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

CSE 495/496 - Final Presentation

Eren Torlak

210104004090

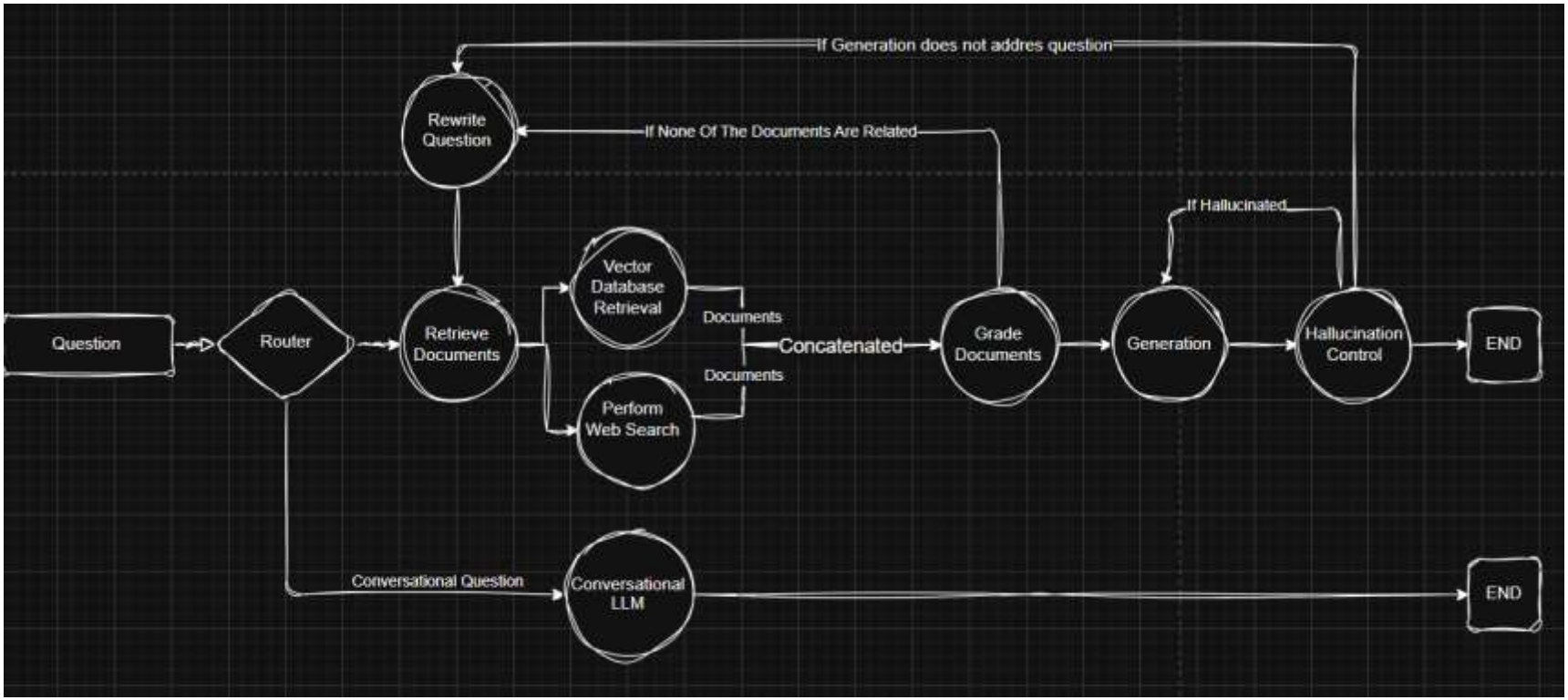
Project Supervisor:
Dr. Salih Sarp

14.01.2025

Contents

Project Description

- The goal of this project is to integrate **Self-RAG** so that the model can **leverage** an **internal knowledge** base and 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

The dynamic workflow in the system is designed to handle two main query cases: retrieval workflows and conversational language model workflows.

