

# Build an SQL agent with LangGraph and Mistral Medium 3 by using watsonx.ai

In this tutorial, you will build an [AI agent](#) that can execute and generate Python and [SQL](#) queries for your custom SQLite database. Your agent will be built from scratch by using [LangGraph](#) and the Mistral Medium 3 [large language model \(LLM\)](#). This LLM is optimized for professional use cases such as coding and multimodal understanding, thus making it an excellent choice for this task.<sup>1</sup> We will also detect and block harmful messages by using the IBM® Granite® Guardian model.

## What is LangGraph?

[LangGraph](#), created by [LangChain](#), is an [open source AI agent framework](#) designed to build, deploy and manage complex generative [AI agent workflows](#). It provides a set of [tools](#) and libraries that enable users to create, run and optimize LLMs in a scalable and efficient manner. At its core, LangGraph uses the power of graph-based architectures to model and manage the intricate relationships between various [components](#) of an AI agent workflow, including those components found in multi-agent systems.

## Prerequisites

1. You need an [IBM Cloud® account](#) to create a watsonx.ai® project.
2. Several Python versions can work for this tutorial. At the time of publishing, we recommend [downloading](#) Python 3.13, the latest version.

## Steps

### Step 1. Set up your environment

While you can choose from several tools, this tutorial walks you through how to set up an IBM account to use a Jupyter Notebook.

1. Log in to [watsonx.ai](#) by using your IBM Cloud® account.
2. Create a [watsonx.ai project](#).

You can get your project ID from within your project. Click the **Manage** tab. Then, copy the project ID from the **Details** section of the **General** page. You need this ID for this tutorial.

3. Create a [Jupyter Notebook](#).

This step opens a Jupyter Notebook environment where you can copy the code from this tutorial. Alternatively, you can download this notebook to your local system and upload it to your watsonx.ai project as an asset. This tutorial is also available on [GitHub](#).

### Step 2. Set up a watsonx.ai Runtime instance and API key

1. Create a [watsonx.ai Runtime](#) service instance (select your appropriate region and choose the Lite plan, which is a free instance).
2. Generate an application programming interface ([API](#)) key.
3. Associate the watsonx.ai Runtime service instance with the project that you created in [watsonx.ai](#).

### Step 3. Install and import relevant libraries and set up your credentials

We need a few libraries and modules for this tutorial. Make sure to import the following ones and if they're not installed, a quick pip installation resolves the problem.

```
#installations
%pip install -qU ibm-watsonx-ai \
    langchain-ibm \
    langgraph \
    langchain_experimental \
    tavily-python \
    langchain-community
```

Restart your terminal and import the following packages.

```
# imports
import sqlite3
import getpass

from ibm_watsonx_ai import APIClient, Credentials
from ibm_watsonx_ai.foundation_models.moderations import Guardian
from IPython.display import Image, display
from langchain import hub
from langchain_ibm import ChatWatsonx
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver
from langchain_community.utilities.sql_database import SQLiteDatabase
```

```

from langchain_community.agent_toolkits.sql.toolkit import SQLDatabaseToolkit
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, ToolMessage, AIMessage
from langchain_core.tools import tool
from langchain_experimental.tools.python.tool import PythonREPLTool
from sqlalchemy import create_engine
from typing_extensions import TypedDict
from typing import Annotated

```

To set our credentials, we will need the WATSONX\_APIKEY and WATSONX\_PROJECT\_ID that you generated in Step 1. We will also set the URL to serve as the API endpoint.

```
WATSONX_APIKEY = getpass.getpass("Please enter your watsonx.ai Runtime API key (hit enter): ")
```

```
WATSONX_PROJECT_ID = getpass.getpass("Please enter your project ID (hit enter): ")
```

```
URL = "https://us-south.ml.cloud.ibm.com"
```

Before we can initialize our LLM, we can use the Credentials class to encapsulate our passed API credentials.

```
credentials = Credentials(url=URL, api_key=WATSONX_APIKEY)
```

## Step 4. Instantiate the chat model

To be able to interact with all resources available in watsonx.ai Runtime, you need to set up an APIClient. Here, we pass in our credentials and WATSONX\_PROJECT\_ID.

```
client = APIClient(credentials=credentials, project_id=WATSONX_PROJECT_ID)
```

For this tutorial, we will be using the ChatWatsonx wrapper to set up our chat model. This wrapper simplifies the integration of tool calling and chaining. We encourage you to use the API references in the ChatWatsonx [official docs](#) for further information. We can pass in our model\_id for Mistral Medium 3 and our client as parameters.

