Technical Integration Guide: CNTRagSystem

Carbon Nanotube Research Assistant Team

1 Introduction

Purpose

This document provides a technical overview of the Carbon Nanotube (CNT) Research Assistant's core component, the `CNTRagSystem`. It outlines the necessary steps to decouple the system from its Streamlit-based interface and integrate it into other applications, such as a backend API, an alternative web framework, or a command-line tool.

System Overview

The CNT Research Assistant is a sophisticated Retrieval-Augmented Generation (RAG) system for answering questions about Carbon Nanotubes. It integrates multiple components:

- **LLM Interface:** Interfaces with Large Language Models (e.g., OpenAI GPT, Google Gemini).
- **Vector Store:** A similarity search database (e.g., FAISS) storing document embeddings.
- **Graph Database:** A Neo4j database modeling CNT knowledge as interconnected entities.
- **CNTRagSystem:** Central orchestrator managing the question-to-answer lifecycle.

2 Core Component: CNTRagSystem

The `CNTRagSystem` class handles the full RAG workflow. It is portable and can be initialized independently of the frontend.

Responsibilities

- Contextualizing new questions using chat history.
- Querying the Vector Store for relevant document chunks.
- Querying the Graph Database for structured knowledge.
- Synthesizing information and generating a final answer via LLM.
- Streaming real-time events throughout the process.

3 Setup and Initialization

To initialize the `CNTRagSystem`, you must construct its core dependencies. The `load_rag_system()` function in `app.py` serves as a reference.

Dependencies

- `llm_interface`: An instance of `LLMInterface`.
- `vector_store`: Loaded instance of the VectorStore.
- `graph_db`: Instance of `Neo4jGraphDB`.
- `logger`: Python `logging` instance.
- `feedback_db_path` (optional): Path to feedback database.
- `feedback_history` (optional): Pre-loaded feedback.

Example Initialization

```python
import os
from utils import setup_logging
from llm_interface import LLMInterface
from vector_store import get_vector_store
from graph_db import Neo4jGraphDB
from rag_core import CNTRagSystem
import config

logger = setup_logging(config.DEFAULT_LOG_LEVEL, config.DEFAULT_LOG_FILE_PATH)

try:
    llm = LLMInterface(
        llm_provider=config.DEFAULT_GENERATIVE_LLM_PROVIDER,
        google_api_key=config.GOOGLE_API_KEY,
        openai_api_key=config.OPENAI_API_KEY,
        logger=logger
    )

    vector_store = get_vector_store(
        vector_db_type=config.DEFAULT_VECTOR_DB_TYPE,
        vector_db_path=config.DEFAULT_VECTOR_DB_PATH,
        logger=logger
    )

    vector_store.load_or_build(
        documents_path_pattern=config.DEFAULT_DOCUMENTS_PATH_PATTERN,
        chunk_settings={
            ’size’: config.DEFAULT_CHUNK_SIZE,
            ’overlap’: config.DEFAULT_CHUNK_OVERLAP
        },
        embedding_interface=llm
    )

    graph_db = Neo4jGraphDB(logger=logger)

    rag_system = CNTRagSystem(
        llm_interface=llm,
        vector_store=vector_store,
        graph_db=graph_db,
        logger=logger,
        feedback_db_path=config.DEFAULT_FEEDBACK_DB_PATH
    )

    logger.info("RAG System is ready for integration.")

except Exception as e:
    logger.critical(f"Failed to initialize RAG system: {e}")
    rag_system = None
```

Listing 1: Initialization Code

4 Primary Integration Method: `stream_query_process()`

This method is the main interface for question-answering. It yields real-time events for integration with web or CLI frontends.

Signature

`stream_query_process(question: str) -> Generator[Dict[str, Any], None, None]`

Event Stream Format

Each event is a dictionary with keys:

- `event`: One of `"trace"`, `"sources"`, `"final_answer"`, `"suggestions"`, `"done"`
- `data`: Payload of the event

Example Usage

```
if rag_system:
    user_question = "What are the applications of single-walled carbon nanotubes?"

    for event in rag_system.stream_query_process(question=user_question):
        event_type = event.get("event")
        data = event.get("data")

        if event_type == "trace":
            print(f"-> LOG: {data}")
        elif event_type == "sources":
            print(f"-> SOURCES FOUND: {len(data)} sources.")
        elif event_type == "final_answer":
            print(f"\n--- FINAL ANSWER ---\n{data}\n")
        elif event_type == "suggestions":
            print(f"\n--- SUGGESTIONS ---\n{data}\n")
        elif event_type == "done":
            print("\n--- PROCESS COMPLETE ---")
            break
```

5 Handling Chat History

To maintain context across turns, use the following method:

Signature

`generate_contextual_query(chat_history: List[Dict], new_question: str) -> str`

Data Format

- `chat_history`: List of dictionaries with keys:
  - `"role"`: Either `"user"` or `"assistant"`
  - `"content"`: Message text

Integration Flow

1. Store chat history in your application state.
2. Call `generate_contextual_query()` with the chat history and user's new question.
3. Use the resulting query with `stream_query_process()`.

Example

```
chat_history = [
    {"role": "user", "content": "What are the main types of CNTs?"},
    {"role": "assistant", "content": "The main types are Single-Walled
   (SWCNTs) and Multi-Walled (MWCNTs)."}
]

new_question = "Tell me more about their electrical properties."

if rag_system:
    contextual_query = rag_system.generate_contextual_query(
        chat_history=chat_history,
        new_question=new_question
    )

    for event in rag_system.stream_query_process(question=
   contextual_query):
        # Handle events...
        pass
```

Conclusion

This document outlined the technical steps required to integrate the `CNTRagSystem` into a non-Streamlit environment. By following the initialization, event streaming, and chat history handling patterns, you can extend this powerful RAG system into your own applications with minimal changes.