If the query requires retrieval, the system engages both the vector database and the web search tool to gather relevant information

The system retrieves two documents from the database that most closely match the query's semantic content. Simultaneously, the web search tool is activated to perform a real-time search, retrieving two additional documents from external sources.

```
---ROUTE QUESTION---
---ROUTE QUESTION TO RETRIEVER---

---INIT STATE---
---VECTOR DATABASE RETRIEVE---
---WEB SEARCH---
---CHECK DOCUMENT RELEVANCE TO QUESTION---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT NOT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---GRADE: DOCUMENT RELEVANT---
---ASSESS GRADED DOCUMENTS---
---DECISION: GENERATE---
---GENERATE---
---CHECK HALLUCINATIONS---
---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---
---GRADE GENERATION vs QUESTION---
---DECISION: GENERATION ADDRESSES QUESTION---
```

Routing Mechanism

After retrieval, the system evaluates the relevance of each document using a grading mechanism. This mechanism employs a binary scoring model, which assesses whether a document is relevant (yes) or irrelevant (no) to the query.

Irrelevant documents are filtered out, preventing them from contributing to the response generation process. This initial grading step is critical to maintaining the quality of the system's outputs, as it ensures that only meaningful and contextually accurate data is used.

```
# Grader prompt
system = """You are a grader assessing relevance of a retrieved document to a user question. \n
    It does not need to be a stringent test. The goal is to filter out erroneous retrievals. \n
    If the document contains keyword(s) or semantic meaning related to the user question, grade it as relevant. \n
    Give a binary score 'yes' or 'no' score to indicate whether the document is relevant to the question."""
grade_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system),
        ("human", "Retrieved document: \n\n {document} \n\n User question: {question}"),
    ]
)

# LLM with function call
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
structured_llm_grader = llm.with_structured_output(GradeDocuments)
retrieval_grader = grade_prompt | structured_llm_grader
```

Routing Mechanism

The Question Rewriter Node in my system takes an input query and reformulates it into a more optimized version for retrieval.

It specifically improves the query's semantic alignment with the vector database and web search.

```
# Question Re-writer
system = """You a question re-writer that converts an input question to a better version that is optimized \n
     for vectorstore retrieval. Look at the input and try to reason about the underlying semantic intent / meaning."""
re_write_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system),
        (
            "human",
            "Here is the initial question: \n\n {question} \n Formulate an improved question.",
        ),
    ]
)

question_rewriter = re_write_prompt | llm | StrOutputParser()
```