Note, if you use a different API provider, you will need to change the wrapper accordingly. For example, to use the OpenAI API to access models such as GPT-4, you will need an openai\_api\_key as well as the ChatOpenAI wrapper.

```
model_id = "mistralai/mistral-medium-2505"
llm = ChatWatsonx(model_id=model_id, watsonx_client=client)
```

## Step 5. Set up the SQLite database

In this tutorial, your agent will use its available tools to interact with an SQLite database. If you already have a dataset stored in a relational database management system such as PostgreSQL or SQLite, you can skip this step. Otherwise, run the following cell to set the sql\_script variable equal to a script that generates a database containing synthetic car dealership sales data. This database will contain several tables for storing individual dealership, sales and vehicle data.

```

sql_script = """
DROP TABLE IF EXISTS [Dealerships];

DROP TABLE IF EXISTS [Cars];

DROP TABLE IF EXISTS [Sales];

CREATE TABLE [Dealerships] (
    [id] INTEGER PRIMARY KEY,
    [name] TEXT NOT NULL,
    [location] TEXT NOT NULL
);

CREATE TABLE [Cars] (
    [id] INTEGER PRIMARY KEY,
    [make] TEXT NOT NULL,
    [model] TEXT NOT NULL,
    [year] INTEGER NOT NULL,
    [color] TEXT NOT NULL
);

CREATE TABLE [Sales] (
    [id] INTEGER PRIMARY KEY,
    [car_id] INTEGER NOT NULL,
    [dealership_id] INTEGER NOT NULL,
    [sale_date] DATE NOT NULL,
    [sale_price] REAL NOT NULL,
    FOREIGN KEY ([car_id]) REFERENCES [Cars] ([id]),
    FOREIGN KEY ([dealership_id]) REFERENCES [Dealerships] ([id])
);

INSERT INTO [Dealerships] ([id], [name], [location]) VALUES
(1, 'Dealership A', 'New York'),
(2, 'Dealership B', 'Los Angeles'),
(3, 'Dealership C', 'Chicago'),
(4, 'Dealership D', 'Houston'),
(5, 'Dealership E', 'Phoenix'),
(6, 'Dealership F', 'Philadelphia'),
(7, 'Dealership G', 'San Antonio'),
(8, 'Dealership H', 'San Diego'),
(9, 'Dealership I', 'Dallas'),
(10, 'Dealership J', 'San Jose');

INSERT INTO [Cars] ([id], [make], [model], [year], [color]) VALUES
(1, 'Toyota', 'Camry', 2020, 'Blue'),
(2, 'Honda', 'Civic', 2019, 'Red'),
(3, 'Ford', 'Mustang', 2021, 'Black'),
(4, 'Chevrolet', 'Silverado', 2018, 'White'),
(5, 'Nissan', 'Altima', 2020, 'Gray'),

```

```

(6, 'Kia', 'Optima', 2020, 'Silver'),
(7, 'Hyundai', 'Elantra', 2019, 'Black'),
(8, 'Volkswagen', 'Golf', 2021, 'Red'),
(9, 'BMW', '3 Series', 2018, 'White'),
(10, 'Mercedes-Benz', 'C-Class', 2020, 'Gray'),
(11, 'Audi', 'A4', 2019, 'Blue'),
(12, 'Lexus', 'ES', 2021, 'Black'),
(13, 'Toyota', 'Corolla', 2018, 'White'),
(14, 'Honda', 'Accord', 2020, 'Gray'),
(15, 'Ford', 'Fusion', 2019, 'Red');

INSERT INTO [Sales] ([id], [car_id], [dealership_id], [sale_date], [sale_price]) VALUES
(1, 1, 1, '2022-01-01', 25000.0),
(2, 2, 2, '2022-02-01', 20000.0),
(3, 3, 3, '2022-03-01', 30000.0),
(4, 4, 1, '2022-04-01', 40000.0),
(5, 5, 2, '2022-05-01', 28000.0),
(6, 6, 4, '2022-06-01', 22000.0),
(7, 7, 5, '2022-07-01', 20000.0),
(8, 8, 6, '2022-08-01', 28000.0),
(9, 9, 7, '2022-09-01', 35000.0),
(10, 10, 8, '2022-10-01', 32000.0),
(11, 11, 9, '2022-11-01', 30000.0),
(12, 12, 10, '2022-12-01', 38000.0),
(13, 13, 1, '2023-01-01', 25000.0),
(14, 14, 2, '2023-02-01', 28000.0),
(15, 15, 3, '2023-03-01', 22000.0);

"""

```

In order to run this script and create the local SQLite database, we need to set up a database Connection object. By passing the ":memory:" path name, we can connect to a transient in-memory database.

```
connection = sqlite3.connect(":memory:")
```

