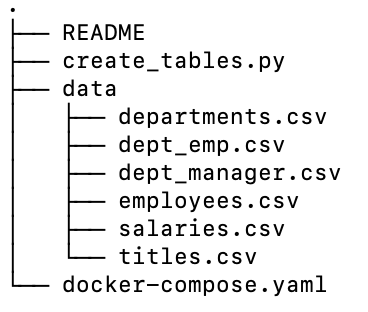
**Building an AI Agent for Text-to-SQL with LangGraph: Part I**

In Part II, we built a LangGraph-based AI agent that translates natural language queries into SQL (Text-to-SQL agent), executes them, and retrieves the results. To enable the agent to function end-to-end, setting up the database is a crucial first step, allowing seamless interaction and ex We will use a PostgreSQL Docker container to build the database, so Docker must be installed beforehand. Once Docker is set up, download all CSV files and place them in the data folder within our working directory, as shown below. Additionally, ensure you have the necessary files: README, docker-compose.yaml, and create\_tables.py in the current working directory as shown below. Here is the [**data source**](https://github.com/cmoeser5/Employee-Database-SQL/tree/main/SQL).



The docker-compose.yml file configures a PostgreSQL database service with persistent storage, making it accessible on port 5432. It initializes the database with predefined user credentials and creates a database named my\_database. Below is the Docker Compose file.

version: "3.8"  
services:  
 postgres:  
 image: postgres:latest  
 container\_name: postgres\_container  
 restart: always  
 ports:  
 - "5432:5432"  
 environment:  
 POSTGRES\_USER: postgres  
 POSTGRES\_PASSWORD: postgres  
 POSTGRES\_DB: my\_database  
 volumes:  
 - postgres\_data:/var/lib/postgresql/data  
volumes:  
 postgres\_data:

Below is the create\_tables.py script, which is responsible for uploading and updating data in the PostgreSQL database.

import psycopg2  
import os  
  
# PostgreSQL connection parameters  
conn = psycopg2.connect(  
 host="localhost",  
 port="5432",  
 database="my\_database",  
 user="postgres",  
 password="postgres"  
)  
cursor = conn.cursor()  
  
# Directory containing CSV files  
csv\_dir = "data/"  
  
# Loop through each CSV file and import  
for file in ['titles.csv', 'employees.csv', 'departments.csv', 'dept\_emp.csv', 'dept\_manager.csv', 'salaries.csv']:  
#for file in os.listdir(csv\_dir):  
 print(file)  
 if file.endswith(".csv"):  
 table\_name = os.path.splitext(file)[0] # Table name based on file name  
 file\_path = os.path.join(csv\_dir, file)  
 print(table\_name)  
  
 # Create table (adjust schema as needed)  
 if table\_name == "employees":  
 cursor.execute("""  
 CREATE TABLE employees (  
 emp\_no INT NOT NULL,  
 emp\_title\_id VARCHAR NOT NULL,  
 birth\_date DATE NOT NULL,  
 first\_name VARCHAR NOT NULL,  
 last\_name VARCHAR NOT NULL,  
 sex VARCHAR NOT NULL,  
 hire\_date DATE NOT NULL,  
 FOREIGN KEY (emp\_title\_id) REFERENCES titles (title\_id),  
 PRIMARY KEY (emp\_no)  
 );  
 """)  
 elif table\_name == "departments":  
 cursor.execute("""  
 CREATE TABLE departments (  
 dept\_no VARCHAR NOT NULL,  
 dept\_name VARCHAR NOT NULL,  
 PRIMARY KEY (dept\_no)  
 );  
 """)  
 elif table\_name == "titles":  
 cursor.execute("""  
 CREATE TABLE titles (  
 title\_id VARCHAR NOT NULL,  
 title VARCHAR NOT NULL,  
 PRIMARY KEY (title\_id)  
 );  
 """)  
 elif table\_name == "dept\_emp":  
 cursor.execute("""  
 CREATE TABLE dept\_emp (  
 emp\_no INT NOT NULL,  
 dept\_no VARCHAR NOT NULL,  
 FOREIGN KEY (emp\_no) REFERENCES employees (emp\_no),  
 FOREIGN KEY (dept\_no) REFERENCES departments (dept\_no),  
 PRIMARY KEY (emp\_no, dept\_no)  
 );  
 """)  
 elif table\_name == "salaries":  
 cursor.execute("""  
 CREATE TABLE salaries (  
 emp\_no INT NOT NULL,  
 salary INT NOT NULL,  
 FOREIGN KEY (emp\_no) REFERENCES employees (emp\_no),  
 PRIMARY KEY (emp\_no)  
 );  
 """)  
 elif table\_name == "dept\_manager":  
 cursor.execute("""  
 CREATE TABLE dept\_manager (  
 dept\_no VARCHAR NOT NULL,  
 emp\_no INT NOT NULL,  
 FOREIGN KEY (emp\_no) REFERENCES employees (emp\_no),  
 FOREIGN KEY (dept\_no) REFERENCES departments (dept\_no),  
 PRIMARY KEY (dept\_no, emp\_no)  
 );  
 """)  
 # Import CSV into table  
 with open(file\_path, "r") as f:  
 cursor.copy\_expert(f"""  
 COPY {table\_name} FROM STDIN  
 DELIMITER ',' CSV HEADER;  
 """, f)  
  
# Commit changes and close connection  
conn.commit()  
cursor.close()  
conn.close()

Once Docker is installed, and all necessary CSV files are placed in the data folder, along with the required files (create\_tables.py and docker-compose.yml), we can proceed step by step as outlined in the README file. By following these instructions, we will successfully create a PostgreSQL database, update all necessary tables, and have the database fully ready for Part II.

# Steps-   
   
## Download the Data: Retrieve all necessary data from the following links   
   
https://github.com/cmoeser5/Employee-Database-SQL/tree/main/employee\_sql  
https://github.com/cmoeser5/Employee-Database-SQL/blob/main/SQL/sql\_bonus.ipynb   
   
## Start PostgreSQL Using Docker: Run the docker-compose.yaml file from your working directory with   
  
docker compose up -d  
   
Note: If a new volume is created, Docker prepends the directory name, so the volume name will be texttosql\_postgres\_data. You can verify it by checking the Docker Compose configuration - 'docker compose config'  
   
## Verify the Volume Name: Inspect the volume using:   
   
docker volume inspect texttosql\_postgres\_data   
   
## Upload Data into PostgreSQL: Run the script to create tables and upload CSV files into the database. This script reads all CSV files from the data folder and uploads them one by one.   
  
python create\_tables.py  
   
## view the database - entering sql consol from the terminal   
  
docker exec -it postgres\_container psql -U postgres -d my\_database  
   
## find a list of tables  
  
\dt  
   
## Find elements from a given table  
  
SELECT \* FROM employees limit 10;  
   
## all databases   
  
\l  
## exit sql consol  
  
\q

All the required steps are outlined in the README file as below.

