LM Studio

Documentation

API

Tool Use

Enable LLMs to interact with external functions and APIs.

Tool use enables LLMs to request calls to external functions and APIs through the `/v1/chat/completions` endpoint, via LM Studio's REST API (or via any OpenAI client). This expands their functionality far beyond text output.

Quick Start

1. Start LM Studio as a server

To use LM Studio programmatically from your own code, run LM Studio as a local server.

You can turn on the server from the "Developer" tab in LM Studio, or via the `lms` CLI:

```
lms server start
```

Install `lms` by running `npx lmstudio install-cli`

This will allow you to interact with LM Studio via an OpenAI-like REST API. For an intro to LM Studio's OpenAI-like API, see Running LM Studio as a server.

2. Load a Model

You can load a model from the "Chat" or "Developer" tabs in LM Studio, or via the `lms` CLI:

```
lms load
```

3. Copy, Paste, and Run an Example!

- `Curl`

LM Studio

Documentation

- Advanced Agent Example

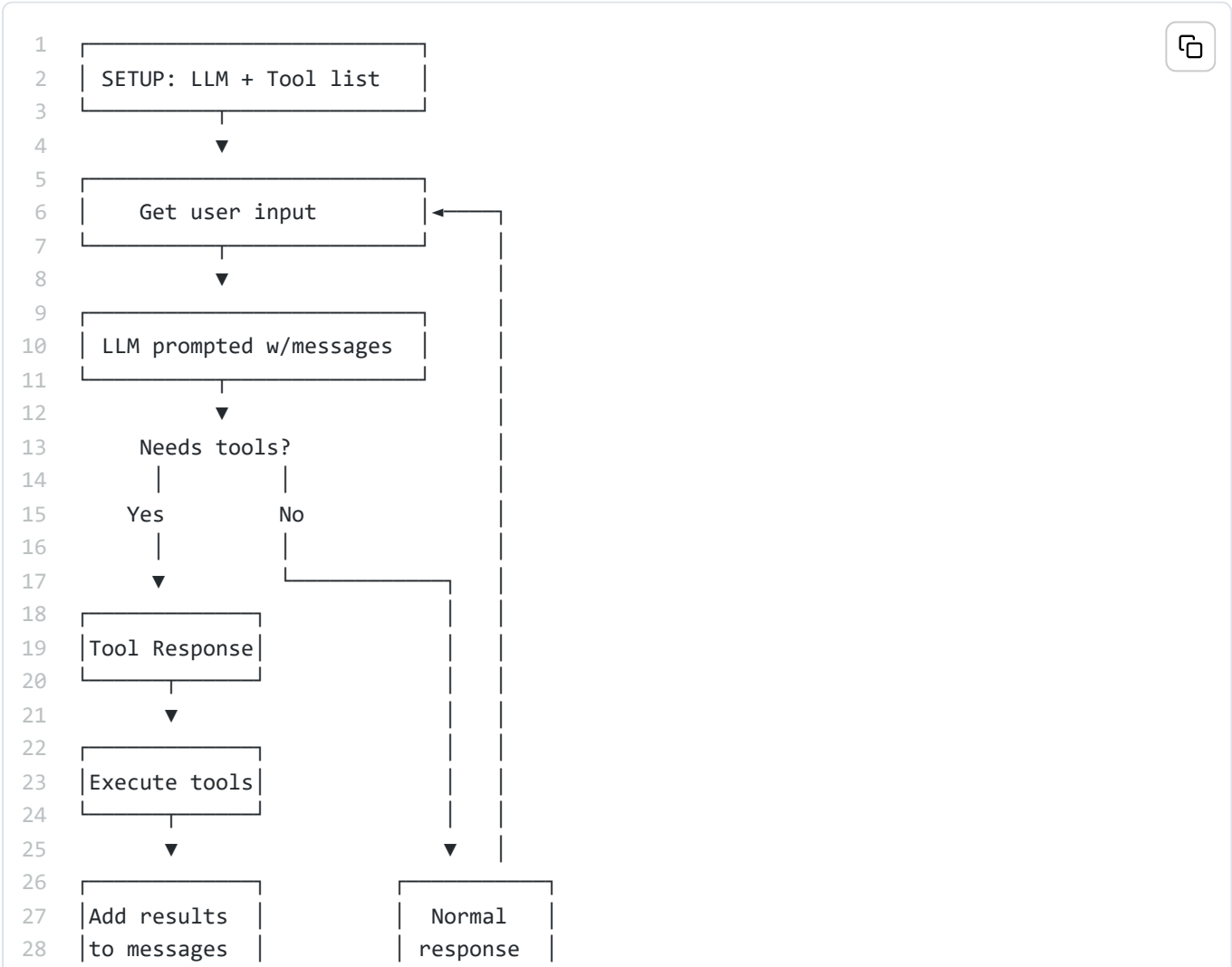
Tool Use

What really is "Tool Use"?

