Assignment 4: Simplified Agentic RAG System with LangGraph

1. Background

Retrieval-Augmented Generation (RAG) combines a knowledge base (KB) with a large language model (LLM) so the model can ground its answers in factual, pre-indexed content. An "agentic" RAG system adds a simple self-review loop: after producing an initial answer, the model checks its own output against the KB, identifies any missing or incorrect pieces, and (if needed) retrieves a bit more context to refine its response.

In this assignment, you will build a **lighter-weight** agentic RAG pipeline using LangGraph's Graph API. You will:

1. Load a small JSON-based KB (`self_critique_loop_dataset.json`) of software best-practice snippets.
2. Index those snippets into a vector store for fast semantic search.
3. Define a LangGraph graph containing:
    - A *retriever* node (fetches top-k relevant snippets),
    - A single *LLM-generator* node (produces an answer using retrieved context),
    - A *self-critique* node (checks if the answer is complete), and
    - A *refinement* node (if critique finds gaps, retrieve one more snippet and regenerate).
4. Expose a minimal interface (Jupyter notebook or simple Python script) to enter a question and see the end-to-end pipeline run (initial answer → critique → optional refinement → final answer).

---

2. Problem Statement & Detailed Tasks

> ***"Build a simplified Agentic RAG system using LangGraph that, for any user question about software engineering best practices, retrieves up to 5 relevant KB snippets, generates an initial LLM answer, self-critiques it, and—only when necessary—retrieves one extra snippet to refine the answer, finally returning a citation-backed response."***

2.1. Detailed Tasks

1. **Preprocessing & Indexing**
    - **Load the KB**: Read `self_critique_loop_dataset.json` (≈30 entries). Each entry has:

    ```
    {
      "doc_id": "KB001",
      "question": "What are best practices for debugging?",
      "answer_snippet": "When debugging, start with a minimal reproducible test, use logging, and apply divide-and-conquer to
      "source": "debugging_guide.md",
      "last_updated": "2024-01-10"
    }
    ```

    - **Compute Embeddings**: Use OpenAI's `text-embedding-3-small` (or a local sentence-transformer) to embed each `answer_snippet`.
    - **Upsert to Vector Store**: Choose Pinecone, Weaviate, or Qdrant. Create an index (e.g. `kb_index`) and upsert each record:

    ```
    {
      "id": "<doc_id>",
      "vector": <embedding array>,
      "metadata": { "source": "<filename>", "last_updated": "<date>" }
    }
    ```

    - Confirm all ~30 entries are indexed.
2. **LangGraph Graph Definition** Create a file named `agentic_rag_simplified.py` (or define inside a Jupyter notebook). In it, define a LangGraph graph with these four nodes:
    1. **Retriever Node**
        - **Name**: `retrieve_kb`
        - **Type**: "vector_retriever" (wraps a simple Python function).

- **Input**:
  - `user_question: str`
- **Operation**:
  1. Embed `user_question`.
  2. Query the vector index for the top 5 most similar `answer_snippet` vectors.
  3. Return a list of up to 5 hits, each `{ "doc_id": <id>, "answer_snippet": <text>, "source": <filename> }`.
- **Output**:
  - `kb_hits: List[Dict]`

2. **LLM Answer Node**
   - **Name**: `generate_answer`
   - **Type**: `LLMChainNode`
   - **Inputs**:
     - `user_question: str`
     - `kb_hits: List[Dict]`
   - **Prompt Template** (example):

```
You are a software best-practices assistant.
User Question:
{user_question}

Retrieved Snippets:
{for hit in kb_hits: print(f"[{hit['doc_id']}] {hit['answer_snippet']}")}

Task:
Based on these snippets, write a concise answer to the user’s question.
Cite each snippet you use by its doc_id in square brackets (e.g., [KB004]).
Return only the answer text.
```

   - **LLM Settings**:
     - Model: `gpt-4` (or `gpt-3.5-turbo`)
     - Temperature: `0`
   - **Output**:
     - `initial_answer: str`

