Teradata · Follow publication

Build a Data Analyst AI Agent from Scratch

9 min read · Feb 7, 2025

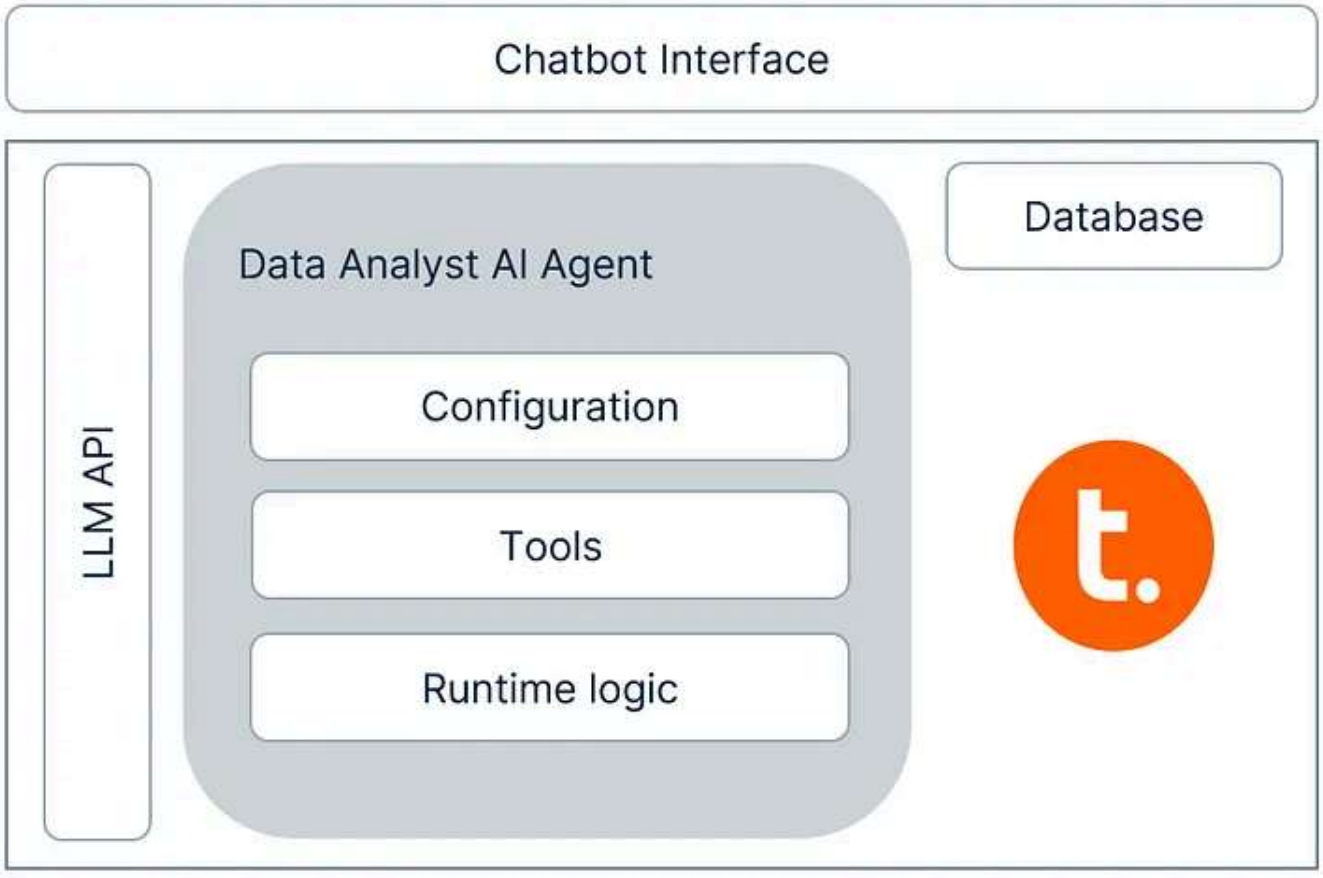
Daniel Herrera Follow

Listen Share More

As presented in the Open Data Science Conference AI Builders Summit 2025

Agentic AI and AI Agents are widely discussed topics within the data community. If you work as a data analyst, data scientist or data engineer, you see multiple architectures, frameworks, techniques and use cases for Agentic AI mentioned daily in your professional network.