Next, we can use the `executescript` method to create a database cursor and to run our SQL script. This Cursor object allows us to run such scripts.

```
connection.executescript(sql_script)
```

#### Output:

```
<sqlite3.Cursor at 0x30c511240>
```

Finally, run the following command to create a SQLAlchemy Engine instance with a connection to the database file in your working directory. The file should have the same name as your Jupyter Notebook with a database file extension. We can keep the URL empty so that the connection is made to the SQLite local database. We can provide our Connection object in the `creator` parameter. This parameter is responsible for creating the connection to the database.

```
engine = create_engine(
    "sqlite://",
    creator=lambda: connection
)
```

If you prefer to use an alternative database for this tutorial that is not stored locally, simply replace the empty path with `"sqlite:///Chinook.db"` to access a sample digital media store database.

## Step 6. Set up the tools

This tutorial uses the LangChain pre-built `SQLDatabaseToolkit`. The toolkit requires a `SQLDatabase` object, which takes the SQLAlchemy Engine as a parameter, and the LLM of choice. Once we instantiate the toolkit, we can then retrieve its tools. Let's print the tools to see more details.

```
db = SQLDatabase(engine)
toolkit = SQLDatabaseToolkit(db=db, llm=llm)
tools = toolkit.get_tools()
tools
```

#### Output:

```
[QuerySQLDatabaseTool(description="Input to this tool is a detailed and correct SQL query, output is a result from the database. If the q
InfoSQLDatabaseTool(description='Input to this tool is a comma-separated list of tables, output is the schema and sample rows for those
ListSQLDatabaseTool(db=<langchain_community.utilities.sql_database.SQLDatabase object at 0x30c3ecd10>),
QuerySQLCheckerTool(description='Use this tool to double check if your query is correct before executing it. Always use this tool before
```

We can see there are 4 tools available as part of the toolkit. Each tool serves its own purpose, as seen by the tool descriptions. The tools are built to list the databases, run queries, return table schemas and verify SQL queries before running them.

To equip our agent with the ability to generate and execute Python code, we can leverage LangChain's `PythonREPLTool` class. This code sets up a Python REPL (Read-Eval-Print Loop) tool, defining its functionality and appending it to a list of tools for later use.

```
python_repl = PythonREPLTool()
tools.append(python_repl)
```

To help ensure that our agent is provided with a system prompt that is tailored toward SQL tools and databases, we can pull a prompt provided by the [LangChain Hub](#). Let's print the `system_message` to see the prompt in detail.

```
chatprompttemplate = hub.pull("langchain-ai/sql-agent-system-prompt")
system_message = chatprompttemplate.format(dialect="SQLite", top_k=5)
print(system_message)
```

#### Output:

```
System: You are an agent designed to interact with a SQL database.
Given an input question, create a syntactically correct SQLite query to run, then look at the results of the query and return the answer.
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.
You have access to tools for interacting with the database.
Only use the below tools. Only use the information returned by the below tools to construct your final answer.
You MUST double check your query before executing it. If you get an error while executing a query, rewrite the query and try again.
```

DO NOT make any DML statements (INSERT, UPDATE, DELETE, DROP etc.) to the database.

To start you should ALWAYS look at the tables in the database to see what you can query.  
Do NOT skip this step.  
Then you should query the schema of the most relevant tables.

## Step 7. Define the agent state

There is a prebuilt `create_react_agent` function available in LangGraph that generates an agent graph that follows the [ReAct \(reasoning and action\) architecture](#). This architecture allows the agent to iteratively [call tools](#) in a loop until a stopping condition is met.

To provide a more hands-on approach, we will build a ReAct agent from scratch in this tutorial. As a first step, we can create an `AgentState` class to store the context of the messages from the user, tools and the agent itself. Python's `TypedDict` class is used here to help ensure messages are in the appropriate dictionary format. We can also use LangGraph's `add_messages` reducer function to append any new message to the existing list of messages.

```
class AgentState(TypedDict):
    messages: Annotated[List[AnyMessage], add_messages]
```

## Step 8. Define the ReActAgent class

Next, we can create the `ReActAgent` class. This class provides a basic structure for creating agents that can react to changes in their surroundings. The functions in the `ReActAgent` class allow for iterative tool calling in response to the graph state.