**Download the Data:**Retrieve all required data (csv format) from the following links: [Employee Database SQL Repository](https://github.com/cmoeser5/Employee-Database-SQL/tree/main/employee_sql) & [SQL Bonus Notebook](https://github.com/cmoeser5/Employee-Database-SQL/blob/main/SQL/sql_bonus.ipynb)

**Start PostgreSQL Using Docker:**Run the docker-compose.yaml file from your working directory:

docker compose up -d

**Note:** If a new volume is created, Docker automatically prefixes the directory name, making the volume name texttosql\_postgres\_data. Verify this using:

docker compose config

**Verify the Volume Name:**Check the created volume with:

docker volume inspect texttosql\_postgres\_data

**Upload Data into PostgreSQL:**Run the script to create tables and upload CSV files into the database:

python create\_tables.py

This script reads all CSV files from the data folder and loads them one by one into the database.

**Access the Database via SQL Console:**Enter the PostgreSQL console from the terminal:

docker exec -it postgres\_container psql -U postgres -d my\_database

**View Available Tables:**List all tables in the database:

\dt

**Retrieve Sample Data from a Table:**Fetch the first 10 rows from the employees table:

SELECT \* FROM employees LIMIT 10;

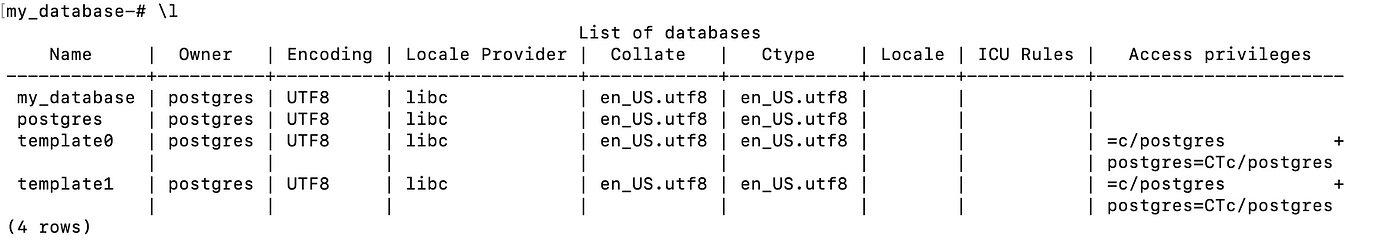
**List All Databases:**View all databases:

\l

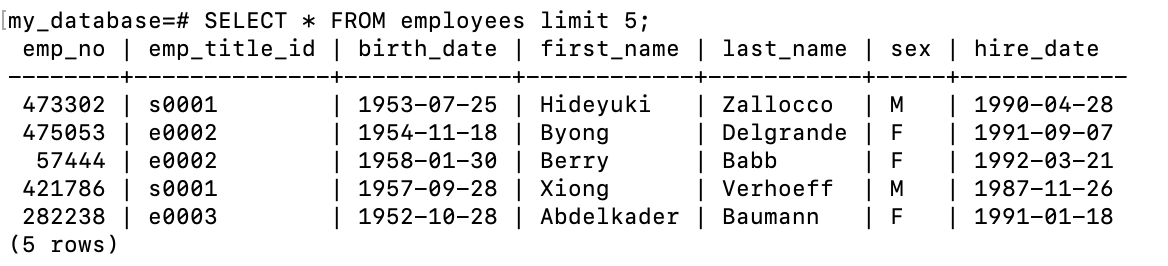
**Exit the SQL Console:**Use the following command to exit:

\q

Once these steps are completed, we can verify that the data is successfully loaded into the database. For instance, using the \l command will display the available databases as follows:



We can easily execute SQL operations directly in the console.



By completing these steps, we have successfully prepared the data for the text-to-SQL AI agent. In Part II, we will focus on building the agent and testing it on this dataset. You can find Part II [here](https://medium.com/@kbdhunga/building-an-ai-agent-for-text-to-sql-with-langgraph-part-ii-5685b95a7415).

**Building an AI Agent for Text-to-SQL with LangGraph: Part II**

19

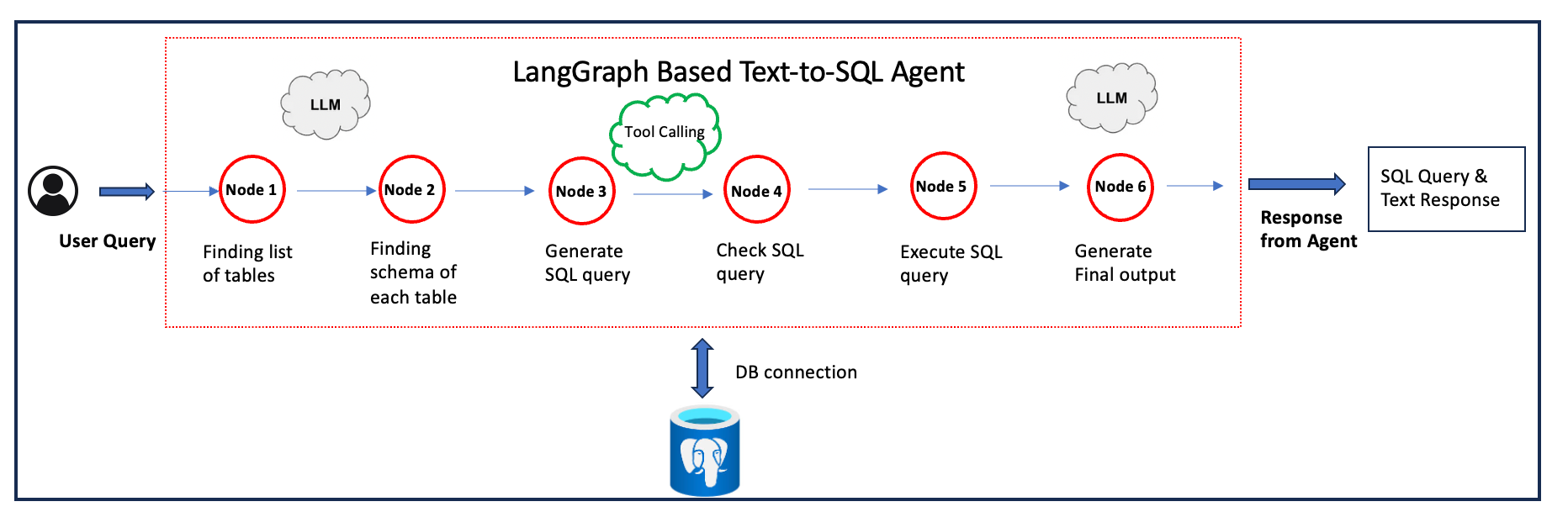


Image created by the author.

A Text-to-SQL AI agent is a system that translates natural language queries into SQL statements, enabling users to interact with databases without needing to know SQL syntax. This technology is crucial as it democratizes data access, allowing non-technical users to retrieve and analyze information directly, thereby enhancing decision-making and operational efficiency.

