Tools and Agents in LangChain

1. Understanding Large Language Models (LLMs) and Their Limitations

LLMs are powerful Natural Language Processing (NLP) systems with two core capabilities:

- **Reasoning Capability ("To Think"):** LLMs can understand a question, break it down, and plan how to answer it.
- **Language Generation ("To Speak"):** Once an LLM understands how to answer a question, it can generate a word-by-word response.

However, LLMs have significant limitations:

- **Lack of Action/Execution:** LLMs cannot perform tasks in the real world. For example, while an LLM can suggest travel options from Delhi to Mumbai, it cannot book a train ticket.
- **Quote:** In a way, you can say that an LLM is like a human body that has the capability to think and speak, but that human body doesn't have hands or legs, which means it cannot execute any task on its own.
- **Inability to Access Real-time Data:** They cannot fetch live weather data.
- **Unreliable Mathematical Calculations:** While basic arithmetic might work, complex math problems often yield unreliable results because LLMs are trained on language generation, not mathematical problem-solving.
- **No External API Interaction:** They cannot call external APIs (e.g., tweeting on behalf of a user).
- **No Code Execution:** They cannot run code.
- **No Database Interaction:** They cannot interact with databases.

2. What are Tools in LangChain?

Tools are the mechanism to overcome LLM limitations by providing them with the "hands and legs" to perform real-world tasks.

- **Purpose:** A tool is created for a specific task and connected to an LLM. When the LLM needs to perform that task, it uses the tool.
- **Technical Definition:** A tool is just a Python function that is packaged in a way the LLM can understand and call when needed.
- **Functionality:** An LLM will autonomously decide when to use a tool, provide it with the necessary inputs, the tool executes the task, returns the result to the LLM, and the LLM then communicates the completion to the user.
- **Analogy:** If an LLM is a brain that can think and speak, tools give it the ability to "do things." The more tools an LLM has access to, the wider range of tasks it can perform.

3. Types of Tools in LangChain

LangChain offers two main categories of tools:

3.1. Built-in Tools

These are pre-built, production-ready tools provided by the LangChain team for common use cases, requiring minimal or no setup. Users just need to import and use them.

Examples of Built-in Tools:

- **DuckDuckGo Search:** For web searches (e.g., fetching real-time news).
- **Usage Example:** Import DuckDuckGoSearchRun from langchain_community.tools. Create an object and call its invoke() method with a search query.
- **Wikipedia Query Run:** For searching and summarizing Wikipedia articles.
- **Python REPL Tool:** For running raw Python code (e.g., reliable factorial calculation).
- **Shell Tool:** For executing command-line commands on the host machine (e.g., listing directory contents, identifying the current user). *Note: This tool is powerful but risky and should be used with caution in production.***Dependency:** Requires langchain_experimental to be installed.
- **Requests Get Tool:** For making HTTP requests.
- **Gmail Send Message Tool:** For sending emails.
- **Slack Tool, SQL Database Query Tool:** Other domain-specific tools.

Common Attributes of Any Tool: All tools (built-in or custom) possess three key attributes that the LLM uses to understand them:

- **name:** The name of the tool (usually the function name for custom tools).
- **description:** A human-readable description of what the tool does. This is crucial for the LLM to understand when and how to use the tool.
- **args (Arguments Schema):** Defines the required inputs for the tool, including their names and types. This information is typically sent to the LLM as a JSON schema.

3.2. Custom Tools

Custom tools are built by users for specific, unique use cases that are not covered by built-in tools.

Common Scenarios for Creating Custom Tools:

- **Calling Internal Company APIs:** To allow an agent to interact with a company's existing infrastructure (e.g., booking systems, internal databases).
- **Encapsulating Business Logic:** For functionalities specific to a company's unique operations.
- **Interacting with Custom Databases, Products, or Apps:** Enabling agents to communicate with an organization's proprietary systems.

