Simplifying the AI Agent Loop with Function Calling

Function Calling and the Simplified Agent Loop

Now that we have explored function calling, we can significantly simplify our AI agent loop. Traditionally, parsing the LLM’s responses required engineering strict output formats, validating them, and handling errors when the model deviated from the expected structure. With function calling, the model now natively supports structured responses, ensuring we receive either a valid function call or a standard text reply.

This dramatically reduces the complexity of parsing responses and handling actions. Instead of manually extracting commands from unstructured text, we can directly interpret the function calls provided by the model and execute them with confidence.

How Function Calling Simplifies the Agent Loop

Let’s revisit the agent loop, but now using function calling. Instead of handling multiple layers of parsing, we let the model’s API return structured function calls when needed.

```python
import json
import os
from typing import List

from litellm import completion

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)}"
```

```python
def terminate(message: str) -> None:
    """Terminate the agent loop and provide a summary message."""
    print(f"Termination message: {message}")

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

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"]
            }
        }
    },
    {
        "type": "function",
        "function": {
            "name": "terminate",
            "description": "Terminates the conversation. No further 
actions or interactions are possible after this. Prints the provided 
message for the user.",
            "parameters": {
                "type": "object",
                "properties": {
                    "message": {"type": "string"},
                },
                "required": ["message"]
            }
        }
    }
]

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.

When you are done, terminate the conversation by using the
"terminate" tool and I will provide the results to the user.
"""
}]

# Initialize agent parameters
iterations = 0
max_iterations = 10

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

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

# The Agent Loop
while iterations < max_iterations:

    messages = agent_rules + memory

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

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

        action = {
            "tool_name": tool_name,
            "args": tool_args
        }
```

```
            if tool_name == "terminate":
                print(f"Termination message: {tool_args['message']}")
                break
            elif tool_name in tool_functions:
                try:
                    result = {"result": tool_functions[tool_name]
(**tool_args)}
                except Exception as e:
                    result = {"error":f"Error executing {tool_name}:
{str(e)}"}
            else:
                result = {"error": f"Unknown tool: {tool_name}"}

            print(f"Executing: {tool_name} with args {tool_args}")
            print(f"Result: {result}")
            memory.extend([
                {"role": "assistant", "content": json.dumps(action)},
                {"role": "user", "content": json.dumps(result)}
            ])
        else:
            result = response.choices[0].message.content
            print(f"Response: {result}")
            break
```

What This Changes

1. **No More Custom Parsing Logic** - We don't need to engineer strict text output parsing; the model always returns structured JSON for function calls.
2. **Dynamic Execution** - Instead of a rigid loop that manually checks what the agent should do, we simply read and execute the function call.
3. **Unified Text & Action Handling** - If no function call is needed, the model responds with a message, allowing mixed conversational and action-driven workflows.
4. **Automated Function Execution** - The agent dynamically maps the tool name from the model to its corresponding Python function and executes it with the provided arguments.

Errors Can Still Occur

While function calling improves structured execution, it does not eliminate all potential issues. One common problem is that the model may still sometimes return improperly formatted JSON or a tool call that isn't valid. For example, the response could contain syntax errors, missing required fields, or incorrectly formatted values. If the agent attempts to parse this malformed JSON, it will fail.

To address this, we could add error handling around `json.loads()` to catch `json.JSONDecodeError`. If a parsing error occurs, the agent retries by sending another request. This prevents the entire loop from crashing and ensures robustness in execution. You can do this as a practice exercise to enhance the agent's reliability.

Additionally, even when the JSON is correctly formatted, errors can still arise at runtime due to missing files, incorrect arguments, or unexpected edge cases in function execution. Our error handling ensures that these issues do not halt execution entirely but instead provide meaningful feedback for debugging and improvement.

Key Takeaways

By leveraging function calling, we remove unnecessary complexity from the agent loop, allowing the AI to interact with its environment more reliably. This simplification makes AI agents more robust, scalable, and easier to integrate into real-world applications.

In a later lesson, we will explore **how to register tools dynamically with decorators**, further improving flexibility and maintainability in AI agent design.