences."
    )
    possible_queries: List[str] = Field(
        description="List of possible queries that users might ask about this topic."
    )
```

```
# Prompt
system = """You are an expert at technical documentation.
Analyze the given documentation and provide:
1. A concise summary (5-7 sentences)
2. List of 5-7 possible queries that users might ask about this topic.

Use the document to generate the summary and possible queries.
Query is a question or a statement that a user might ask about the topic.
Query examples should be relevant to the document content.
"""
```

How I Solved Chunking Problem

- Finally, I combined the **summary** and **query examples** with the **original metadata** (e.g., title, description) and stored as page content in a **vector database.**

- Original **page content** is saved as **metadata** because when we **retrieve** the **chunk** we will get the **real content** from **metadata.**

- This approach ensured that each page was properly summarized, indexed, and made easier to search, without losing important context or detail.

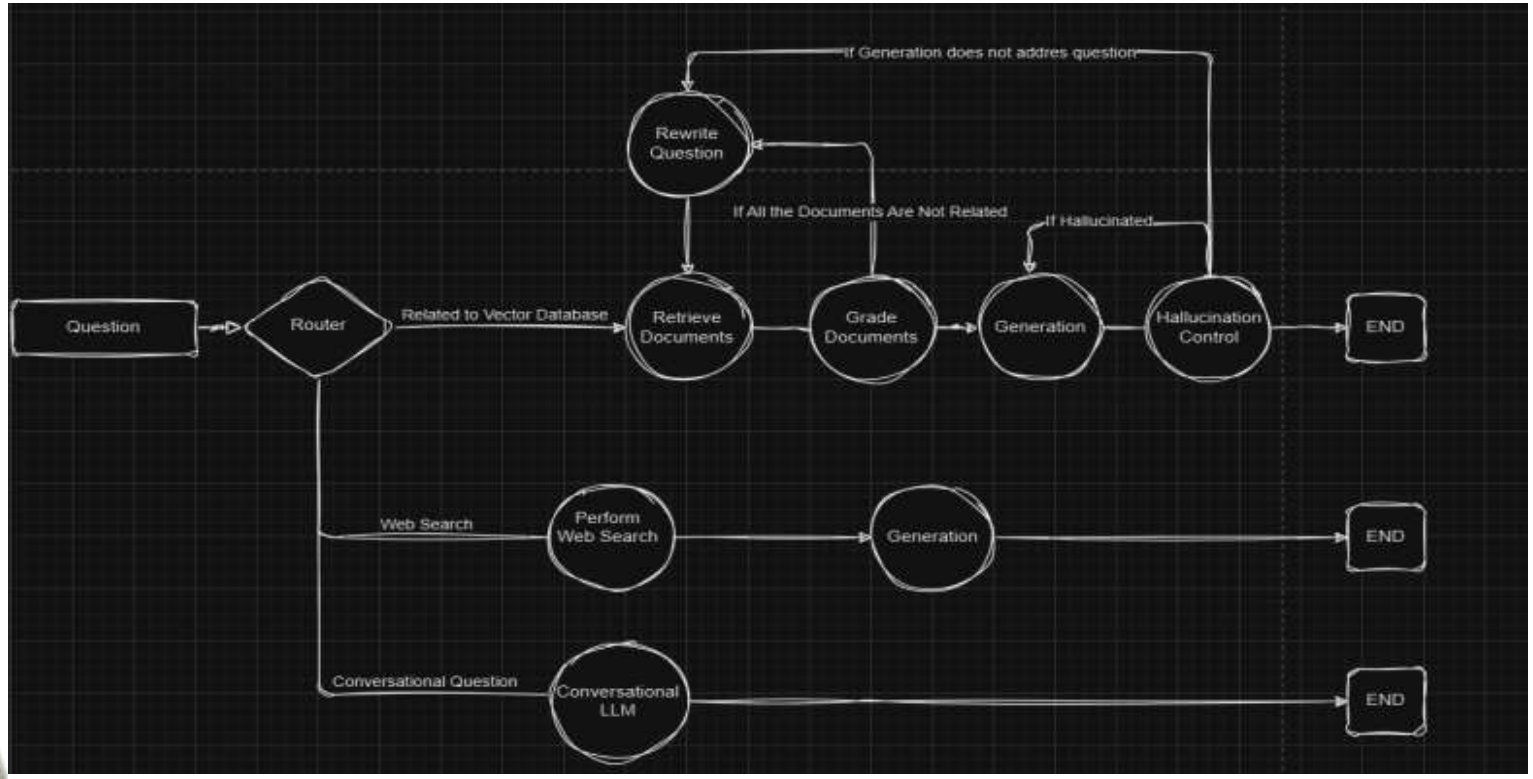
```
# Initialize empty list to store summaries
new_docs = []

