Absolutely! Let's dive \*\*deep\*\* into the code you provided and explain \*\*every component\*\* in full detail — including \*\*functions, tools, nodes, edges, prompts, state management, logging, and control flow\*\*. This is a \*\*LangGraph-based SQL Agent\*\*, and we'll break it down \*\*step by step\*\*, so you understand not just \*what\* it does, but \*why\* and \*how\*.

---

# 🧠 \*\*Overview of the System\*\*

This is a \*\*stateful, LLM-powered SQL agent\*\* that:

- Connects to a \*\*SQLite database\*\*.

- Uses \*\*Groq's LLM (e.g., Llama-3.1-8b-instant)\*\* to interpret natural language questions.

- Generates \*\*safe SELECT queries\*\*.

- Executes them and returns \*\*natural language answers\*\*.

- Uses \*\*LangGraph\*\* for workflow orchestration.

- Includes \*\*intent detection\*\*, \*\*error handling\*\*, \*\*logging\*\*, and \*\*recursion protection\*\*.

It’s designed to be \*\*robust, auditable, and production-ready\*\*.

---

## 🔹 1. \*\*Imports & Setup\*\*

```python

import os

import logging

from datetime import datetime

from typing import Annotated, Literal, TypedDict, List, Dict, Any, Optional

from langchain\_community.utilities import SQLDatabase

from langchain\_community.agent\_toolkits import SQLDatabaseToolkit

from langchain\_core.messages import AIMessage, ToolMessage, HumanMessage

from langchain\_groq import ChatGroq

from langchain\_core.tools import tool

from langgraph.graph import END, StateGraph, START

from langgraph.graph.message import add\_messages

from langchain\_core.runnables import RunnableLambda, RunnableWithFallbacks

from langgraph.prebuilt import ToolNode

from langgraph.errors import GraphRecursionError

from langchain\_core.prompts import ChatPromptTemplate

import time

```

### 🔍 Why These?

| Module | Purpose |

|-------|--------|

| `SQLDatabase` | Connect to SQLite and run queries |

| `SQLDatabaseToolkit` | Provides pre-built tools like `sql\_db\_list\_tables`, `sql\_db\_schema` |

| `ChatGroq` | Fast LLM inference via Groq API |

| `ToolNode` | Executes tools (functions) in LangGraph |

| `StateGraph` | Defines a state machine workflow |

| `ChatPromptTemplate` | Structured prompts for LLM |

| `add\_messages` | Reducer function to merge messages into state |

| `RunnableWithFallbacks` | Adds error recovery to tool execution |

---

## 🔹 2. \*\*Logging System\*\*

```python

def get\_logger(name: str) -> logging.Logger:

logger = logging.getLogger(name)

logger.setLevel(logging.DEBUG)

logger.propagate = False

if logger.handlers:

return logger

log\_dir = "logs"

os.makedirs(log\_dir, exist\_ok=True)

log\_file = os.path.join(log\_dir, f"{datetime.now().strftime('%Y-%m-%d')}.log")

fh = logging.FileHandler(log\_file, encoding='utf-8')

fh.setLevel(logging.DEBUG)

formatter = logging.Formatter(

"%(asctime)s [%(levelname)-8s] %(name)s: %(message)s",

datefmt="%Y-%m-%d %H:%M:%S"

)

fh.setFormatter(formatter)

logger.addHandler(fh)

return logger

```

### 🔍 What It Does:

- \*\*No console output\*\* — all logs go to a file (`logs/YYYY-MM-DD.log`).

- Logs \*\*everything\*\*: DEBUG, INFO, WARNING, ERROR, CRITICAL.

- Prevents \*\*duplicate handlers\*\* (important in Jupyter).

- Used throughout the agent to \*\*track execution\*\*.

### ✅ Example Log:

```

2024-07-28 10:15:23 [INFO ] SQLAgent: Received query: 'hi whats upp...'

```

---

## 🔹 3. \*\*State Definition\*\*

```python

class State(TypedDict):

messages: Annotated[list[AnyMessage], add\_messages]

query\_attempts: int

final\_answer: Optional[str]

```

### 🔍 Purpose:

This defines the \*\*memory/state\*\* of the agent as it runs.

| Field | Type | Purpose |

|------|------|--------|

| `messages` | `list[AnyMessage]` | Chat history: Human, AI, Tool messages |

| `query\_attempts` | `int` | Prevent infinite loops during SQL generation |

| `final\_answer` | `str` | Final human-readable answer |

> `Annotated[..., add\_messages]` means: when new messages come in, \*\*append them\*\* to the existing list.

---

## 🔹 4. \*\*Main Class: `SQLAgent`\*\*

This is the core agent.

