How to convert your existing microservices into Langchain Tools

Imagine a world where your chatbot can not only understand your requests but also take action on your behalf. Sounds futuristic? It's not. It's the power of Large Language Models (LLM) combined with your existing microservices.

The Evolution of LLM: From Text Generators to Action-Oriented Assistants 

Large Language Models (LLM) have evolved from simple text generators to advanced Agents capable of understanding complex tasks and breaking them down into smaller, actionable steps.

But the agent alone can't interact directly with the real world or execute code. However, **Tools** bridge this gap by enabling agents to perform actions such as searching the web, accessing databases, or controlling devices.

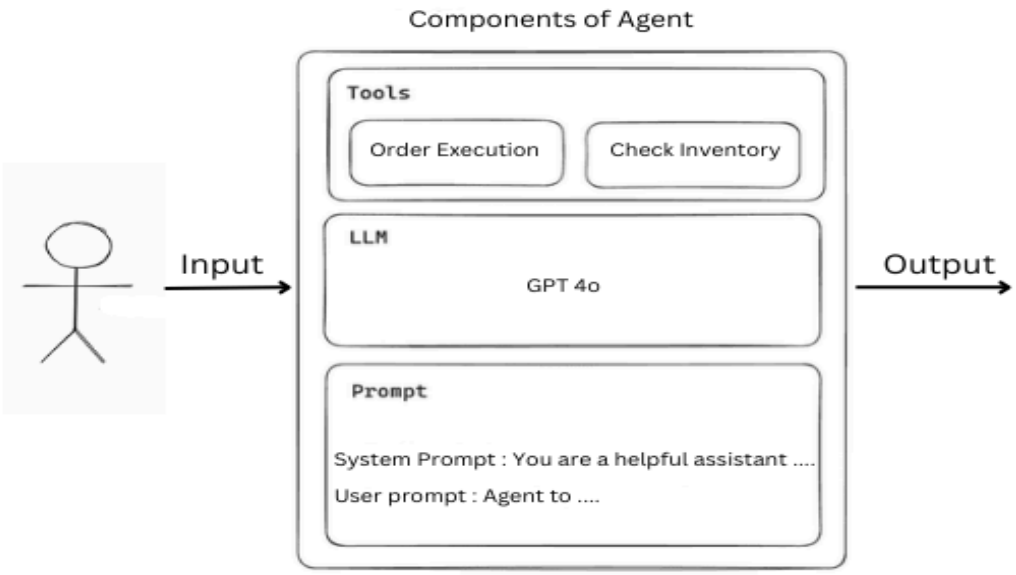
Objective of this Blog 🎯

In this blog, we'll explore how to build an AI-driven agent step-by-step. This agent will interact with users, complete tasks using various tools, and communicate with microservice APIs. We'll also cover how to set up prompts to ensure the agent remains focused on specific tasks, enhancing its efficiency and effectiveness.

Lets Go 🏁

Imagine an e-commerce platform built on a microservices architecture, with one service dedicated to order processing. With the development of LLM agents as task executors, a customer could simply ask a chatbot, "Place order for two laptops." The chatbot would understand the request and automatically place the order by communicating with the order processing microservice.

The diagram below illustrates the agentic architecture which we will be building in this blog.



Step 1 : **Let's install the necessary Python libraries :**

```
Unset

pip install flask requests json pandas snowflake-connector langchain openai
```

Step 2 : Constructing a Python Flask-based order execution API

This API will form the backbone for showcasing how our LLM Agent chatbot can effectively execute orders based on user interactions.

Create the file '*orders.py*'

```python
Python
from flask import Flask, request, jsonify

app = Flask(__name__)

# In-memory order storage
```

```
orders = []

@app.route('/order', methods=['POST'])
def create_order():
    data = request.get_json()
    product = data.get('product')
    quantity = data.get('quantity')

    # Create a new order dictionary
    new_order = {
        'product': product,
        'quantity': quantity
    }

    # Add the order to the in-memory storage
    orders.append(new_order)

    # Return a success message with HTTP status code 201
    return jsonify({'message': 'Order created'}), 201

if __name__ == '__main__':
    app.run(debug=True)
```

Step 3 : Building Tool to execute order

Langchain offers various pre-built tools for tasks like web searching, Slack collaboration, and GitHub management. You can explore a few of them here: Langchain Tools.