In cases like this, where information abounds and the subject is complex, it is worth it to build an understanding of the topic starting from its most basic implementation which is the purpose of this article and the accompanying notebook.



Basic Architecture Diagram of the Data Analyst AI Agent

Agentic AI in simple terms

Agents, intelligent agents, and autonomous agents are concepts that have been discussed in academia well before commercial Large Language Models (LLMs) became widely available. An agent, in simple terms, is a system that, when provided with a goal and a set of tools, works towards achieving the goal by using the tools provided.

The use of large language models (LLMs) to determine the path to achieve a goal, sequence tool usage, and manage unexpected states is commonly referred to as Agentic AI.

Two key properties of LLMs make them particularly effective as an orchestration engine for agents, compared to rule-based approaches or other machine learning alternatives:

Embedded knowledge:

LLMs contain a vast amount of information extracted from their training data. This enables them to manage a variety of states with minimal instructions.

Structured responses:

LLMs can be instructed to generate structured outputs. This simplifies their implementation within an agent system, which, at its core, is a program processing these outputs.

Our data focused AI Agent

In our example, we build an AI Agent with the following characteristics:

Goal: Answer business-related questions provided by the user in English, based on the information contained in a database.

Tools: A function that executes SQL statements in a Teradata Vantage database.

Requirements

To build this agent, we will need the following resources:

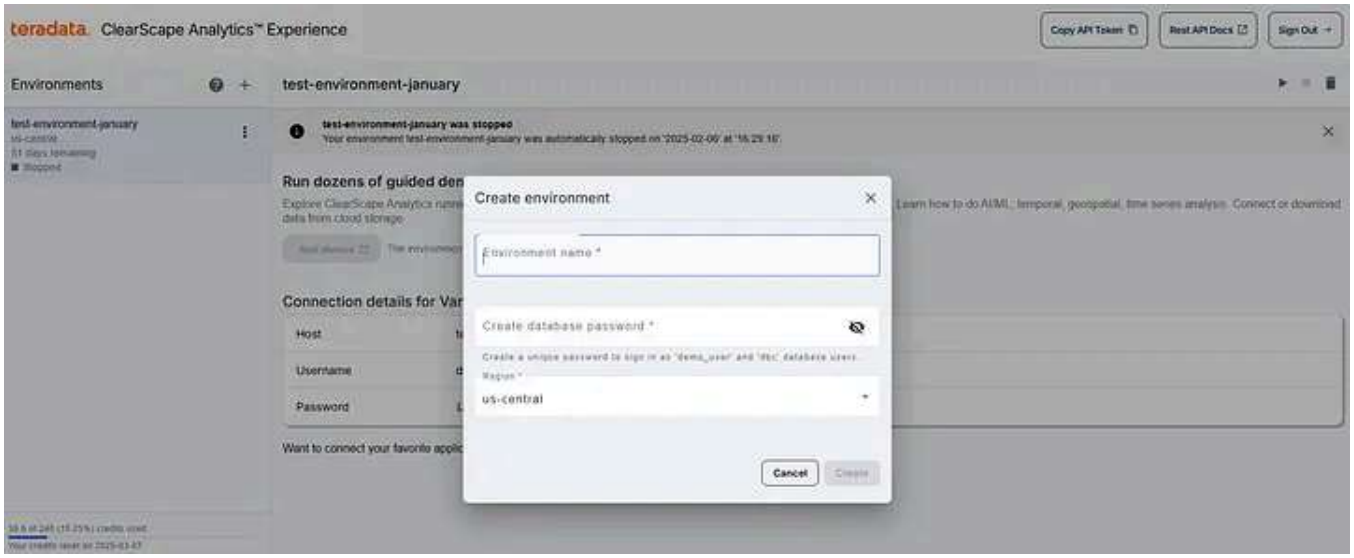
- A Teradata VantageCloud development environment with Jupyter Notebooks integration. Get one for free at ClearScape Analytics Experience.
- API access to an LLM.

In the sample notebook and this article, we use OpenAI, but this can be changed. Doing so requires modifying the API calls accordingly.

