Using LLM Function Calling for AI-Agent Interaction

Step 3: Using LLM Function Calling for Structured Execution

One of the most challenging aspects of integrating AI agents with tool execution is ensuring that the model consistently produces structured output that can be parsed correctly. Traditionally, developers would attempt to engineer prompts to make the model output well-formed JSON, but this approach is unreliable—models can introduce variations, omit required fields, or output unstructured text that breaks parsing logic.

To solve this, most LLMs offer **function calling APIs** that guarantee structured execution. Instead of treating function execution as a free-form text generation task, function calling APIs allow us to explicitly define the tools available to the model using **JSON Schema**. The model then decides when and how to call these functions, ensuring structured and predictable responses.

How Function Calling Simplifies AI-Agent Interfaces

When using function calling, the model returns either:

1. **A function call** that includes the tool name and arguments as structured JSON.
2. **A standard text response** if the model decides a function is unnecessary.

This approach removes the need for manual prompt engineering to enforce structured output and allows the agent to focus on decision-making rather than syntax compliance.

Example: Executing Function Calls

Below is a typical way to invoke function calling using OpenAI's API:

```
import json
import os
from typing import List

from litellm import completion
```

```python
def list_files() -> List[str]:
    """List files in the current directory."""
    return os.listdir(".")

def read_file(file_name: str) -> str:
    """Read a file's contents."""
    try:
        with open(file_name, "r") as file:
            return file.read()
    except FileNotFoundError:
        return f"Error: {file_name} not found."
    except Exception as e:
        return f"Error: {str(e)}"


tool_functions = {
    "list_files": list_files,
    "read_file": read_file
}

tools = [
    {
        "type": "function",
        "function": {
            "name": "list_files",
            "description": "Returns a list of files in the
directory.",
            "parameters": {"type": "object", "properties": {},
"required": []}
        }
    },
    {
        "type": "function",
        "function": {
            "name": "read_file",
            "description": "Reads the content of a specified file in
the directory.",
            "parameters": {
                "type": "object",
                "properties": {"file_name": {"type": "string"}},
                "required": ["file_name"]
            }
        }
    }
]

# Our rules are simplified since we don't have to worry about
```

```
getting a specific output format
agent_rules = [{
    "role": "system",
    "content": """
You are an AI agent that can perform tasks by using available tools.

If a user asks about files, documents, or content, first list the
files before reading them.
"""
}]

user_task = input("What would you like me to do? ")

memory = [{"role": "user", "content": user_task}]

messages = agent_rules + memory

response = completion(
    model="openai/gpt-4o",
    messages=messages,
    tools=tools,
    max_tokens=1024
)

# Extract the tool call from the response, note we don't have to
parse now!
tool = response.choices[0].message.tool_calls[0]
tool_name = tool.function.name
tool_args = json.loads(tool.function.arguments)
result = tool_functions[tool_name](**tool_args)

print(f"Tool Name: {tool_name}")
print(f"Tool Arguments: {tool_args}")
print(f"Result: {result}")
```

Breaking Down Function Calling Step by Step

Let's examine how function calling works in detail:

1. Define the Tool Functions

```
def list_files() -> List[str]:
    """List files in the current directory."""
    return os.listdir(".")

def read_file(file_name: str) -> str:
    """Read a file's contents."""
    try:
        with open(file_name, "r") as file:
            return file.read()
    except FileNotFoundError:
        return f"Error: {file_name} not found."
    except Exception as e:
        return f"Error: {str(e)}"
```

First, we define the actual Python functions that will be executed. These contain the business logic for each tool and handle the actual operations the AI agent can perform.

2. Create a Function Registry

```
tool_functions = {
    "list_files": list_files,
    "read_file": read_file
}
```

We maintain a dictionary that maps function names to their corresponding Python implementations. This registry allows us to easily look up and execute the appropriate function when the model calls it.

3. Define Tool Specifications Using JSON Schema

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "list_files",
            "description": "Returns a list of files in the
directory.",
            "parameters": {"type": "object", "properties": {},
"required": []}
        }
    },
    {
        "type": "function",
        "function": {
            "name": "read_file",
            "description": "Reads the content of a specified file in
the directory.",
            "parameters": {
                "type": "object",
                "properties": {"file_name": {"type": "string"}},
                "required": ["file_name"]
            }
        }
    }
]
```

This is where we describe our tools to the model. Each tool specification includes:

- A **name** that matches a key in our `tool_functions` dictionary
- A **description** that helps the model understand when to use this tool
- **Parameters** defined using JSON Schema, specifying the expected input format

Note how the `list_files` function takes no parameters (empty "properties" object), while `read_file` requires a "file_name" string parameter. The model will use these specifications to generate properly structured calls.

4. Set Up the Agent's Instructions

```
agent_rules = [{
    "role": "system",
    "content": """
You are an AI agent that can perform tasks by using available tools.

If a user asks about files, documents, or content, first list the
files before reading them.
"""
}]
```

The system message provides guidance on how the agent should behave. With function calling, we don't need to instruct the model on how to format its outputs - we only need to focus on the decision-making logic.

5. Prepare the Conversation Context

```
user_task = input("What would you like me to do? ")
memory = [{"role": "user", "content": user_task}]
messages = agent_rules + memory
```

We combine the system instructions with the user's input to create the conversation context.

6. Make the API Call with Function Definitions

```
response = completion(
    model="openai/gpt-4o",
    messages=messages,
    tools=tools,
    max_tokens=1024
)
```

The critical difference here is the inclusion of the `tools` parameter, which tells the model what functions it can call. This is what activates the function calling mechanism.

7. Process the Structured Response

```
tool = response.choices[0].message.tool_calls[0]
tool_name = tool.function.name
tool_args = json.loads(tool.function.arguments)
```

When using function calling, the response comes back with a dedicated `tool_calls` array rather than free-text output. This ensures that:

- The **function name** is properly identified

- The **arguments** are correctly formatted as valid JSON
- We don’t need to parse or extract from unstructured text

8. Execute the Function with the Provided Arguments

```
result = tool_functions[tool_name](**tool_args)
```

Finally, we look up the appropriate function in our registry and call it with the arguments the model provided. The `**tool_args` syntax unpacks the JSON object into keyword arguments.

Key Benefits of Function Calling APIs

1. **Eliminates prompt engineering for structured responses** – No need to force the model to output JSON manually.
2. **Uses standardized JSON Schema** – The same format used in API documentation applies seamlessly to AI interactions.
3. **Allows mixed text and tool execution** – The model can decide whether a tool is necessary or provide a natural response.
4. **Simplifies parsing logic** – Instead of handling inconsistent outputs, developers only check for `tool_calls` in the response.
5. **Guarantees syntactically correct arguments** – The model automatically ensures arguments match the expected parameter format.

Conclusion

Function calling APIs significantly improve the reliability of AI-agent interactions by enforcing structured execution. By defining tools with JSON Schema and letting the model determine when to use them, we build a more predictable and maintainable AI environment interface. In the next section, we will explore **how to register these tools dynamically using decorators** to further streamline agent development.