---

### ✅ `\_\_init\_\_()` — Initialization

```python

def \_\_init\_\_(self, db\_path: str, model\_name: str = "llama-3.1-8b-instant", groq\_api\_key: Optional[str] = None):

self.connection\_string = f"sqlite:///{db\_path}"

self.db = SQLDatabase.from\_uri(self.connection\_string)

self.llm = ChatGroq(model=model\_name, api\_key=..., temperature=0)

self.\_setup\_tools()

self.\_setup\_prompts()

self.\_build\_graph()

```

#### 🔍 Steps:

1. \*\*Connect to SQLite\*\* via `SQLDatabase`.

2. \*\*Initialize LLM\*\* (Groq) with `temperature=0` for \*\*deterministic output\*\*.

3. \*\*Setup tools, prompts, and graph\*\*.

---

## 🔹 5. \*\*Tool Setup: `\_setup\_tools()`\*\*

```python

toolkit = SQLDatabaseToolkit(db=self.db, llm=self.llm)

tools = toolkit.get\_tools()

```

### 🔍 Tools Retrieved:

| Tool | Purpose |

|------|--------|

| `sql\_db\_list\_tables` | Lists all table names |

| `sql\_db\_schema` | Gets DDL schema of a table |

| `db\_query\_tool` | \*\*Custom tool\*\* to run SQL queries |

### ✅ Custom Tool: `db\_query\_tool`

```python

@tool

def db\_query\_tool(query: str) -> str:

result = self.db.run\_no\_throw(query)

return str(result) if result else "No results."

```

- `run\_no\_throw`: Prevents crashes on bad SQL.

- Logs every query and result.

- Returns string or error.

> This is \*\*not\*\* a LangChain auto-tool — it's \*\*manually defined\*\* to give full control.

---

## 🔹 6. \*\*Prompts: `\_setup\_prompts()`\*\*

### ✅ 1. \*\*Query Generation Prompt\*\*

```python

"""You are an expert SQLite assistant...

Return ONLY the SQL query, nothing else."""

```

- Forces LLM to output \*\*only SQL\*\*.

- Rules: `SELECT` only, use `LIMIT`, correct syntax.

### ✅ 2. \*\*Interpretation Prompt\*\*

```python

"""You are a data analyst. Interpret SQL results...

Start with 'Answer: '"""

```

- Converts raw results into \*\*natural language\*\*.

- Handles empty results and errors.

### ✅ 3. \*\*Intent Classification Prompt (NEW)\*\*

```python

"""Determine if user message is:

- 'greeting' → small talk

- 'query' → data request

Respond ONLY with 'greeting' or 'query'."""

```

- Prevents full workflow for "Hi", "Hello".

- Uses LLM to \*\*classify intent\*\*.

---

## 🔹 7. \*\*Tool Node with Fallback\*\*

```python

def \_create\_tool\_node\_with\_fallback(self, tools: list) -> RunnableWithFallbacks:

def handle\_tool\_error(state: Dict) -> Dict:

return {

"messages": [

ToolMessage(content=f"Error: {repr(error)}", tool\_call\_id=tc["id"])

]

}

return ToolNode(tools).with\_fallbacks([RunnableLambda(handle\_tool\_error)])

```

### 🔍 What It Does:

- Wraps tool execution.

- If a tool fails (e.g., invalid SQL), \*\*catches the error\*\*.

- Adds an error message to the chat history.

- Prevents agent crash.

> This is \*\*critical\*\* for robustness.

---

## 🔹 8. \*\*Graph Workflow: `\_build\_graph()`\*\*

This is where \*\*LangGraph\*\* orchestrates everything.

### 🧱 Nodes in the Graph

| Node | Type | Purpose |

|------|------|--------|

| `first\_tool\_call` | Function Node | Start by listing tables |

| `list\_tables\_tool` | ToolNode | Run `sql\_db\_list\_tables` |

| `model\_get\_schema` | Function Node | Ask LLM which tables to inspect |

| `get\_schema\_tool` | ToolNode | Run `sql\_db\_schema` |

| `query\_gen` | Function Node | Generate SQL |

| `execute\_query` | Function Node | Run SQL |

| `interpret\_results` | Function Node | Convert result to natural language |

---

## 🔹 9. \*\*Detailed Node Functions\*\*

### ✅ `first\_tool\_call(state)`

```python

def first\_tool\_call(state: State) -> Dict:

return {

"messages": [

AIMessage(

content="",

tool\_calls=[{"name": "sql\_db\_list\_tables", "args": {}, "id": "tool\_abcd123"}]

)

],

"query\_attempts": 0,

"final\_answer": None

}

```

- \*\*Starts the workflow\*\* by asking for table names.

