Tool Calling

vLLM currently supports named function calling, as well as the `auto` and `none` options for the `tool_choice` field in the chat completion API. The `tool_choice` option `required` is **not yet supported** but on the roadmap.

Quickstart

Start the server with tool calling enabled. This example uses Meta's Llama 3.1 8B model, so we need to use the llama3 tool calling chat template from the vLLM examples directory:

```bash
vllm serve meta-llama/Llama-3.1-8B-Instruct \
    --enable-auto-tool-choice \
    --tool-call-parser llama3_json \
    --chat-template examples/tool_chat_template_llama3.1_json.jinja
```

Next, make a request to the model that should result in it using the available tools:

```python
from openai import OpenAI
import json

client = OpenAI(base_url="http://localhost:8000/v1", api_key="dummy")

def get_weather(location: str, unit: str):
    return f"Getting the weather for {location} in {unit}..."

tool_functions = {"get_weather": get_weather}


tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {"type": "string", "description": "City and state, e.g., 'San Francisco, CA'"},
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]}
            },
            "required": ["location", "unit"]
        }
    }
}]


response = client.chat.completions.create(
    model=client.models.list().data[0].id,
    messages=[{"role": "user", "content": "What's the weather like in San Francisco?"}],
    tools=tools,
    tool_choice="auto"
)


tool_call = response.choices[0].message.tool_calls[0].function
```

```
print(f"Function called: {tool_call.name}")
print(f"Arguments: {tool_call.arguments}")
print(f"Result: {get_weather(**json.loads(tool_call.arguments))}")
```

Example output:

```text
Function called: get_weather
Arguments: {"location": "San Francisco, CA", "unit": "fahrenheit"}
Result: Getting the weather for San Francisco, CA in fahrenheit...
```

This example demonstrates:

* Setting up the server with tool calling enabled
* Defining an actual function to handle tool calls
* Making a request with `tool_choice="auto"`
* Handling the structured response and executing the corresponding function

You can also specify a particular function using named function calling by setting `tool_choice={"type": "function", "function": {"name": "get_weather"}}`. Note that this will use the guided decoding backend - so the first time this is used, there will be several seconds of latency (or more) as the FSM is compiled for the first time before it is cached for subsequent requests.

Remember that it's the callers responsibility to:

1. Define appropriate tools in the request
2. Include relevant context in the chat messages
3. Handle the tool calls in your application logic

For more advanced usage, including parallel tool calls and different model-specific parsers, see the sections below.

Named Function Calling

vLLM supports named function calling in the chat completion API by default. It does so using Outlines through guided decoding, so this is
enabled by default, and will work with any supported model. You are guaranteed a validly-parsable function call - not a
high-quality one.

vLLM will use guided decoding to ensure the response matches the tool parameter object defined by the JSON schema in the `tools` parameter.
For best results, we recommend ensuring that the expected output format / schema is specified in the prompt to ensure that the model's intended generation is aligned with the schema that it's being forced to generate by the guided decoding backend.

To use a named function, you need to define the functions in the `tools` parameter of the chat completion request, and
specify the `name` of one of the tools in the `tool_choice` parameter of the chat completion request.

Automatic Function Calling

To enable this feature, you should set the following flags:

* `--enable-auto-tool-choice` -- **mandatory** Auto tool choice. tells vLLM that you want to enable the model to generate its own tool calls when it
deems appropriate.
* `--tool-call-parser` -- select the tool parser to use (listed below). Additional tool parsers
will continue to be added in the future, and also can register your own tool parsers in the `--tool-parser-plugin`.
* `--tool-parser-plugin` -- **optional** tool parser plugin used to register user defined tool parsers into vllm, the registered tool parser name can be specified in `--tool-call-parser`.
* `--chat-template` -- **optional** for auto tool choice. the path to the chat template which handles `tool`-role messages and `assistant`-role messages
that contain previously generated tool calls. Hermes, Mistral and Llama models have tool-compatible chat templates in their
`tokenizer_config.json` files, but you can specify a custom template. This argument can be set to `tool_use` if your model has a tool use-specific chat
template configured in the `tokenizer_config.json`. In this case, it will be used per the `transformers` specification. More on this [here](https://huggingface.co/docs/transformers/en/chat_templating#why-do-some-models-have-multiple-templates)
from HuggingFace; and you can find an example of this in a `tokenizer_config.json` [here](https://huggingface.co/NousResearch/Hermes-2-Pro-Llama-3-8B/blob/main/tokenizer_config.json)

If your favorite tool-calling model is not supported, please feel free to contribute a parser & tool use chat template!

