A black background with white text

AI-generated content may be incorrect.

Let me summarize so you can see how it naturally maps to a **LangGraph pipeline**:

**🔄 Current Flow in answer\_query\_with\_validation**

1. **RAG Retrieval**
   * Uses FAISS/Chroma vectorstore
   * Retrieves schema/semantic context from embeddings
2. **KG Context Prep**
   * Extract table names from KG (get\_table\_names\_from\_kg)
   * Collect relationships (edges between tables)
3. **LLM SQL Generation**
   * Prompt with NL query + schema + joins
   * Constrained to SQLite syntax
   * No aliases, no mutations (only SELECT)
4. **Validation & Retry**
   * Parse via sqlglot AST → check syntax
   * Validate using KG → catch wrong columns/joins
   * If invalid → retry with hints
5. **Execution**
   * Run final SQL with run\_sql\_query
   * Return sql\_query, df

**⚡ LangGraph Flow Proposal**

In LangGraph, this translates neatly into **nodes**:

* **Start → Retriever → LLM\_SQL → Validator → Executor → End**
* With a **retry loop** if Validator fails

**📐 LangGraph Node Mapping**

* **Retriever Node**  
  Input: NL query  
  Output: semantic\_context
* **KG Node**  
  Input: schema graph kg  
  Output: canonical\_tables, rels
* **LLM\_SQL Node**  
  Input: nl\_query, semantic\_context, rels  
  Output: raw SQL proposal
* **Validator Node**  
  Input: SQL  
  Runs sqlglot.parse\_one + validate\_sql\_with\_kg  
  If invalid → return hints
* **Retry Edge**  
  Back to LLM\_SQL with augmented user\_prompt
* **Executor Node**  
  Input: valid SQL  
  Runs run\_sql\_query  
  Output: dataframe

**🔗 Control Flow in LangGraph**

Start

│

▼

Retriever ──► KG Node

│ │

▼ ▼

LLM\_SQL

│

▼

Validator

│ │

│ └─invalid→ Retry Edge → LLM\_SQL

▼

Executor

│

▼

End

**🔹 Step 1: Store and Represent Schema Properly**

When admin uploads tables:

1. **Extract schema** → table names, column names, data types, foreign keys (if any).
2. **Generate embeddings** for:
   * Table descriptions (if available).
   * Column names + sample values (to capture semantics).  
     Example: "orders.customer\_id → integer, links to customers.id".
3. **Build Knowledge Graph**:
   * Nodes = Tables & Columns.
   * Edges = Relationships (foreign keys, semantic links).  
     Store this in Neo4j or even as a NetworkX graph.

👉 Output: Vector store + Knowledge Graph.

**🔹 Step 2: User Query Handling**

When user asks a query:

1. **Vector Search**: Use embeddings to retrieve top-k relevant tables & columns.
2. **Knowledge Graph Traversal**: Ensure retrieved tables are connected → infer join paths.

👉 Output: Minimal relevant schema context.

**🔹 Step 3: LangGraph Orchestration**

Define a flow with nodes:

1. **Schema Retrieval Node**: Collects relevant tables & columns from vector + KG.
2. **Reasoning Node**: Convert NL query into a structured reasoning plan.  
   Example: “Find top 5 customers → need customers + orders + group by + sort”.
3. **SQL Generation Node**: LLM generates query using schema context.
4. **Validation Node**:
   * Parse with sqlglot or sqlparse.
   * Dry run → catch errors, auto-correct via re-prompt.
5. **Execution Node**: Run SQL and return results.

👉 Output: Valid SQL + Answer Table.

**🔹 Step 4: Feedback & Improvement Loop**

* Log user query, generated SQL, results.
* If wrong, allow user to give feedback → store correction.
* Use these logs to fine-tune later or improve prompts.

**🔹 Step 5: Tech Stack Choices**

* **Vector DB**: FAISS (local, simple) or Pinecone/Weaviate.
* **KG Storage**: Neo4j (best), or NetworkX (lighter).
* **Orchestration**: LangGraph (stateful) or LangChain (simpler).
* **Validation**: sqlglot, sqlparse.
* **DB Execution**: SQLAlchemy for multi-DB support.

👉 Suggested implementation order for you:

1. Add **vector search** over schema embeddings.
2. Add a **knowledge graph** to capture table relationships.
3. Use **LangGraph** to orchestrate retrieval → reasoning → SQL generation → validation.