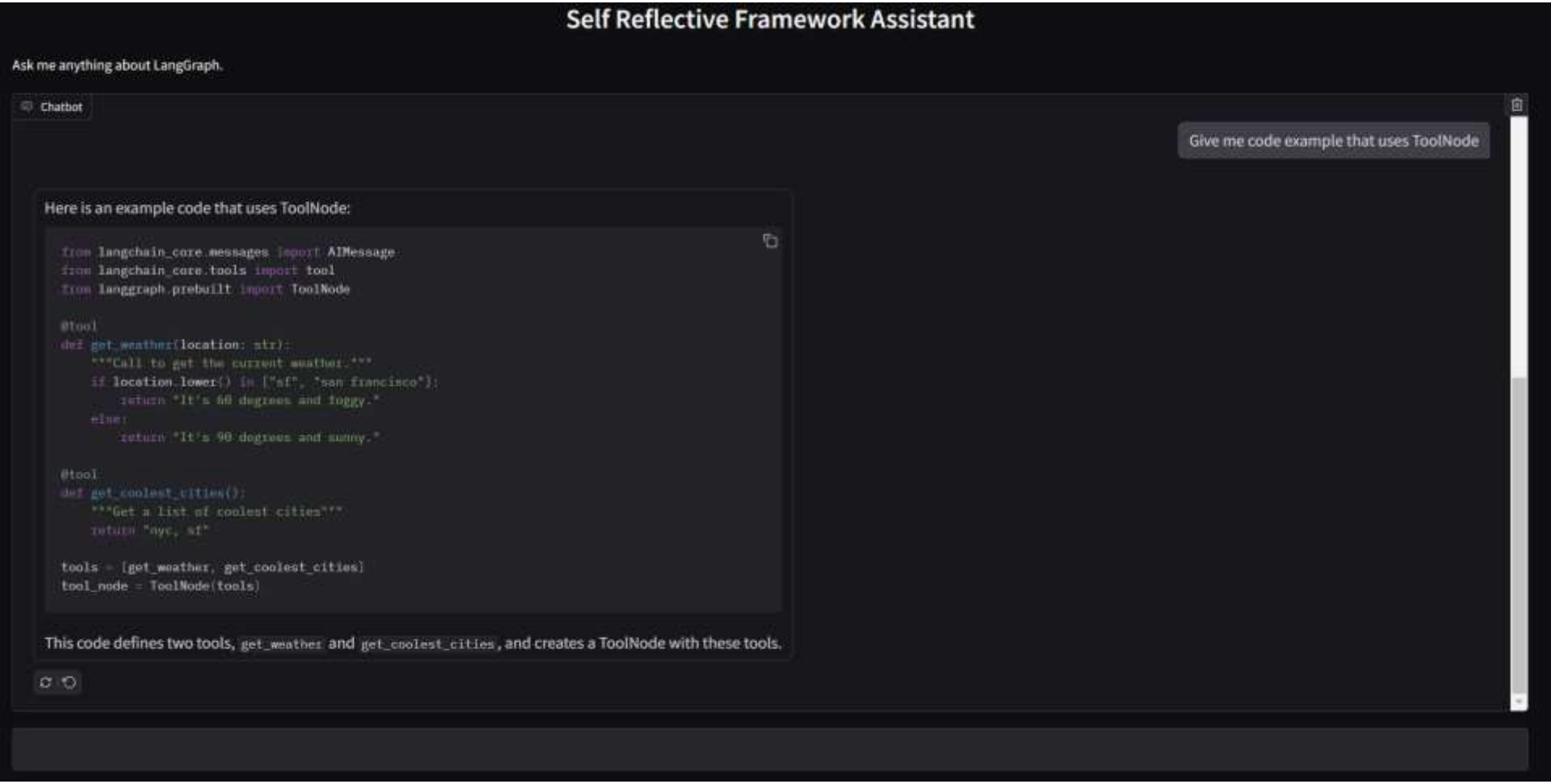
Routing Mechanism

If the query is casual, conversational, or a follow-up question, it bypasses the retrieval mechanism entirely and is routed directly to the conversational language model. Examples of such queries include informal statements like “Can you explain that?” or “What do you mean by this?”