But Since we're aiming for a unique interaction with our microservice, we'll create a custom tool. This allows us to tailor the functionality to our specific needs.

Components of a Custom Tool

1. **Decorator (@tool)**:
    - The decorator is used to register the function as a tool.
2. **Tool Name**:
    - The name of the tool is a unique identifier used to reference the tool within the system.
3. **Description**:
    - A brief description of what the tool does. This helps agents to understand the purpose and functionality of the tool.

4. **JSON Schema(Arguments)**:
    - Defines the input structure, including required fields, types, and validation rules.
5. **Result Handling**:
    - Specifies how the tool's result is processed or returned to the user.
6. **Function Implementation**:
    - Contains the core logic of the tool.

Let's explore the construction of our first tool `create_order`!

Python

```python
import requests
from langchain.agents import tool

@tool
def create_order(product: str, quantity: int) -> str:
    """
    Creates an order by sending a POST request to a Flask API endpoint.

    Args:
        product (str): Name of the product.
        quantity (int): Quantity of the product.


    Returns:
        str: A message indicating success or failure based on the API response.
    """
    url = 'http://localhost:5000/order'  # Replace with your actual API
endpoint
    data = {'product': product, 'quantity': quantity}

    try:
        response = requests.post(url, json=data)
        response.raise_for_status()  # Raise an exception for HTTP error
responses

        # Parse the response
        response_data = response.json()
        message = response_data.get('message', 'Order created successfully.')

        return message
    except requests.exceptions.RequestException as e:
        return f"Error creating order: {str(e)}"
```

Step 4 : Building a Inventory Checker Tool

Let’s ensure our agent doesn’t place random orders. Before processing any order, we'll create a tool to verify product availability by querying our Snowflake database to check the current stock.

we are using snowflake, but you can connect with any other databases by changing the connector and connection details.

```python
import pandas as pd
import snowflake.connector
from langchain.agents import tool

snowflake_account = 'abc'
snowflake_user = 'dev'
snowflake_password = 'dev#123'
snowflake_warehouse = 'COMPUTE_'
snowflake_database = 'e_commerce'
snowflake_schema = 'products'
Snowflake_table = 'products_inventory'

conn = snowflake.connector.connect(
    user=snowflake_user,
    password=snowflake_password,
    account=snowflake_account,
    warehouse=snowflake_warehouse,
    database=snowflake_database,
    schema=snowflake_schema
)

@tool
def query_snowflake(user_query: str) -> str:
  """
  This table has a list of products up for sale on my ecommerce website.
  Generates and executes a SQL query based on user input. Note these are the
column in table - PRODUCT,AVAILABLE_QUANTITY
  snowflake_database = 'e_commerce'
  snowflake_schema = 'products'
  Snowflake_table = 'products_inventory'

  Args:
    user_query: The user's query in natural language.

  Returns:
    The query results as a pandas DataFrame.
```

```
    """

    # Basic query generation (replace with more complex logic)
    sql_query = f"{user_query}"

    try:
      cur = conn.cursor()
      cur.execute(sql_query)
      result = cur.fetchall()
      df = pd.DataFrame(result, columns=[col[0] for col in cur.description])
      return df
    except snowflake.connector.errors.DatabaseError as e:
      return f"Snowflake error: {str(e)}"
    except Exception as e:
      return f"Unexpected error: {str(e)}"
```

Step 5 : Setting Up OpenAI Model

For security reasons, ensure that your OpenAI API key is set as an environment variable, not directly in your code.

```
Unset
For linux/Mac OS : export OPENAI_API_KEY="enter-your-your-api-key-here"

For Windows : set OPENAI_API_KEY="enter-your-your-api-key-here"
```

Configure the powerful OpenAI language model.

```
Python
import os
from langchain.chat_models import ChatOpenAI

api_key = os.getenv("OPENAI_API_KEY")

# Set up the turbo LLM
turbo_llm = ChatOpenAI(
    temperature=0,
```

```
    model_name='gpt-4o'
)
```

Step 6 : Build Agent

Now, we're ready to assemble the core of our system: the order-placing agent. This intelligent assistant will leverage the tools and memory we've defined to effectively handle user requests.

