Documentation

Tool Use

Introduction to Tool Use

Tool use is a powerful feature that allows Large Language Models (LLMs) to interact with external resources, such as APIs, databases, and the web, to gather dynamic data they wouldn't otherwise have access to in their pre-trained (or static) state and perform actions beyond simple text generation.

Tool use bridges the gap between the data that the LLMs were trained on with dynamic data and real-world actions, which opens up a wide array of realtime use cases for us to build powerful applications with, especially with Groq's insanely fast inference speed. 🚀

How Tool Use Works

Groq API tool use structure is compatible with OpenAI's tool use structure, which allows for easy integration. See the following cURL example of a tool use request:

```
curl https://api.groq.com/openai/v1/chat/completions \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $GROQ_API_KEY" \
-d '{
  "model": "llama-3.3-70b-versatile",
  "messages": [
    {
      "role": "user",
      "content": "What'\''s the weather like in Boston today?"
    }
  ],
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
              "type": "string",
              "enum": ["celsius", "fahrenheit"]
            }
          },
          "required": ["location"]
        }
      }
    }
  ],
  "tool_choice": "auto"
}'
```

To integrate tools with Groq API, follow these steps:

1. Provide tools (or predefined functions) to the LLM for performing actions and accessing external data in real-time in addition to your user prompt within your Groq API request
2. Define how the tools should be used to teach the LLM how to use them effectively (e.g. by defining input and output formats)
3. Let the LLM autonomously decide whether or not the provided tools are needed for a user query by evaluating the user query, determining whether the tools can enhance its response, and utilizing the tools accordingly
4. Extract tool input, execute the tool code, and return results
5. Let the LLM use the tool result to formulate a response to the original prompt

This process allows the LLM to perform tasks such as real-time data retrieval, complex calculations, and external API interaction, all while maintaining a natural conversation with our end user.

Tool Use with Groq

Groq API endpoints support tool use to almost instantly deliver structured JSON output that can be used to directly invoke functions from desired external resources.

Supported Models

The following models are recommended for tool use due to their versatility and performance:

- **deepseek-r1-distill-llama-70b**
- **llama-3.3-70b-versatile**
- **llama-3.1-8b-instant**

Other Supported Models

The following models powered by Groq also support tool use:

- **`mixtral-8x7b-32768`** (parallel tool use not supported)
- **`gemma2-9b-it`** (parallel tool use not supported)

Tools Specifications

Tool use is part of the Groq API chat completion request payload.

Tool Call and Tool Response Structure

Tool Call Structure

Groq API tool calls are structured to be OpenAI-compatible. The following is an example tool call structure:

```
{
  "model": "llama-3.3-70b-versatile",
  "messages": [
    {
      "role": "system",
      "content": "You are a weather assistant. Use the get_weather function to retrieve weather information for a
    },
    {
      "role": "user",
      "content": "What's the weather like in New York today?"
    }
  ],
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Get the current weather for a location",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
              "type": "string",
              "enum": ["celsius", "fahrenheit"],
              "description": "The unit of temperature to use. Defaults to fahrenheit."
            }
          },
          "required": ["location"]
        }
      }
    }
  ],
  "tool_choice": "auto",
  "max_completion_tokens": 4096
}'
```

Tool Call Response

The following is an example tool call response based on the above:

```
"model": "llama-3.3-70b-versatile",
"choices": [{
    "index": 0,
    "message": {
        "role": "assistant",
        "tool_calls": [{
            "id": "call_d5wg",
            "type": "function",
            "function": {
                "name": "get_weather",
                "arguments": "{\"location\": \"New York, NY\"}"
            }
        }]
    },
    "logprobs": null,
    "finish_reason": "tool_calls"
}],
```

When a model decides to use a tool, it returns a response with a `tool_calls` object containing:

- `id`: a unique identifier for the tool call
- `type`: the type of tool call, i.e. function
- `name`: the name of the tool being used
- `parameters`: an object containing the input being passed to the tool

Setting Up Tools

To get started, let's go through an example of tool use with Groq API that you can use as a base to build more tools on your own.

Step 1: Create Tool

Let's install Groq SDK, set up our Groq client, and create a function called `calculate` to evaluate a mathematical expression that we will represent as a tool.

Note: In this example, we're defining a function as our tool, but your tool can be any function or an external resource (e.g. dabatase, web search engine, external API).