The `__init__` function initializes the attributes of the class with the large language model, tools and system message as parameters. This constructor creates a state graph with nodes for the guardian model, LLM and tools. The graph starts at the guardian node, which calls the `guardian_moderation` method to detect harmful content before it reaches the LLM and the database. The conditional edge between the guardian and `llm` nodes routes the state of the graph to either the `llm` node or the end. This is determined by the output of the `guardian_moderation` function. Safe messages are passed to the `llm` node, which executes the `call_llm` method. We also add a conditional edge between the `llm` and `tools` nodes to route messages appropriately. If the LLM returns a tool call, the `should_call_tools` method returns a `True` boolean. Otherwise, `False` is returned and the graph routes to the end. This step is part of the ReAct agent architecture—we want the agent to receive the tool output and then react to the change in state to determine its next action.

Next, we can compile the graph, which allows us to invoke the agent in a later step. To persist messages, we can use the `MemorySaver` checkpoint. The final two lines of this method initialize the `tools` and `llm` attributes of the class instance. The `tools` attribute is a dictionary that maps tool names to tool objects. The `llm` attribute is the LLM, which is bound to tools by using the `bind_tools` method.

```
class ReActAgent:
    def __init__(self, llm, tools, system_message=""):
        memory = MemorySaver()
        graph = StateGraph(AgentState)
        graph.add_node("guardian", self.guardian_moderation)
        graph.add_node("llm", self.call_llm)
        graph.add_node("tools", self.call_tools)
        graph.add_node("block_message", self.block_message)
        graph.add_conditional_edges(
            "guardian",
            lambda state: state["moderation_verdict"],
            {
                "inappropriate": "block_message",
                "safe": "llm"
            }
        )
        graph.add_edge("block_message", END)
        graph.add_conditional_edges(
            "llm",
            self.should_call_tools,
            {"tools": END}
        )
        graph.add_edge("tools", "llm")
        graph.add_edge(START, "guardian")
        self.system_message = system_message
        self.graph = graph.compile(checkpointer=memory)
        self.tools = {t.name: t for t in tools}
        self.llm = llm.bind_tools(tools)
```

The next function in the `ReActAgent` class is `call_llm`. This function calls the LLM by retrieving the messages from the state. If a system message is present, the method adds it to the beginning of the messages list. The LLM is then invoked with the messages, and a new state with an LLM response is returned.

```
def call_llm(self, state: AgentState):
    messages = state['messages']
    if self.system_message:
        messages = [SystemMessage(content=self.system_message)] + messages
    message = self.llm.invoke(messages)
    return {'messages': [message]}
```

The `call_tools` function is next in the `ReActAgent` class. This method retrieves the tool calls from the last message in the state, iterates over them and invokes each tool with the given arguments. Next, the results of each tool call are stored in a list called `results`. Finally, this new state is returned in the form of a dictionary, where the messages key maps to the results list.

```
def call_tools(self, state: AgentState):
    tool_calls = state['messages'][-1].tool_calls
    results = []
    for t in tool_calls:
        result = self.tools[t['name']].invoke(t['args'])
        results.append(ToolMessage(tool_call_id=t['id'],
                                   name=t['name'],
                                   content=str(result)))
    return {'messages': results}
```

The following function in the `ReActAgent` class is `should_call_tools`. This function determines whether to call the tools based on the state by retrieving the previous LLM response from the state and checking if it contains any tool calls.

```
def should_call_tools(self, state: AgentState):
    result = state['messages'][-1]
    return "tools" if len(result.tool_calls) > 0 else END
```

The `guardian_moderation` function executed in the `guardrain` node is designed to moderate messages using a guardian system, for the purpose of detecting and blocking unwanted or sensitive content. First, the last message is retrieved. Next, a dictionary named `detectors` is defined, which contains the detector configurations and their threshold values. These detectors identify specific types of content in messages, such as personally identifiable information (PII) as well as hate speech, abusive language, and profanity (HAP). Next, an instance of the `Guardian` class is created, passing in an `api_client` object named `client` and the `detectors` dictionary. The `detect` method of the `Guardian` instance is called, passing in the content of the last message and the `detectors` dictionary. The method then returns a dictionary in which the `moderation_verdict` key stores a value of either "safe" or "inappropriate," depending on the Granite Guardian model's output.

```
def guardian_moderation(self, state: AgentState):
    message = state['messages'][-1]
    detectors = {
        "granite_guardian": {"threshold": 0.4},
        "hap": {"threshold": 0.4},
        "pii": {},
    }
    guardian = Guardian(
        api_client=client,
        detectors=detectors
    )
    response = guardian.detect(
        text=message.content,
        detectors=detectors
    )
    if len(response['detections']) != 0 and response['detections'][0]['detection'] == "Yes":
        return {"moderation_verdict": "inappropriate"}
    else:
        return {"moderation_verdict": "safe"}
```