We will build an AI agent using LangGraph that translates natural language queries into SQL statements, executes them, and retrieves the results. This process consists of two main phases:

**Part I - Database Setup**

We set up a PostgreSQL database and load multiple datasets as tables. Part I is documented separately and [**can be accessed here**](https://medium.com/@kbdhunga/building-an-ai-agent-for-text-to-sql-with-langgraph-part-i-6e43c2f9b203). We highly recommend completing Part I before proceeding to Part II, as the database is crucial for running the graph seamlessly from start to finish.

We have adopted this agentic framework from the official LangGraph documentation, [**available here**](https://langchain-ai.github.io/langgraph/tutorials/sql-agent/). For those who wish to run the text-to-SQL agent without setting up the database as outlined in Part I, an alternative data source is provided [**here**](https://langchain-ai.github.io/langgraph/tutorials/sql-agent/). Additionally, some nodes and edges have been modified here from the original example to better suit this implementation.

**[Building an AI Agent for Text-to-SQL with LangGraph: Part I](https://medium.com/@kbdhunga/building-an-ai-agent-for-text-to-sql-with-langgraph-part-i-6e43c2f9b203?source=post_page-----5685b95a7415---------------------------------------" \t "_blank)**

[In Part II, we built a LangGraph-based AI agent that translates natural language queries into SQL (Text-to-SQL agent)…](https://medium.com/@kbdhunga/building-an-ai-agent-for-text-to-sql-with-langgraph-part-i-6e43c2f9b203?source=post_page-----5685b95a7415---------------------------------------" \t "_blank)

[medium.com](https://medium.com/@kbdhunga/building-an-ai-agent-for-text-to-sql-with-langgraph-part-i-6e43c2f9b203?source=post_page-----5685b95a7415---------------------------------------" \t "_blank)

In Part I, we set up the following PostgreSQL database. Using this database, we can evaluate the performance of the text-to-SQL agent.