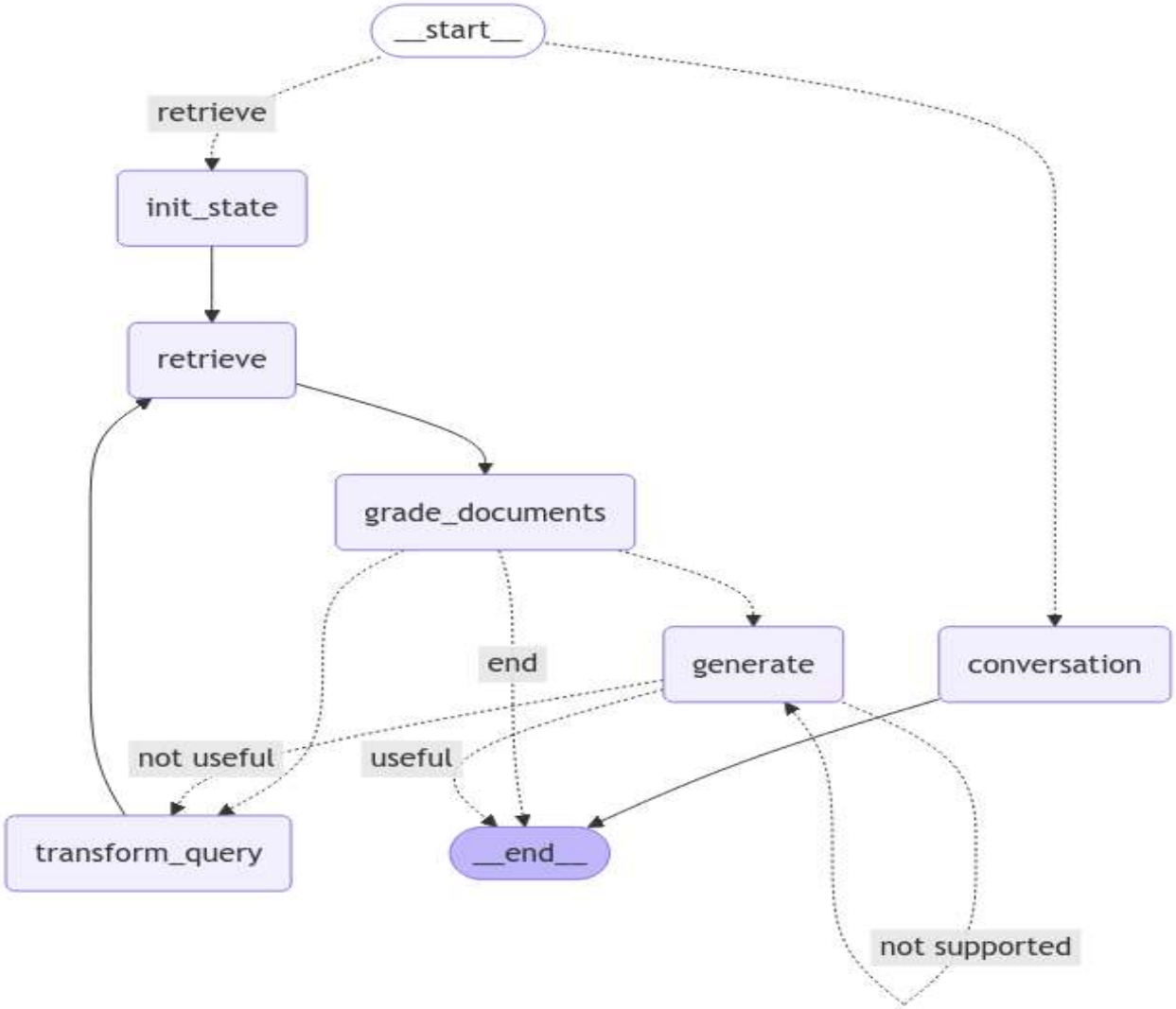
In these cases, the conversational model generates responses using the context of the ongoing interaction, without involving the vector database or web search. This ensures smooth and efficient handling of informal queries while keeping the system’s resources optimized

```
---ROUTE QUESTION---
--ROUTE QUESTION TO CONVERSATION
---CONVERSATION---
```

