LLM Tool Pattern Implementation Report

Chong Chen

November 11, 2024

1 Introduction

In recent years, Large Language Models (LLMs) have advanced significantly in their ability to process natural language and generate human-like responses. However, despite their extensive training data, these models are limited to the information available at their training cutoff date and cannot interact with real-time data or perform tasks that require external processing. To address this limitation, the tool pattern allows LLMs to interact with external functions and APIs, enabling them to access up-to-date information and perform dynamic operations.

This assignment focuses on implementing a tool pattern that enables an LLM to interact with external functions through XML-structured prompts. Specifically, the objective is to create a system in which LLMs can utilize tools for various purposes, such as retrieving data, performing calculations, and interacting with external APIs. By using a structured XML-based format, the system can accurately parse tool calls from the model, validate inputs, and handle errors, ensuring reliable and robust interactions.

The key components of this assignment include:

- **Tool System Prompt:** A structured prompt format using XML tags that defines the available tools and how the LLM can call them.
- **Tool Definition Format:** A standardized format for defining tool metadata, such as function name, parameters, and return types.
- **Tool Decorator:** A Python decorator to wrap functions as tools, allowing for type validation and XML parsing.
- **Tool Agent:** A centralized agent that manages tool execution, handles user queries, and formats responses.

This report will detail the design and implementation of the tool pattern system, covering the structure of the Tool System Prompt, the Tool Definition Format, the implementation of the Tool Decorator, and the functionality of the Tool Agent. In addition, three tools will be developed to demonstrate different aspects of tool usage, including API interaction, data processing, and external service integration. Basic tests will be implemented to validate each tool's functionality and error handling capabilities.

This project demonstrates how LLMs can be extended beyond their training limitations, making them more versatile and capable of handling real-world applications that require up-to-date data and complex interactions.

2 Design and Implementation

2.1 Tool System Prompt

2.1.1 The TOOL_SYSTEM_PROMPT and its Purpose

The `TOOL_SYSTEM_PROMPT` is a template string that serves as an instructional message to a function-calling AI model, guiding it on how to interact with the tools provided. It specifies that the AI model should not make assumptions about the arguments but rather adhere strictly to the expected data types and formats. This prompt includes instructions for the AI to use the tool functions provided in an XML-based format. The goal of this prompt is to ensure that the AI model understands the tools available and their usage, enabling it to accurately formulate function calls.

2.1.2 Details on XML Tags

The XML tags `<tool_call></tool_call>` and `<tools></tools>` encapsulate the tool call responses and available tool definitions, respectively. Each function call initiated by the AI model is embedded within `<tool_call></tool_call>` tags in JSON format, containing the function name, arguments, and a unique call ID. This structure facilitates easy parsing and processing of tool calls, enabling systematic execution and response handling by the `ToolAgent` class.

2.2 Tool Definition Format

2.2.1 The EXAMPLE_TOOL_DEFINITION Format

In this context, the format of each tool's definition is encapsulated in a dictionary (or JSON format) that describes the function name, description, and a schema of the parameters. This schema specifies each parameter's type and its expected structure (e.g., basic types such as `int`, `str`, or complex types like `List`, `Dict`). The `EXAMPLE_TOOL_DEFINITION` format enables the AI to understand the function's capabilities and expected inputs without directly inspecting the function implementation.

2.2.2 Overview of Parameter Specification and Return Types

Each parameter in the tool definition includes a "type" field that indicates the expected data type for validation. For complex data structures like lists or dictionaries, additional fields, such as `items_type` or `key_type` and `value_type`, are provided to clarify the expected structure. Return types are documented but not strictly enforced in this definition format since the focus is on input validation and successful tool execution.

2.3 Tool Decorator

2.3.1 The Tool Decorator Function

The `tool` decorator function wraps a regular Python function into a `Tool` object. This decorator applies the `get_fn_signature` function to extract metadata from the function's annotations, generating a JSON schema that defines the tool's name, description, and parameter types. This structured format helps the `ToolAgent` accurately map user queries to function calls. Additionally, it stores the tool definition within the `Tool` object, making it accessible for testing and validation.

2.3.2 Type Validation and XML Parsing Steps

The decorator also indirectly supports type validation by establishing a structured signature in the `Tool` object. During tool execution, the `validate_arguments` function checks argument types based on this signature. XML parsing is used by `ToolAgent` to identify and isolate each `<tool_call>` from the AI's response, extracting the JSON payload for further processing. This design promotes modularity by separating tool registration, type validation, and XML parsing.