Preparing the development environment

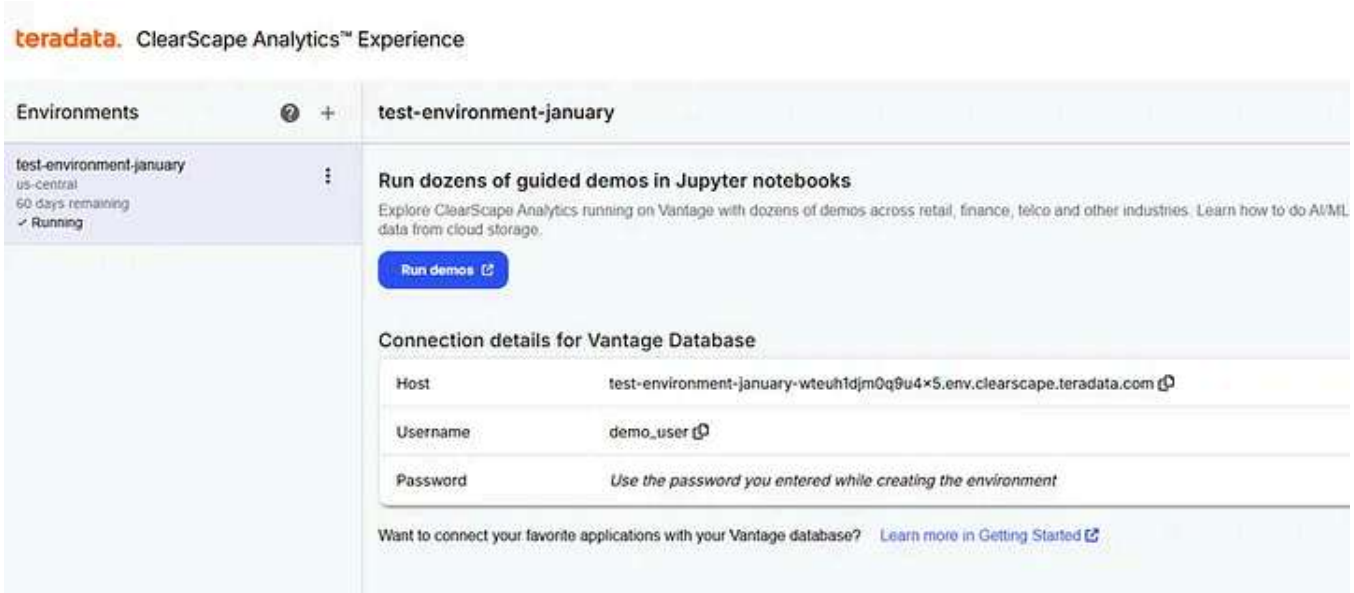
1. Log in to ClearScape Analytics Experience.
2. Create an environment in the ClearScape Analytics Experience console.

> *Take note of your chosen password as you will need it to interact with the database*