Hermes Models (`hermes`)

All Nous Research Hermes-series models newer than Hermes 2 Pro should be supported.

* `NousResearch/Hermes-2-Pro-*`

* `NousResearch/Hermes-2-Theta-*`

* `NousResearch/Hermes-3-*`

_Note that the Hermes 2 **Theta** models are known to have degraded tool call quality & capabilities due to the merge
step in their creation_.

Flags: `--tool-call-parser hermes`

Mistral Models (`mistral`)

Supported models:

* `mistralai/Mistral-7B-Instruct-v0.3` (confirmed)

* Additional mistral function-calling models are compatible as well.

Known issues:

1. Mistral 7B struggles to generate parallel tool calls correctly.

2. Mistral's `tokenizer_config.json` chat template requires tool call IDs that are exactly 9 digits, which is
much shorter than what vLLM generates. Since an exception is thrown when this condition

is not met, the following additional chat templates are provided:

* `examples/tool_chat_template_mistral.jinja` - this is the "official" Mistral chat template, but tweaked so that
it works with vLLM's tool call IDs (provided `tool_call_id` fields are truncated to the last 9 digits)
* `examples/tool_chat_template_mistral_parallel.jinja` - this is a "better" version that adds a tool-use system prompt
when tools are provided, that results in much better reliability when working with parallel tool calling.

Recommended flags: `--tool-call-parser mistral --chat-template examples/tool_chat_template_mistral_parallel.jinja`

Llama Models (`llama3_json`)

Supported models:

* `meta-llama/Meta-Llama-3.1-8B-Instruct`
* `meta-llama/Meta-Llama-3.1-70B-Instruct`
* `meta-llama/Meta-Llama-3.1-405B-Instruct`
* `meta-llama/Meta-Llama-3.1-405B-Instruct-FP8`

The tool calling that is supported is the [JSON based tool calling](https://llama.meta.com/docs/model-cards-and-prompt-formats/llama3_1/#json-based-tool-calling). For [pythonic tool calling](https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/text_prompt_format.md#zero-shot-function-calling) in Llama-3.2 models, see the `pythonic` tool parser below.
Other tool calling formats like the built in python tool calling or custom tool calling are not supported.

Known issues:

1. Parallel tool calls are not supported.
2. The model can generate parameters with a wrong format, such as generating
   an array serialized as string instead of an array.

The `tool_chat_template_llama3_json.jinja` file contains the "official" Llama chat template, but tweaked so that
it works better with vLLM.

Recommended flags: `--tool-call-parser llama3_json --chat-template examples/tool_chat_template_llama3_json.jinja`

IBM Granite

Supported models:

* `ibm-granite/granite-3.0-8b-instruct`

Recommended flags: `--tool-call-parser granite --chat-template examples/tool_chat_template_granite.jinja`

`examples/tool_chat_template_granite.jinja`: this is a modified chat template from the original on Huggingface. Parallel function calls are supported.

* `ibm-granite/granite-3.1-8b-instruct`

Recommended flags: `--tool-call-parser granite`

The chat template from Huggingface can be used directly. Parallel function calls are supported.

* `ibm-granite/granite-20b-functioncalling`

Recommended flags: `--tool-call-parser granite-20b-fc --chat-template examples/tool_chat_template_granite_20b_fc.jinja`

`examples/tool_chat_template_granite_20b_fc.jinja`: this is a modified chat template from the original on Huggingface, which is not vLLM compatible. It blends function description elements from the Hermes template and follows the same system prompt as "Response Generation" mode from [the paper](https://arxiv.org/abs/2407.00121). Parallel function calls are supported.