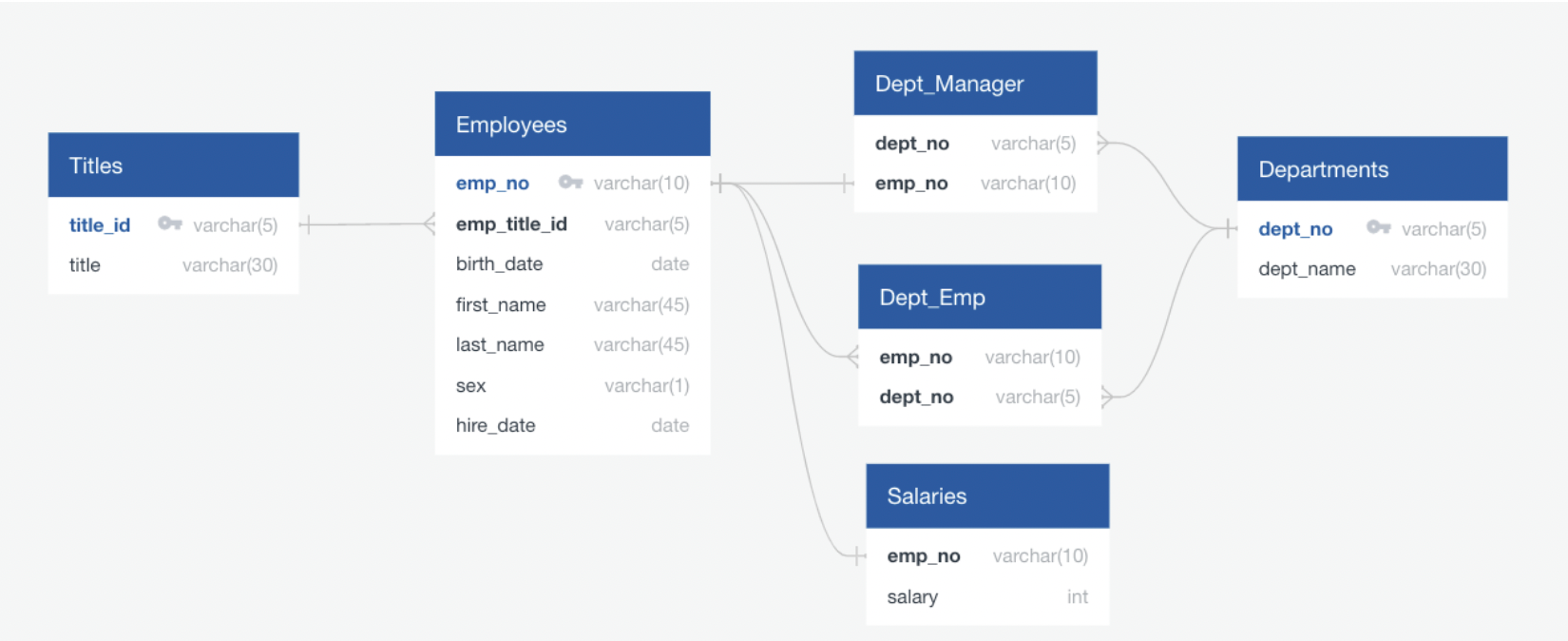


Image taken from https://github.com/cmoeser5/Employee-Database-SQL

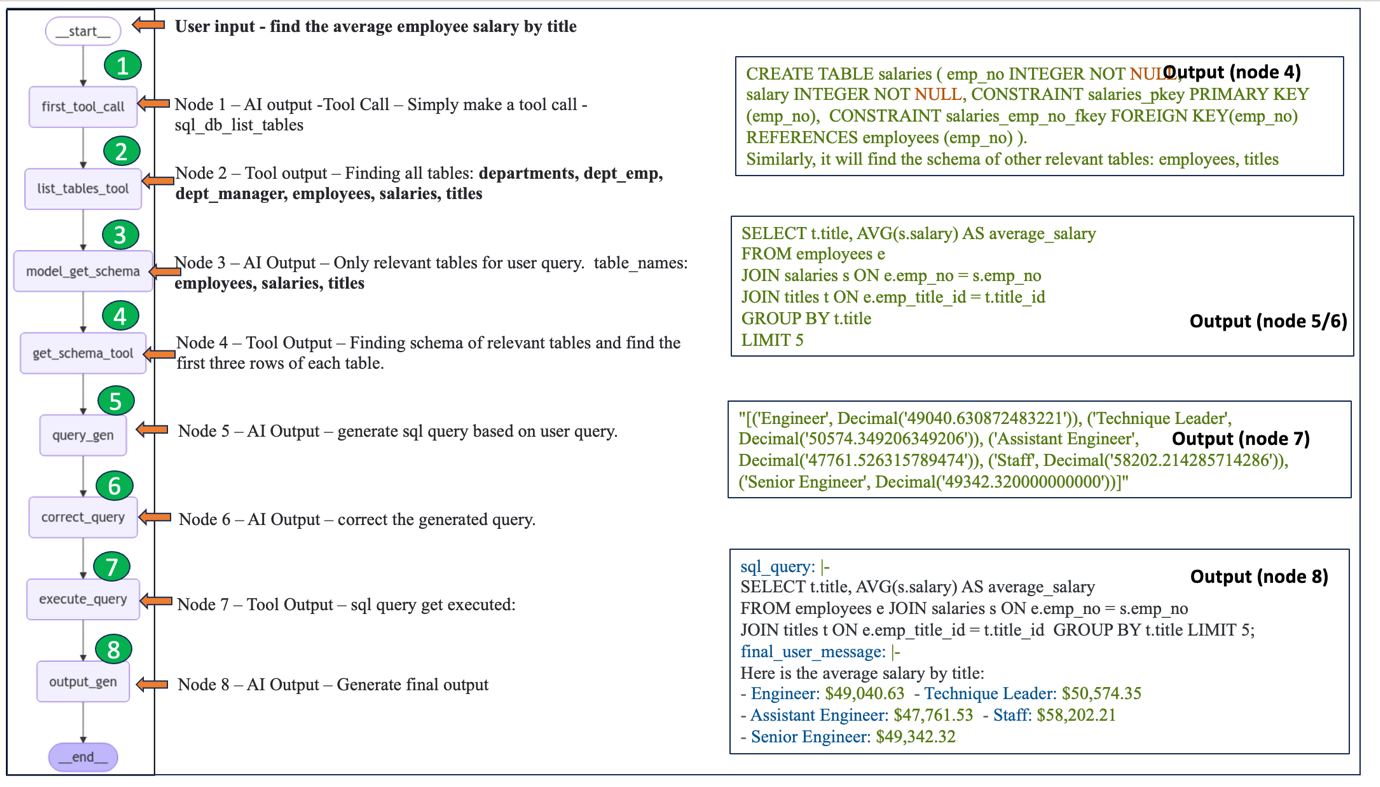
**Part II - Building an AI Agent**

We have built a LangGraph-based text-to-SQL agent that interacts with the database, generates SQL queries from user input, executes them, and retrieves the results.

In this article, **we will focus on Part II**, where we develop a LangGraph-based text-to-SQL agent and explore its various components in detail. Let’s break it down step by step.

[**A full Jupyter Notebook is available here.**](https://github.com/DhunganaKB/OpenAI-Chat/blob/main/LangGraph/text_to_sql.ipynb)

The image below illustrates the complete agent with various nodes, demonstrating how each node processes the user input and generates its corresponding output.



**Connecting to the Database and Testing**

As discussed in part I, first we should run the docker container so that agent can access the database. To start, ensure all environment variables are stored in a .env file located in the current working directory. Once the variables are set in the .env file, they can be loaded using the load\_dotenv function from the dotenv library. Here's an example of how the .env file might look:

OPENAI\_API\_KEY="sk-proj-yZq6wV54x1-4D4WMOnOPqEoCaRItFPYsSHOGdWtyB-1Lixc4SHVJIRPQRrLnUT3BlbkFJkYqEChWv0SwkoH9Mvd1tAVVKs\_tzOkmufiAMw98NtPktwani18VMetUQQA"  
LANGCHAIN\_API\_KEY="lsv2\_pt\_03bbeafb31944ebd3e738\_2b9dea39f"

The following piece of code allows us to connect with the PostgresSQL database. Also we can find different avaiable tables and schema once the connection is established. All PostgreSQL credentials are specified in the docker-compose file in Part I. Using these credentials, we created a connection string and utilized the SQLDatabase library to establish a connection with the database. Additionally, we have used OpenAI’s GPT-4o model for query generation and processing.

from sqlalchemy import create\_engine  
import pandas as pd  
#import matplotlib.pyplot as plt  
import numpy as np  
from dotenv import load\_dotenv  
load\_dotenv()  
from langchain\_openai import ChatOpenAI  
import os  
from langchain\_community.utilities import SQLDatabase  
from typing import Any  
import json  
  
from langchain\_core.messages import ToolMessage, HumanMessage, AIMessage  
from langchain\_core.runnables import RunnableLambda, RunnableWithFallbacks  
from langgraph.prebuilt import ToolNode  
from langchain\_community.agent\_toolkits import SQLDatabaseToolkit  
from langchain\_core.tools import tool  
from langchain\_core.prompts import ChatPromptTemplate  
  
from typing import Annotated, Literal  
from pydantic import BaseModel, Field  
from typing\_extensions import TypedDict  
from langgraph.graph import END, StateGraph, START  
from langgraph.graph.message import AnyMessage, add\_messages  
from IPython.display import Image, display  
from langchain\_core.output\_parsers import JsonOutputParser  
from typing import List, Optional, Dict  
from langchain\_core.prompts import PromptTemplate  
  
## Connection String for PostGres DB - please check the credentials from docker-compose file in part I  
# username, password, port, database  
connection\_str=f"postgresql://postgres:postgres@localhost:5432/my\_database"  
engine = create\_engine(connection\_str)  
  
# query all records in employees  
employees\_data = pd.read\_sql("SELECT \* FROM employees", engine)  
employees\_data.head()  
  
# use the folloing command with the database connectivity  
  
# Optional, add tracing in LangSmith  
os.environ["LANGSMITH\_TRACING"] = "true"  
os.environ["LANGSMITH\_PROJECT"] = "Text-To-SQL"  
  
llm = ChatOpenAI(model="gpt-4o")  
  
db = SQLDatabase.from\_uri(connection\_str)  
  
# check the connection:  
print(db.dialect, db.get\_usable\_table\_names())  
  
db.run("SELECT \* FROM employees LIMIT 5")  
  
# here are the output: - this provides the list of tables   
  
## postgresql ['departments', 'dept\_emp', 'dept\_manager', 'employees', 'salaries', 'titles']  
  
## "[(473302, 's0001', datetime.date(1953, 7, 25), 'Hideyuki', 'Zallocco', 'M', datetime.date(1990, 4, 28)), (475053, 'e0002', datetime.date(1954, 11, 18), 'Byong', 'Delgrande', 'F', datetime.date(1991, 9, 7)), (57444, 'e0002', datetime.date(1958, 1, 30), 'Berry', 'Babb', 'F', datetime.date(1992, 3, 21)), (421786, 's0001', datetime.date(1957, 9, 28), 'Xiong', 'Verhoeff', 'M', datetime.date(1987, 11, 26)), (282238, 'e0003', datetime.date(1952, 10, 28), 'Abdelkader', 'Baumann', 'F', datetime.date(1991, 1, 18))]"

To effectively track and analyze the agent’s output, we have configured credentials for LangSmith. This setup enables detailed monitoring, debugging, and evaluation of the agent’s performance, ensuring accurate query generation and execution.

We have now successfully connected to the database. The next step is to build the agent using LangGraph. To do this, we first define some utility functions and tools that facilitate various database and table-related operations.