Create an environment on ClearScape Analytics Experience

3. Start the Jupyter Notebook environment by clicking “Run demos”.

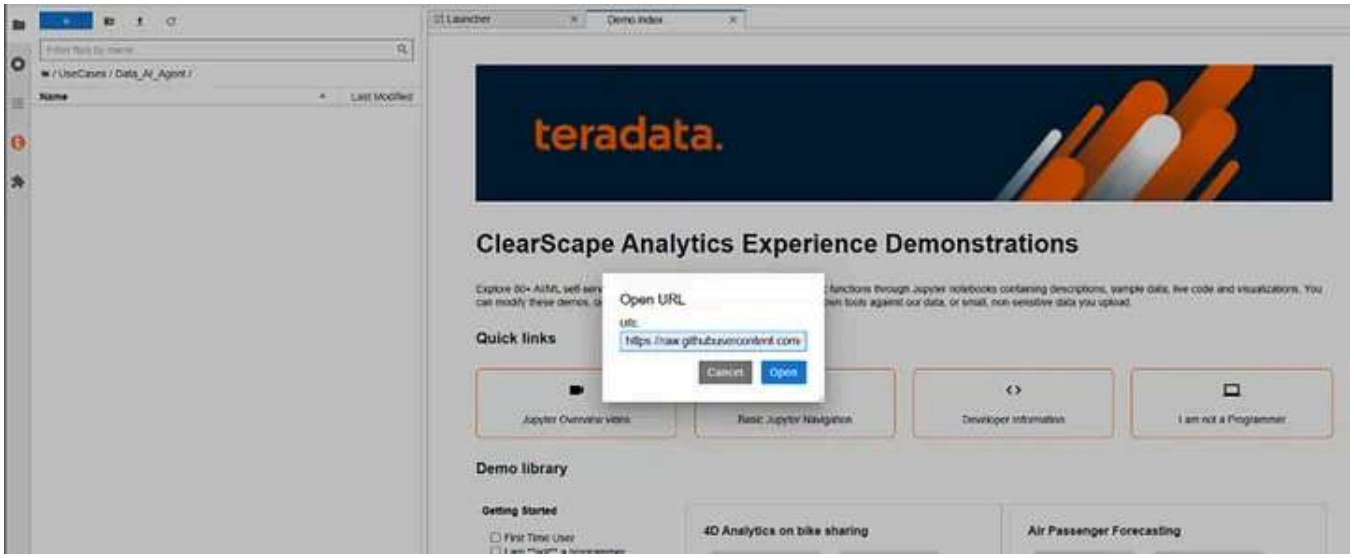


Run demos on ClearScape Analytics Experience

4. In the Jupyter Notebook environment, open the “Use Cases” folder and create a folder with a name of your choice.

5. Open the folder you’ve created, click on “Open from URL” under the “File” menu, and paste the following URL in the dialog box:

- https://raw.githubusercontent.com/Teradata/simple_data_ai_agent/refs/heads/main/notebooks/simple_data_agent.ipynb



Loading sample notebook on ClearScape Analytics Experience

- This will load the project's notebook into your environment.

6. In the folder you created, create or load a `configs.json` file with the following structure, replacing `"your-api-key-here"` with your actual LLM API key:

```
{
"llm-api-key": "your-api-key-here"
}
```

7. Notes on dependencies

- If you are running the project on ClearScape Analytics Experience the above are all the required prerequisites, the notebook can run as it is.
- If you are using a different LLM provider, you will need to install the corresponding SDK and adjust the code in the notebook accordingly.

Building a Data Analyst AI Agent

Project setup:

In this section we install and import the necessary libraries, load the LLM service API key to memory, create a database connection, and load sample data.

The first step is to install the SDK of our chosen LLM API provider.

```
!pip install openai
```

Running the project in ClearScape Analytics experience simplifies this setup significantly since the database comes integrated in the notebook environment. Creating a connection to this database is straightforward as seen in the snippet below.

The password needed to establish the connection is the one selected for the corresponding environment, as mentioned in the prerequisites.

```
%run -i ../startup.ipynb
eng = create_context(host = 'host.docker.internal', username='demo_user', passw
print(eng)
```

Once the database connection is created, the `data_loading_queries` can be run against the database to create the tables needed by the example.

Agent Configuration:

The configuration of the agent is comprised of two pieces:

1. Definition of the Agent's routine.
2. Definition of the tools available to the Agent for achieving the goal.

The Agent's routine is typically defined in a system prompt, outlining both its goal and the necessary actions to achieve it.

```
# Define the system prompt
system_prompt = f"""
You are a data analyst for a retail company working with a Teradata system.

1. Users send you business questions in plain English, and you provide answers
2. To generate answers, you must construct an SQL statement to query the Terada
   - The SQL query must be written as a single line of text without carriage re
   - Ensure that the query adheres to Teradata's SQL dialect and does not inclu
   - Joins across tables should be used when necessary to fulfill the user's re
3. Execute the SQL query in Teradata.
```

```
    4. Present the query results to the user in plain English.
    """
```

The system prompt, for our purposes, should embed the data catalog needed to fulfill the user's request.

The function `query_teradata_dictionary()` is responsible for retrieving the data catalog. The DBC database in a Teradata system contains relevant information that can be easily retrieved to form a data catalog.

```
databases = ["teddy_retailers"]
def query_teradata_dictionary(databases_of_interest):
    query = f'''
            SELECT DatabaseName, TableName, ColumnName, ColumnFormat, ColumnTyp
            FROM DBC.ColumnsV
            WHERE DatabaseName IN ('{', '.join(databases_of_interest)}')
            '''
    table_dictionary = DataFrame.from_query(query)
    return json.dumps(table_dictionary.to_pandas().to_json())
```

The tools available to the Agent are defined as functions. Knowledge of these functions is provided to the LLM through data structures called function signatures, enabling it to generate appropriate function calls.