2.4 Tool Agent

2.4.1 ToolAgent Class for Tool Execution

The `ToolAgent` class acts as an intermediary between the AI model and the tools, handling user queries, invoking tools, and generating responses. It stores a list of registered tools, each defined by a `Tool` object, and builds a consolidated prompt to present all available tool signatures to the model. Upon receiving a tool call, the `ToolAgent` extracts, validates, and executes the function with the appropriate arguments, gathering results for user feedback.

2.4.2 Response Formatting and Error Handling

The `ToolAgent` formats responses by encapsulating function calls in `<tool_call>` tags, helping the AI parse tool invocation requests. It also manages error handling by validating argument types before tool execution. If the AI requests a tool call without a matching tool or invalid arguments, `ToolAgent` captures and returns an error message, indicating that no suitable tools were available or that inputs were mismatched. This error reporting helps maintain robust interaction and improves user guidance by providing feedback on unrecognized requests.

2.5 Example Tools

This section provides an overview of the three tools implemented in this project: an API interaction tool for HackerNews, a data processing tool for analyzing and cleaning datasets, and an external service integration tool for interacting with GitHub.

2.5.1 API Interaction Tool: HackerNews Search

The `search_hackernews` tool enables the Large Language Model (LLM) to interact with the HackerNews API to search for recent articles based on a specified query. The tool accepts a search term and an optional limit on the number of results to return. The results are filtered to only include articles with a minimum number of points to ensure relevance. Each article returned includes the title, URL, and score.

- **Functionality:** This tool queries the HackerNews API to retrieve articles matching the specified search term.
- **Parameters:**
  - `query` (str): The search keyword.
  - `limit` (int): Maximum number of results to return.
- **Return Value:** A list of dictionaries, each containing the article title, URL, and score.

2.5.2 Data Processing Tool: Time Series Analysis and Dataset Cleaning

The data processing tool set consists of two main functions: `analyze_time_series` and `clean_dataset`. The `analyze_time_series` function performs statistical analysis on time series data, calculating metrics such as mean, median, standard deviation, minimum, maximum, and identifying trends based on a specified date and value column. The `clean_dataset` function, on the other hand, preprocesses a dataset by selecting specified columns, handling missing values, and removing duplicates.

- **Functionality:**
  - `analyze_time_series` provides statistical insights for time series data.
  - `clean_dataset` cleans the dataset by selecting columns, handling null values, and removing duplicates.
- **Parameters:**
  - `data` (List[Dict]): Input dataset.
  - `date_column` (str): Date column for time series analysis.

  - `value_column` (str): Value column for analysis.
  - `columns` (List[str]): Columns to keep in dataset cleaning.
  - `handle_nulls` (str): Strategy for handling null values (default is 'drop').

- **Return Value:**
  - `analyze_time_series` returns a dictionary of statistical metrics.
  - `clean_dataset` returns a cleaned dataset as a list of dictionaries.

2.5.3 External Service Integration Tool: GitHub Interaction

The GitHub toolset includes three functions: `get_trending_repos`, `get_repo_contributors`, and `search_repo_issues`. These functions interact with GitHub's API to retrieve trending repositories based on language and timeframe, fetch top contributors for a specific repository, and search for issues within a repository. The `get_trending_repos` function retrieves repositories based on popularity within a specified date range, the `get_repo_contributors` function lists the top contributors, and the `search_repo_issues` function retrieves issues based on criteria such as state and labels.

- **Functionality:**
  - `get_trending_repos` fetches trending GitHub repositories.
  - `get_repo_contributors` retrieves the top contributors for a repository.
  - `search_repo_issues` searches for issues within a repository.

- **Parameters:**
  - `language` (str): Programming language filter for trending repositories.
  - `since` (str): Time range for trending repositories (e.g., 'daily', 'weekly', 'monthly').
  - `repo_name` (str): Repository name in the format 'owner/repo' for contributor and issue search.
  - `state` (str): Issue state for filtering (e.g., 'open', 'closed').
  - `label` (str): Label to filter issues by.
  - `limit` (int): Maximum number of results to return.

- **Return Value:**
  - `get_trending_repos` returns a list of trending repositories.
  - `get_repo_contributors` returns a list of top contributors for the specified repository.
  - `search_repo_issues` returns a list of issues based on search criteria.