**Define Fallback function to handel errors**

In LangGraph, a **fallback** mechanism is used to handle errors gracefully during agent execution. When a tool or node encounters an error, a fallback can be defined to manage the exception and provide a meaningful error message to the agent. This ensures that the agent can continue operating smoothly, even when unexpected issues arise. For instance, when creating a ToolNode, we can specify a fallback function to handle potential errors:

def handle\_tool\_error(state) -> dict:  
 error = state.get("error")  
 tool\_calls = state["messages"][-1].tool\_calls  
 return {  
 "messages": [  
 ToolMessage(  
 content=f"Error: {repr(error)}\n please fix your mistakes.",  
 tool\_call\_id=tc["id"],  
 )  
 for tc in tool\_calls  
 ]  
 }  
  
def create\_tool\_node\_with\_fallback(tools: list):  
 """  
 Create a ToolNode with a fallback to handle errors and surface them to the agent.  
 """  
 return ToolNode(tools).with\_fallbacks(  
 [RunnableLambda(handle\_tool\_error)], exception\_key="error"  
 )

In this example, the handle\_tool\_error function generates a ToolMessage containing the error details and prompts the agent to correct the mistake. The create\_tool\_node\_with\_fallback function then associates this error-handling mechanism with the ToolNode.

**Define Tools For Agent**

In developing an AI agent to interact with a SQL database, it’s essential to define specific tools that facilitate seamless communication between the agent and the database. The SQLDatabaseToolkit from the langchain\_community package offers a suite of such tools, enabling efficient database interactions.

# considering following tools  
toolkit = SQLDatabaseToolkit(db=db, llm=ChatOpenAI(model="gpt-4o"))  
tools = toolkit.get\_tools()  
# tool find the tables present in the database  
list\_tables\_tool = next(tool for tool in tools if tool.name == "sql\_db\_list\_tables")  
#tool finding the schema of the tables  
get\_schema\_tool = next(tool for tool in tools if tool.name == "sql\_db\_schema")

**List Tables Tool (sql\_db\_list\_tables):** Retrieves a list of available tables in the database.

**Get Schema Tool (sql\_db\_schema):** Fetches the Data Definition Language (DDL) statements for a specified table, providing its schema details.

Let’s create an additional customized tool for executing SQL queries.

## building a customized tool for sql query  
@tool  
def db\_query\_tool(query: str) -> str:  
 """  
 Execute a SQL query against the database and get back the result.  
 If the query is not correct, an error message will be returned.  
 If an error is returned, rewrite the query, check the query, and try again.  
 """  
 result = db.run\_no\_throw(query)  
 if not result:  
 return "Error: Query failed. Please rewrite your query and try again."  
 return result  
  
  
# building a query check chain with tools binds  
query\_check\_system = """You are a PostgreSQL expert with a strong attention to detail.  
Double check the PostgreSQL query for common mistakes, including:  
- Using NOT IN with NULL values  
- Using UNION when UNION ALL should have been used  
- Using BETWEEN for exclusive ranges  
- Data type mismatch in predicates  
- Properly quoting identifiers  
- Using the correct number of arguments for functions  
- Casting to the correct data type  
- Using the proper columns for joins  
  
If there are any of the above mistakes, rewrite the query. If there are no mistakes, just reproduce the original query.  
  
You will call the appropriate tool to execute the query after running this check."""  
  
query\_check\_prompt = ChatPromptTemplate.from\_messages(  
 [("system", query\_check\_system), ("placeholder", "{messages}")]  
)  
query\_check = query\_check\_prompt | ChatOpenAI(model="gpt-4o", temperature=0).bind\_tools(  
 [db\_query\_tool], tool\_choice="required"  
)  
  
  
# sql=query\_check.invoke({"messages": [("user", "SELECT \* FROM employees LIMIT 3;")]})  
# print(sql)

We have defined three tools: list\_tables\_tool, get\_schema\_tool, and db\_query\_tool.

**Building a Graph**

Now, let’s define an agent using LangGraph. In LangGraph, establishing the agent’s state and constructing a state graph are fundamental steps in creating workflows for AI agents.

# Define the state for the agent  
class State(TypedDict):  
 messages: Annotated[list[AnyMessage], add\_messages]  
  
# Define a new graph  
workflow = StateGraph(State)  
  
# Add a node for the first tool call  
def first\_tool\_call(state: State) -> dict[str, list[AIMessage]]:  
 # print("from first\_tool\_call")  
 # print(state)  
 return {  
 "messages": [  
 AIMessage(  
 content="",  
 tool\_calls=[  
 {  
 "name": "sql\_db\_list\_tables",  
 "args": {},  
 "id": "tool\_abcd123",  
 }  
 ],  
 )  
 ]  
 }  
  
def model\_check\_query(state: State):  
 """  
 Use this tool to double-check if your query is correct before executing it.  
 """  
 return {"messages": [query\_check.invoke({"messages": [state["messages"][-1]]})]}

**Adding Nodes to the Graph**

To further refine the workflow, we will introduce several new nodes, each responsible for specific tasks in processing the user query and interacting with the database.

**First Tool Call Node (first\_tool\_call):**This node serves as an initial tool execution step. Regardless of the user’s input, it will always call the predefined sql\_db\_list\_tables tool to retrieve a list of available tables in the database.

**List Tables Execution Node (list\_tables\_tool):**This node executes the sql\_db\_list\_tables tool and retrieves the complete list of tables currently present in the database.

**Schema Selection Node (model\_generate\_schema):**This node processes the retrieved list of tables and selects only the relevant ones that align with the user’s query, ensuring that unnecessary tables are filtered out.

**Schema Retrieval Node (get\_schema\_tool):**Once the relevant tables are identified, this node fetches their schema details, providing essential metadata such as column names and data types for the selected tables.

# Add first node  
workflow.add\_node("first\_tool\_call", first\_tool\_call)  
  
# Add second node  
workflow.add\_node(  
 "list\_tables\_tool", create\_tool\_node\_with\_fallback([list\_tables\_tool])  
)  
  
# Add a node for a model to choose the relevant tables based on the question and available tables  
model\_get\_schema = ChatOpenAI(model="gpt-4o", temperature=0).bind\_tools(  
 [get\_schema\_tool]  
)  
# Add third node  
workflow.add\_node(  
 "model\_get\_schema",  
 lambda state: {  
 "messages": [model\_get\_schema.invoke(state["messages"])],  
 },  
)  
# Add fourth node  
workflow.add\_node("get\_schema\_tool", create\_tool\_node\_with\_fallback([get\_schema\_tool]))

Here are the outputs produced by these nodes:

user input = "find the average employee salary by title"  
  
## first node: table list tool  
output from node1 = sql\_db\_list\_tables tool\_abcd123  
  
## second nodes - all tables  
output from node2 = departments, dept\_emp, dept\_manager, employees, salaries, titles  
  
## third node - find only the relevant tables  
output from node3 = table\_names: employees, salaries, titles  
  
## fourth node - find schema for those relevant tables  
output from node4 =   
  
CREATE TABLE employees (  
 emp\_no INTEGER NOT NULL,   
 emp\_title\_id VARCHAR NOT NULL,   
 birth\_date DATE NOT NULL,   
 first\_name VARCHAR NOT NULL,   
 last\_name VARCHAR NOT NULL,   
 sex VARCHAR NOT NULL,   
 hire\_date DATE NOT NULL,   
 CONSTRAINT employees\_pkey PRIMARY KEY (emp\_no),   
 CONSTRAINT employees\_emp\_title\_id\_fkey FOREIGN KEY(emp\_title\_id) REFERENCES titles (title\_id)  
)  
  
