

## Let's Build an Advanced RAG combining RAG and Text-to-SQL

---

Before we begin, here's a quick demo of what we're building!

We will be using

@OpenAI: OpenAI model for language processing

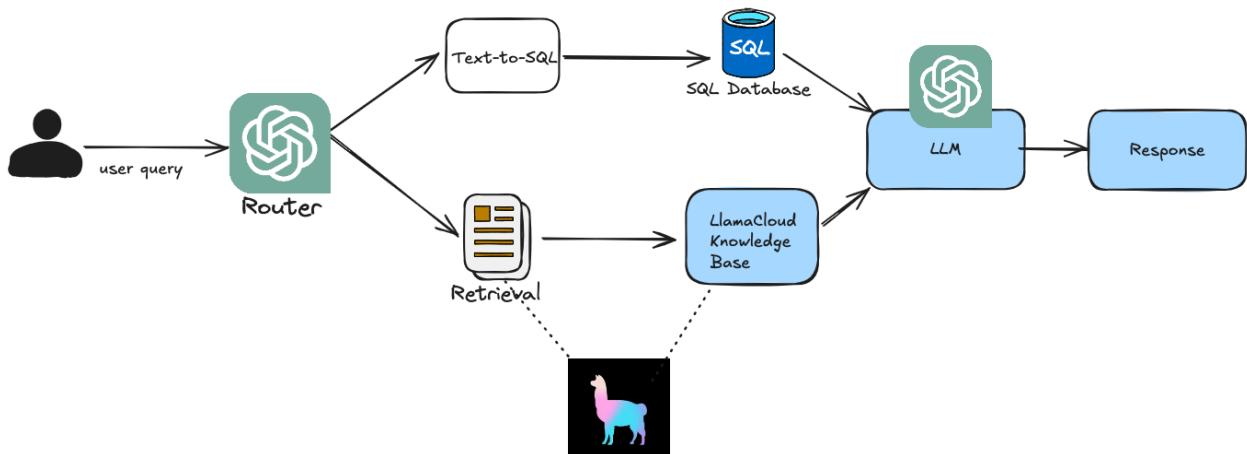
@llama\_index: LlamaCloud index for RAG-based retrieval

Demo: [https://github.com/afizs/ai-engineering-hub/blob/main/llamacloud\\_sql\\_router/demo.mp4](https://github.com/afizs/ai-engineering-hub/blob/main/llamacloud_sql_router/demo.mp4)

---

How it works:

1. User inputs a query (e.g., "Which city has the highest population?")
2. AI selects the appropriate tool (SQL for structured queries, Llama Cloud for semantic ones)
3. The tool executes the query and retrieves the response.
4. Results are returned to the user



0. Setup your API keys and project and index name of Llamaindex

# Setup your API keys

.env

```
OPENAI_API_KEY="YOUR OPENAI API KEY"  
LLAMA_INDEX_API_KEY=""  
LLAMA_INDEX_ORGANIZATION_ID="YOUR_ORGANIZATION_ID"  
LLAMA_INDEX_PROJECT_NAME="YOUR_PROJECT_NAME"  
LLAMA_INDEX_NAME="YOUR_INDEX_NAME"
```



Afiz ⚡  
𝕏 @itsafiz

Step - 0

## 1. Create SQL Database and Insert Data

We will use SQLite for creating the Database. This is a 3 step process.

- Create the SQL Engine
- Create table and with three columns
- Insert the sample data

### Create SQL Database & Insert Data

The diagram illustrates the flow of the Python code into four main steps:

