Contents

Self-RAG Implementation Report 2
- 1. Overview 2
- 2. Objectives 2
- 3. High-Level Architecture 3
  - Core Components 3
- 4. Graph State Design 3
  - State Fields 3
- 5. Node-Level Description 3
  - 5.1 Retrieve Node 3
  - 5.2 Document Relevance Grading Node 4
  - 5.3 Generate Node 4
  - 5.4 Query Transformation Node 4
- 6. Control Flow Logic 4
- 7. Alignment with Self-RAG Framework 4
  - Architectural Differences 5
- 9. Limitations 5
- 10. Conclusion 5

Self-RAG Implementation Report

1. Overview

This report documents the design, logic, and architecture of the Self-RAG (Self-Reflective Retrieval-Augmented Generation) system implemented using LangGraph. The system is designed to improve answer quality by iteratively retrieving documents, generating answers, and self-evaluating relevance, groundedness, and usefulness before producing a final response.

The implementation follows the core ideas proposed in the Self-RAG framework while adapting them for a production-oriented environment using HuggingFace models and a local document corpus.

2. Objectives

The primary goals of this implementation are:

- Reduce hallucinations in RAG-based systems
- Ensure retrieved documents are relevant to the query
- Verify that generated answers are grounded in retrieved content
- Ensure answers directly address the user's question
- Automatically recover from failure cases using query rewriting and regeneration

3. High-Level Architecture

The system is implemented as a stateful directed graph using LangGraph. Each node performs a specific operation, and conditional edges determine the control flow based on intermediate evaluation results.

Core Components

- **Retriever**: Fetches top-k relevant document chunks from a Chroma vector store
- **LLM Generator**: Produces answers using retrieved context
- **Self-Grading Modules**:
    - Document relevance grader
    - Groundedness (hallucination) checker
    - Answer usefulness checker
- **Query Rewriter**: Reformulates queries when retrieval or generation fails

4. Graph State Design

The system maintains a shared mutable state across nodes:

```
class GraphState(TypedDict):
    question: str
    documents: list
    generation: str
    steps: List[str]
```

State Fields

- **question**: Current user question (may be rewritten)
- **documents**: Retrieved and filtered document chunks
- **generation**: Latest LLM-generated answer
- **steps**: Execution trace for debugging and observability

5. Node-Level Description

5.1 Retrieve Node

- Retrieves candidate documents using vector similarity search
- Adds retrieval logs to the execution trace

Purpose: Provide candidate evidence for answer generation

5.2 Document Relevance Grading Node

- Uses an LLM-based binary classifier
- Filters out documents not relevant to the question

Decision Outcome: - If no relevant documents remain → trigger query rewriting - Else → proceed to answer generation

5.3 Generate Node

- Generates an answer using retrieved documents
- Immediately evaluates groundedness and usefulness

Possible Outcomes: - Hallucinated answer → regenerate - Grounded but not useful → rewrite query - Grounded and useful → terminate successfully

5.4 Query Transformation Node

- Rewrites the original question to improve retrieval quality
- Clears document state before re-retrieval

Purpose: Recovery from poor retrieval or misaligned answers

6. Control Flow Logic

The graph follows this execution loop:

1. Start → Retrieve
2. Retrieve → Grade Documents
3. If no relevant docs → Rewrite Query → Retrieve
4. If relevant docs → Generate
5. If hallucinated → Regenerate
6. If not useful → Rewrite Query → Retrieve
7. If grounded & useful → End

This loop ensures robustness and self-correction.

7. Alignment with Self-RAG Framework

The implementation is follows the Self-RAG design:

- Retrieval decision-making
- Document relevance validation
- Groundedness verification
- Answer usefulness scoring
- Iterative refinement loop

Architectural Differences

- Uses HuggingFace (Mistral-7B-Instruct) instead of OpenAI models
- Consolidates some grading logic within the generation node
- Operates on a local enterprise document corpus

9. Limitations

- LLM-based graders may occasionally misclassify
- Increased latency due to multiple LLM calls (40s+ for a unit test)
- Requires careful prompt tuning for graders

10. Conclusion

This Self-RAG LangGraph implementation demonstrates a robust, self-correcting RAG pipeline suitable for enterprise knowledge systems. While architecturally adapted for practical constraints, it faithfully implements the core Self-RAG reasoning paradigm and significantly improves answer reliability and relevance.