/\*  
3 rows from employees table:  
emp\_no emp\_title\_id birth\_date first\_name last\_name sex hire\_date  
473302 s0001 1953-07-25 Hideyuki Zallocco M 1990-04-28  
475053 e0002 1954-11-18 Byong Delgrande F 1991-09-07  
57444 e0002 1958-01-30 Berry Babb F 1992-03-21  
\*/  
  
  
CREATE TABLE salaries (  
 emp\_no INTEGER NOT NULL,   
 salary INTEGER NOT NULL,   
 CONSTRAINT salaries\_pkey PRIMARY KEY (emp\_no),   
 CONSTRAINT salaries\_emp\_no\_fkey FOREIGN KEY(emp\_no) REFERENCES employees (emp\_no)  
)  
  
/\*  
3 rows from salaries table:  
emp\_no salary  
10001 60117  
10002 65828  
10003 40006  
\*/  
  
  
CREATE TABLE titles (  
 title\_id VARCHAR NOT NULL,   
 title VARCHAR NOT NULL,   
 CONSTRAINT titles\_pkey PRIMARY KEY (title\_id)  
)  
  
/\*  
3 rows from titles table:  
title\_id title  
s0001 Staff  
s0002 Senior Staff  
e0001 Assistant Engineer  
\*/

**Expanding the Graph with Four Additional Nodes**

To complete the workflow, we are adding four new nodes to the graph, each playing a crucial role in generating, validating, executing, and displaying the SQL query results.

**Query Generation Node (query\_gen):**This node collects all messages from the previous nodes and generates an SQL query based on the provided inputs.

**Query Correction Node (correct\_query):**This node reviews the generated SQL query to ensure its accuracy. If any inconsistencies or errors are detected in the previous step, this node corrects the SQL query before execution.

**Query Execution Node (execute\_query):**This node takes the validated SQL query from the previous step and executes it against the database, retrieving the corresponding results.

**Output Generation Node (output\_gen):**This final node compiles both the executed SQL query and its output, ensuring that the user receives both the generated SQL statement and the query results in a structured response.

By implementing these additional nodes, we create a streamlined process that ensures the SQL query is properly generated, validated, executed, and returned with its results, improving accuracy and transparency for the user.

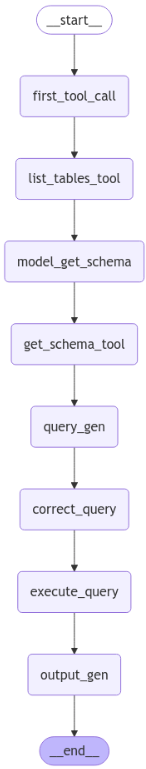
# Describe a tool to represent the end state  
class SubmitFinalAnswer(BaseModel):  
 """Submit the final answer to the user based on the query results."""  
  
 final\_answer: str = Field(..., description="The final answer to the user")  
  
  
# Add a node for a model to generate a query based on the question and schema  
query\_gen\_system = """You are a SQL expert with a strong attention to detail.  
  
Given an input question, output a syntactically correct Postgres query to run, then look at the results of the query and return the answer.  
  
DO NOT call any tool besides SubmitFinalAnswer to submit the final answer.  
  
When generating the query:  
  
Output the SQL query that answers the input question without a tool call.  
  
Unless the user specifies a specific number of examples they wish to obtain, always limit your query to at most 5 results.  
You can order the results by a relevant column to return the most interesting examples in the database.  
Never query for all the columns from a specific table, only ask for the relevant columns given the question.  
  
If you get an error while executing a query, rewrite the query and try again.  
  
If you get an empty result set, you should try to rewrite the query to get a non-empty result set.   
NEVER make stuff up if you don't have enough information to answer the query... just say you don't have enough information.  
  
If you have enough information to answer the input question, simply invoke the appropriate tool to submit the final answer to the user.  
  
DO NOT make any DML statements (INSERT, UPDATE, DELETE, DROP etc.) to the database."""  
query\_gen\_prompt = ChatPromptTemplate.from\_messages(  
 [("system", query\_gen\_system), ("placeholder", "{messages}")]  
)  
  
  
query\_gen = query\_gen\_prompt | ChatOpenAI(model="gpt-4o", temperature=0)  
  
  
def query\_gen\_node(state: State):  
 message = query\_gen.invoke(state)  
 return {"messages": [message]}  
  
  
### Node for final output - we want both   
class OutputFormat(BaseModel):  
 """generate the output in this format:"""  
 sql\_query: Optional[str] = Field(..., description="final sql query to user ")  
 final\_user\_message: Optional[str] = Field(..., description="final message to user")  
  
output\_gen\_system = """You are a SQL expert with a strong attention to detail. Analyze the all provided input messages   
and generate the final output.  
"""  
output\_gen\_prompt = ChatPromptTemplate.from\_messages(  
 [("system", output\_gen\_system), ("placeholder", "{messages}")]  
)  
# query\_gen = query\_gen\_prompt | ChatOpenAI(model="gpt-4o", temperature=0).bind\_tools(  
# [SubmitFinalAnswer]  
# )  
output\_gen = output\_gen\_prompt | ChatOpenAI(model="gpt-4o", temperature=0).bind\_tools([OutputFormat])  
  
def output\_gen\_node(state: State):  
 message = output\_gen.invoke(state)  
 return {"messages": [message]}  
  
### Fifth node  
workflow.add\_node("query\_gen", query\_gen\_node)  
  
# Add a node for the model to check the query before executing it  
workflow.add\_node("correct\_query", model\_check\_query)  
  
# Add node for executing the query  
workflow.add\_node("execute\_query", create\_tool\_node\_with\_fallback([db\_query\_tool]))  
  
# Final Node:  
workflow.add\_node("output\_gen", output\_gen\_node)

Next, we will link the nodes by defining edges within our graph. For this agent, we will not incorporate any conditional edges. The following are the edges we are setting up:

# Specify the edges between the nodes  
workflow.add\_edge(START, "first\_tool\_call")  
workflow.add\_edge("first\_tool\_call", "list\_tables\_tool")  
workflow.add\_edge("list\_tables\_tool", "model\_get\_schema")  
workflow.add\_edge("model\_get\_schema", "get\_schema\_tool")  
workflow.add\_edge("get\_schema\_tool", "query\_gen")  
# workflow.add\_conditional\_edges(  
# "query\_gen",  
# should\_continue,  
# )  
workflow.add\_edge("query\_gen", "correct\_query")  
workflow.add\_edge("correct\_query", "execute\_query")  
workflow.add\_edge("execute\_query", 'output\_gen')  
workflow.add\_edge("output\_gen", END)  
  
# Compile the workflow into a runnable  
app = workflow.compile()

Let’s generate a visual representation of the defined graph.

display(Image(app.get\_graph(xray=True).draw\_mermaid\_png()))



Now, let’s run the graph with the user query.