Python JavaScript

```
pip install groq
```

```
from groq import Groq
import json

# Initialize the Groq client
client = Groq()
# Specify the model to be used (we recommend Llama 3.3 70B)
MODEL = 'llama-3.3-70b-versatile'

def calculate(expression):
    """Evaluate a mathematical expression"""
    try:
        # Attempt to evaluate the math expression
        result = eval(expression)
        return json.dumps({"result": result})
    except:
        # Return an error message if the math expression is invalid
        return json.dumps({"error": "Invalid expression"})
```

Step 2: Pass Tool Definition and Messages to Model

Next, we'll define our `calculate` tool within an array of available `tools` and call our Groq API chat completion. You can read more about tool schema and supported required and optional fields above in **Tool Specifications.**

By defining our tool, we'll inform our model about what our tool does and have the model decide whether or not to use the tool. We should be as descriptive and specific as possible for our model to be able to make the correct tool use decisions.

In addition to our `tools` array, we will provide our `messages` array (e.g. containing system prompt, assistant prompt, and/or user prompt).

Step 3: Receive and Handle Tool Results

After executing our chat completion, we'll extract our model's response and check for tool calls.

If the model decides that no tools should be used and does not generate a tool or function call, then the response will be a normal chat completion (i.e. `response_message = response.choices[0].message`) with a direct model reply to the user query.

If the model decides that tools should be used and generates a tool or function call, we will:

1. Define available tool or function,
2. Add the model's response to the conversation by appending our message
3. Process the tool call and add the tool response to our message
4. Make a second Groq API call with the updated conversation
5. Return the final response

Python JavaScript

```
# imports calculate function from step 1
def run_conversation(user_prompt):
    # Initialize the conversation with system and user messages
    messages=[
```

```
        {
            "role": "system",
            "content": "You are a calculator assistant. Use the calculate function to perform mathematical ope
        },
        {
            "role": "user",
            "content": user_prompt,
        }
    ]
    # Define the available tools (i.e. functions) for our model to use
    tools = [
        {
            "type": "function",
            "function": {
                "name": "calculate",
                "description": "Evaluate a mathematical expression",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "expression": {
                            "type": "string",
                            "description": "The mathematical expression to evaluate",
                        }
                    },
                    "required": ["expression"],
                },
            },
        }
    ]
    # Make the initial API call to Groq
    response = client.chat.completions.create(
        model=MODEL, # LLM to use
        messages=messages, # Conversation history
        stream=False,
        tools=tools, # Available tools (i.e. functions) for our LLM to use
        tool_choice="auto", # Let our LLM decide when to use tools
        max_completion_tokens=4096 # Maximum number of tokens to allow in our response
    )
    # Extract the response and any tool call responses
    response_message = response.choices[0].message
    tool_calls = response_message.tool_calls
    if tool_calls:
        # Define the available tools that can be called by the LLM
        available_functions = {
            "calculate": calculate,
        }
        # Add the LLM's response to the conversation
        messages.append(response_message)
        # Process each tool call
        for tool_call in tool_calls:
            function_name = tool_call.function.name
            function_to_call = available_functions[function_name]
            function_args = json.loads(tool_call.function.arguments)
            # Call the tool and get the response
            function_response = function_to_call(
                function_args.get("expression")
            )
            # Add the tool response to the conversation
            messages.append(
                {
                    "tool_call_id": tool_call.id,
                    "role": "tool", # Indicates this message is from tool use
                    "name": function_name,
                    "content": function_response,
                }
            )
        # Make a second API call with the updated conversation
        second_response = client.chat.completions.create(
            model=MODEL,
            messages=messages
            max_completion_tokens=20  # We only need a short response
        )
```

Routing System

If you use our models fine-tuned for tool use, we recommended to use them as part of a routing system:

1. **Query Analysis**: Implement a routing system that analyzes incoming user queries to determine their nature and requirements.
2. **Model Selection**: Based on the query analysis, route the request to the most appropriate model:
   - For queries involving function calling, API interactions, or structured data manipulation, use the Llama 3 Groq Tool Use models.
   - For general knowledge, open-ended conversations, or tasks not specifically related to tool use, route to a general-purpose language model such as Llama 3 70B.

