

Cheat Sheet: Manual Tool Calling in LangChain

Estimated time needed: 20 minutes

I. What is manual tool calling?

Manual tool calling in LangChain gives you precise control over how external tools are used. Instead of relying on the LLM to autonomously invoke tools, developers parse the LLM's output to extract tool calls, validate inputs, and execute functions manually.

This approach is particularly beneficial in production environments where reliability, security, and auditability are paramount.

II. Key concepts

| Term | Definition |
|------------------------|---|
| Tool | A Python function paired with a schema that defines its name, description, and expected arguments. Tools can be created using the <code>@tool</code> decorator or by defining a class inherited from <code>BaseTool</code> . |
| Tool Schema | A structured definition (often using Pydantic models) that spells out exactly what input a tool expects. It helps ensure the information is correct and easy to work with. |
| Tool Call | An instruction generated by the LLM indicating which tool to invoke and with what arguments. Typically represented in a structured format like JSON. |
| Automatic Tool Calling | The model autonomously decides to invoke tools based on the input and handles execution without developer intervention. |
| Manual Tool Calling | The developer/user intercepts the model's tool call suggestions, validates inputs, and executes the tools, providing greater control over the process. |
| AIMessage | A message type that represents the model's response, which may include tool call instructions in the <code>.tool_calls</code> attribute. |
| ToolMessage | A message type used to convey the result of a tool execution back to the model, maintaining context and enabling the model to generate informed subsequent responses, containing the tool output and associated <code>tool_call_id</code> . |
| tool_call_id | A unique identifier for each tool call, allowing the system to match the tool's output (ToolMessage) with the corresponding request (AIMessage). This is particularly useful when handling multiple tool calls concurrently. |

III. How to manually call a tool

Here's a step-by-step look at how you can take control and manually call a tool.

| | |
|-------------------------|--|
| Define your tools | First, define a simple tool using the handy <code>@tool</code> shortcut. <pre>from langchain_core.tools import tool @tool def multiply(a: int, b: int) -> int: """Multiply two numbers.""" return a * b</pre> |
| Bind tools to the model | Attach the tools to a chat model that supports tool calling. <pre>model_with_tools = model.bind_tools([multiply])</pre> |
| Parse tool calls | After invoking the model, parse its output to extract tool calls. <pre>response = model_with_tools.invoke("What is 2 multiplied by 3?") tool_calls = response.tool_calls # Contains tool name and arguments</pre> <p>This is what <code>tool_calls</code> looks like:</p> <pre>[{'name': 'multiply', 'args': {'a': 2, 'b': 3}, 'id': 'chatmpl-tool-94d27a8e35b44212bfe6c8d26553c149', 'type': 'tool_call'}]</pre> |
| Validate tool arguments | Use a Pydantic model or perform a manual check to validate inputs. <pre>from pydantic import BaseModel class MultiplyInput(BaseModel): a: int b: int validated_input = MultiplyInput(**tool_calls[0]['args'])</pre> |

This is what `validated_input` looks like:

```
MultiplyInput(a=2, b=3)
```

There are two ways tools return results:

- `tool.invoke(args_dict)` → Returns a raw result (for example, 6)
- `tool.invoke(tool_call_object)` → Returns a `ToolMessage` automatically

The "manual" part of manual tool calling represents best practices for production tool calling, where you want control over execution:

- **Deciding WHETHER** to execute the tool calls (security, validation, business logic)
- **Choosing WHICH** tools to execute
- **Controlling WHEN** to execute them

When you invoke a LangChain Tool with a `ToolCall` object, you automatically get back a `ToolMessage`, so you don't need to manually create a `ToolMessage` in most cases.

`ToolMessage` is used to maintain context and state throughout the conversation between the user and the model.

`ToolMessages` are essential for feeding tool results back to the LLM so it can continue the conversation. The purpose is **NOT** just getting the tool result, but feeding that result back to the LLM so it can:

- See what the tool returned
- **Continue the conversation** with that context
- **Give a final answer** to the user

```
from langchain_core.messages import HumanMessage, AIMessage, ToolMessage
# Complete conversation flow
messages = [
    HumanMessage("What is 2 multiplied by 3?"),
    AIMessage("I'll use the multiply tool", tool_calls=[{"name": "multiply", "args": {"a": 2, "b": 3}, "id": "call_123"}]),
    ToolMessage(content="6", tool_call_id="call_123") # This tells LLM the result!
]
final_response = model.invoke(messages)
# LLM: "The result of 2 multiplied by 3 is 6."
```

When you run the statement below, it's not truly a manual tool calling. It's more like a "semi-automatic" tool calling because the tools are always executed. When you run it, the output is returned wrapped in an `AIMessage`.

```
result = multiply.invoke(validated_input.model_dump())
```

Here is how you can implement controlled tool calling:

```
from langchain_core.messages import ToolMessage
# Manual decision making and validation for multiply tool
for tool_call in response.tool_calls:
    if tool_call['name'] == 'multiply':
        # Check if we should execute this specific call
        a, b = tool_call['args']['a'], tool_call['args']['b']
        # Example: Only allow positive number multiplication
        if a > 0 and b > 0:
            # Validate and execute
            validated_input = MultiplyInput(**tool_call['args'])
            tool_msg = multiply.invoke(tool_call)
            messages.append(tool_msg)
        else:
            # Reject negative numbers
            error_msg = ToolMessage(
                content="Multiplication with negative numbers not allowed",
                tool_call_id=tool_call['id']
            )
            messages.append(error_msg)
    else:
        # Skip unknown tools
        skip_msg = ToolMessage(
            content=f"Tool '{tool_call['name']}' execution skipped",
            tool_call_id=tool_call['id']
        )
        messages.append(skip_msg)
```

Let's look at what makes the above code "manual"?

- Conditional execution:** You decide whether to run the tool. You don't just run every tool the LLM asks for.

```
# MANUAL DECISION: Should we execute this tool call?  
if a > 0 and b > 0:  
    # YES - execute the tool  
    tool_msg = multiply.invoke(tool_call)  
  
else:  
    # NO - reject the tool call  
    error_msg = ToolMessage(content="Multiplication with negative numbers not  
allowed")
```

ii. Custom business logic: You've implemented a rule that overrides the LLM's decision. The LLM might want to multiply negative numbers, but you decide it's not allowed.

```
# CUSTOM RULE: Only allow positive number multiplication  
if a > 0 and b > 0:
```

iii. Tool filtering: You can decide which tools are allowed to run. Any tools you don't recognize or approve get skipped.

```
if tool_call['name'] == 'multiply':  
    # Handle multiply tool  
else:  
    # DECISION: Skip unknown tools  
    skip_msg = ToolMessage(content=f"Tool '{tool_call['name']}' execution skipped")
```

iv. Custom error handling: Instead of letting a tool run and possibly fail, you can proactively reject it with a custom error message that makes more sense to the user.

Author(s)

IBM Skills Network Team



Skills Network