3. **Self-Critique Node**
   - **Name**: `critique_answer`
   - **Type**: `LLMChainNode`
   - **Inputs**:
     - `user_question: str`
     - `initial_answer: str`
     - `kb_hits: List[Dict]`
   - **Prompt Template** (example):

```
You are a critical QA assistant. The user asked: {user_question}

Initial Answer:
{initial_answer}

KB Snippets:
{for hit in kb_hits: print(f"[{hit['doc_id']}] {hit['answer_snippet']}")}

Task:
Determine if the initial answer fully addresses the question using only these snippets.
- If it does, respond exactly: COMPLETE
- If it misses any point or cites missing info, respond: REFINE: <short list of missing topic keywords>

Return exactly one line.
```

- **LLM Settings**:
  - Model: `gpt-4`
  - Temperature: `0`
- **Output**:
  - `critique_result: str` (either "`COMPLETE`" or "`REFINE: ...`")

4. **Refinement Node**
   - **Name**: `refine_answer`
   - **Type**: `LLMChainNode`
   - **Inputs**:
     - `user_question: str`
     - `initial_answer: str`
     - `critique_result: str`
     - `kb_hits: List[Dict]`
   - **Operation**:
     1. Extract missing-topic keywords from `critique_result` (e.g., “cache invalidation”).
     2. Build a new query string:

        ```
        new_query = f"{user_question} and information on {missing_keywords}"
        ```

     3. Call the same retriever function to get **one additional snippet** (`top_k=1`) for that `new_query`.
   - **Prompt Template** (example):

```
You are a software best-practices assistant refining your answer. The user asked: {user_question}

Initial Answer:
{initial_answer}

Critique: {critique_result}

Additional Snippet:
[Code to display the single additional snippet’s doc_id and text]

Task:
Incorporate this snippet into the answer, covering the missing points.
Cite any snippet you use by doc_id in square brackets.
Return only the final refined answer.
```

   - **LLM Settings**:
     - Model: `gpt-4`
     - Temperature: `0`

- **Output**:
  - `refined_answer: str`

5. **Graph Control Flow**
   - **Wire nodes in sequence**:
     1. `retrieve_kb` → `generate_answer` → `critique_answer`.
     2. Add a simple **Decision** node (or an `if` check in your driver script) that:
        - If `critique_result == "COMPLETE"`, take `initial_answer` as final.
        - If `critique_result.startswith("REFINE")`, call the refinement logic (retrieve+refine) to produce `refined_answer`.
     3. Wrap whichever answer (initial or refined) into a JSON response:

```
{ "answer": "<final_answer_text>" }
```

---

3. Tools & Technologies

- **LangGraph** (Graph API)
  - Define nodes (retriever + LLM chains + decision logic) and connect them in a directed graph.
- **Vector Database** (choose one)
  - **Chroma**, **Weaviate**, or **Qdrant** for storing and querying embeddings.
- **Embeddings**
  - OpenAI's `text-embedding-ada-002` (or a publicly available sentence-transformer of your choice).
- **OpenAI LLM**
  - Use `gpt-4` (or `gpt-4o-mini`) via the OpenAI Python SDK.
  - All calls should specify `temperature=0`.
- **Python ≥ 3.10**
  - Develop either in a Jupyter notebook (`.ipynb`) or as a small folder of `.py` modules, then zip them up.
- **Pip-installed packages**:

```
langgraph
openai
pinecone-client         # or weaviate-client / qdrant-client
pydantic
```

---

4. Sample Input Queries

Use these queries to test the entire pipeline. Log each step (hits, initial answer, critique, possible refinement).

1. **"What are best practices for caching?"**
   - Expect retrieval of snippets such as "KB003: [Snippet about caching patterns]" and so on.
   - Initial answer cites one or two KB IDs (e.g., `[KB003]`).
   - Critique might return `REFINE: cache invalidation` if that subtopic was missing.
   - Refinement then retrieves one snippet on invalidation (`KB013`) and produces a final answer citing both `[KB003]` and `[KB013]`.
2. **"How should I set up CI/CD pipelines?"**
   - Expect hits like `KB007`, `KB017`.
   - Initial answer cites those snippets.
   - Critique likely returns `COMPLETE`.
   - Final JSON just returns the initial answer.
3. **"What are performance tuning tips?"**
   - Expect hits `KB002`, `KB012`.
   - Initial answer might cite `[KB002]`.