- Create a Database engine, which stores data locally in db.sqlite**: Points to the first block of code where the database engine is created.
- Create city\_stats table with three columns. city\_name, population and state**: Points to the second block of code where the table structure is defined.
- Inset the sample data into the table**: Points to the third block of code where the sample data is inserted into the table.
- Fetch the data from the DB**: Points to the fourth block of code where the data is retrieved from the database.

```
DB_PATH = "db.sqlite"
engine = create_engine(f"sqlite:/// {DB_PATH}", future=True)
metadata_obj = MetaData()

# create city SQL table
table_name = "city_stats"
city_stats_table = Table(
    table_name,
    metadata_obj,
    Column("city_name", String(16), primary_key=True),
    Column("population", Integer),
    Column("state", String(16), nullable=False),
)

metadata_obj.create_all(engine)

rows = [
    {"city_name": "New York City", "population": 8336000, "state": "New York"}, 
    {"city_name": "Los Angeles", "population": 3822000, "state": "California"}, 
    {"city_name": "Chicago", "population": 2665000, "state": "Illinois"}, 
    {"city_name": "Houston", "population": 2303000, "state": "Texas"}, 
    {"city_name": "Miami", "population": 449514, "state": "Florida"}, 
    {"city_name": "Seattle", "population": 749256, "state": "Washington"}, 
]
for row in rows:
    stmt = insert(city_stats_table).values(**row)
    with engine.begin() as connection:
        cursor = connection.execute(stmt)

with engine.connect() as connection:
    cursor = connection.exec_driver_sql("SELECT * FROM city_stats")
    print(cursor.fetchall())
```

Afiz ✨  
X @itsafiz

Step - 1

## 2. AI powered SQL Query Engine

OpenAI gpt-3.5-tubo model is used for natural language processing.

### Setup AI powered SQL Query Engine

```
● ● ● _engine.py
```

```
from llama_index.core import SQLDatabase, Settings
from llama_index.llms.openai import OpenAI

# Set OpenAI model for language processing
Settings.llm = OpenAI("gpt-3.5-turbo")

# Initialize SQL Query Engine for Natural Language Processing.
# This AI powered SQL Query Engine automatically translates natural
# language into SQL queries, which can be executed on the given DB
sql_database = SQLDatabase(engine, include_tables=["city_stats"])
sql_query_engine = NLSQLTableQueryEngine(
    sql_database=sql_database,
    tables=["city_stats"]
)
```

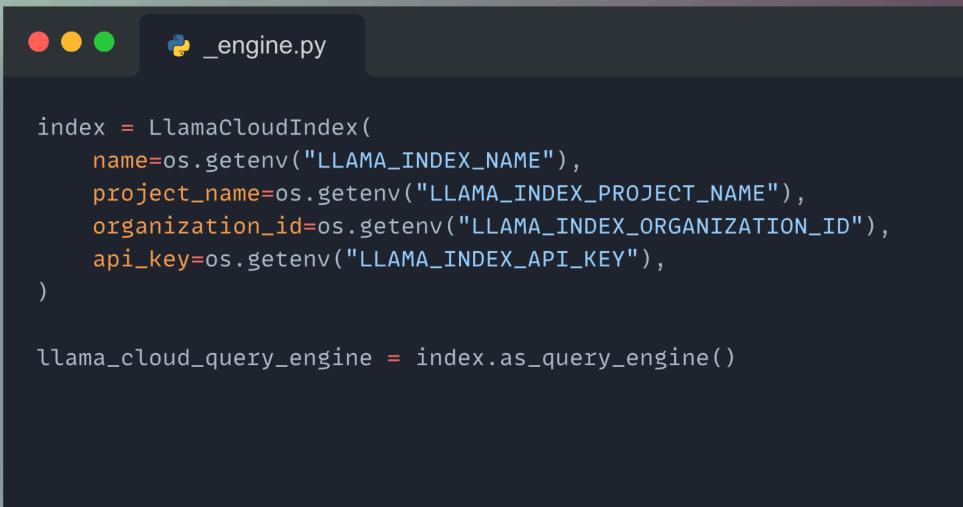
Afiz ✨  
𝕏 @itsafiz

Step - 2

### 3. Setup LlamaIndex Query Engine for RAG

Checkout the Github repo for the steps to create index on LlamaIndex Cloud

## Setup LlamaIndex Query Engine for RAG



```
index = LlamaCloudIndex(  
    name=os.getenv("LLAMA_INDEX_NAME"),  
    project_name=os.getenv("LLAMA_INDEX_PROJECT_NAME"),  
    organization_id=os.getenv("LLAMA_INDEX_ORGANIZATION_ID"),  
    api_key=os.getenv("LLAMA_INDEX_API_KEY"),  
)  
  
llama_cloud_query_engine = index.as_query_engine()
```

Ensure you download the city information PDFs from Wikipedia. Then, create a new index in LlamaCloud and upload the PDFs.