The `block_message` function serves as a notification mechanism, informing the user that their input query contains inappropriate content and has been blocked.

```
def block_message(self, state: AgentState):
    return {"messages": [AIMessage(content="This message has been blocked due to inappropriate content.")]}
```

We can now put all of this code together and run the following cell.

```
class ReActAgent:

    def __init__(self, llm, tools, system_message=""):
        memory = MemorySaver()
        graph = StateGraph(AgentState)
        graph.add_node("guardian", self.guardian_moderation)
        graph.add_node("llm", self.call_llm)
        graph.add_node("tools", self.call_tools)
        graph.add_node("block_message", self.block_message)
        graph.add_conditional_edges(
            "guardian",
            lambda state: state["moderation_verdict"],
            {
                "inappropriate": "block_message",
                "safe": "llm"
            }
        )
        graph.add_edge("block_message", END)
        graph.add_conditional_edges(
            "llm",
            self.should_call_tools,
            ["tools", END]
        )
        graph.add_edge("tools", "llm")
        graph.add_edge(START, "guardian")
        self.system_message = system_message
        self.graph = graph.compile(checkpointer=memory)
        self.tools = {t.name: t for t in tools}
        self.llm = llm.bind_tools(tools)

    def call_llm(self, state: AgentState):
        messages = state['messages']
        if self.system_message:
            messages = [SystemMessage(content=self.system_message)] + messages
        message = self.llm.invoke(messages)
        return {'messages': [message]}

    def call_tools(self, state: AgentState):
        tool_calls = state['messages'][-1].tool_calls
        results = []
        for t in tool_calls:
            result = self.tools[t['name']].invoke(t['args'])
            results.append(ToolMessage(tool_call_id=t['id'],
                                      name=t['name'],
                                      content=str(result)))
        return {'messages': results}

    def should_call_tools(self, state: AgentState):
        result = state['messages'][-1]
        return "tools" if len(result.tool_calls) > 0 else END

    def guardian_moderation(self, state: AgentState):
        message = state['messages'][-1]
        detectors = {
            "granite_guardian": {"threshold": 0.4},
            "hap": {"threshold": 0.4},
            "pii": {},
        }
        guardian = Guardian(
```

```

        api_client=client,
        detectors=detectors
    )
    response = guardian.detect(
        text=message.content,
        detectors=detectors
    )
    if len(response['detections']) != 0 and response['detections'][0]['detection'] == "Yes":
        return {"moderation_verdict": "inappropriate"}
    else:
        return {"moderation_verdict": "safe"}

def block_message(self, state: AgentState):
    return {"messages": [AIMessage(content="This message has been blocked due to inappropriate content.")]}

```

## Step 9. Create and invoke the ReActAgent object

The first line in the following code block creates an instance of the ReActAgent class, passing in the LLM, SQL tools and system message as parameters. Next, we specify a thread to store the graph states in memory. Think of each thread\_id as representing a new chat window. We can also define the user input to any string of choice. Next, we can pass a list consisting of the user input in HumanMessage type to invoke the agent.

First, let's try a prompt that should be blocked by the Granite Guardian model.

```

agent = ReActAgent(llm, tools, system_message=system_message)

config = {"configurable": {"thread_id": "1"}}

user_input = "What is the home address of the customer who purchased the most expensive car last month?"

result = agent.graph.invoke({'messages': [HumanMessage(content=user_input)]}, config)

for message in result["messages"]:
    message.pretty_print()

```

### Output:

```

===== [1m Human Message] [0m=====

What is the home address of the customer who purchased the most expensive car last month?
===== [1m Ai Message] [0m=====

This message has been blocked due to inappropriate content.

```

The Granite Guardian model was able to successfully block the user from requesting sensitive client information. We can see that the graph did not reach the LLM node before terminating the conversation. Next, let's ask an appropriate question in a different thread. For example, "What is the total sales revenue for the top 5 performing dealerships in the year 2022?" as the user input.

user\_input = "What is the total sales revenue for the top 5 performing dealerships in the year 2022?"

```

config2 = {"configurable": {"thread_id": "2"}}

result = agent.graph.invoke({'messages': [HumanMessage(content=user_input)]}, config2)

for message in result["messages"]:
    message.pretty_print()

```

### Output:

```

===== [1m Human Message] [0m=====

What is the total sales revenue for the top 5 performing dealerships in the year 2022?
===== [1m Ai Message] [0m=====

Tool Calls:
  sql_db_list_tables (H2irrlsPM)
  Call ID: H2irrlsPM
  Args:
    tool_input:
===== [1m Tool Message] [0m=====
Name: sql_db_list_tables

Cars, Dealerships, Sales
===== [1m Ai Message] [0m=====

Tool Calls:
  sql_db_schema (ILZWivGX)
  Call ID: ILZWivGX
  Args:
    table_names: Cars, Dealerships, Sales
===== [1m Tool Message] [0m=====
Name: sql_db_schema

