LLM Wrapper for Legal Documents: GraphRAG System Documentation

Technical Documentation

January 13, 2026

Contents

1 System Overview

This document describes a **Graph Retrieval-Augmented Generation (GraphRAG)** system designed to process PDF documents, extract knowledge graphs, and enable intelligent question-answering. The system combines:

- **Neo4j** – Graph database for storing entities and relationships
- **Vector Stores** – For semantic similarity search
- **Multiple LLMs** – OpenAI, Claude, and Gemini for different tasks
- **LangChain** – Orchestration framework for LLM pipelines

2 Core Module: PDFGraphRAG

The `PDFGraphRAG` class (in `pdf_graphrag.py`) is the main entry point for processing documents and querying the knowledge graph.

2.1 Constructor and Initialization

Listing 1: Constructor signature

```
def __init__(self, vector_store_chunk_name, vector_store_nodes_name,
             vector_store_relationships_name, neo4j_uri, neo4j_user,
             neo4j_password, openai_api_key, google_api_key,
             claude_api_key, advanced_search=False)
```

The constructor initializes:

1. **Neo4j Graph Connection** – Connects to the graph database
2. **OpenAI Embeddings** – Uses `text-embedding-3-large` model
3. **Vector Stores** – Three separate stores for chunks, nodes, and relationships
4. **LLM Clients**:
   - `openai_client` – GPT for question processing
   - `claude_client` – Claude Sonnet for graph transformation
   - `gemini_client` – Gemini Flash for general tasks
5. **Graph Transformer** – `LLMGraphTransformer` using Claude for entity extraction

2.2 PDF Processing Pipeline

2.2.1 Loading PDFs

Listing 2: PDF loading function

```
def load_pdf(self, pdf_path: str):
    loader = PyPDFLoader(pdf_path)
    return loader.load()
```

Uses LangChain's `PyPDFLoader` to extract text content from PDF files.

2.2.2 Processing Pipeline

The `process_pdf()` method executes the following pipeline:

1. **Load PDF** – Extract raw text from document
2. **Chunk Text** – Split using `SpacyTextSplitter` for sentence-aware chunking
3. **For each chunk**:
   - Generate embedding vector
   - Create a `Chunk` node with text, embedding, and page metadata
   - Transform chunk into graph documents using `LLMGraphTransformer`
   - Create `HAS` relationships between chunks and extracted entities
4. **Store in Neo4j** – Add all graph documents to the database
5. **Update Vector Stores** – Index nodes and relationships for similarity search

Listing 3: Chunk node creation

```
chunk_node = Node(
    id=chunk_id,
    type="Chunk",
    properties={
        "text": document.page_content,
        "embedding": chunk_embedding,
        "page": document.metadata.get("page", 0)
    }
)
```

2.3 Querying Functions

2.3.1 Graph Database Query

`query_graph_database(question)` converts natural language to Cypher queries using an LLM agent:

1. Retrieves graph schema (node labels, relationship types, sample data)
2. Creates an agent with a `search_database` tool
3. Agent iteratively explores the database with Cypher queries
4. Returns structured response with query, explanation, and data
5. Formats results into natural language answer

The agent uses a structured output schema:

Listing 4: Response schema structure

```
response_schema = {
    "cypher_query": "Final Cypher query",
    "explanation": "Query strategy explanation",
    "data": "JSON string of results",
    "nodes_found": ["list", "of", "node", "ids"],
    "relationships_found": ["nodeA -[REL]-> nodeB"]
}
```

2.3.2 Vector Database Query

Listing 5: Vector similarity search

```
def query_vector_database(self, database: Neo4jVector,
                          question: str, k: int = 5):
    return database.similarity_search(query=question, k=k)
```

Performs semantic similarity search on vector stores to find relevant nodes or relationships.

2.3.3 Chunk Similarity Query

Listing 6: Cosine similarity search on chunks

```
def query_chunks_by_similarity(self, question: str, k: int = 5):
    question_embedding = self.embeddings.embed_query(question)
    result = self.graph.query("""
␣␣␣␣␣␣␣␣MATCH␣(c:Chunk)
␣␣␣␣␣␣␣␣WITH␣c,␣gds.similarity.cosine(c.embedding,␣$embedding)␣AS␣score
␣␣␣␣␣␣␣␣ORDER␣BY␣score␣DESC
␣␣␣␣␣␣␣␣LIMIT␣$k
␣␣␣␣␣␣␣␣RETURN␣c.text␣AS␣text,␣c.page␣AS␣page,␣score
␣␣␣␣""", {"embedding": question_embedding, "k": k})
    return result
```

Uses Neo4j's Graph Data Science library for cosine similarity computation directly in the database.

2.4 Answer Generation

2.4.1 Validation and Final Answer

`validate_and_answer()` combines results from multiple sources:

- Node vector search results
- Relationship vector search results
- Chunk vector search results
- Graph query results
- Optional advanced search results

All results are formatted into a prompt, and the LLM generates a comprehensive natural language answer.