Afiz ✨  
𝕏 @itsafiz

Step - 3

#### 4. Create Query Engine Tools

- **sql\_tool** – Uses `sql_query_engine` to retrieve city population and state data via **SQL queries**.
- **llama\_cloud\_tool** – Uses `llama_cloud_query_engine` to answer **semantic questions** about US cities.

The image shows two terminal windows side-by-side. The top window is titled "Create Query Engine Tools" and has a yellow header bar with the text "SQL query tool". It contains Python code for creating an "sql\_tool". The bottom window is titled "Llama Query Tool" and contains Python code for creating an "llama\_cloud\_tool". Both windows show the code being typed into the terminal, with the file name "sql\_engine\_tool.py" or "llama\_engine\_tool.py" visible in the title bar. The code uses the `QueryEngineTool.from_defaults` method to define the tools, specifying the query engine and a descriptive comment. The bottom window also includes a watermark "Afiz ✨" and "X @itsafiz" in the bottom left corner, and "Step - 4" in the bottom right corner.

```
from llama_index.core.tools import QueryEngineTool

sql_tool = QueryEngineTool.from_defaults(
    query_engine=sql_query_engine,
    description=(
        '''Useful for translating a natural language query into
        a SQL query over a table containing: city_stats, containing
        the population/state of each city located in the USA.'''
    ),
    name="sql_tool"
)

llama_cloud_tool = QueryEngineTool.from_defaults(
    query_engine=llama_cloud_query_engine,
    description=(
        f'''Useful for answering semantic questions about certain
        cities in the US'''
    ),
    name="llama_cloud_tool"
)
```

5. Let's create a workflow that acts as an agent around the two query engines.

First in the workflow, we need 4 events as follows

## Define Event Classes for the Workflow

```
● ● ●   Python event_classes.py

class InputEvent(Event):
    """Represents an input event."""

class GatherToolsEvent(Event):
    """Event triggered to gather tools for execution."""
    tool_calls: Any

class ToolCallEvent(Event):
    """Event representing a tool call."""
    tool_call: ToolSelection

class ToolCallEventResult(Event):
    """Event capturing the result of a tool execution."""
    msg: ChatMessage
```

## 5.1 Define the `RouterOutputAgentWorkflow` Class

This class **manages AI tool selection and execution**

It initializes:

- `tools`: A list of available **query tools** (SQL, Llama Cloud)
- `chat_history`: Stores past **user queries and responses**
- `llm`: Uses **GPT-3.5-turbo** as the default language model

