Tool Calling and AI Agents in LangChain

1. The Core Problem: LLM Limitations

Large Language Models (LLMs) possess two significant capabilities:

- **Reasoning:** LLMs can "breakdown and understand what is being asked of them."
- **Output Generation:** Based on their understanding, LLMs "access their parametric knowledge and generate an answer."

However, LLMs have a critical limitation: **they cannot perform actions or tasks on their own.** As the source states, "LLMs, although they are good at reasoning and output generation, they cannot do tasks on their own." This is likened to a human who can think and speak but "does not have hands and feet" to perform physical actions. Examples of tasks LLMs *cannot* do natively include:

- Making changes to a database.
- Posting on social media platforms like LinkedIn or Twitter.
- Hitting an API to fetch real-time data, such as current weather conditions.

To make LLMs truly powerful and capable of performing tasks in the software world, they need "hands and feet" – which are provided by **Tools**.

2. Introduction to Tools

- **Definition:** Tools are explicitly created functionalities (essentially special Python functions) that enable LLMs to interact with the external world and perform specific tasks.
- **Purpose:** Each tool is designed to carry out a particular task. Examples mentioned include:
- DuckDuckGo Search: For searching the DuckDuckGo search engine.
- Shell Tool: For running command-line commands.
- **Customization:** Programmers can create custom tools tailored to their specific needs.
- **Tool Specification:** When creating a tool, key information must be provided:
- **Name:** The tool's identifier.
- **Description:** A clear explanation of what the tool does, allowing the LLM to understand its function.
- **Input Schema:** Defines the format and type of input the tool expects.

3. Tool Binding: Connecting LLMs and Tools

- **Definition:** Tool Binding is the process of "registering tools with a Large Language Model." This step makes the LLM aware of the tools at its disposal.
- **Functionality Gained:** Through tool binding, the LLM learns:
- "What tools are available."
- "What each tool does" (via its description).
- "What input format to use" (via its input schema).
- **Mechanism:** LLMs in LangChain have a bind_tools() function that takes a list of tools as an argument. The result of this binding is a new LLM instance (LLM with tools) that now has access to the specified tools.

- **LLM Compatibility:** Not all LLMs support tool binding; only specific models possess this capability.

4. Tool Calling: LLM Decision-Making

- **Definition:** Tool Calling is the process where "the LLM decides during a conversation or task that it needs to use a specific tool and generates a structured output with the name of the tool and the arguments to call it with."
- **When it Happens:**If a query can be answered using the LLM's inherent knowledge (e.g., "Hi, how are you?"), no tool is called.
- If a query requires an external action or specific computation (e.g., "What's 8 multiplied by 7?"), the LLM identifies the relevant tool(s) from its bound set.
- **Output:** When an LLM decides to call a tool, its output is not a direct textual answer but a structured "Tool Call" object. This object specifies:
- name: The name of the tool to be called (e.g., multiply).
- args: The input arguments for that tool, derived from the user's query and the tool's input schema (e.g., {'A': 8, 'B': 7}).
- Other metadata like id and type.
- **Crucial Distinction:** It is vital to understand that **"The LLM does not actually run the tool. It just suggests the tool and the input arguments."** The actual execution is handled by the LangChain framework or the programmer. This design choice provides a safety layer, preventing LLMs from executing potentially risky or incorrect actions autonomously.

5. Tool Execution: Running the Tools

- **Definition:** Tool Execution is the step where "the actual Python function (or tool) is run using the input arguments that the LLM suggested during Tool Calling."
- **Programmer's Role:** After the LLM suggests a tool call, the programmer (or the LangChain agent framework) takes this structured output and explicitly invokes the suggested tool with the provided arguments.
- **Output of Execution:** The result of tool execution is a **Tool Message**.
- **Tool Message:** A special type of message in LangChain that encapsulates the output of a tool's execution.
- **Maintaining Context:** The Tool Message is crucial because it can be sent back to the LLM. This provides the LLM with the necessary context (the result of the tool's action) to generate a final, coherent response to the original user query. This involves maintaining a "conversation history" that includes human messages, AI messages (tool calls), and tool messages (tool execution results).

6. Practical Application: Real-time Currency Conversion

The video demonstrates building a practical application to convert currency in real-time using LangChain tools.

- **Problem:** LLMs have outdated conversion rates stored in their parametric knowledge. To get real-time rates, an external API is needed.
- **Solution:** Two tools are created:
- **get_conversion_factor:Purpose:** Hits an external API (ExchangeRate-API) to fetch the real-time conversion factor between a base and target currency.
- **Inputs:** base_currency (string), target_currency (string).
- **Output:** float (the conversion rate).
- **Mechanism:** Uses the requests Python library to make an HTTP GET request to the API.

- **convert:Purpose:** Multiplies a base currency value by a given conversion rate to calculate the target currency value.
- **Inputs:** base_currency_value (integer), conversion_rate (float).
- **Output:** float.
- **Handling Sequential Tool Calls (Injected Tool Arguments):Challenge:** If a query asks for both the conversion factor AND the conversion of a value (e.g., "What is the conversion factor between USD and INR and convert 10 USD to INR?"), the LLM might try to call both tools simultaneously. For the convert tool, it might "hallucinate" or use an outdated conversion_rate from its training data, because the get_conversion_factor tool hasn't been executed yet to provide the *real-time* rate.
- **Solution: Annotated[float, InjectableToolArgument]**: By marking the conversion_rate argument of the convert tool as an InjectableToolArgument, the programmer tells the LLM: "Do not try to fill this argument. I, the developer, will inject this value after running earlier tools." This ensures that the LLM only suggests the base_currency_value for the convert tool, leaving the conversion_rate to be programmatically inserted after get_conversion_factor is executed.

1. **Workflow for Complex Queries:Human Message:** User asks a multi-step query.
2. **LLM Invocation:** The LLM (with bound tools) is invoked with the human message.
3. **AI Message (Tool Calls):** The LLM returns an AI Message containing *multiple* tool calls (e.g., first get_conversion_factor, then convert with a missing conversion_rate).

- **Iterative Execution:**The program loops through the tool_calls suggested by the LLM.
- For get_conversion_factor, it invokes the tool, extracts the real-time conversion_rate from the Tool Message output, and appends the Tool Message to the conversation history.
- For convert, it takes the extracted conversion_rate and **injects** it into the args of the convert tool call. Then, it invokes the convert tool and appends its Tool Message output to the conversation history.

1. **Final LLM Invocation:** The entire updated conversation history (including human message, AI message with tool calls, and subsequent tool messages with execution results) is sent back to the LLM.
2. **Final Answer:** The LLM uses this complete context to generate a coherent, real-time answer.

7. Distinction: Tool Calling vs. AI Agents

While the application demonstrates powerful use of tools and tool calling, it is **not yet a full AI Agent**.

- **Reason:** The example application still requires significant manual decision-making and coding by the programmer (e.g., explicitly looping through tool calls, injecting arguments, managing conversation history).
- **Characteristic of an AI Agent:** An AI Agent is **autonomous**. It "breaks down a problem itself and solves it step by step, and it does not need anyone's help in between."
- **Agent Workflow:** A true agent would:

1. Receive a user query (e.g., "Convert 10 USD to INR").
2. **Autonomously reason:** "I don't know the conversion rate, so first I need to call get_conversion_factor."
3. Execute get_conversion_factor, get the rate.
4. **Autonomously reason:** "Now I know the rate. Next, I should call convert with 10 and 85.34."
5. Execute convert, get the final result, and present it.