The following is the `calculate` tool we built in the above steps enhanced to include a routing system that routes our request to Llama 3 70B if the user query does not require the tool:

Python JavaScript

```
from groq import Groq
import json
# Initialize the Groq client
client = Groq()
# Define models
ROUTING_MODEL = "llama3-70b-8192"
TOOL_USE_MODEL = "llama-3.3-70b-versatile"
GENERAL_MODEL = "llama3-70b-8192"

def calculate(expression):
    """Tool to evaluate a mathematical expression"""
    try:
        result = eval(expression)
        return json.dumps({"result": result})
    except:
        return json.dumps({"error": "Invalid expression"})

def route_query(query):
    """Router logic to let LLM decide if tools are needed"""
    routing_prompt = f"""
    Given the following user query, determine if any tools are needed to answer it.
    If a calculation tool is needed, respond with 'TOOL: CALCULATE'.
    If no tools are needed, respond with 'NO_TOOL'.

    User query: {query}
    Response:
    """

    response = client.chat.completions.create(
        model=ROUTING_MODEL,
        messages=[
            {"role": "system", "content": "You are a routing assistant. Determine if tools are needed based on
            {"role": "user", "content": routing_prompt}
        ],
        max_completion_tokens=20
    )
```

Python JavaScript

Parallel Tool Use

We learned about tool use and built single-turn tool use examples above. Now let's take tool use a step further and imagine a workflow where multiple tools can be called simultaneously, enabling more efficient and effective responses.

This concept is known as **parallel tool use** and is key for building agentic workflows that can deal with complex queries, which is a great example of where inference speed becomes increasingly important (and thankfully we can access fast inference speed with Groq API).

Note: Parallel tool use is natively enabled for all Llama 3 and Llama 3.1 models!

Here's an example of parallel tool use with a tool for getting the temperature and the tool for getting the weather condition to show parallel tool use with Groq API in action:

```
    )   # Return the final response
    return second_response.choices[0].message.content

# Example usage
response = response.choices[0].message.content.strip()
user_prompt = "What is 25 * 4 + 10?"
print(f"TOOL: CALCULATE\nUser prompt: {user_prompt}")
if "TOOL: CALCULATE" in decision:
    return "calculate tool needed"
else:
    return "no tool needed"
```

Error Handling

Groq API tool use is designed to verify whether a model generates a valid tool call object. When a model fails to generate a valid tool call object, Groq API will return a 400 error with an explanation in the "failed_generation" field of the JSON body that is returned.

Next Steps

For more information and examples of working with multiple tools in parallel using Groq API and Instructor, see our Groq API Cookbook tutorial here.

Tool Use with Structured Outputs (Python)

Groq API offers best-effort matching for parameters, which means the model could occasionally miss parameters or misinterpret types for more complex tool calls. We recommend the Instuctor library to simplify the process of working with structured data and to ensure that the model's output adheres to a predefined schema.

Here's an example of how to implement tool use using the Instructor library with Groq API:

```
pip install instructor pydantic
```

```
import instructor
from pydantic import BaseModel, Field
from groq import Groq

# Define the tool schema object
tool_schema = {
    "name": "get_weather_info",
    "description": "Get the weather information for any location.",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The location for which we want to get the weather information (e.g., New York)"
            },
        },
        "required": ["location"]
    }
}

# Make the initial request
response = client.chat.completions.create(
    model=model, messages=messages, tools=tools, tool_choice="auto", max_completion_tokens=4096
)

# Define the Pydantic model for the tool call
class ToolCall(BaseModel):
    tool_name: str = Field(description="The name of the tool to call")
    tool_parameters: str = Field(description="JSON string of tool parameters")

# Process tool calls
```

Benefits of Using Structured Outputs

- Type Safety: Pydantic models ensure that output adheres to the expected structure, reducing the risk of errors.
- Automatic Validation: Instructor automatically validates the model's output against the defined schema.

Next Steps

For more information and examples of working with structured outputs using Groq API and Instructor, see our Groq API Cookbook tutorial.

Best Practices

- Provide detailed tool descriptions for optimal performance.
- We recommend tool use with the Instructor library for structured outputs.
- Use the fine-tuned Llama 3 models by Groq or the Llama 3.1 models for your applications that require tool use.
- Implement a routing system when using fine-tuned models in your workflow.
- Handle tool execution errors by returning error messages with "is_error": true.