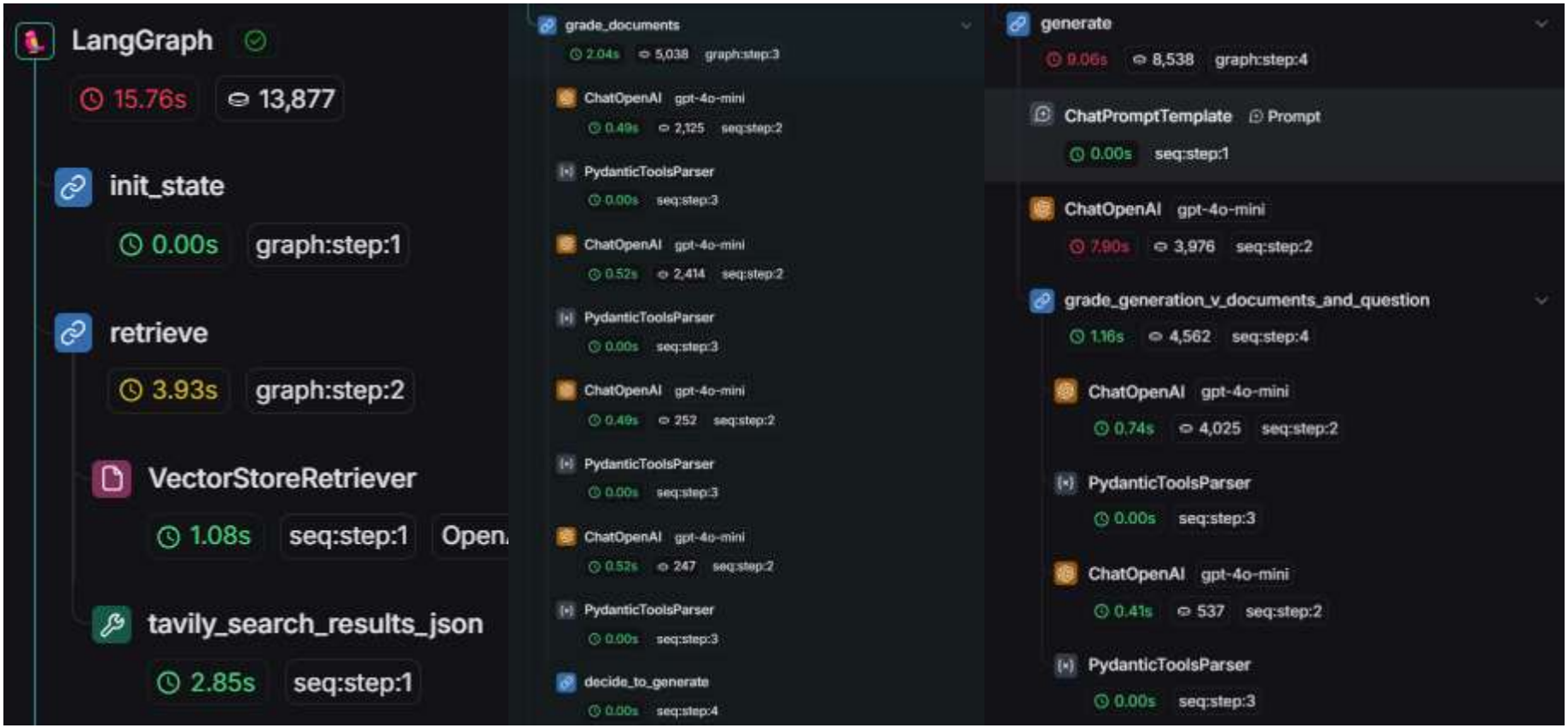
Example


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.

Workflow


__start__
retrieve
init_state
retrieve
grade_documents
generate
conversation
end
not useful
useful
transform_query
__end__
not supported

Monitoring


LangGraph
15.76s
13,877
init_state
0.00s
graph:step:1
retrieve
3.93s
graph:step:2
VectorStoreRetriever
1.08s
seq:step:1
tavily_search_results_json
2.85s
seq:step:1
grade_documents
2.04s
5,038
graph:step:3
ChatOpenAI
gpt-4o-mini
0.49s
2,125
seq:step:2
PydanticToolsParser
0.00s
seq:step:3
ChatOpenAI
gpt-4o-mini
0.52s
2,414
seq:step:2
PydanticToolsParser
0.00s
seq:step:3
ChatOpenAI
gpt-4o-mini
0.49s
252
seq:step:2
PydanticToolsParser
0.00s
seq:step:3
ChatOpenAI
gpt-4o-mini
0.52s
247
seq:step:2
PydanticToolsParser
0.00s
seq:step:3
decide_to_generate
0.00s
seq:step:4
generate
9.06s
8,538
graph:step:4
ChatPromptTemplate
Prompt
0.00s
seq:step:1
ChatOpenAI
gpt-4o-mini
7.90s
3,976
seq:step:2
grade_generation_v_documents_and_question
1.16s
4,562
seq:step:4
ChatOpenAI
gpt-4o-mini
0.74s
4,025
seq:step:2
PydanticToolsParser
0.00s
seq:step:3
ChatOpenAI
gpt-4o-mini
0.41s
537
seq:step:2
PydanticToolsParser
0.00s
seq:step:3

