**1. Initial Setup & Dependencies**

* **Imports**: Loads necessary libraries (langchain, neo4j, openai, etc.) for NLP, graph operations, and chatbot functionality.
* **Environment Variables**: Sets API keys and database credentials for OpenAI and Neo4j.

**2. Neo4j Graph Initialization**

* **Neo4jGraph()**: Connects to the Neo4j database using the provided URI and credentials.
* **Purpose**: Manages graph data (e.g., courses, faculty, requirements) for structured retrieval.

**3. Data Loading & Preprocessing**

* **WebBaseLoader**: Fetches content from NWMSU webpages (e.g., MSACS program details).
* **TokenTextSplitter**: Initializes TokenTextSplitter to break data into chunks of 512 tokens with 24 -token overlap.
  + **512 tokens** is a common limit for tokenization (e.g., for models like BERT,GPT) to avoid truncation.
  + **24-token overlap** ensures smooth transitions between chunks, helping models understand context better.
* **Output**: Processed text stored in documents for graph transformation.

**4. Graph Construction with LLM**

* **LLMGraphTransformer**: Converts text chunks (output of step 3) into a knowledge graph using OpenAI's LLM.
* **graph.add\_graph\_documents()**: Stores the graph in Neo4j (run once during initial setup).
* **Purpose**: Extracts entities (courses, professors) and relationships for structured queries.

**5. Graph Visualization**

* **showGraph()**: Visualizes Neo4j graph data in Google Colab using Cypher queries.

**6. Vector Index Setup**

* **Neo4jVector.from\_existing\_graph()**: Creates a hybrid (text + vector) search index on Neo4j. Enables hybrid search combining text and embeddings for efficient retrieval.
* **OpenAIEmbeddings() :** It converts text into numerical representations that preserve the original meaning and relationships between words and phrases.
* **Purpose**: Enables similarity-based retrieval of unstructured content.

**7. Entity Extraction & Structured Retrieval**

* **Entities Class**: – Defines categories for school-related things (courses, people, requirements).
* **ChatPromptTemplate.from\_messages()** method initializes a prompt template for interacting with users.
* **Entity Chain** – LLM pipeline that reads a question and picks out the important school-related things mentioned.
* **generate\_full\_text\_query** splits input into words and builds search query to finds similar words even with small typos or spelling mistakes, with ~2 tolerance.

Also finds *similar* words ("**apple" can find "appl", "aple", or "apples**")

* structured\_retriever extracts entities from a question and queries Neo4j for related graph data.

­

* Returns formatted relationships (e.g., node-to-node connections) as a string.

**8. Retriever**

* **retriever()**: Combines:
  + **Structured data**: From Neo4j graph (e.g., "Minimum Duolingo score: 110").
  + **Unstructured data**: From vector similarity search
    - **Unstructured data** refers to the raw text content (like paragraphs from websites) that hasn’t been converted into a specific format. Even after building a graph, it's used to provide extra context, explanations, or missing details that the graph might not capture.
* **Output**: Fused context for the LLM to generate answers.

**9. Question Condensing (Chat History)**

* **\_search\_query**: Reformulates follow-up questions (e.g., "What’s his email?" → "What is Dr. Smith’s email?") using chat history.
* **Uses**: CONDENSE\_QUESTION\_PROMPT to make questions standalone.

**10. Answer Generation Chain**

* **chain**: Pipeline to:
  1. Retrieve context (structured + unstructured).
  2. Pass the question and context to the LLM.
  3. Generate a natural language answer
* **chain.invoke({"question": "..."})** Each chain.invoke() tests how the RAG pipeline handles different types of questions like **Factual Questions**, **True/False Verification**, **Follow-up Questions and Edge cases.**
* **warnings.filterwarnings("ignore"):** The code hides small warnings and quiet logs from Neo4j unless it’s a big problem. It makes things less noisy and easier to read.
* **main.py** : consolidates the full pipeline (entity extraction, retrieval, and answering) into one file for simplicity and efficiency.

**11. Streamlit Chatbot UI**

* **app.py**: Launches a web interface with:
  + Creates a app.py file for a Streamlit web interface titled "NWMSU QA Chatbot"
  + Manages chat history and displays user/assistant messages interactively.

**Key Workflow Summary:**

1. **User Query** → 2. **Retrieve Graph + Text Context** → 3. **LLM Generates Answer** → 4. **Streamlit Displays Response**.

**Technical Workflow Summary**

1. **User Query Ingestion**
   * The user submits a natural language query (e.g., *"What is the minimum Duolingo score for MSACS?"*) via the Streamlit frontend.
   * The input is passed to the LangChain-based retrieval and generation pipeline.
2. **Context Retrieval (Hybrid Search)**
   * **Structured Retrieval (Neo4j Graph DB)**
     + The query is parsed using an LLM (entity\_chain) to extract entities (e.g., "Duolingo score").
     + A **full-text Cypher query** (structured\_retriever) retrieves connected nodes and relationships (e.g., (Requirement)-[HAS\_SCORE]->(Duolingo: 110)).
   * **Unstructured Retrieval (Vector Similarity Search)**
     + The same query is embedded using **OpenAI’s text-embedding** and matched against indexed document chunks in Neo4j (vector\_index.similarity\_search).
     + Returns semantically relevant text snippets (e.g., *"The MSACS program requires a minimum Duolingo score of 110."*).
3. **LLM-Based Answer Synthesis**
   * The retrieved structured (graph) and unstructured (text) contexts are formatted into a prompt.
   * OpenAI’s gpt-3.5-turbo generates a concise, natural language response conditioned on the provided evidence (e.g., *"The minimum Duolingo score required for MSACS is 110."*).
4. **Response Rendering (Streamlit UI)**
   * The generated answer is displayed in the chat interface.
   * Conversation history is preserved in st.session\_state for follow-up queries.

**Key Technical Components**

* **Neo4j**: Serves as both a knowledge graph (structured data) and vector store (unstructured embeddings).
* **LangChain**: Orchestrates retrieval, prompt engineering, and LLM interaction.
* **Hybrid Search**: Combines exact graph traversal with approximate vector search for comprehensive context retrieval.
* **Streamlit**: Provides a lightweight, interactive frontend for real-time Q&A.

This pipeline ensures **accurate, context-aware responses** by leveraging structured program rules (Neo4j) and unstructured documentation (vector search), synthesized via LLM reasoning.