- Sends an `AIMessage` with a \*\*tool call\*\* to `sql\_db\_list\_tables`.

- Resets `query\_attempts`.

> This is the \*\*trigger\*\* that kicks off the entire process.

---

### ✅ `model\_get\_schema(state)`

```python

def model\_get\_schema(state: State) -> Dict:

chat\_with\_get\_schema = self.llm.bind\_tools([self.get\_schema\_tool])

result = chat\_with\_get\_schema.invoke(messages)

return {"messages": [result]}

```

- After getting table names, this node asks the LLM:

> "Which tables should I inspect?"

- The LLM responds by \*\*calling `sql\_db\_schema`\*\* on relevant tables.

- Uses `bind\_tools([self.get\_schema\_tool])` so LLM knows it can use that tool.

> This is \*\*smart routing\*\* — LLM decides what schema to fetch.

---

### ✅ `query\_gen\_node(state)`

```python

def query\_gen\_node(state: State) -> Dict:

query\_attempts = state.get("query\_attempts", 0) + 1

if query\_attempts > 3:

return {"messages": [AIMessage("Unable to generate valid SQL.")], "query\_attempts": query\_attempts}

query\_response = (self.query\_gen\_prompt | self.llm).invoke({"messages": messages})

return {"messages": [query\_response], "query\_attempts": query\_attempts}

```

- Uses the \*\*query generation prompt\*\* to ask LLM for SQL.

- Increments `query\_attempts`.

- Stops after 3 attempts to \*\*prevent infinite loops\*\*.

> This is where the \*\*SQL is born\*\*.

---

### ✅ `execute\_query\_node(state)`

```python

def execute\_query\_node(state: State) -> Dict:

sql\_query = last\_message.content.strip()

# Extract SELECT line

if "SELECT" in sql\_query.upper():

for line in sql\_query.split('\n'):

if 'SELECT' in line.upper():

sql\_query = line.strip().rstrip('.')

break

result = self.db.run\_no\_throw(sql\_query)

return {

"messages": [ToolMessage(content=str(result), tool\_call\_id="manual\_query\_execution")]

}

```

- Extracts SQL from the last message.

- Cleans it (removes periods, gets `SELECT` line).

- Runs it with `run\_no\_throw`.

- Returns result as a `ToolMessage`.

> This is \*\*safe execution\*\* — no crashes on bad SQL.

---

### ✅ `interpret\_results\_node(state)`

```python

def interpret\_results\_node(state: State) -> Dict:

interpretation = (self.interpret\_prompt | self.llm).invoke({"messages": messages})

return {

"messages": [interpretation],

"final\_answer": interpretation.content

}

```

- Takes the SQL result and asks LLM:

> "What does this mean in plain English?"

- Uses the \*\*interpretation prompt\*\*.

- Stores answer in `final\_answer`.

> This is the \*\*final human-facing output\*\*.

---

## 🔹 10. \*\*Conditional Edges (Decision Logic)\*\*

These control \*\*where the workflow goes next\*\*.

### ✅ `should\_continue\_after\_query\_gen(state)`

```python

if 'SELECT' in last\_message.content.upper():

return "execute\_query"

else:

return END

```

- If the LLM output contains `SELECT`, go to `execute\_query`.

- Else, \*\*end\*\* (e.g., if LLM said "I don’t know").

---

### ✅ `should\_continue\_after\_execution(state)`

```python

if result starts with "Error":

if query\_attempts >= 3: return END

else: return "query\_gen" # Retry

elif result is ToolMessage: return "interpret\_results"

else: return END

```

- If query \*\*fails\*\*, retry up to 3 times.

- If \*\*succeeds\*\*, go to interpretation.

- Prevents infinite retries.

---

### ✅ `should\_continue\_after\_interpretation(state)`

```python

return END

```

- Always ends after interpretation.

- Final answer is ready.

---

## 🔹 11. \*\*Graph Construction\*\*

```python

workflow = StateGraph(State)

workflow.add\_node("first\_tool\_call", first\_tool\_call)

workflow.add\_node("list\_tables\_tool", ToolNode(...))

...

workflow.add\_edge(START, "first\_tool\_call")

workflow.add\_edge("first\_tool\_call", "list\_tables\_tool")

workflow.add\_conditional\_edges("query\_gen", should\_continue\_after\_query\_gen)

self.app = workflow.compile()

```

### 🔍 Workflow Flow:

```

START

↓

first\_tool\_call → (calls tool)

↓

list\_tables\_tool → (returns tables)

↓

model\_get\_schema → (LLM decides which tables to inspect)

↓

get\_schema\_tool → (fetches schema)

↓

query\_gen → (LLM generates SQL)

↙ ↘

[has SELECT?] → execute\_query → [success?] → interpret\_results → END

↓ ↓

END [error?] → retry (back to query\_gen)

```

This is a \*\*looping, retrying, intelligent workflow\*\*.