3 Testing

3.1 Testing Approach

The testing approach aims to verify the functionality of each tool, check basic operations, and ensure graceful error handling. Three primary tests were implemented:

3.1.1 Basic Functionality Tests

These tests ensure that each tool works as expected under normal conditions. For instance, they verify that the functions are executed correctly, return the expected results, and exhibit the intended behavior.

3.1.2 Error Handling Tests

The error handling tests assess the system's ability to respond to invalid inputs or unexpected conditions without failing. By simulating incorrect inputs, these tests confirm that the tools can manage errors gracefully and provide useful feedback.

3.2 Test Cases

3.2.1 Test 1: Tool Creation and Attribute Check

This test verifies that the `tool` decorator correctly sets up the necessary attributes on functions that are registered as tools. Specifically, it checks that the `tool_definition` attribute is added to decorated functions, confirming that they are properly prepared for integration within the `ToolAgent`.

- **Objective:** Ensure that the `tool` decorator adds the `tool_definition` attribute to each decorated function.
- **Test Code:** The `add` function is decorated with `tool` and checked for the presence of `tool_definition`.
- **Expected Result:** The function should include the `tool_definition` attribute.

3.2.2 Test 2: Basic Function Execution

This test verifies that a tool registered with the `ToolAgent` can execute a basic function correctly. The `add` function, which adds two integers, is registered, and the agent is instructed to perform an addition operation. The test checks if the tool correctly returns the result.

- **Objective:** Confirm that the tool function executes correctly and returns the expected result.

- **Test Code:** The `ToolAgent` registers the `add` function and performs an addition of 5 and 3.
- **Expected Result:** The result should contain the correct answer, which is 8.

3.2.3 Test 3: Error Handling

The error handling test examines how the system responds to invalid inputs. Here, the `add` function is given non-integer inputs (3.14 and 0.99). The test checks if the system identifies the invalid input type and returns an appropriate error message.

- **Objective:** Verify that the tool can handle invalid inputs and return an error message.
- **Test Code:** The test runs the `add` function with floating-point numbers and checks for an error in the response.
- **Expected Result:** The response should contain an error message indicating invalid input.

4 Conclusion

4.1 Summary of Project Completion and Success Criteria

This project successfully implemented a tool pattern system to extend the capabilities of a Large Language Model (LLM) by enabling interaction with external functions. The system allows an LLM to accurately call predefined tools, validate inputs, and handle errors using an XML-based prompt and a structured tool definition format. Key components, including the `ToolAgent`, tool decorator, and error-handling mechanisms, were developed and tested to ensure robust performance. Through basic functionality tests and error-handling scenarios, the system demonstrated its ability to execute tool functions reliably and to provide useful feedback in the event of invalid input. Overall, the project met its success criteria by enhancing the LLM's flexibility and extending its real-world applicability through external integrations.

4.2 Potential Improvements and Future Work

While the tool system achieved its primary objectives, several improvements could be made to increase functionality and flexibility:

- **Enhanced Tool Discovery:** Currently, tools must be pre-registered with the `ToolAgent`. Implementing a dynamic tool discovery mechanism could allow for easier registration and integration of new tools without requiring manual updates to the `ToolAgent`.

- **More Sophisticated Error Handling:** Adding a more granular error-handling framework could allow the system to distinguish between various types of errors (e.g., missing arguments, type mismatches) and provide tailored feedback for each case, improving user guidance.
- **Multi-step Tool Execution:** Some queries may require sequential calls to multiple tools (e.g., retrieving data, then processing it). Future work could implement a workflow manager that supports multi-step operations, allowing the LLM to chain tool calls dynamically.
- **Context-aware Responses:** Integrating context-aware responses could improve the system's ability to manage session state, enabling the LLM to make tool calls based on previous interactions within a conversation, further enhancing the user experience.
- **Broader Tool Integration:** Expanding the range of tools (e.g., for web scraping, advanced data visualization) could open up new use cases, making the system applicable across more domains.

In summary, this project provides a foundational framework for tool-based LLM interactions, demonstrating how LLMs can be made more practical and adaptable for real-world applications. By continuing to develop and refine this approach, future systems can leverage the strengths of LLMs in combination with external tools, enabling more comprehensive and responsive solutions across various fields.