## Define the `RouterOutputAgentWorkflow`

This class manages AI tool selection and execution.

```
class RouterOutputAgentWorkflow(Workflow):
    """Custom workflow that handles tool selection and message routing."""

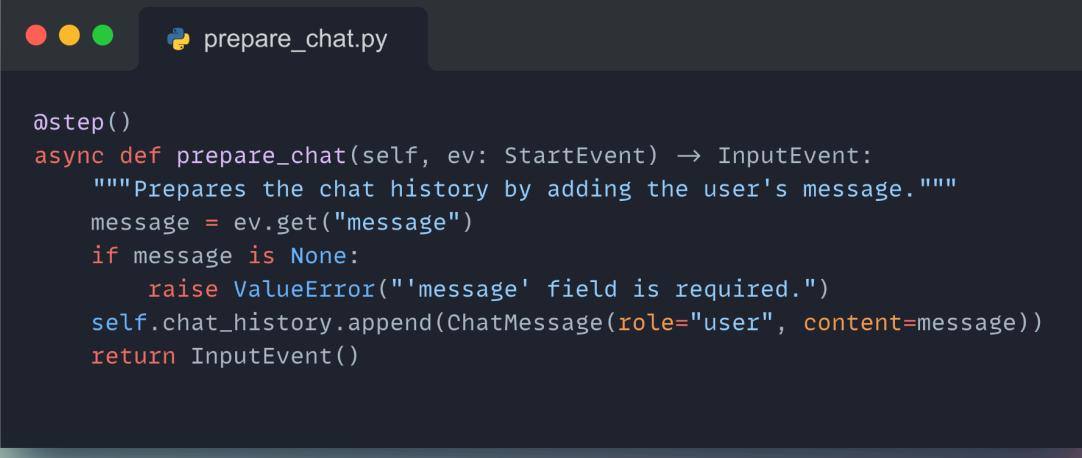
    def __init__(
        self,
        tools: List[BaseTool],
        timeout: Optional[float] = 10.0,
        disable_validation: bool = False,
        verbose: bool = False,
        llm: Optional[LLM] = None,
        chat_history: Optional[List[ChatMessage]] = None,
    ):
        """Initialize the workflow with available tools and configurations."""
        super().__init__(timeout=timeout, disable_validation=disable_validation,
                         verbose=verbose)
        self.tools: List[BaseTool] = tools
        self.tools_dict: Optional[Dict[str, BaseTool]] = {
            tool.metadata.name: tool for tool in self.tools
        }
        self.llm: LLM = llm or OpenAI(temperature=0, model="gpt-3.5-turbo")
        self.chat_history: List[ChatMessage] = chat_history or []

    def reset(self) -> None:
        """Clear chat history."""
        self.chat_history = []
```

## 6. Define Workflow Steps for Processing Messages and Calling Tools

This workflow consists of 5 steps.

### 1. Process User Message



The screenshot shows a terminal window with a dark theme. The title bar says "Workflow Step: Process User Message". The window contains the following Python code:

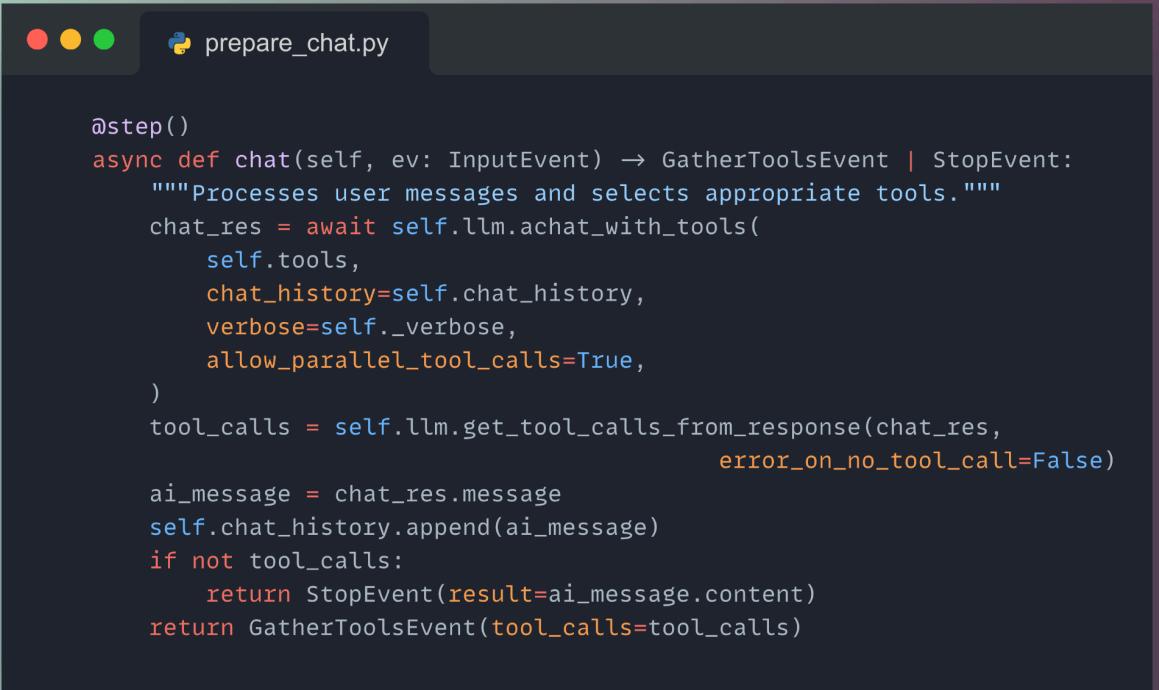
```
@step()
async def prepare_chat(self, ev: StartEvent) -> InputEvent:
    """Prepares the chat history by adding the user's message."""
    message = ev.get("message")
    if message is None:
        raise ValueError("'message' field is required.")
    self.chat_history.append(ChatMessage(role="user", content=message))
    return InputEvent()
```