Evaluation

```python
class ScoreModel(BaseModel):
    """Relevance and performance score for a response"""

    score: int = Field(
        ...,
        description="Score between 0-5",
        ge=0,    # greater or equal
        le=5     # less or equal
    )
    justification: str = Field(
        ...,
        description="Justification for the score"
    )
```

Evaluation

```python
def score_relevance(query, response):
    scoring_prompt = f"""
    You are a grader assessing the relevance of a response to a query.
    In no way response should contain mock up or hallucination.

    Response should not be a hypothetical answer.

    If question can be answered with the code and response contains code, it is relevant but if it is not, it is not relevant.

    0: Hypothetical answer or irrelevant
    1: Slightly relevant
    2: Moderately relevant
    3: Relevant
    4: Highly relevant
    5: Contains real code example and directly answers the question

    Dont be generous, be strict.
    You should tend to give lower scores with no code responses.

    If the response doesnt need any code to answer the question make sure answer is confidently answered score accordingly.

    If the response is saying that like "I am not sure, not explicitly mentioned, I think, I believe" etc. give lower scores.
    Query: {query}
    Response: {response}
    Score the relevance on a scale of 0-5.
    """
    scoring_llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
    scoring_llm = scoring_llm.with_structured_output(ScoreModel)
    score = scoring_llm.invoke(scoring_prompt)
    return score
```

LLM and SELF-RAG Comparision

- The evaluation of Self-RAG against standard LLMs and traditional RAG systems revealed its superior performance in relevance and accuracy. This evaluation done by using a LLM with provided context.

Query: Give me code example that uses ToolNode in langgraph
- Method: Standard LLM
- Relevance Score: 0
- Latency: 1.34s
- Justification: The response does not provide any code example and expresses uncertainty about the implementation of ToolNode in langgraph, making it irrelevant to the query.