---

## 🔹 12. \*\*Main Method: `query()`\*\*

```python

def query(self, question: str, recursion\_limit: int = 10) -> Dict[str, Any]:

```

### 🔍 Steps:

#### 1. \*\*Intent Classification\*\*

```python

intent = self.intent\_prompt | self.llm.invoke({"input": question})

if intent == "greeting":

return {"sql\_query": None, "answer": "Hello! How can I assist you today?"}

```

- Uses LLM to detect if user is just saying "Hi".

- If yes, \*\*skip full workflow\*\*.

#### 2. \*\*Run the Graph\*\*

```python

self.app.invoke({

"messages": [HumanMessage(content=question)],

"query\_attempts": 0,

"final\_answer": None

}, config={"recursion\_limit": 15})

```

- Starts the agent with the user question.

- Sets recursion limit to prevent infinite loops.

#### 3. \*\*Extract Results\*\*

- Calls `\_extract\_final\_sql\_query()` to get the SQL.

- Gets `final\_answer` from state.

#### 4. \*\*Error Handling\*\*

- `GraphRecursionError`: Too many steps → user-friendly message.

- Other errors → generic fallback.

---

## 🔹 13. \*\*Helper Methods\*\*

### ✅ `\_extract\_final\_sql\_query(messages)`

- Scans messages in reverse.

- Finds the last message with `SELECT`.

- Returns cleaned SQL.

### ✅ `get\_table\_info()`

- Returns full DDL schema of all tables.

### ✅ `list\_tables()`

- Returns list of table names.

> Useful for debugging or external use.

---

## 🔹 14. \*\*Example Usage\*\*

```python

agent = SQLAgent(db\_path="final\_ecommerce.db")

result = agent.query("show me Arun Pandey's email")

print(result["sql\_query"])

print(result["answer"])

```

### 🔍 What Happens:

1. Intent → `query` (not greeting).

2. List tables → `users`, `orders`, etc.

3. Get schema of `users`.

4. Generate SQL: `SELECT email FROM users WHERE name = 'Arun Pandey';`

5. Execute → get result.

6. Interpret → "Answer: Arun Pandey's email is arun@example.com."

---

## 🔐 \*\*Safety & Best Practices\*\*

| Feature | Why It Matters |

|-------|----------------|

| `temperature=0` | Deterministic SQL output |

| `run\_no\_throw` | No crashes on bad SQL |

| Max 3 query attempts | Prevent infinite loops |

| Recursion limit | Prevent stack overflow |

| Only `SELECT` | No destructive queries |

| Logging to file | Full audit trail |

| Intent detection | Better UX |

---

## 🧩 \*\*Why This Architecture Works\*\*

| Component | Benefit |

|---------|--------|

| \*\*LangGraph\*\* | Stateful, conditional, retryable workflows |

| \*\*ToolNode + Fallback\*\* | Robust tool execution |

| \*\*Custom Prompts\*\* | Precise control over LLM behavior |

| \*\*Logging\*\* | Debugging and monitoring |

| \*\*Intent Detection\*\* | Smarter UX |

| \*\*Separation of Concerns\*\* | Easy to maintain and extend |

---

## ✅ Summary: What Happens When You Ask a Question?

1. \*\*Intent Check\*\*: Is it a greeting? If yes → quick reply.

2. \*\*List Tables\*\*: "What tables exist?"

3. \*\*Get Schema\*\*: "What columns are in `users`?"

4. \*\*Generate SQL\*\*: "Write a SELECT query for Arun Pandey’s email."

5. \*\*Execute\*\*: Run the SQL.

6. \*\*Interpret\*\*: "His email is arun@example.com."

7. \*\*Return\*\*: Final answer + SQL.

---

## 🚀 Suggestions for Improvement

- Add \*\*query validation\*\* (e.g., reject `DROP`).

- Cache schema to avoid repeated calls.

- Support \*\*JOINs\*\* with schema-aware prompts.

- Add \*\*chat history\*\* for multi-turn conversations.

- Use \*\*structured output\*\* (Pydantic) for better parsing.

---

Let me know if you'd like:

- A \*\*diagram\*\* of the graph flow.

- A version using \*\*Gemini\*\* instead of Groq.

- A \*\*Flask API\*\* wrapper.

- Support for \*\*PostgreSQL\*\* or \*\*MySQL\*\*.

I'm happy to help!