At the bottom left, it says "Afiz ⚡" and "X @itsafiz". At the bottom right, it says "Step - 6.1".

## 6.2 Choose an AI Tool

GPT-3.5-turbo model analyzes the user message. It selects the most relevant tool (`sql_tool` or `llama_cloud_tool`).

### Workflow Step: Choose an AI Tool



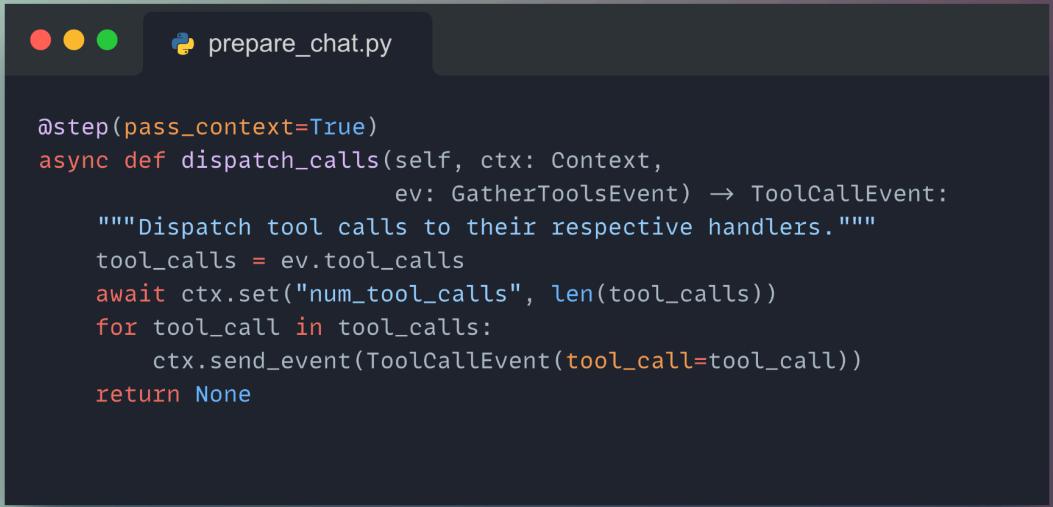
```
async def chat(self, ev: InputEvent) -> GatherToolsEvent | StopEvent:
    """Processes user messages and selects appropriate tools."""
    chat_res = await self.llm.agchat_with_tools(
        self.tools,
        chat_history=self.chat_history,
        verbose=self._verbose,
        allow_parallel_tool_calls=True,
    )
    tool_calls = self.llm.get_tool_calls_from_response(chat_res,
                                                       error_on_no_tool_call=False)
    ai_message = chat_res.message
    self.chat_history.append(ai_message)
    if not tool_calls:
        return StopEvent(result=ai_message.content)
    return GatherToolsEvent(tool_calls=tool_calls)
```

Afiz ✨  
X @itsafiz

### 6.3. Dispatch Tool Calls

Sends the tool request to the appropriate AI tool.