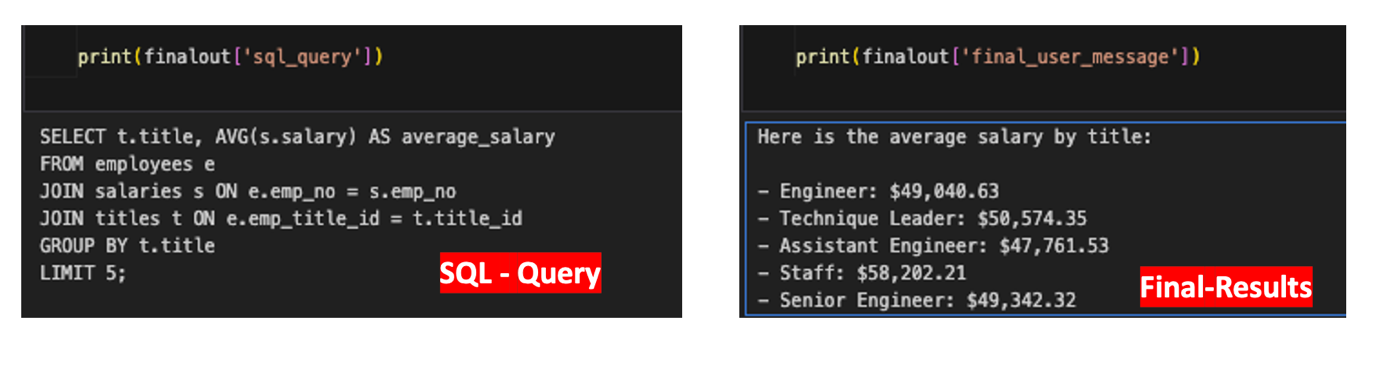
query="find the total number of employees"  
# query="how many distinct employees are there?"  
query="find the average employee salary by title"  
  
messages = app.invoke(  
 {"messages": [("user", query)]}  
)  
  
print(messages)

Here is the final state after the graph has been fully executed.