InternLM Models (`internlm`)

Supported models:

* `internlm/internlm2_5-7b-chat` (confirmed)
* Additional internlm2.5 function-calling models are compatible as well

Known issues:

* Although this implementation also supports InternLM2, the tool call results are not stable when testing with the `internlm/internlm2-chat-7b` model.

Recommended flags: `--tool-call-parser internlm --chat-template examples/tool_chat_template_internlm2_tool.jinja`

Jamba Models (`jamba`)

AI21's Jamba-1.5 models are supported.

* `ai21labs/AI21-Jamba-1.5-Mini`
* `ai21labs/AI21-Jamba-1.5-Large`

Flags: `--tool-call-parser jamba`

Models with Pythonic Tool Calls (`pythonic`)

A growing number of models output a python list to represent tool calls instead of using JSON. This has the advantage of inherently supporting parallel tool calls and removing ambiguity around the JSON schema required for tool calls. The `pythonic` tool parser can support such models.

As a concrete example, these models may look up the weather in San Francisco and Seattle by generating:

```python
[get_weather(city='San Francisco', metric='celsius'), get_weather(city='Seattle', metric='celsius')]
```

Limitations:

* The model must not generate both text and tool calls in the same generation. This may not be hard to change for a specific model, but the community currently lacks consensus on which tokens to emit when starting and ending tool calls. (In particular, the Llama 3.2 models emit no such tokens.)
* Llama's smaller models struggle to use tools effectively.

Example supported models:

* `meta-llama/Llama-3.2-1B-Instruct`* (use with `examples/tool_chat_template_llama3.2_pythonic.jinja`)
* `meta-llama/Llama-3.2-3B-Instruct`* (use with `examples/tool_chat_template_llama3.2_pythonic.jinja`)
* `Team-ACE/ToolACE-8B` (use with `examples/tool_chat_template_toolace.jinja`)
* `fixie-ai/ultravox-v0_4-ToolACE-8B` (use with `examples/tool_chat_template_toolace.jinja`)

Flags: `--tool-call-parser pythonic --chat-template {see_above}`

WARNING
Llama's smaller models frequently fail to emit tool calls in the correct format. Your mileage may vary.

How to write a tool parser plugin

A tool parser plugin is a Python file containing one or more ToolParser implementations. You can write a ToolParser similar to the `Hermes2ProToolParser` in vllm/entrypoints/openai/tool_parsers/hermes_tool_parser.py.

Here is a summary of a plugin file:

```python

# import the required packages

# define a tool parser and register it to vllm
# the name list in register_module can be used
# in --tool-call-parser. you can define as many
# tool parsers as you want here.
@ToolParserManager.register_module(["example"])
class ExampleToolParser(ToolParser):
    def __init__(self, tokenizer: AnyTokenizer):
        super().__init__(tokenizer)

    # adjust request. e.g.: set skip special tokens
    # to False for tool call output.
    def adjust_request(
            self, request: ChatCompletionRequest) -> ChatCompletionRequest:
        return request

    # implement the tool call parse for stream call
    def extract_tool_calls_streaming(
        self,
        previous_text: str,
        current_text: str,
        delta_text: str,
        previous_token_ids: Sequence[int],
        current_token_ids: Sequence[int],
        delta_token_ids: Sequence[int],
        request: ChatCompletionRequest,
    ) -> Union[DeltaMessage, None]:
        return delta

    # implement the tool parse for non-stream call
    def extract_tool_calls(
        self,
        model_output: str,
        request: ChatCompletionRequest,
    ) -> ExtractedToolCallInformation:
        return ExtractedToolCallInformation(tools_called=False,
                                            tool_calls=[],
                                            content=text)

```

Then you can use this plugin in the command line like this.

```console
    --enable-auto-tool-choice \
    --tool-parser-plugin <absolute path of the plugin file>
    --tool-call-parser example \
    --chat-template <your chat template> \
```