Methods for Creating Custom Tools in LangChain:

There are three primary ways to create custom tools:

a) Using the @tool Decorator (Simplest and Most Common)

1. **Process: Define a Python function:** This function contains the logic for the task the tool will perform.
2. **Add a Docstring:** Highly recommended to provide a clear description of the function's purpose. This docstring becomes the tool's description for the LLM.
3. **Add Type Hinting:** Specify the types of input parameters and the return type. This helps the LLM understand the required inputs and expected outputs, forming the tool's args schema.
4. **Apply @tool Decorator:** Import tool from langchain_core.tools and place it above the function definition. This decorator transforms the function into a LangChain tool that can communicate with an LLM.

- **Example (Multiplication Tool):**from langchain_core.tools import tool

- @tool
- def multiply(a: int, b: int) -> int:
- """Multiplies two numbers."""
- return a * b
- **Invocation:** Once created, a tool object can be invoked using its invoke() method, passing a dictionary of arguments.
- **Benefits:** Simple, straightforward, and sufficient for most experimental and 80-90% of scenarios.

b) Using StructuredTool and Pydantic (More Strict and Mature)

- **Purpose:** This method allows for a more structured and enforced definition of the tool's input schema using Pydantic models. Useful for production-ready agents requiring strict input validation.

1. **Process:Define a Python function** for the tool's logic.
2. **Create a Pydantic BaseModel class:** This class defines the exact structure and types of the tool's input arguments, including descriptions and required fields.
3. **Use StructuredTool.from_function():** This class method is used to create the tool, specifying the function, a custom name, a description, and crucially, the Pydantic model as the args_schema.

- **Benefits:** Stronger type enforcement and better input validation, which is beneficial for complex, production-level applications.

c) Inheriting from BaseTool Class (Most Flexible for Deep Customization)

- **Concept:** BaseTool is an abstract base class that defines the core structure and interface for *all* tools in LangChain. Both @tool decorated functions and StructuredTool implicitly inherit from BaseTool.
- **Purpose:** Allows direct, deep-level customization and the creation of asynchronous versions of tools (e.g., for handling concurrency).

1. **Process:Create a Python class** that inherits from BaseTool.
2. **Define name and description attributes** within the class.
3. **Define args_schema:** This is typically a Pydantic model that defines the input schema, similar to StructuredTool.
4. **Implement the _run() method:** This is the core method where the tool's synchronous execution logic resides.
5. **(Optional) Implement the _arun() method:** For asynchronous execution.

- **Benefits:** Provides the highest level of control and is essential for applications requiring advanced features like concurrency handling.

4. Toolkits: Collections of Related Tools

A **Toolkit** is a collection of related tools that serve a common purpose, packaged together for convenience and reusability.

- **Purpose:** Organize multiple tools that logically belong together (e.g., all Google Drive related tools: upload, search, read).
- **Benefit:** Enhances reusability across different applications.

1. **Implementation: Create individual tools:** (e.g., using the @tool decorator).

2. **Create a Toolkit class:** This class contains a get_tools() method that returns a list of the tools that are part of the toolkit.

- **Example (Math Toolkit):** A toolkit containing add and multiply tools, as both perform arithmetic operations.

5. Tools and Agents: The Core Connection

AI Agents are LLM-powered systems that can **autonomously think, decide, and take actions** using external tools or APIs to achieve a goal.

- **Quote:** An AI agent is an LLM-powered system that can autonomously think, decide, and take actions using external tools or APIs to achieve a goal.
- **Agent Capabilities: Reasoning & Decision-Making:** Provided by the LLM. The LLM thinks step-by-step to solve a problem.
- **Action Performance:** Powered by Tools. Once the LLM decides on a course of action, a tool executes it.
- **Conclusion:** The marriage of LLMs and Tools is what we call an Agent. Therefore, a strong understanding of tools is fundamental for building agents with LangChain and LangGraph.

Next Steps: "Tool Calling," which focuses on how LLMs and tools are connected and interact to enable agents to perform tasks.