{'messages': [HumanMessage(content='find the average employee salary by title', additional\_kwargs={}, response\_metadata={}, id='b232b73e-f3eb-4bec-8229-398b9d6ee4f0'),  
 AIMessage(content='', additional\_kwargs={}, response\_metadata={}, id='b4cdcc52-cc5e-4e71-9409-b75941cf1410', tool\_calls=[{'name': 'sql\_db\_list\_tables', 'args': {}, 'id': 'tool\_abcd123', 'type': 'tool\_call'}]),  
 ToolMessage(content='departments, dept\_emp, dept\_manager, employees, salaries, titles', name='sql\_db\_list\_tables', id='fc508eff-1b07-4dfe-85bb-10c78cf26b06', tool\_call\_id='tool\_abcd123'),  
 AIMessage(content='', additional\_kwargs={'tool\_calls': [{'id': 'call\_jFBKxmAYIIr54QLDJ5VZNudx', 'function': {'arguments': '{"table\_names":"employees, salaries, titles"}', 'name': 'sql\_db\_schema'}, 'type': 'function'}], 'refusal': None}, response\_metadata={'token\_usage': {'completion\_tokens': 21, 'prompt\_tokens': 161, 'total\_tokens': 182, 'completion\_tokens\_details': {'accepted\_prediction\_tokens': 0, 'audio\_tokens': 0, 'reasoning\_tokens': 0, 'rejected\_prediction\_tokens': 0}, 'prompt\_tokens\_details': {'audio\_tokens': 0, 'cached\_tokens': 0}}, 'model\_name': 'gpt-4o-2024-08-06', 'system\_fingerprint': 'fp\_50cad350e4', 'finish\_reason': 'tool\_calls', 'logprobs': None}, id='run-2a839125-1343-47ec-be46-d0dfb61a07f2-0', tool\_calls=[{'name': 'sql\_db\_schema', 'args': {'table\_names': 'employees, salaries, titles'}, 'id': 'call\_jFBKxmAYIIr54QLDJ5VZNudx', 'type': 'tool\_call'}], usage\_metadata={'input\_tokens': 161, 'output\_tokens': 21, 'total\_tokens': 182, 'input\_token\_details': {'audio': 0, 'cache\_read': 0}, 'output\_token\_details': {'audio': 0, 'reasoning': 0}}),  
 ToolMessage(content='\nCREATE TABLE employees (\n\temp\_no INTEGER NOT NULL, \n\temp\_title\_id VARCHAR NOT NULL, \n\tbirth\_date DATE NOT NULL, \n\tfirst\_name VARCHAR NOT NULL, \n\tlast\_name VARCHAR NOT NULL, \n\tsex VARCHAR NOT NULL, \n\thire\_date DATE NOT NULL, \n\tCONSTRAINT employees\_pkey PRIMARY KEY (emp\_no), \n\tCONSTRAINT employees\_emp\_title\_id\_fkey FOREIGN KEY(emp\_title\_id) REFERENCES titles (title\_id)\n)\n\n/\*\n3 rows from employees table:\nemp\_no\temp\_title\_id\tbirth\_date\tfirst\_name\tlast\_name\tsex\thire\_date\n473302\ts0001\t1953-07-25\tHideyuki\tZallocco\tM\t1990-04-28\n475053\te0002\t1954-11-18\tByong\tDelgrande\tF\t1991-09-07\n57444\te0002\t1958-01-30\tBerry\tBabb\tF\t1992-03-21\n\*/\n\n\nCREATE TABLE salaries (\n\temp\_no INTEGER NOT NULL, \n\tsalary INTEGER NOT NULL, \n\tCONSTRAINT salaries\_pkey PRIMARY KEY (emp\_no), \n\tCONSTRAINT salaries\_emp\_no\_fkey FOREIGN KEY(emp\_no) REFERENCES employees (emp\_no)\n)\n\n/\*\n3 rows from salaries table:\nemp\_no\tsalary\n10001\t60117\n10002\t65828\n10003\t40006\n\*/\n\n\nCREATE TABLE titles (\n\ttitle\_id VARCHAR NOT NULL, \n\ttitle VARCHAR NOT NULL, \n\tCONSTRAINT titles\_pkey PRIMARY KEY (title\_id)\n)\n\n/\*\n3 rows from titles table:\ntitle\_id\ttitle\ns0001\tStaff\ns0002\tSenior Staff\ne0001\tAssistant Engineer\n\*/', name='sql\_db\_schema', id='c5451fb4-66ce-49d7-878f-7beb33f33d19', tool\_call\_id='call\_jFBKxmAYIIr54QLDJ5VZNudx'),  
 AIMessage(content='```sql\nSELECT t.title, AVG(s.salary) AS average\_salary\nFROM employees e\nJOIN salaries s ON e.emp\_no = s.emp\_no\nJOIN titles t ON e.emp\_title\_id = t.title\_id\nGROUP BY t.title\nLIMIT 5;\n```', additional\_kwargs={'refusal': None}, response\_metadata={'token\_usage': {'completion\_tokens': 55, 'prompt\_tokens': 683, 'total\_tokens': 738, 'completion\_tokens\_details': {'accepted\_prediction\_tokens': 0, 'audio\_tokens': 0, 'reasoning\_tokens': 0, 'rejected\_prediction\_tokens': 0}, 'prompt\_tokens\_details': {'audio\_tokens': 0, 'cached\_tokens': 0}}, 'model\_name': 'gpt-4o-2024-08-06', 'system\_fingerprint': 'fp\_50cad350e4', 'finish\_reason': 'stop', 'logprobs': None}, id='run-c21002f8-c6cd-4789-b160-848073b2ad49-0', usage\_metadata={'input\_tokens': 683, 'output\_tokens': 55, 'total\_tokens': 738, 'input\_token\_details': {'audio': 0, 'cache\_read': 0}, 'output\_token\_details': {'audio': 0, 'reasoning': 0}}),  
 AIMessage(content='', additional\_kwargs={'tool\_calls': [{'id': 'call\_JZapUFwccgxHpoNA9ukTPM1f', 'function': {'arguments': '{"query":"SELECT t.title, AVG(s.salary) AS average\_salary\\nFROM employees e\\nJOIN salaries s ON e.emp\_no = s.emp\_no\\nJOIN titles t ON e.emp\_title\_id = t.title\_id\\nGROUP BY t.title\\nLIMIT 5;"}', 'name': 'db\_query\_tool'}, 'type': 'function'}], 'refusal': None}, response\_metadata={'token\_usage': {'completion\_tokens': 65, 'prompt\_tokens': 267, 'total\_tokens': 332, 'completion\_tokens\_details': {'accepted\_prediction\_tokens': 0, 'audio\_tokens': 0, 'reasoning\_tokens': 0, 'rejected\_prediction\_tokens': 0}, 'prompt\_tokens\_details': {'audio\_tokens': 0, 'cached\_tokens': 0}}, 'model\_name': 'gpt-4o-2024-08-06', 'system\_fingerprint': 'fp\_4691090a87', 'finish\_reason': 'tool\_calls', 'logprobs': None}, id='run-dfa9051c-09ff-492f-9a60-49244b06cc37-0', tool\_calls=[{'name': 'db\_query\_tool', 'args': {'query': 'SELECT t.title, AVG(s.salary) AS average\_salary\nFROM employees e\nJOIN salaries s ON e.emp\_no = s.emp\_no\nJOIN titles t ON e.emp\_title\_id = t.title\_id\nGROUP BY t.title\nLIMIT 5;'}, 'id': 'call\_JZapUFwccgxHpoNA9ukTPM1f', 'type': 'tool\_call'}], usage\_metadata={'input\_tokens': 267, 'output\_tokens': 65, 'total\_tokens': 332, 'input\_token\_details': {'audio': 0, 'cache\_read': 0}, 'output\_token\_details': {'audio': 0, 'reasoning': 0}}),  
 ToolMessage(content="[('Engineer', Decimal('49040.630872483221')), ('Technique Leader', Decimal('50574.349206349206')), ('Assistant Engineer', Decimal('47761.526315789474')), ('Staff', Decimal('58202.214285714286')), ('Senior Engineer', Decimal('49342.320000000000'))]", name='db\_query\_tool', id='ffca8ec9-f051-4fd5-b1a7-b54f4cfd2eb0', tool\_call\_id='call\_JZapUFwccgxHpoNA9ukTPM1f'),  
 AIMessage(content='', additional\_kwargs={'tool\_calls': [{'id': 'call\_yEU4fFXdTzO3tl4cAdzTaS4K', 'function': {'arguments': '{"sql\_query":"SELECT t.title, AVG(s.salary) AS average\_salary\\nFROM employees e\\nJOIN salaries s ON e.emp\_no = s.emp\_no\\nJOIN titles t ON e.emp\_title\_id = t.title\_id\\nGROUP BY t.title\\nLIMIT 5;","final\_user\_message":"Here is the average salary by title:\\n\\n- Engineer: $49,040.63\\n- Technique Leader: $50,574.35\\n- Assistant Engineer: $47,761.53\\n- Staff: $58,202.21\\n- Senior Engineer: $49,342.32"}', 'name': 'OutputFormat'}, 'type': 'function'}], 'refusal': None}, response\_metadata={'token\_usage': {'completion\_tokens': 132, 'prompt\_tokens': 713, 'total\_tokens': 845, 'completion\_tokens\_details': {'accepted\_prediction\_tokens': 0, 'audio\_tokens': 0, 'reasoning\_tokens': 0, 'rejected\_prediction\_tokens': 0}, 'prompt\_tokens\_details': {'audio\_tokens': 0, 'cached\_tokens': 0}}, 'model\_name': 'gpt-4o-2024-08-06', 'system\_fingerprint': 'fp\_4691090a87', 'finish\_reason': 'tool\_calls', 'logprobs': None}, id='run-4b86b552-f6d2-41d5-b6f9-9a6c148a1b53-0', tool\_calls=[{'name': 'OutputFormat', 'args': {'sql\_query': 'SELECT t.title, AVG(s.salary) AS average\_salary\nFROM employees e\nJOIN salaries s ON e.emp\_no = s.emp\_no\nJOIN titles t ON e.emp\_title\_id = t.title\_id\nGROUP BY t.title\nLIMIT 5;', 'final\_user\_message': 'Here is the average salary by title:\n\n- Engineer: $49,040.63\n- Technique Leader: $50,574.35\n- Assistant Engineer: $47,761.53\n- Staff: $58,202.21\n- Senior Engineer: $49,342.32'}, 'id': 'call\_yEU4fFXdTzO3tl4cAdzTaS4K', 'type': 'tool\_call'}], usage\_metadata={'input\_tokens': 713, 'output\_tokens': 132, 'total\_tokens': 845, 'input\_token\_details': {'audio': 0, 'cache\_read': 0}, 'output\_token\_details': {'audio': 0, 'reasoning': 0}})]}

Here is the final output generated by agent.

finalout=json.loads(messages['messages'][-1].additional\_kwargs['tool\_calls'][0]['function']['arguments'])



In this article, we built a LangGraph-based AI agent that translates natural language queries into SQL, executes them, and returns results. We structured this process into two key phases: setting up a PostgreSQL database (covered in Part I) and developing the text-to-SQL agent (covered in Part II).

By leveraging LangGraph, we created a structured workflow consisting of multiple nodes responsible for retrieving database metadata, generating SQL queries, validating them, executing the queries, and returning the final output. Through systematic integration of tools and error-handling mechanisms, the agent efficiently processes queries while ensuring accuracy and adaptability.

This framework demonstrates the potential of AI-powered database interaction, enabling users to query complex databases without SQL expertise. With further refinements, such a system can be extended to support more advanced query generation, enhanced error handling, and integration with diverse data sources.