# Process each document
for doc in docs:

    # Generate summary for current document
    result = question_router.invoke({"document": doc.page_content})

    queries = ' '.join(result.possible_queries) # Convert list to string

    new_page_content = doc.metadata.get("title") + doc.metadata.get("description") + result.summary + queries

    new_metadata = {
        "source": doc.metadata.get("source"),
        "title": doc.metadata.get("title"),
        "description": doc.metadata.get("description"),
        "summary": result.summary,
        "possible_queries": queries,
        "content": doc.page_content,
    }

    document = Document(page_content=new_page_content, metadata=new_metadata)

    # Append to new list
    new_docs.append(document)
```

Routing Mechanism

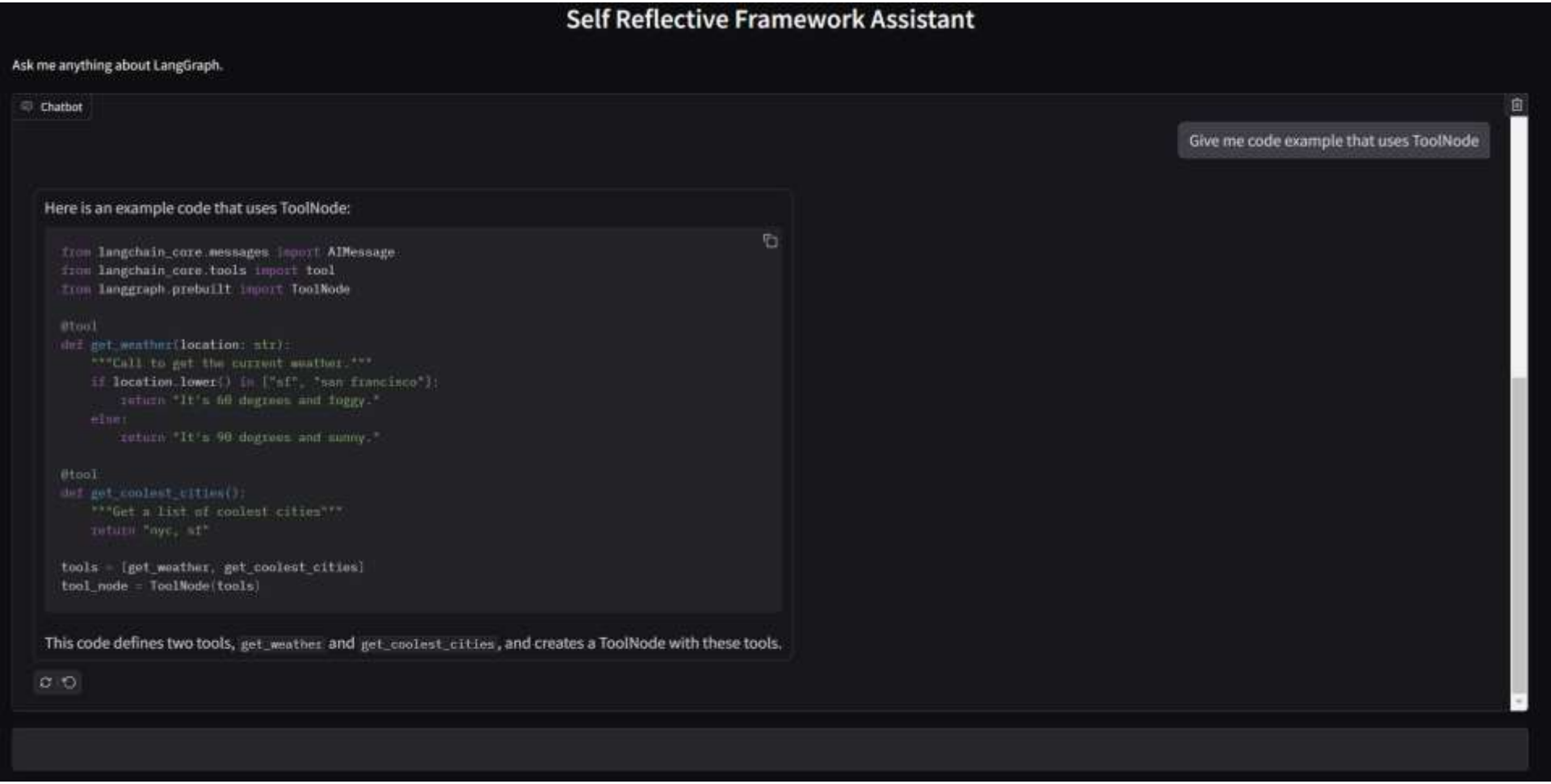
This workflow manages dynamic, decision-driven tasks such as web searches, document retrieval, grading, and content generation. It starts by selecting the route based on the user's needs, whether that's **web_search**, **init_state**, or **conversation**.