To construct this agent, we'll combine the following elements:

- **List Tools:** These are the actions the agent can perform. In our case, we have `query_snowflake` to fetch product information and `create_order` to place orders.
- **Define Memory Window:** This component stores conversation history, allowing the agent to refer to past interactions. We're using `ConversationBufferWindowMemory`.
- **Prompts to the Agent**: We provide system and user prompts to guide the agent's behavior and clarify the task at hand.
- **Agent Type:** Agent types are designed to interact with various tools and environments for specific tasks. We are using the `STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION` agent. This type handles structured chat interactions using a zero-shot approach, reacting to user inputs based on a predefined description without needing specific training examples for each scenario.

```python
from langchain.agents import initialize_agent, AgentType
from langchain.chains.conversation.memory import ConversationBufferWindowMemory

# Tools
tools = [query_snowflake, create_order]


# Memory
memory = ConversationBufferWindowMemory(k=5)

# System prompt
system_prompt = """You are a helpful assistant for placing orders on a
e-commerce website"""

# User prompt
```

```
user_prompt = """Agent to create order based on available product in database,
Don't allow product if not available for sale.
If the product is not in the product_inventory or not available, don't place an
order and give a message to the user product not available.
Leverage your memory & history to access past conversations and decisions when
responding to user queries."""

# Messages
messages = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": user_prompt}
]

# Create the agent
agent = initialize_agent(
    agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,  # Agent type
    messages=messages,  # Initial system and user prompts
    tools=tools,  # Tools available to the agent
    llm=turbo_llm,  # Language model to be used
    verbose=True,  # Print out the agent's reasoning process
    max_iterations=3,  # Maximum number of iterations for the agent
    early_stopping_method="generate",  # Stop the agent when it generates a
final response
    memory=memory  # Memory to store conversation history
)
```

Step 7: Interacting with the Agent

Let's see our agent in action!

Example 1 :

Python

```
agent("Place order for 5 Laptop")
```

Unset

```
> Entering the new AgentExecutor chain...
I need to check the available quantity of laptops before placing the
order. Let me query the database to confirm the availability.
Action:
```
{
  "action": "query_snowflake",
  "action_input": "SELECT PRODUCT, AVAILABLE_QUANTITY FROM
YOUTUBE.PUBLIC.product_inventory WHERE PRODUCT = 'Laptop'"
}
```
Observation:   PRODUCT  AVAILABLE_QUANTITY
0  Laptop                 100
Thought:I have confirmed that there are 100 Laptops available for
purchase. Now, I can proceed to place the order for 5 Laptops.
Action:
```
{
  "action": "create_order",
  "action_input": {"product": "Laptop", "quantity": 5}
}
```
Observation: Order created
Thought:{
  "action": "Final Answer",
  "action_input": "Order for 5 Laptops has been successfully
created."
}

> Finished chain.


{'input': 'Place order for 5 Laptop',
 'history': '',
 'output': '{\n  "action": "Final Answer",\n  "action_input": "Order
for 5 Laptops has been successfully created."\n}'}
```

Example 2 :

```python
agent("which product are up for sale")
```

```
{'input': 'which product are up for sale,
 'history': 'Human: Place order for 5 Laptop\nAI: {\n  "action": "Final
Answer",\n  "action_input": "Order for 5 Laptops has been successfully
created."\n}\nHuman: Did any order is placed for laptops before\nAI: There are
no records of orders placed for laptops in the e-commerce website\'s
database.',
 'output': 'I have retrieved the list of products available for sale on your
e-commerce website. The products and their available quantities are as
follows:\n\n- Laptop: 100 available\in- Smartphone: 50 available\n- Tablet: 25
available\n\nIf you need to create an order for any of these products, feel
free to let me know!'}
```

Wrapping up ✅

By effectively combining Langchain's powerful tools and the flexibility of LLMs, we've constructed a robust order-processing agent. This agent can intelligently handle order requests, interact with databases, and provide real-time feedback to users.

Remember, the possibilities are endless. You can expand this framework by creating additional tools for tasks like product recommendations, inventory management, or customer support. The key is to identify the necessary actions and encapsulate them as Langchain tools.

With this foundation, you're well-equipped to build sophisticated AI-driven applications that revolutionize your web applications.

Related Articles

Introduction to Langchain Agents

Large Language Models: A Comprehensive Exploration

Microservices Architecture