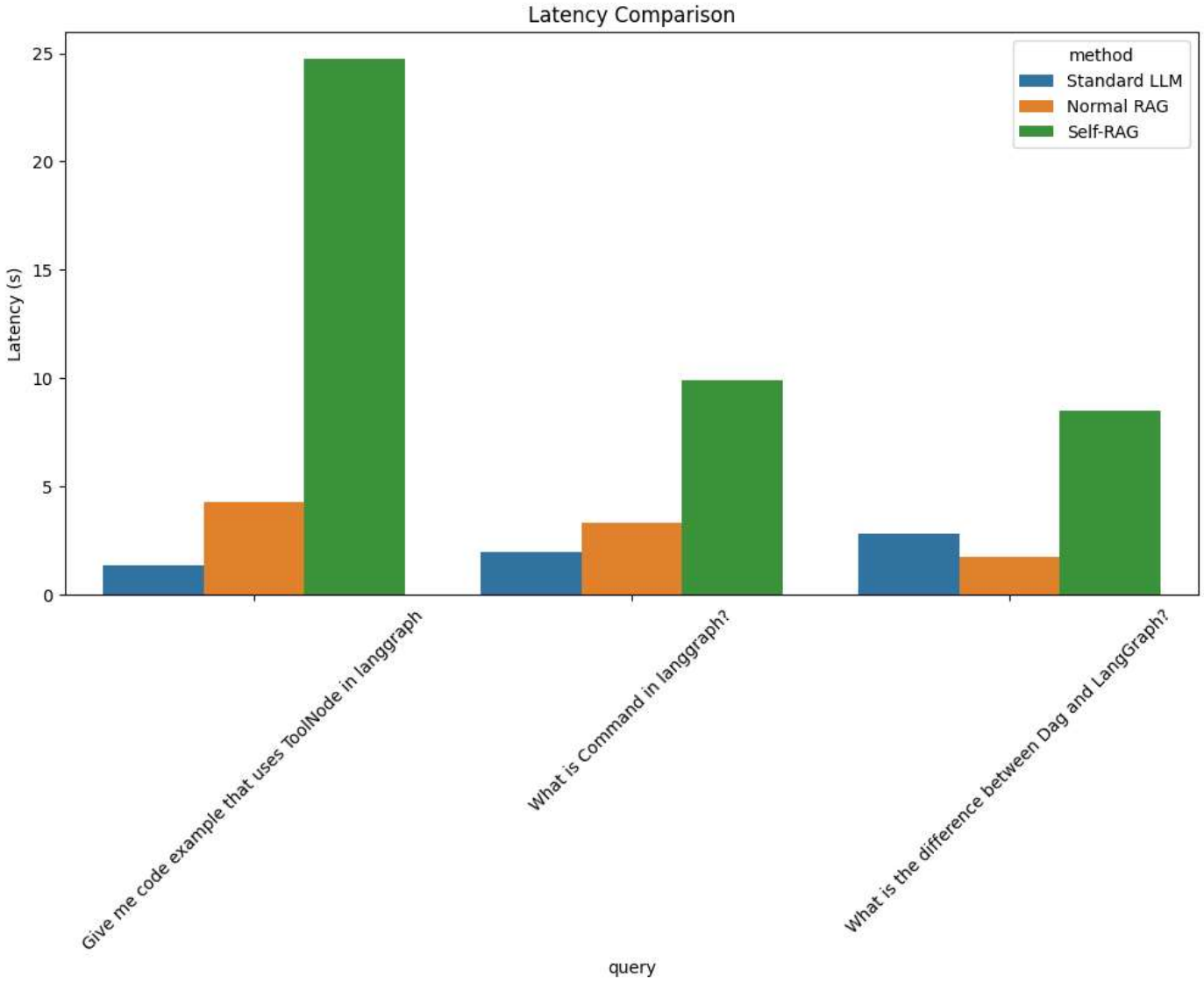
Query: Give me code example that uses ToolNode in langgraph
- Method: Self-RAG
- Relevance Score: 5
- Latency: 24.74s
- Justification: The response provides a clear and relevant code example that demonstrates the use of `ToolNode` in LangGraph. It includes the definition of tools and how to invoke them using `ToolNode`, directly addressing the query.