We need to provide the agent only with one tool, the function needed to query the database.

```
def query_teradata_database(sql_statement):
    query_statement = sql_statement.split('ORDER BY',1)[0]
    query_result = DataFrame.from_query(query_statement)
    return json.dumps(query_result.to_pandas().to_json())
```

Due to the way we need to return the results of the query to the Agent, as a JSON structure, it is very convenient to use Teradata Dataframes to process the SQL query. This approach, however, precludes the usage of ORDER BY statements in the SQL

queries to be processed. This is not known by the LLM and will require prompting tests and trial and error to make it produce SQL queries compatible with the constraint.

Prompts don't provide consistent results, rules do. When working with LLMs if something can be formulated as a rule, it is always a good idea to make it a rule, thus the small string manipulation at the beginning of the function.

As mentioned, tools are known to the LLM through their corresponding signatures. Function signatures can be created manually or generated using a helper function. The latter is a more scalable and less error-prone approach than manual definition.

```
def function_to_schema(func) -> dict:
    type_map = {
        str: "string",
        int: "integer",
        float: "number",
        bool: "boolean",
        list: "array",
        dict: "object",
        type(None): "null",
    }

    try:
        signature = inspect.signature(func)
    except ValueError as e:
        raise ValueError(
            f"Failed to get signature for function {func.__name__}: {str(e)}"
        )

    parameters = {}
    for param in signature.parameters.values():
        try:
            param_type = type_map.get(param.annotation, "string")
        except KeyError as e:
            raise KeyError(
                f"Unknown type annotation {param.annotation} for parameter {par
            )
        parameters[param.name] = {"type": param_type}

    required = [
        param.name
        for param in signature.parameters.values()
        if param.default == inspect._empty
    ]

    return {
```

```
            "type": "function",
            "function": {
                "name": func.__name__,
                "description": (func.__doc__ or "").strip(),
                "parameters": {
                    "type": "object",
                    "properties": parameters,
                    "required": required,
                },
            },
        }
```

Agent Runtime:

For the agent runtime we need the following:

1. Extract the data structure of our tool functions.

```
tools = [query_teradata_database]
tool_schemas = [function_to_schema(tool) for tool in tools]
```

2. Create a map between the names of our functions which are strings and the actual function objects in Python. The LLM returns the name of the functions the agent will need to execute as strings, but what we need to call as part of our code is the function objects. This map bridges the LLM response to our Agent runtime.

```
tools_map = {tool.__name__: tool for tool in tools}
```

3. A tool_call executor, which is a helper function that takes a function name, as a string, and function parameters, and executes the functions referenced with the corresponding parameters, through the tools_map we've just created.

```
def execute_tool_call(tool_call, tools_map):
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)
    print(f"Assistant: {name}({args})")
```

```
    # call corresponding function with provided arguments
    return tools_map[name](**args)
```

4. Our runtime function proper:

This function takes as parameters the system prompt and the user prompt. Its inner workings are contained in two loops.
The outer loop interacts with the LLM, appending the LLM responses to the message chain.

```
def run_full_turn(system_message, messages):
    while True:
        print(f"just logging messages {messages}")
        response = client.chat.completions.create(
            model="gpt-4o",
            messages=[{"role": "system", "content": system_message}] + messages
            tools=tool_schemas or None,
            seed = 2
        )
message = response.choices[0].message
messages.append(message)
```

The inner loop iterates over all the needed tool_calls, calling the corresponding tools and appending their results to the messages that will be sent to the LLM the next iteration.

```
if message.tool_calls:  # if finished handling tool calls, break
            # === 2. handle tool calls ===
            for tool_call in message.tool_calls:
                result = execute_tool_call(tool_call, tools_map)

                result_message = {
                    "role": "tool",
                    "tool_call_id": tool_call.id,
                    "content": result,
                }
                messages.append(result_message)
        else:
            break
```

Putting it all together

Outer loop first iteration:

Message chain: At this moment only the user query

```
[{'role': 'user', 'content': 'what is the month with the greatest amount of ord
```

LLM Response:


```
"SELECT EXTRACT(MONTH FROM order_date) AS order_month, COUNT(order_id) AS order
, name='query_teradata_database'), type='function')]))]
```

The LLM has identified the tools and produced the SQL statement that will be executed. This response is appended to the message chain.

