**1. Initialization Phase (Code executed when main.py starts):**

* **import os, from loguru import logger, etc.:** These lines import necessary Python modules and libraries. os for environment variables, loguru for logging, langchain components for LLM interactions, langgraph for workflow management, and custom modules from utils.
* **os.environ["TOKENIZERS\_PARALLELISM"] = "false":** This line disables tokenizer parallelism, which can sometimes cause issues in multi-threaded environments.
* **llm = initialize\_openai(os.getenv("OPENAI\_API\_KEY")):**
  + Calls initialize\_openai from utils.llm to initialize the OpenAI language model (like GPT-4o).
  + It uses the OPENAI\_API\_KEY environment variable to authenticate with OpenAI.
  + The initialized LLM object is stored in the llm variable.
* **embeddings = initialize\_embeddings(os.getenv("HUGGINGFACE\_MODEL")):**
  + Calls initialize\_embeddings from utils.llm to initialize the HuggingFace embeddings model.
  + It uses the HUGGINGFACE\_MODEL environment variable to specify the model to use (e.g., carrick113/autotrain-wsucv-dqrgc).
  + The initialized embeddings model is stored in the embeddings variable.
* **db = initialize\_database(os.getenv("DATABASE\_URI")):**
  + Calls initialize\_database from utils.maria to establish a connection to the MariaDB database.
  + It uses the DATABASE\_URI environment variable (e.g., mysql+pymysql://...) to connect to your database.
  + The database connection object (using Langchain's SQLDatabase) is stored in the db variable.
* **qdrant\_store = qdrant\_on\_prem(embeddings, os.getenv("COLLECTION\_NAME")):**
  + Calls qdrant\_on\_prem from utils.vector\_database to initialize the Qdrant vector store.
  + It uses the embeddings model initialized earlier and the COLLECTION\_NAME environment variable to connect to your Qdrant instance and collection.
  + The initialized Qdrant vector store object is stored in qdrant\_store.
* **sql\_examples = load\_sql\_examples(os.getenv("EXAMPLES\_FILE\_PATH")):**
  + Calls load\_sql\_examples from utils.tools to load SQL examples from a file specified by EXAMPLES\_FILE\_PATH (e.g., efg.jsonl).
  + These examples are used for example-based query retrieval and are stored in the sql\_examples list.

**In summary, the initialization phase sets up all the necessary components: the LLM, embeddings model, database connection, vector store, and loads example queries. These components are then used by the SQL agent workflow.**

**2. run\_interactive() Function (Interactive Chat Loop):**

* **config = {"configurable": {"thread\_id": "sql\_agent\_thread"}}:** Sets up a configuration for LangGraph, specifically assigning a thread\_id for checkpointing and state management.
* **print("Hi I am SQL Agent. How can I help you today?"):** Prints a welcome message to the user.
* **while True::** Starts an infinite loop to continuously interact with the user until they exit.
  + **user\_query = input("User: ").strip():** Prompts the user to enter a question and reads their input from the command line. .strip() removes leading/trailing whitespace.
  + **if not user\_query: continue:** If the user enters nothing (just presses Enter), the loop continues to the next iteration, prompting for input again.
  + **print(f"User: {user\_query}"):** Prints the user's query for confirmation.
  + **result = sql\_agent\_workflow.invoke({"question": user\_query}, config=config):**
    - This is the core of the interaction. It calls the sql\_agent\_workflow function, passing the user's question as a dictionary {"question": user\_query}.
    - config=config passes the LangGraph configuration for state management.
    - sql\_agent\_workflow processes the question and returns a dictionary containing the response.
    - The returned dictionary is stored in the result variable.
  + **print(f"Assistant: {result['response']}"):** Prints the response from the result dictionary, which is the SQL agent's answer to the user's question.
  + **except KeyboardInterrupt::** Handles Ctrl+C (KeyboardInterrupt) to allow the user to gracefully exit the chat loop.
    - print("\nExiting chat..."): Prints an exit message.
    - break: Exits the while True loop, ending the interactive session.
  + **except Exception as e::** Catches any other exceptions that might occur during the chat loop (e.g., errors in sql\_agent\_workflow).
    - logger.error(f"Error in chat loop: {str(e)}"): Logs the error using loguru.
    - print(f"Assistant: Error: {str(e)}"): Prints a generic error message to the user.
* **if \_\_name\_\_ == "\_\_main\_\_": run\_interactive():** This standard Python construct ensures that the run\_interactive() function is called only when main.py is executed directly (not when imported as a module).

**In essence, run\_interactive() sets up a command-line interface for the user to ask questions to the SQL agent and receive responses in a loop.**

**3. sql\_agent\_workflow(inputs: dict) -> dict Function (Core SQL Agent Workflow):**

This function is the heart of the SQL agent. It takes the user's question as input and orchestrates the process of answering it.

