Structuring LLM Tool Calls with Pydantic and JSON Serialization

Skills Network

Estimated Reading Time: 10 minutes

Why This Matters

We know that LLMs can output text, and when we bind tools to our LLM, we can extract parameters and call functions to process the output or perform specific tasks. A schema or data model is a code-level extension of this idea—it allows you to enforce that the LLM outputs data in a specific format, such as a Python class, dictionary, or JSON. This ensures that if you need to feed the LLM output into an API, database, or another function, the output is structured, predictable, and integrates seamlessly with other systems.

Consider a weather API example: it might expect data such as weather conditions ("sunny", "rainy", "cloudy"), temperature as an integer, and the temperature unit ("celsius" or "fahrenheit").

For this, you could create a Pydantic class like this:

```
from pydantic import BaseModel, Field
class WeatherSchema(BaseModel):
    condition: str = Field(description="Weather condition such as sunny, rainy, cloudy")
    temperature: int = Field(description="Temperature value")
    unit: str = Field(description="Temperature unit such as fahrenheit or celsius")
```

You can then bind this schema as a tool to your LLM and call the LLM as follows:

```
from langchain_openai import ChatOpenAI
# Create an LLM instance
llm = ChatOpenAI(model="gpt-4.1-nano")  # or your preferred model
weather_llm = llm.bind_tools(tools=[WeatherSchema])
response = weather_llm.invoke("It's sunny and 75 degrees")
# Returns: {"condition": "sunny", "temperature": 75, "unit": "fahrenheit"}
```

In the output, the response of one of the attributes will be a dictionary of key-value pairs the weather API will use.

Similarly, for a spam detection task, you could define a schema with classification, confidence score, and reasoning as follows:

```
class SpamSchema(BaseModel):
    classification: str = Field(description="Email classification: spam or not_spam")
    confidence: float = Field(description="Confidence score between 0 and 1")
    reason: str = Field(description="Reason for the classification")
spam_llm = llm.bind_tools(tools=[SpamSchema])
response = spam_llm.invoke("I'm a Nigerian prince, you want to be rich")
# Returns: {"classification": "spam", "confidence": 0.95, "reason": "Nigerian prince scam"}
```

Note: These are conceptual examples to demonstrate the approach.

The schema is one of many attributes in the `AIMessage`.

Let's better understand it by extracting it from a LLM response.

Real Example: Addition Tool with Pydantic and LangChain

```
In a real-world example, we might ask a language model to book a flight, where the schema could include fields like destination, starting point, time
```

```
from pydantic import BaseModel, Field
from langchain_core.messages import HumanMessage
from langchain_openai import ChatOpenAI
# Define BaseModel class for addition
class Add(BaseModel):
    """Add two numbers together"""
    a: int = Field(description="First number")
    b: int = Field(description="Second number")
# Setup LLM and bind the Add tool
llm = ChatOpenAI(model="gpt-4.1-nano")
initial_chain = llm.bind_tools(tools=[Add])
# Ask LLM to add numbers
question = "add 1 and 10"
response = initial_chain.invoke([HumanMessage(content=question)])
# Extract and calculate from the LLM response
def extract_and_add(response):
    tool_call = response.tool_calls[0]
    a = tool_call["args"]['a']
    b = tool_call["args"]['b']
    return a + b
# Execute and print results
result = extract_and_add(response)
print(f"LLM extracted: a={response.tool_calls[0]['args']['a']}, b={response.tool_calls[0]['args']['b']}")
print(f"Result: {result}")
```

Why Use Pydantic Models for LLM Tool Calls?

In applications where LLMs call external tools or functions—such as in agentic systems or tool-augmented reasoning—it is critical that both inputs and outputs are:

- Structured
- Validated
- Easily serialized to and from JSON

Pydantic lets you define Python classes with strict type validation and built-in JSON serialization, which helps ensure reliable data exchange between your LLM and other systems.

Example: Defining Reusable Math Tool Schemas

```
from pydantic import BaseModel
from typing import Literal
class TwoOperands(BaseModel):
    a: float
    b: float
class AddInput(TwoOperands):
    operation: Literal['add']
class SubtractInput(TwoOperands):
    operation: Literal['subtract']
class MathOutput(BaseModel):
    result: float
```

Tool Functions Using Pydantic Models

```
def add_tool(data: AddInput) -> MathOutput:
    return MathOutput(result=data.a + data.b)
def subtract_tool(data: SubtractInput) -> MathOutput:
    return MathOutput(result=data.a - data.b)
```

Dispatching Tool Calls from JSON Input

```
incoming_json = '{"a": 7, "b": 3, "operation": "subtract"}'
def dispatch_tool(json_payload: str) -> str:
    base = SubtractInput.parse_raw(json_payload)
    if base.operation == "add":
        output = add_tool(AddInput.parse_raw(json_payload))
    elif base.operation == "subtract":
        output = subtract_tool(SubtractInput.parse_raw(json_payload))
    else:
        raise ValueError("Unsupported operation")
    return output.json()
result_json = dispatch_tool(incoming_json)
print(result_json)  # {"result": 4.0}
```

What Does `Literal` Do?

The `Literal` type from Python's `typing` module restricts a variable to one or more specific constant values. In the examples above, it ensures that the `operation` field only accepts 'add' or 'subtract'. This helps validate that the input operation matches a known tool and prevents invalid operations from reaching your system.

Here is an example:

```
from typing import Literal
# Define a schema with Literal to restrict operation types
class CalculatorSchema(BaseModel):
    operation: Literal['add', 'subtract', 'multiply', 'divide'] = Field(
        description="The mathematical operation to perform"
    )
    a: float = Field(description="First number")
    b: float = Field(description="Second number")

calculator_llm = llm.bind_tools(tools=[CalculatorSchema])
# Test with valid operations
response1 = calculator_llm.invoke("Add 15 and 23")
print(response1.tool_calls[0]['args'])
# Output: {"operation": "add", "a": 15.0, "b": 23.0}
response2 = calculator_llm.invoke("Multiply 7 by 8")
print(response2.tool_calls[0]['args'])
# Output: {"operation": "multiply", "a": 7.0, "b": 8.0}
```

Why JSON-Serializable Pydantic Models Are Powerful

Feature	Benefit
Type Validation	Ensures inputs and outputs conform to expected schema
Reusability	Use base classes such as `TwoOperands` across multiple tools
JSON Serialization	`.json()` and `.parse_raw()` simplify tool chaining and I/O
Extensibility	Easily add more tools (e.g., multiply, divide)

Final Thoughts and Alternatives

Using Pydantic models to define JSON-serializable tool inputs and outputs makes your LLM applications:

- Robust and error-proof.
- Compatible with orchestration frameworks such as LangChain, CrewAI, Watsonx, and so on.
- Easy to test, maintain, and extend.

This forms the foundation of building reliable, multi-tool LLM agents.

Optional Note: Pydantic vs. Python Dataclasses

You **don't need** to use Pydantic exclusively. Since Python 3.7, `dataclasses` offer a lightweight alternative for defining data models with built-in parsing and serialization. However, Pydantic provides more advanced features such as:

- Validators for complex constraints
- Extra configuration options
- Better integration with libraries like LangChain and FastAPI

Pydantic is generally recommended due to its maturity and feature set, and it's the default choice in popular frameworks.

Author(s)

Faranak Heidari
Karan Goswami