Tool use describes:

- LLMs output text requesting functions to be called (LLMs cannot directly execute code)
- Your code executes those functions
- Your code feeds the results back to the LLM.

High-level flow



LM Studio

Documentation

LM Studio supports tool use through the `/v1/chat/completions` endpoint when given function definitions in the `tools` parameter of the request body. Tools are specified as an array of function definitions that describe their parameters and usage, like:

It follows the same format as OpenAI's Function Calling API and is expected to work via the OpenAI client SDKs.

We will use lmstudio-community/Qwen2.5-7B-Instruct-GGUF as the model in this example flow.

1. You provide a list of tools to an LLM. These are the tools that the model can *request* calls to. For example:

```
// the list of tools is model-agnostic
[
  {
    "type": "function",
    "function": {
      "name": "get_delivery_date",
      "description": "Get the delivery date for a customer's order",
      "parameters": {
        "type": "object",
        "properties": {
          "order_id": {
            "type": "string"
          }
        },
        "required": ["order_id"]
      }
    }
  }
]
```

This list will be injected into the `system` prompt of the model depending on the model's chat template. For `Qwen2.5-Instruct`, this looks like:

```
<|im_start|>system
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{"type": "function", "function": {"name": "get_delivery_date", "description": "Get th
```

Important: The model can only *request* calls to these tools because LLMs *cannot* directly call functions, APIs, or any other tools. They can only output text, which can then be parsed to programmatically call the functions.

2. When prompted, the LLM can then decide to either:
   - (a) Call one or more tools

```
User: Get me the delivery date for order 123
Model: <tool_call>
{"name": "get_delivery_date", "arguments": {"order_id": "123"}}
</tool_call>
```

   - (b) Respond normally

```
User: Hi
Model: Hello! How can I assist you today?
```

3. LM Studio parses the text output from the model into an OpenAI-compliant `chat.completion` response object.
   - If the model was given access to `tools`, LM Studio will attempt to parse the tool calls into the `response.choices[0].message.tool_calls` field of the `chat.completion` response object.
   - If LM Studio cannot parse any **correctly formatted** tool calls, it will simply return the response to the standard `response.choices[0].message.content` field.
   - **Note**: Smaller models and models that were not trained for tool use may output improperly formatted tool calls, resulting in LM Studio being unable to parse them into the `tool_calls` field. This is useful for troubleshooting when you do not receive `tool_calls` as expected. Example of an improperly formatting `Qwen2.5-Instruct` tool call:

```
<tool_call>
["name": "get_delivery_date", function: "date"]
</tool_call>
```

> Note that the brackets are incorrect, and the call does not follow the `name, argument` format.

2. The result of the tool call

To the `messages` array to send back to the model

```
# pseudocode, see examples for copy-paste snippets
if response.has_tool_calls:
    for each tool_call:
        # Extract function name & args
        function_to_call = tool_call.name     # e.g. "get_delivery_date"
        args = tool_call.arguments            # e.g. {"order_id": "123"}

        # Execute the function
        result = execute_function(function_to_call, args)

        # Add result to conversation
        add_to_messages([
            ASSISTANT_TOOL_CALL_MESSAGE,      # The request to use the tool
            TOOL_RESULT_MESSAGE               # The tool's response
        ])
else:
    # Normal response without tools
    add_to_messages(response.content)
```

5. The LLM is then prompted again with the updated messages array, but without access to tools. This is because:
   - The LLM already has the tool results in the conversation history
   - We want the LLM to provide a final response to the user, not call more tools

```
# Example messages
messages = [
    {"role": "user", "content": "When will order 123 be delivered?"},
    {"role": "assistant", "function_call": {
        "name": "get_delivery_date",
        "arguments": {"order_id": "123"}
    }},
    {"role": "tool", "content": "2024-03-15"},
]
response = client.chat.completions.create(
    model="lmstudio-community/qwen2.5-7b-instruct",
    messages=messages
)
```

The `response.choices[0].message.content` field after this call may be something like:

```
Your order #123 will be delivered on March 15th, 2024
```

LM Studio

Documentation

Supported Models

Through LM Studio, **all** models support at least some degree of tool use.

However, there are currently two levels of support that may impact the quality of the experience: Native and Default.