2.4.2 Interactive Question Interface

`invoke_question()` provides an interactive CLI:

- `-h` – Display help instructions
- `-s` – Show graph schema
- `exit` – Exit the interface
- Any other input – Process as a question

The function orchestrates:

1. Vector searches on nodes, relationships, and chunks
2. Graph query via `GraphCypherQAChain`
3. Optional advanced search (if enabled)
4. Final answer validation and generation

3 Testing Module: RomeoJulietGraphTester

The `test_romeo_juliet_graph.py` module validates knowledge graph accuracy by comparing graph query results against authoritative sources.

3.1 Test Pipeline Overview

The test suite executes a 4-step process for each test iteration:

1. **Generate Question** – LLM creates varied test questions
2. **Query Graph** – Execute question against Neo4j
3. **Get Ground Truth** – Search web for authoritative answer
4. **Compare & Score** – Evaluate accuracy (0-100 scale)

3.2 Question Generation

Listing 7: Question generation with schema constraints

```
def generate_test_question(self, iteration: int) -> Dict[str, Any]:
    # Returns: question, question_type, expected_nodes,
    #          expected_relationships
```

The function:

- Rotates through question categories (relationships, attributes, events, locations, multi-hop)
- Uses graph schema to constrain expected node/relationship types
- Tracks previous questions to avoid duplicates
- Returns structured output with validation constraints

Question types:

- `relationship` – Character relationships
- `character_attribute` – Traits, roles, affiliations
- `event` – Plot events and connections
- `location` – Settings and places
- `multi_hop` – Complex traversal queries

3.3 Graph Querying with Agent

The `query_graph_database()` method uses an LLM agent with tool access:

Listing 8: Database search tool

```
@tool
def search_database(cypher_query: str) -> str:
    """Execute a Cypher query against Neo4j.
    Returns JSON string of results or error message."""
    # Handles Neo4j object serialization
    # Returns JSON-formatted results
```

The agent follows this strategy:

1. Analyze question to identify relevant nodes/relationships
2. Start with exploration queries
3. Refine iteratively based on results
4. Return best Cypher query with found data

3.4 Comparison and Scoring

`compare_and_score()` evaluates graph accuracy:

Listing 9: Scoring rubric

```
Scoring Rubric:
- 100:   Perfect match, all information correct
- 80-99: Mostly accurate, minor details missing
- 60-79: Partially accurate, some key info missing
- 40-59: Significantly inaccurate or incomplete
- 0-39:  Mostly incorrect or no useful data
```

Output includes:

- `score` – Numeric accuracy score (0-100)
- `accuracy_assessment` – Detailed explanation
- `discrepancies` – Specific errors found
- `missing_data` – Information gaps
- `correct_elements` – What the graph got right
- `recommendations` – Improvement suggestions

3.5 Report Generation

`generate_final_report()` produces:

- JSON file with all test results
- Statistics: average, min, max scores
- Aggregated recommendations and issues
- Letter grade (A-F) based on average score

4 Utility Functions

4.1 JSON Serialization

`serialize_for_json()` handles Neo4j object conversion:

Listing 10: Object type handling

```
Supported types:
- Neo4j Node     -> {_type, labels, properties}
- Neo4j Relationship -> {_type, type, properties}
- Neo4j Path     -> {_type, nodes[], relationships[]}
- datetime       -> ISO format string
- dict/list      -> Recursive serialization
- primitives     -> Pass through
- other          -> String representation
```

4.2 SpaCy Graph Extraction

`spacy_to_graph_document()` extracts knowledge from text:

1. **Entity Extraction** – Named entities become nodes
2. **SVO Triple Extraction** – Subject-Verb-Object patterns
   - Find ROOT verbs in dependency parse
   - Extract subjects (`nsubj`, `nsubjpass`)
   - Extract objects (`dobj`, `pobj`, `attr`)
   - Create relationships with verb lemma as type

5 Architecture Diagram

```
+-------------------+     +--------------------+     +-------------------+
|   PDF Document    | --> | SpacyTextSplitter | --> |      Chunks       |
+-------------------+     +--------------------+     +-------------------+
                                                           |
                                                           v
                                              +-------------------------+
                                              | LLMGraphTransformer     |
                                              | (Claude)                |
                                              +-------------------------+
                                                           |
                          +---------------------------------+
                          |                                |
                          v                                v
              +--------------------+          +---------------------+
              |   Neo4j Graph      |          |   Vector Stores     |
              |  (Nodes + Rels)    |          |  (Embeddings)       |
              +--------------------+          +---------------------+
                          |                                |
                          +----------------+---------------+
                                           |
                                           v
```

```
+------------------------+
|   Query Processing     |
| - Cypher Generation    |
| - Similarity Search    |
| - Answer Synthesis     |
+------------------------+
            |
            v
+------------------------+
|   Natural Language     |
|       Answer           |
+------------------------+
```

6 Key Dependencies

- `langchain-neo4j` – Neo4j integration with LangChain
- `langchain-experimental` – Graph transformers
- `langchain-openai` – OpenAI models and embeddings
- `langchain-anthropic` – Claude models
- `langchain-google-genai` – Gemini models
- `spacy` – NLP and text splitting
- `neo4j` – Python driver for Neo4j
- `python-dotenv` – Environment variable management