* **question = inputs.get("question", "").strip():** Extracts the user's question from the inputs dictionary and removes any leading/trailing whitespace.
* **if not question: return {"response": "Please provide a question."}:** Handles the case where the user provides an empty question.
* **if is\_database\_question(question)::** Calls is\_database\_question from utils.tools to determine if the user's question is likely related to the database.
  + **If is\_database\_question returns True (Database Question Path):**
    - **logger.info(f"Database question detected: {question}"):** Logs that a database question is detected.
    - **Step 1: similar\_examples\_future = find\_similar\_examples(...) & similar\_examples = similar\_examples\_future.result():**
      * Calls find\_similar\_examples from utils.task to search the Qdrant vector store for examples similar to the user's question.
      * This uses vector embeddings to find semantically similar questions from your example set.
      * The results (similar examples) are stored in similar\_examples.
    - **Step 2: Check for Exact Match (Vector Search):**
      * Iterates through similar\_examples to see if any example question exactly matches the user's question (case-insensitive).
      * If an exact match is found:
        + sql\_query = example["sql"]: Extracts the SQL query from the matching example.
        + tables = extract\_tables\_from\_sql(sql\_query): Extracts table names from the SQL query using extract\_tables\_from\_sql from utils.task.
        + logger.info(f"Exact match found in Qdrant: '{sql\_query}'"): Logs that an exact match was found.
        + break: Exits the loop as an exact match is found.
    - **Step 3: Fallback to File-Based Examples:**
      * if not sql\_query:: If no exact match was found in vector search.
      * sql\_query, tables = search\_examples\_for\_sql(question, sql\_examples): Calls search\_examples\_for\_sql from utils.task to search for an exact match in the file-based sql\_examples list.
    - **Step 4: Generate SQL (If No Matches):**
      * if not sql\_query:: If still no sql\_query found (no exact matches in vector or file examples).
      * logger.info("No exact match found; generating SQL as fallback."): Logs that SQL generation is being used as a fallback.
      * tables = extract\_relevant\_tables(question, db): Calls extract\_relevant\_tables from utils.task to identify relevant tables in the database based on the user's question.
      * **check\_tables\_exist(tables, db):** Checks if the extracted tables actually exist in the database. If not, it returns an error response.
      * sql\_query\_future = generate\_dynamic\_sql(...) & sql\_query = sql\_query\_future.result():
        + Calls generate\_dynamic\_sql from utils.task to generate an SQL query dynamically using the LLM.
        + It provides the question, relevant tables, similar examples (for context), database connection, and the LLM to the function.
        + The generated sql\_query is obtained.
        + tables = extract\_tables\_from\_sql(sql\_query): Extracts table names from the generated SQL query.
    - **Step 5: Validate SQL Query:**
      * if sql\_query is None or sql\_query == "":: Checks if sql\_query is still empty or None after all attempts to find or generate it. If so, returns an error response.
      * **check\_tables\_exist(tables, db):** Again checks if the tables extracted from the (generated) SQL query exist in the database. Returns error if not.
      * validation\_future = validate\_sql\_with\_llm(...) & if not validation\_future.result()::
        + Calls validate\_sql\_with\_llm from utils.task to validate the generated SQL query using the LLM.
        + If validation fails, returns an error response.
    - **Step 6: Execute SQL and Get Response:**
      * response = execute\_sql\_with\_no\_data\_handling(...): Calls execute\_sql\_with\_no\_data\_handling from utils.task to:
        + Execute the sql\_query against the database using execute\_sql\_query.
        + Format the SQL result into a user-friendly natural language response using format\_response (implicitly called within execute\_sql\_with\_no\_data\_handling).
        + Handle cases where the SQL query returns no data, providing a polite "no data found" message.
        + The final formatted response is obtained.
  + **Else (Non-Database Question Path):**
    - logger.info(f"Non-database question detected: {question}"): Logs that a non-database question is detected.
    - response = generate\_polite\_response(question): Calls generate\_polite\_response to generate a polite response for non-database questions.
* **return {"response": response}:** Returns a dictionary containing the final response (either the SQL answer or a polite non-database/error message).

**In summary, sql\_agent\_workflow is the main function that orchestrates the entire process of handling user questions, from classifying them, searching for examples, generating SQL if needed, validating it, executing it, and finally formatting a user-friendly response, including handling cases where no data is found or errors occur.**

**4. generate\_polite\_response(question: str) -> str Function:**

* This is a simple function that generates a polite response when the user's question is not classified as a database question.
* It uses a ChatPromptTemplate to instruct the LLM to act as a friendly SQL agent and politely guide the user to ask database-related questions.
* It returns the LLM-generated polite response as a string.

**Overall Flow:**

1. **Initialization:** Sets up all necessary components (LLM, embeddings, DB, vector store, examples).
2. **Interactive Loop (run\_interactive()):** Continuously prompts the user for questions.
3. **Question Processing (sql\_agent\_workflow()):**
   * Classifies the question as database-related or not.
   * **For database questions:**
     + Searches for similar examples (vector and file-based).
     + Generates SQL if no examples are found.
     + Validates the SQL query.
     + Executes the SQL query.
     + Formats the SQL result into a natural language response (handling no-data cases).
   * **For non-database questions:**
     + Generates a polite, non-database-related response.
4. **Response Output:** Prints the agent's response to the user in the interactive loop.