After initialization, relevant data is retrieved, and the documents are graded to determine if content should be generated, a query needs to be refined, or if the process should end.

The **generate** step checks the quality of the generated content, either ending the flow or refining the query for further iteration.



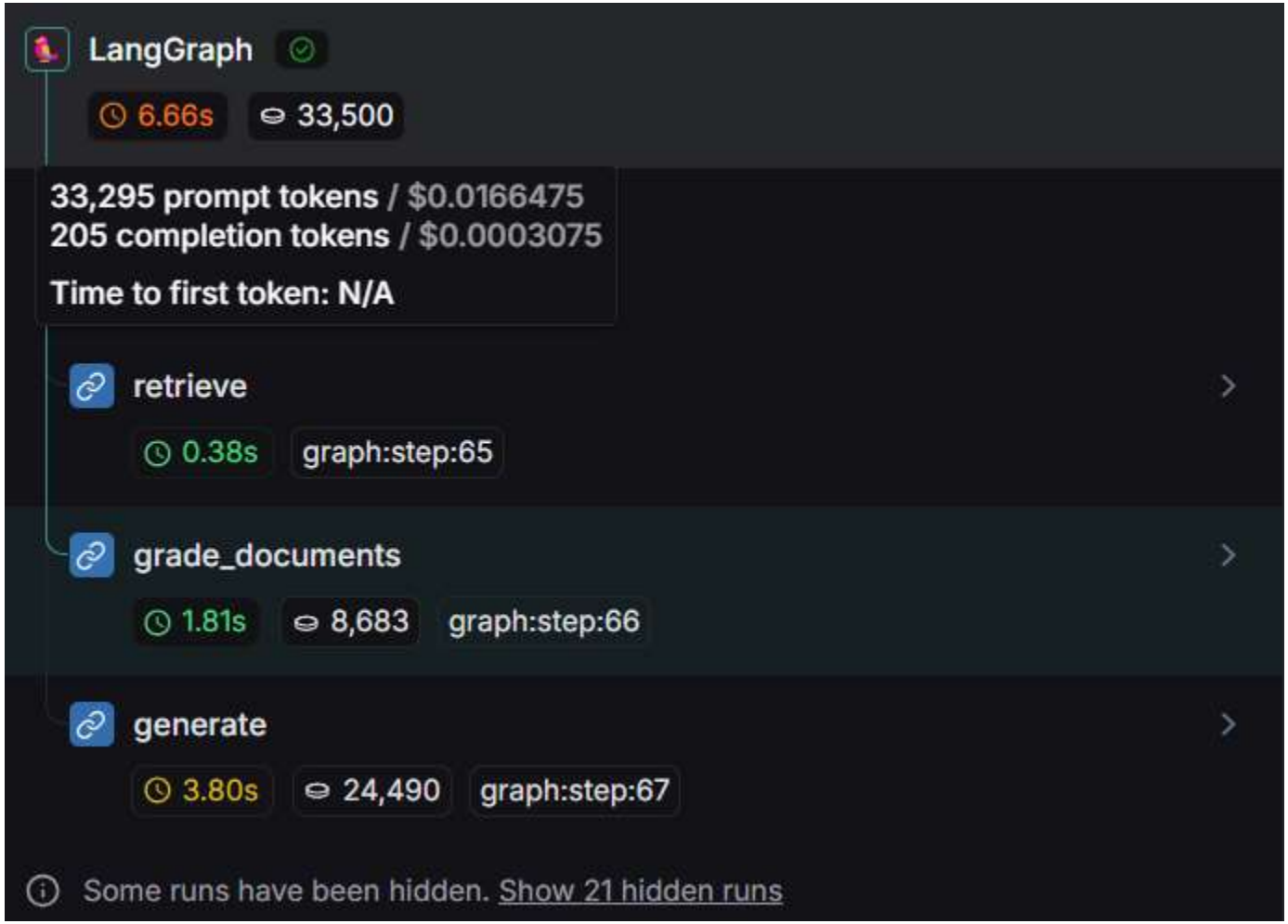
Example


Self Reflective Framework Assistant
Ask me anything about LangGraph.
Chatbot
Give me code example that uses ToolNode
Here is an example code that uses ToolNode:
from langchain_core.messages import AIMessage
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode
@tool
def get_weather(location: str):
"""Call to get the current weather."""
if location.lower() in ["sf", "san francisco"]:
return "It's 60 degrees and foggy."
else:
return "It's 90 degrees and sunny."
@tool
def get_coolest_cities():
"""Get a list of coolest cities"""
return "nyc, sf"
tools = [get_weather, get_coolest_cities]
tool_node = ToolNode(tools)
This code defines two tools, get_weather and get_coolest_cities, and creates a ToolNode with these tools.

Example