```

```

CREATE TABLE "Cars" (
  id INTEGER,
  make TEXT NOT NULL,
  model TEXT NOT NULL,
  year INTEGER NOT NULL,
  color TEXT NOT NULL,
  PRIMARY KEY (id)
)

/*
3 rows from Cars table:
id    make    model  year  color
1     Toyota  Camry  2020  Blue
2     Honda   Civic  2019  Red
3     Ford    Mustang 2021  Black
*/

```

```

CREATE TABLE "Dealerships" (
    id INTEGER,
    name TEXT NOT NULL,
    location TEXT NOT NULL,
    PRIMARY KEY (id)
)

/*
3 rows from Dealerships table:
id      name      location
1      Dealership A    New York
2      Dealership B    Los Angeles
3      Dealership C    Chicago
*/

CREATE TABLE "Sales" (
    id INTEGER,
    car_id INTEGER NOT NULL,
    dealership_id INTEGER NOT NULL,
    sale_date DATE NOT NULL,
    sale_price REAL NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(dealership_id) REFERENCES "Dealerships" (id),
    FOREIGN KEY(car_id) REFERENCES "Cars" (id)
)

/*
3 rows from Sales table:
id      car_id      dealership_id      sale_date      sale_price
1      1      1      2022-01-01      25000.0
2      2      2      2022-02-01      20000.0
3      3      3      2022-03-01      30000.0
*/
===== [1m Ai Message] [0m=====
Tool Calls:
  sql_db_query_checker (yIZ0tk4VP)
Call ID: yIZ0tk4VP
Args:
  query: SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue FROM Dealerships INNER JOIN Sales ON Dealerships.id = Sa
===== [1m Tool Message] [0m=====
Name: sql_db_query_checker

SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue
FROM Dealerships
INNER JOIN Sales ON Dealerships.id = Sales.dealership_id
WHERE strftime('%Y', Sales.sale_date) = '2022'
GROUP BY Dealerships.id
ORDER BY total_sales_revenue DESC
LIMIT 5;
===== [1m Ai Message] [0m=====
Tool Calls:
  sql_db_query (cTzJFfvTL)
Call ID: cTzJFfvTL
Args:
  query: SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue FROM Dealerships INNER JOIN Sales ON Dealerships.id = Sa
===== [1m Tool Message] [0m=====
Name: sql_db_query

[('Dealership A', 65000.0), ('Dealership B', 48000.0), ('Dealership J', 38000.0), ('Dealership G', 35000.0), ('Dealership H', 32000.0)]
===== [1m Ai Message] [0m=====

The total sales revenue for the top 5 performing dealerships in the year 2022 are:

1. Dealership A: $65,000
2. Dealership B: $48,000
3. Dealership J: $38,000
4. Dealership G: $35,000
5. Dealership H: $32,000

===== [1m Human Message] [0m=====

What is the total sales revenue for the top 5 performing dealerships in the year 2022?
===== [1m Ai Message] [0m=====
Tool Calls:
  sql_db_list_tables (H2irrlsPM)
Call ID: H2irrlsPM
Args:
  tool_input:
===== [1m Tool Message] [0m=====
Name: sql_db_list_tables

Cars, Dealerships, Sales
===== [1m Ai Message] [0m=====
Tool Calls:
  sql_db_schema (ILZWivGX)
Call ID: ILZWivGX
Args:
  table_names: Cars, Dealerships, Sales
===== [1m Tool Message] [0m=====
Name: sql_db_schema

CREATE TABLE "Cars" (
    id INTEGER,
    make TEXT NOT NULL,

```

```

        model TEXT NOT NULL,
        year INTEGER NOT NULL,
        color TEXT NOT NULL,
        PRIMARY KEY (id)
    )

/*
3 rows from Cars table:
id      make      model  year  color
1      Toyota  Camry  2020  Blue
2      Honda   Civic  2019  Red
3      Ford    Mustang 2021  Black
*/

```

```

CREATE TABLE "Dealerships" (
    id INTEGER,
    name TEXT NOT NULL,
    location TEXT NOT NULL,
    PRIMARY KEY (id)
)

```

```

/*
3 rows from Dealerships table:
id      name      location
1      Dealership A  New York
2      Dealership B  Los Angeles
3      Dealership C  Chicago
*/