- Critique may respond `REFINE: profiling tools` if missing that detail.
- Refinement retrieves a profiling snippet (`KB022`) and final answer cites `[KB002]` and `[KB022]`.

4. **“How do I version my APIs?”**

   - Expect hits `KB005`, `KB015`.
   - Initial answer cites `[KB005]`.
   - Critique could be `REFINE: semantic versioning` if not mentioned.
   - Refinement snippet (`KB015`) added, final answer cites both IDs.

5. **“What should I consider for error handling?”**

   - Expect hits `KB009`, `KB019`.
   - Initial answer cites `[KB009]`.
   - Critique probably returns `COMPLETE` if logging, retries, and exception best practices are covered.
   - Final answer is the same as the initial.

---

5. Deliverables

You may submit **either** a single Jupyter notebook (`.ipynb`) showing all steps **or** a ZIP folder containing:

1. **Python Code Files**

   - `index_kb.py`

     - Loads `self_critique_loop_dataset.json`, computes embeddings, and upserts records to your chosen vector DB.

   - `agentic_rag_simplified.py` (or modules under a folder named `agentic_rag/`)

     - Defines the LangGraph graph with the four nodes:

       1. `retrieve_kb`
       2. `generate_answer`
       3. `critique_answer`
       4. `refine_answer`

     - Contains any helper functions (e.g., building `new_query` for refinement).

   - `executor.py` (optional if you did a pure Python approach)

     - Prompts the user for a question, runs the LangGraph graph, prints the final JSON.

   - `requirements.txt`

```
langgraph
openai
pinecone-client     # or weaviate-client / qdrant-client
pydantic
```

2. **Jupyter Notebook** (if you choose the notebook route)

   - Should contain all cells that:

     1. Import required libraries and set up API keys (e.g., `OPENAI_API_KEY`).

     2. Run `index_kb.py` logic to create embeddings and upsert.

     3. Define the LangGraph graph inline (preferably in its own cell).

     4. Execute the graph on each of the **5 sample queries**, showing:

        - `kb_hits` (initial retrieval),
        - `initial_answer`,
        - `critique_result`,
        - (if needed) `refined_answer`,
        - Final JSON response.

   - Include commentary/memos in Markdown cells that explain each step.

3. **Readme or Top-Cell Explanation**

- Brief instructions on how to run your notebook or execute the Python scripts.
- Outline any environment variables needed (`OPENAI_API_KEY`, `PINECONE_API_KEY`, etc.).

---

6. Notes & Tips

- **Keep It Simple**: Do not over-engineer. One retrieval pass, one critique, at most one extra snippet for refinement.

- **Citation Format**: Whenever the LLM references a snippet, it must write `[KBxxx]` exactly. If a citation is missing or incorrect, the self-critique node should flag it as a gap.

- **Prompt Consistency**: Always set `temperature=0` to ensure repeatable outputs.

- **Decision Logic**: You can implement the "COMPLETE vs. REFINE" check as a simple Python `if` statement in your driver code rather than a separate LangGraph decision node—either approach is fine.

- **Logging**: For each query, print out (or display) in your notebook:

    1. Retrieved KB IDs & snippets,
    2. Initial answer,
    3. Critique result,
    4. (If used) refined answer,
    5. Final JSON output.

- **Testing**: Before wiring the full graph, manually test:

    - `retrieve_kb_entries("test question", top_k=3)`
    - A small "Answer Generation" prompt by itself,
    - A small "Self-Critique" prompt by itself,
    - A quick "Refinement" prompt by itself.

- **Reproducibility**: Pin your library versions in `requirements.txt` to avoid mismatched LLM or vector-DB client behavior.

Good luck! This simplified assignment will give you a hands-on, clear path to building an agentic RAG pipeline—without excessive complexity.