Models with Native tool use support will have a hammer badge in the app, and generally perform better in tool use scenarios.

Native tool use support

"Native" tool use support means that both:

1. The model has a chat template that supports tool use (usually means the model has been trained for tool use)
   - This is what will be used to format the `tools` array into the system prompt and tell them model how to format tool calls
   - Example: Qwen2.5-Instruct chat template
2. LM Studio supports that model's tool use format
   - Required for LM Studio to properly input the chat history into the chat template, and parse the tool calls the model outputs into the `chat.completion` object

Models that currently have native tool use support in LM Studio (subject to change):

- Qwen
  - `GGUF` lmstudio-community/Qwen2.5-7B-Instruct-GGUF (4.68 GB)
  - `MLX` mlx-community/Qwen2.5-7B-Instruct-4bit (4.30 GB)
- Llama-3.1, Llama-3.2
  - `GGUF` lmstudio-community/Meta-Llama-3.1-8B-Instruct-GGUF (4.92 GB)
  - `MLX` mlx-community/Meta-Llama-3.1-8B-Instruct-8bit (8.54 GB)
- Mistral
  - `GGUF` bartowski/Ministral-8B-Instruct-2410-GGUF (4.67 GB)

1. The model does not have chat template that supports tool use (usually means the model has not been trained for tool use)
2. LM Studio does not currently support that model's tool use format

Under the hood, default tool use works by:

- Giving models a custom system prompt and a default tool call format to use
- Converting `tool` role messages to the `user` role so that chat templates without the `tool` role are compatible
- Converting `assistant` role `tool_calls` into the default tool call format

Results will vary by model.

You can see the default format by running `lms log stream` in your terminal, then sending a chat completion request with `tools` to a model that doesn't have Native tool use support. The default format is subject to change.

▶ Expand to see example of default tool use format

All models that don't have native tool use support will have default tool use support.

Example using `curl`

This example demonstrates a model requesting a tool call using the `curl` utility.

To run this example on Mac or Linux, use any terminal. On Windows, use Git Bash.

```
curl http://localhost:1234/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "lmstudio-community/qwen2.5-7b-instruct",
    "messages": [{"role": "user", "content": "What dell products do you have under $50 in
    "tools": [
      {
        "type": "function",
        "function": {
          "name": "search_products",
          "description": "Search the product catalog by various criteria. Use this wheneve
```

LM Studio

Documentation

```
              "category": {
                "type": "string",
                "description": "Product category to filter by",
                "enum": ["electronics", "clothing", "home", "outdoor"]
              },
              "max_price": {
                "type": "number",
                "description": "Maximum price in dollars"
              }
            },
            "required": ["query"],
            "additionalProperties": false
          }
        }
      }
    ]
  }'
```

All parameters recognized by `/v1/chat/completions` will be honored, and the array of available tools should be provided in the `tools` field.

If the model decides that the user message would be best fulfilled with a tool call, an array of tool call request objects will be provided in the response field, `choices[0].message.tool_calls`.

The `finish_reason` field of the top-level response object will also be populated with `"tool_calls"`.

An example response to the above `curl` request will look like:

```
{
  "id": "chatcmpl-gb1t1uqzefudice8ntxd9i",
  "object": "chat.completion",
  "created": 1730913210,
  "model": "lmstudio-community/qwen2.5-7b-instruct",
  "choices": [
    {
      "index": 0,
      "logprobs": null,
      "finish_reason": "tool_calls",
      "message": {
        "role": "assistant",
        "tool_calls": [
          {
            "id": "365174485",
```

```
        }
      }
    ],
    "usage": {
      "prompt_tokens": 263,
      "completion_tokens": 34,
      "total_tokens": 297
    },
    "system_fingerprint": "lmstudio-community/qwen2.5-7b-instruct"
}
```

In plain english, the above response can be thought of as the model saying:

> "Please call the `search_products` function, with arguments:
>
> - 'dell' for the `query` parameter,
> - 'electronics' for the `category` parameter
> - '50' for the `max_price` parameter
>
> and give me back the results"

The `tool_calls` field will need to be parsed to call actual functions/APIs. The below examples demonstrate how.

Examples using `python`

Tool use shines when paired with program languages like python, where you can implement the functions specified in the `tools` field to programmatically call them when the model requests.