RAG and SELF-RAG Comparision

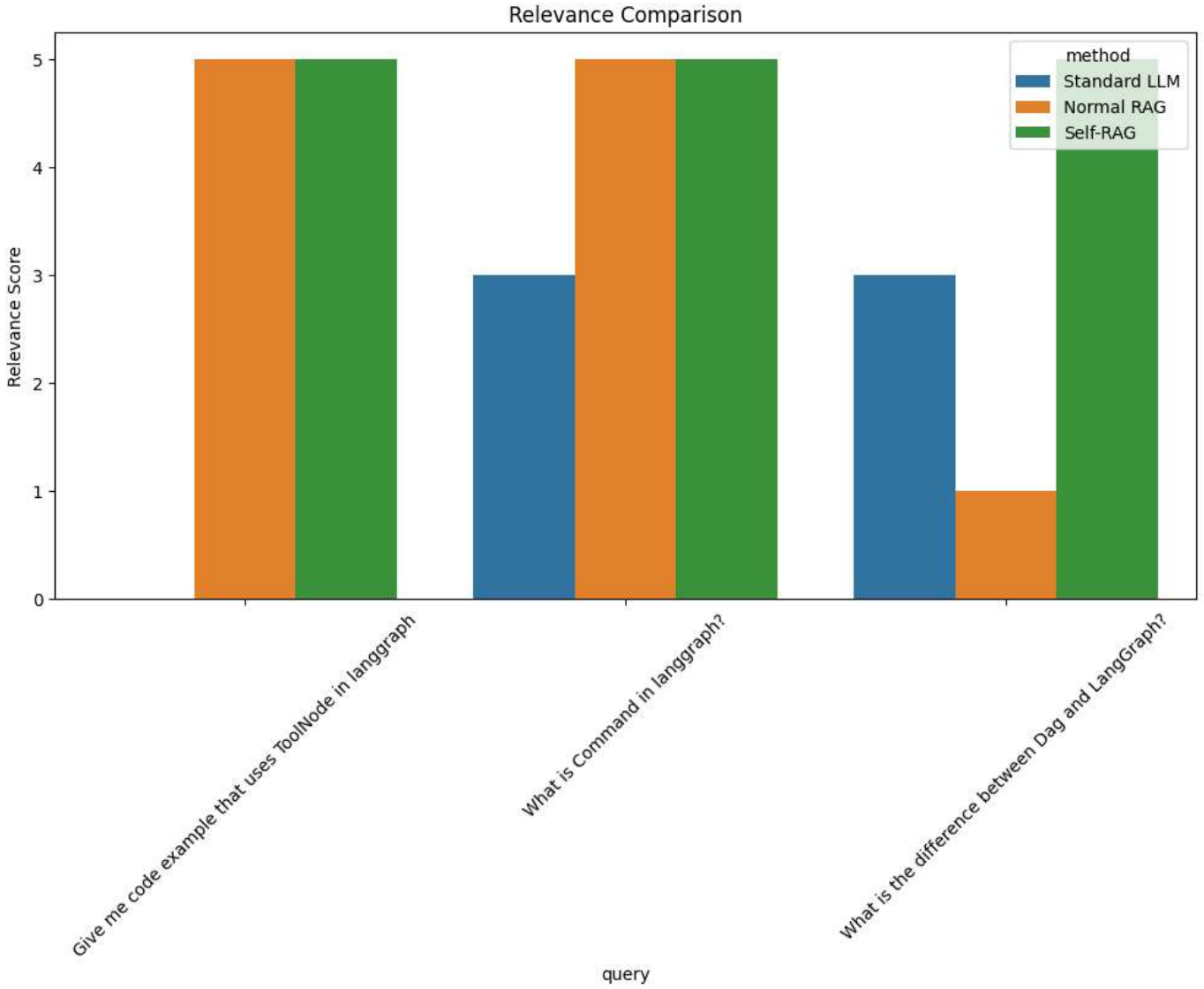
- Query: What is the difference between Dag and LangGraph?
- - Method: Normal RAG
- - Relevance Score: 1
- - Latency: 1.78s
- - Justification: The response attempts to address the query by mentioning LangGraph and its features, but it fails to provide a direct comparison with Dag. The lack of explicit information about Dag and the use of uncertain language ('I cannot provide a direct comparison') detracts from its relevance.

- Query: What is the difference between Dag and LangGraph?
- - Method: Self-RAG
- - Relevance Score: 5
- - Latency: 8.50s
- - Justification: The response directly answers the question by clearly explaining the difference between LangGraph and a directed acyclic graph (DAG), specifically mentioning that LangGraph allows for cycles while a DAG does not. It provides a concise and accurate definition of both concepts without any hypothetical language.

Latency Evaluation



Output Evaluation



Future Work

Future enhancements to the Self-RAG framework will focus on reducing latency through parallel query processing and optimizing grading mechanisms. The prepro cessing phase can be improved with advanced summarization techniques and more effective query generation, ensuring higher retrieval precision.

Also automatic code execution and testing can be added. This can make evaluation easier.

Improving the user interface is another priority. Features like displaying references with links or page numbers for retrieved documents will increase transparency and usability. These improvements aim to make the system faster, more functional, and user-friendly while broadening its applicability

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