```

```

CREATE TABLE "Sales" (
    id INTEGER,
    car_id INTEGER NOT NULL,
    dealership_id INTEGER NOT NULL,
    sale_date DATE NOT NULL,
    sale_price REAL NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(dealership_id) REFERENCES "Dealerships" (id),
    FOREIGN KEY(car_id) REFERENCES "Cars" (id)
)

```

```

/*
3 rows from Sales table:
id      car_id  dealership_id  sale_date  sale_price
1        1        1      2022-01-01    25000.0
2        2        2      2022-02-01    20000.0
3        3        3      2022-03-01    30000.0
*/

```

===== [1m Ai Message] [0m=====

Tool Calls:

sql\_db\_query\_checker (yIZ0tk4VP)

Call ID: yIZ0tk4VP

Args:

query: SELECT Dealerships.name, SUM(Sales.sale\_price) AS total\_sales\_revenue FROM Dealerships INNER JOIN Sales ON Dealerships.id = Sa

===== [1m Tool Message] [0m=====

Name: sql\_db\_query\_checker

```

SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue
FROM Dealerships
INNER JOIN Sales ON Dealerships.id = Sales.dealership_id
WHERE strftime('%Y', Sales.sale_date) = '2022'
GROUP BY Dealerships.id
ORDER BY total_sales_revenue DESC
LIMIT 5;

```

===== [1m Ai Message] [0m=====

Tool Calls:

sql\_db\_query (cTzJFfvTL)

Call ID: cTzJFfvTL

Args:

query: SELECT Dealerships.name, SUM(Sales.sale\_price) AS total\_sales\_revenue FROM Dealerships INNER JOIN Sales ON Dealerships.id = Sa

===== [1m Tool Message] [0m=====

Name: sql\_db\_query

[('Dealership A', 65000.0), ('Dealership B', 48000.0), ('Dealership J', 38000.0), ('Dealership G', 35000.0), ('Dealership H', 32000.0)]

===== [1m Ai Message] [0m=====

The total sales revenue for the top 5 performing dealerships in the year 2022 are:

1. Dealership A: \$65,000
2. Dealership B: \$48,000
3. Dealership J: \$38,000
4. Dealership G: \$35,000
5. Dealership H: \$32,000

Great! The agent has successfully executed a series of tasks, including calling multiple tools to extract the SQL table schema, generating new SQL queries and verifying their correctness before executing them. As a result, the agent returns the correct total sales revenue for the top 5 performing car dealerships within our database in 2022. We can see the agent's multistep reasoning as it prints each generated SQL query. In the response, we also see that the Granite Guardian model determined that the user query is appropriate.

Let's ask a follow-up question. This time let's ask the agent to generate and execute Python code to display a bar graph representation of the results. The agent should retrieve the previous messages since we are using the same thread\_id.

user\_input = "Write Python code to plot these results on a bar graph. Then, you must execute the code and display the bar graph."

result = agent.graph.invoke({'messages': [HumanMessage(content=user\_input)]}, config2)



```
for message in result["messages"]:
    message.pretty_print()
```

**Output:**

```
===== [1m Human Message  [0m=====
```

```
What is the total sales revenue for the top 5 performing dealerships in the year 2022?
```

```
===== [1m Ai Message  [0m=====
```

```
Tool Calls:
```

```
  sql_db_list_tables (H2irrlsPM)
```

```
  Call ID: H2irrlsPM
```

```
  Args:
```

```
    tool_input:
```

```
===== [1m Tool Message  [0m=====
```

```
Name: sql_db_list_tables
```

```
Cars, Dealerships, Sales
```

```
===== [1m Ai Message  [0m=====
```

```
Tool Calls:
```

```
  sql_db_schema (ILZWivGX)
```

```
  Call ID: ILZWivGX
```

```
  Args:
```

```
    table_names: Cars, Dealerships, Sales
```

```
===== [1m Tool Message  [0m=====
```

```
Name: sql_db_schema
```

```
CREATE TABLE "Cars" (
  id INTEGER,
  make TEXT NOT NULL,
  model TEXT NOT NULL,
  year INTEGER NOT NULL,
  color TEXT NOT NULL,
  PRIMARY KEY (id)
)
```

```
/*
```

```
3 rows from Cars table:
```

id	make	model	year	color
1	Toyota	Camry	2020	Blue
2	Honda	Civic	2019	Red
3	Ford	Mustang	2021	Black

```
*/
```

```
CREATE TABLE "Dealerships" (
  id INTEGER,
  name TEXT NOT NULL,
  location TEXT NOT NULL,
  PRIMARY KEY (id)
)
```

```
/*
```

```
3 rows from Dealerships table:
```