Updated message chain

```
[{'role': 'user', 'content': 'what is the month with the greatest amount of ord
ChatCompletionMessage(content=None, refusal=None, role='assistant', audio=None,
function_call=None,
tool_calls=[ChatCompletionMessageToolCall(id='call_OokTYjelsTCfERs9pTfUwA9h',
function=Function(arguments='{"sql_statement":"SELECT EXTRACT(MONTH FROM order_
, name='query_teradata_database'), type='function')]
```

This part is important as the message chain provides the LLM context regarding tools and their results. This is the reason each `tool_call` has an id.

```
ChatCompletionMessageToolCall(id='call_OokTYjelsTCfERs9pTfUwA9h'...)
```

The Agent system performs the tool calls and appends the results to the message chain according to the logic defined in the inner for loop.

Updated message chain, this time with the results of the database query.

```
[{'role': 'user', 'content': 'what is the month with the greatest amount of orc
ChatCompletionMessage(content=None, refusal=None, role='assistant', audio=None,
function_call=None,
tool_calls=[ChatCompletionMessageToolCall(id='call_OokTYjelsTCfERs9pTfUwA9h',
function=Function(arguments='{"sql_statement":"SELECT EXTRACT(MONTH FROM order_
name='query_teradata_database'), type='function')]),
{'role': 'tool', 'tool_call_id': 'call_OokTYjelsTCfERs9pTfUwA9h',
'content':
'"{\\"order_month\\":{\\"0\\":11,\\"1\\":8,\\"2\\":12,\\"3\\":7,\\"4\\":10,\\"5
"order_count\\":{\\"0\\":607,\\"1\\":810,\\"2\\":721,\\"3\\":623,\\"4\\":780,\\
```

Outer loop second iteration:

Message chain: Same as above

```
[{'role': 'user', 'content': 'what is the month with the greatest amount of orc
ChatCompletionMessage(content=None, refusal=None, role='assistant', audio=None,
function_call=None,
tool_calls=[ChatCompletionMessageToolCall(id='call_OokTYjelsTCfERs9pTfUwA9h',
function=Function(arguments='{"sql_statement":"SELECT EXTRACT(MONTH FROM order_
name='query_teradata_database'), type='function')]),
{'role': 'tool', 'tool_call_id': 'call_OokTYjelsTCfERs9pTfUwA9h',
'content':
'"{\\"order_month\\":{\\"0\\":11,\\"1\\":8,\\"2\\":12,\\"3\\":7,\\"4\\":10,\\"5
"order_count\\":{\\"0\\":607,\\"1\\":810,\\"2\\":721,\\"3\\":623,\\"4\\":780,\\
```

LLM response

```
message=ChatCompletionMessage(
content='The month with the greatest number of orders is September, with a tota
refusal=None, role='assistant', audio=None, function_call=None, tool_calls=None
```

At this point, the LLM took the finalized message chain to provide the final response.

Conclusions

As seen in this sample project, Agentic AI is fundamentally about orchestration. Agents are not LLMs; rather, they are systems that integrate a set of tools, rules, and guardrails. Orchestrated by an LLM, these components work together to achieve specific goals.

LLMs cannot execute external tools, the Agent System executes the tools based on the responses provided by the LLM.

LLM providers, such as OpenAI, offer an API with a defined data structure for the following purposes:

- Refer to tools and define tool_calls
- Organize the message chain
- Record tool responses

LLMs running on a user's own infrastructure may not offer all these features, making agent frameworks more necessary in such cases.

This example could be improved by incorporating error-handling mechanisms for both tools and their outputs.

Now it's your turn to recreate the notebook, enhance it, and share your insights with us. Sign up for a ClearScape Analytics Experience account today!

Agentic Ai | Agents | Data Analysis | Teradata | Generative Ai Use Cases