```
---ROUTE QUESTION---
---ROUTE QUESTION TO RAG---

---INIT STATE---
Output received: {'init_state': {'question': 'Give me code example that uses ToolNode', 'query_rewritten_num': 0}}
Processing node: init_state
---RETRIEVE---
Output received: {'retrieve': {'question': 'Give me code example that uses ToolNode', 'documents': ['\n\n\n\n\n\n\n
Processing node: retrieve
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: GENERATE---
Output received: {'grade_documents': {'question': 'Give me code example that uses ToolNode', 'documents': ['\n\n\n\
Processing node: grade_documents
---GENERATE---
---CHECK HALLUCINATIONS---
---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---
---GRADE GENERATION vs QUESTION---
---DECISION: GENERATION ADDRESSES QUESTION---
Output received: {'generate': {'question': 'Give me code example that uses ToolNode', 'generation': 'Here is an exa
Processing node: generate
```

Monitoring


LangGraph
6.66s
33,500
33,295 prompt tokens / $0.0166475
205 completion tokens / $0.0003075
Time to first token: N/A
retrieve
0.38s
graph:step:65
grade_documents
1.81s
8,683
graph:step:66
generate
3.80s
24,490
graph:step:67
Some runs have been hidden. Show 21 hidden runs

Project Timeline

Data Collection

25/10/24 - 10/11/24

10/11/24 - 25/11/24

Creating Vector Database

Designing and Implementing The System

25/11/24 - 10/05/25

10/05/25 - end

Performance Evaluation

References

1. https://selfrag.github.io/
2. https://arxiv.org/pdf/2310.11511
3. https://blog.gopenai.com/advanced-rag-with-self-correction-langgraph-no-hallucination-agents-groq-42cb6e5c0086
4. https://blog.gopenai.com/building-an-effective-rag-pipeline-a-guide-to-integrating-self-rag-corrective-rag-and-adaptive-ab7767f8ead1
5. https://github.com/langchain-ai/langgraph/blob/main/examples/rag/langgraph_self_rag.ipynb
6. https://smith.langchain.com/
7. https://www.gradio.app/docs/gradio/chatinterface