id	name	location
1	Dealership A	New York
2	Dealership B	Los Angeles
3	Dealership C	Chicago

```
*/
```

```
CREATE TABLE "Sales" (
  id INTEGER,
  car_id INTEGER NOT NULL,
  dealership_id INTEGER NOT NULL,
  sale_date DATE NOT NULL,
  sale_price REAL NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY(dealership_id) REFERENCES "Dealerships" (id),
  FOREIGN KEY(car_id) REFERENCES "Cars" (id)
)
```

```
/*
```

```
3 rows from Sales table:
```

id	car_id	dealership_id	sale_date	sale_price
1	1	1	2022-01-01	25000.0
2	2	2	2022-02-01	20000.0
3	3	3	2022-03-01	30000.0

```
*/
```

```
===== [1m Ai Message  [0m=====
```

```
Tool Calls:
```

```
  sql_db_query_checker (yIZ0tk4VP)
```

```
  Call ID: yIZ0tk4VP
```

```
  Args:
```

```
    query: SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue FROM Dealerships INNER JOIN Sales ON Dealerships.id = Sa
```

```
===== [1m Tool Message  [0m=====
```

```
Name: sql_db_query_checker
```

```
SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue
FROM Dealerships
INNER JOIN Sales ON Dealerships.id = Sales.dealership_id
WHERE strftime('%Y', Sales.sale_date) = '2022'
GROUP BY Dealerships.id
ORDER BY total_sales_revenue DESC
LIMIT 5;
```

```
===== [1m Ai Message  [0m=====
```

```
Tool Calls:
```

```

sql_db_query (cTzJFfvTl)
Call ID: cTzJFfvTl
Args:
  query: SELECT Dealerships.name, SUM(Sales.sale_price) AS total_sales_revenue FROM Dealerships INNER JOIN Sales ON Dealerships.id = Sa
===== [1m Tool Message  [0m=====
Name: sql_db_query

[['Dealership A', 65000.0), ('Dealership B', 48000.0), ('Dealership J', 38000.0), ('Dealership G', 35000.0), ('Dealership H', 32000.0)]
===== [1m Ai Message  [0m=====

The total sales revenue for the top 5 performing dealerships in the year 2022 are:

1. Dealership A: $65,000
2. Dealership B: $48,000
3. Dealership J: $38,000
4. Dealership G: $35,000
5. Dealership H: $32,000
===== [1m Human Message  [0m=====

Write Python code to plot these results on a bar graph. Then, you must execute the code and display the bar graph.
===== [1m Ai Message  [0m=====

Tool Calls:
  Python_REPL (5X1fPIODL)
Call ID: 5X1fPIODL
Args:
  query: import matplotlib.pyplot as plt

# Data for total sales revenue
dealerships = ['Dealership A', 'Dealership B', 'Dealership J', 'Dealership G', 'Dealership H']
sales_revenue = [65000, 48000, 38000, 35000, 32000]

# Create a bar graph
plt.bar(dealerships, sales_revenue, color='blue')

# Customize the plot
plt.title('Total Sales Revenue for Top 5 Performing Dealerships in 2022')
plt.xlabel('Dealerships')
plt.ylabel('Total Sales Revenue ($)')
plt.xticks(rotation=45)
plt.grid(axis='y')

# Display the plot
plt.tight_layout()
plt.show()
===== [1m Tool Message  [0m=====
Name: Python_REPL

===== [1m Ai Message  [0m=====

Your requirements have been fulfilled. Code has been executed in repl.
The bar graph has been plotted and shown in the python REPL.
The graph can also be seen below:

As expected, the agent successfully called the python_repl tool to generate and execute Python code, resulting in a graphical representation of the results.
Notice that the agent was also able to determine the appropriate x-axis and y-axis values, labels and title. This highlights the agentic reasoning that sets AI agents
apart from traditional LLM chatbots.

To obtain a visual representation of the agent's graph, we can display the graph flow.
display(Image(agent.graph.get_graph().draw_mermaid_png()))

```

**Output:****Summary**

In this tutorial, we explored how to build a text-to-SQL ReAct agent with LangGraph. Instead of using the prebuilt agent executor function, `create_react_agent`, we built our agent from scratch. After parsing a user's question in natural language, our artificial intelligence agent uses its tools for SQL query generation, validation and execution, demonstrating skill in both SQL and Python. Given the LangChain `PythonREPLTool` class, the agent is equipped with the ability to not only generate but also execute Python code. We saw the successful execution of this tool upon prompting the agent for a graphical representation of its response. The conditional loop between the LLM and the tool nodes enables the ReAct agent architecture. With this tutorial, we have exhibited the coding and multimodal reasoning capabilities of Mistral Medium 3. As a next step, consider experimenting with additional nodes and edges in the graph.