Single-turn example

Below is a simple single-turn (model is only called once) example of enabling a model to call a function called `say_hello` that prints a hello greeting to the console:

`single-turn-example.py`

LM Studio

Documentation

```python
def say_hello(name: str) -> str:
    print(f"Hello, {name}!")

# Tell the AI about our function
tools = [
    {
        "type": "function",
        "function": {
            "name": "say_hello",
            "description": "Says hello to someone",
            "parameters": {
                "type": "object",
                "properties": {
                    "name": {
                        "type": "string",
                        "description": "The person's name"
                    }
                },
                "required": ["name"]
            }
        }
    }
]

# Ask the AI to use our function
response = client.chat.completions.create(
    model="lmstudio-community/qwen2.5-7b-instruct",
    messages=[{"role": "user", "content": "Can you say hello to Bob the Builder?"}],
    tools=tools
)

# Get the name the AI wants to use a tool to say hello to
# (Assumes the AI has requested a tool call and that tool call is say_hello)
tool_call = response.choices[0].message.tool_calls[0]
name = eval(tool_call.function.arguments)["name"]

# Actually call the say_hello function
say_hello(name) # Prints: Hello, Bob the Builder!

```

Running this script from the console should yield results like:

```
→ % python single-turn-example.py
Hello, Bob the Builder!
```

Play around with the name in

LM Studio

Documentation

Multi-turn example

Now for a slightly more complex example.

In this example, we'll:

1. Enable the model to call a `get_delivery_date` function
2. Hand the result of calling that function back to the model, so that it can fulfill the user's request in plain text

▶ `multi-turn-example.py` (click to expand)

Running this script from the console should yield results like:

```
→ % python multi-turn-example.py

Model response requesting tool call:

ChatCompletion(id='chatcmpl-wwpstqqu94go4hvclqnpwn', choices=[Choice(finish_reason='tool_c

get_delivery_date function returns delivery date:

2024-11-19 13:03:17.773298

Final model response with knowledge of the tool call result:

Your order number 1017 is scheduled for delivery on November 19, 2024, at 13:03 PM.
```

Advanced agent example

Building upon the principles above, we can combine LM Studio models with locally defined functions to create an "agent" - a system that pairs a language model with custom functions to understand requests and perform actions beyond basic text generation.

The agent in the below example can:

1. Open safe urls in your default browser
2. Check the current time
3. Analyze directories in your file system

▶ `agent-chat-example.py` (click to expand)

LM Studio

Documentation

```
You: What time is it?

Assistant: The current time is 14:11:40 (EST) as of November 6, 2024.

You: What time is it now?

Assistant: The current time is 14:13:59 (EST) as of November 6, 2024.

You: Open lmstudio.ai

Assistant: The link to lmstudio.ai has been opened in your default web browser.

You: What's in my current directory?

Assistant: Your current directory at `/Users/matt/project` contains a total of 14 files an

- Files without an extension: 3
- `.mjs` files: 2
- `.ts` (TypeScript) files: 3
- Markdown (`md`) file: 1
- JSON files: 4
- TOML file: 1

The total size of these items is 1,566,990,604 bytes.

You: Thank you!

Assistant: You're welcome! If you have any other questions or need further assistance, fee

You:
```

Streaming

When streaming through `/v1/chat/completions` (`stream=true`), tool calls are sent in chunks. Function names and arguments are sent in pieces via `chunk.choices[0].delta.tool_calls.function.name` and `chunk.choices[0].delta.tool_calls.function.arguments`.

For example, to call `get_current_weather(location="San Francisco")`, the streamed `ChoiceDeltaToolCall` in each `chunk.choices[0].delta.tool_calls[0]` object will look like:

```
ChoiceDeltaToolCall(index=0, id='814890118', function=ChoiceDeltaToolCallFunction(argu
ChoiceDeltaToolCall(index=0, id=None, function=ChoiceDeltaToolCallFunction(arguments='{"',
ChoiceDeltaToolCall(index=0, id=None, function=ChoiceDeltaToolCallFunction(arguments='loca
```

signature for execution.

The below example shows how to create a simple tool-enhanced chatbot through the `/v1/chat/completions` streaming endpoint (`stream=true`).

▶ `tool-streaming-chatbot.py` (click to expand)

You can chat with the bot by running this script from the console:

```
→ % python tool-streaming-chatbot.py
Assistant: Hi! I am an AI agent empowered with the ability to tell the current time (Type

You: Tell me a joke, then tell me the current time

Assistant: Sure! Here's a light joke for you: Why don't scientists trust atoms? Because th

Now, let me get the current time for you.

**Calling Tool: get_current_time**

The current time is 18:49:31. Enjoy your day!

You:
```

Community

Chat with other LM Studio users, discuss LLMs, hardware, and more on the LM Studio Discord server.