## Workflow Step 3: Dispatch Tool Calls



```
prepare_chat.py

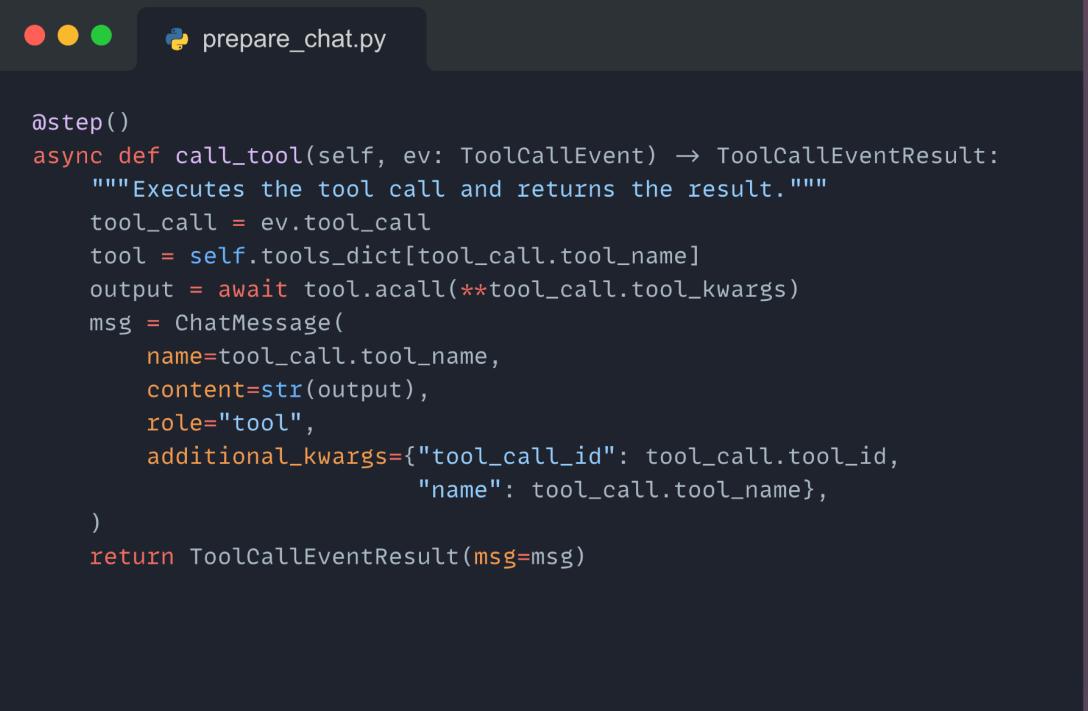
@step(pass_context=True)
async def dispatch_calls(self, ctx: Context,
                       ev: GatherToolsEvent) -> ToolCallEvent:
    """Dispatch tool calls to their respective handlers."""
    tool_calls = ev.tool_calls
    await ctx.set("num_tool_calls", len(tool_calls))
    for tool_call in tool_calls:
        ctx.send_event(ToolCallEvent(tool_call=tool_call))
    return None
```

Afiz ⚡  
X @itsafiz

## 6.4 Execute the Tool Call

- Calls the selected tool (`SQL tool` or `Llama Cloud`).
- Executes the query and stores the result in a chat message.

### Workflow Step 4: Execute the Tool Call



```
async def call_tool(self, ev: ToolCallEvent) -> ToolCallEventResult:
    """Executes the tool call and returns the result."""
    tool_call = ev.tool_call
    tool = self.tools_dict[tool_call.tool_name]
    output = await tool.acall(**tool_call.tool_kwargs)
    msg = ChatMessage(
        name=tool_call.tool_name,
        content=str(output),
        role="tool",
        additional_kwargs={"tool_call_id": tool_call.tool_id,
                           "name": tool_call.tool_name},
    )
    return ToolCallEventResult(msg=msg)
```

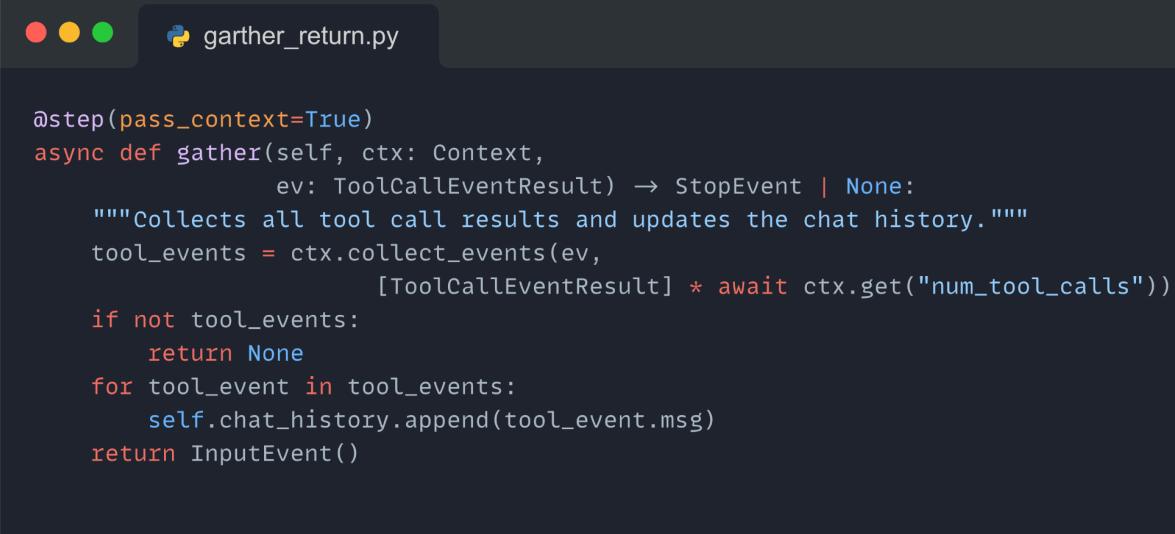
Afiz ✨  
X @itsafiz

## 6.5 Gather and Return the Tool Results

- Retrieves all responses from the AI tool.
- Adds the response to chat history.
- Returns the tool-generated answer

### Workflow Step 5: Gather and Return the Tool Results

python



```
garther_return.py

@step(pass_context=True)
async def gather(self, ctx: Context,
                 ev: ToolCallEventResult) -> StopEvent | None:
    """Collects all tool call results and updates the chat history."""
    tool_events = ctx.collect_events(ev,
                                      [ToolCallEventResult] * await ctx.get("num_tool_calls"))
    if not tool_events:
        return None
    for tool_event in tool_events:
        self.chat_history.append(tool_event.msg)
    return InputEvent()
```

Afiz ✨  
X @itsafiz

## 7. Initialize the workflow

Creates an instance of `RouterOutputAgentWorkflow` with:

- `sql_tool`: SQL query engine for database searches.
- `llama_cloud_tool`: Llama Cloud AI tool for answering questions.

## Workflow Step 6: Initialize the workflow



```
garther_return.py
wf = RouterOutputAgentWorkflow(tools=[sql_tool, llama_cloud_tool],
                                verbose=True, timeout=120)
```

Afiz ✨  
X @itsafiz

## 8. Run the workflow

Run the workflow with a sample query and see the result 😊

The screenshot shows two Jupyter Notebook cells. The top cell is titled "Workflow Step 7: Run the workflow". It contains Python code that runs a workflow and displays the result. A yellow arrow points from the "message" parameter in the code to the "Response" cell below. The bottom cell is titled "Response" and shows the execution log of the workflow steps, ending with the output "New York City has the highest population." A yellow arrow points from the final output line in the log to the text itself.

```
result = await wf.run(message="Which city has the highest population?")
display(Markdown(result))
```

Running step prepare\_chat  
Step prepare\_chat produced event InputEvent  
Running step chat  
Chat message: None  
Step chat produced event GatherToolsEvent  
Running step dispatch\_calls  
Step dispatch\_calls produced no event  
Running step call\_tool  
Calling function sql\_tool with msg  
{'input': 'SELECT city FROM city\_stats ORDER BY population DESC LIMIT 1'}  
Step call\_tool produced event ToolCallEventResult  
Running step gather  
Step gather produced event InputEvent  
Running step chat  
Chat message: New York City has the highest population.  
Step chat produced event StopEvent  
New York City has the highest population.

That's all for today. Please checkout the GitHub for complete code along with Streamlit UI.

[https://github.com/afizs/ai-engineering-hub/tree/main/llamacloud\\_sql\\_router](https://github.com/afizs/ai-engineering-hub/tree/main/llamacloud_sql_router)