Workflow for a Function Calling Agent

This notebook walks through setting up a `Workflow` to construct a function calling agent from scratch.

Function calling agents work by using an LLM that supports tools/functions in its API (OpenAI, Ollama, Anthropic, etc.) to call functions an use tools.

Our workflow will be stateful with memory, and will be able to call the LLM to select tools and process incoming user messages.

```
!pip install -U llama-index
```

```
import os

os.environ["OPENAI_API_KEY"] = "sk-proj-..."
```

[Optional] Set up observability with Llamatrace

Set up tracing to visualize each step in the workflow.

Since workflows are async first, this all runs fine in a notebook. If you were running in your own code, you would want to use `asyncio.run()` to start an async event loop if one isn't already running.

```
async def main():
    <async code>

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Designing the Workflow

An agent consists of several steps

1. Handling the latest incoming user message, including adding to memory and getting the latest chat history
2. Calling the LLM with tools + chat history
3. Parsing out tool calls (if any)
4. If there are tool calls, call them, and loop until there are none
5. When there is no tool calls, return the LLM response

The Workflow Events

To handle these steps, we need to define a few events:

1. An event to handle new messages and prepare the chat history
2. An event to handle streaming responses
3. An event to trigger tool calls
4. An event to handle the results of tool calls

The other steps will use the built-in `StartEvent` and `StopEvent` events.

```python
from llama_index.core.llms import ChatMessage
from llama_index.core.tools import ToolSelection, ToolOutput
from llama_index.core.workflow import Event


class InputEvent(Event):
    input: list[ChatMessage]


class StreamEvent(Event):
    delta: str


class ToolCallEvent(Event):
    tool_calls: list[ToolSelection]


class FunctionOutputEvent(Event):
    output: ToolOutput
```

The Workflow Itself

With our events defined, we can construct our workflow and steps.

Note that the workflow automatically validates itself using type annotations, so the type annotations on our steps are very helpful!

```
from typing import Any, List

from llama_index.core.llms.function_calling import
FunctionCallingLLM
from llama_index.core.memory import ChatMemoryBuffer
from llama_index.core.tools.types import BaseTool
from llama_index.core.workflow import (
    Context,
    Workflow,
    StartEvent,
    StopEvent,
    step,
)
from llama_index.llms.openai import OpenAI


class FuncationCallingAgent(Workflow):
    def __init__(
        self,
        *args: Any,
        llm: FunctionCallingLLM | None = None,
        tools: List[BaseTool] | None = None,
        **kwargs: Any,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.tools = tools or []

        self.llm = llm or OpenAI()
        assert self.llm.metadata.is_function_calling_model

    @step
    async def prepare_chat_history(
        self, ctx: Context, ev: StartEvent
    ) -> InputEvent:
        # clear sources
        await ctx.set("sources", [])

        # check if memory is setup
        memory = await ctx.get("memory", default=None)
        if not memory:
            memory =
ChatMemoryBuffer.from_defaults(llm=self.llm)

        # get user input
        user_input = ev.input
        user_msg = ChatMessage(role="user", content=user_input)
        memory.put(user_msg)
```

```
        # get chat history
        chat_history = memory.get()

        # update context
        await ctx.set("memory", memory)

        return InputEvent(input=chat_history)

    @step
    async def handle_llm_input(
        self, ctx: Context, ev: InputEvent
    ) -> ToolCallEvent | StopEvent:
        chat_history = ev.input

        # stream the response
        response_stream = await
self.llm.astream_chat_with_tools(
            self.tools, chat_history=chat_history
        )
        async for response in response_stream:

ctx.write_event_to_stream(StreamEvent(delta=response.delta or
""))

        # save the final response, which should have all content
        memory = await ctx.get("memory")
        memory.put(response.message)
        await ctx.set("memory", memory)

        # get tool calls
        tool_calls = self.llm.get_tool_calls_from_response(
            response, error_on_no_tool_call=False
        )

        if not tool_calls:
            sources = await ctx.get("sources", default=[])
            return StopEvent(
                result={"response": response, "sources":
[*sources]}
            )
        else:
            return ToolCallEvent(tool_calls=tool_calls)

    @step
    async def handle_tool_calls(
        self, ctx: Context, ev: ToolCallEvent
    ) -> InputEvent:
        tool_calls = ev.tool_calls
        tools_by_name = {tool.metadata.get_name(): tool for tool
in self.tools}
```

```
        tool_msgs = []
        sources = await ctx.get("sources", default=[])

        # call tools -- safely!
        for tool_call in tool_calls:
            tool = tools_by_name.get(tool_call.tool_name)
            additional_kwargs = {
                "tool_call_id": tool_call.tool_id,
                "name": tool.metadata.get_name(),
            }
            if not tool:
                tool_msgs.append(
                    ChatMessage(
                        role="tool",
                        content=f"Tool {tool_call.tool_name}
does not exist",
                        additional_kwargs=additional_kwargs,
                    )
                )
                continue

            try:
                tool_output = tool(**tool_call.tool_kwargs)
                sources.append(tool_output)
                tool_msgs.append(
                    ChatMessage(
                        role="tool",
                        content=tool_output.content,
                        additional_kwargs=additional_kwargs,
                    )
                )
            except Exception as e:
                tool_msgs.append(
                    ChatMessage(
                        role="tool",
                        content=f"Encountered error in tool
call: {e}",
                        additional_kwargs=additional_kwargs,
                    )
                )

        # update memory
        memory = await ctx.get("memory")
        for msg in tool_msgs:
            memory.put(msg)

        await ctx.set("sources", sources)
        await ctx.set("memory", memory)
```

```
        chat_history = memory.get()
        return InputEvent(input=chat_history)
```

And thats it! Let's explore the workflow we wrote a bit.

`prepare_chat_history()` : This is our main entry point. It handles adding the user message to memory, and uses the memory to get the latest chat history. It returns an `InputEvent` .

`handle_llm_input()` : Triggered by an `InputEvent` , it uses the chat history and tools to prompt the llm. If tool calls are found, a `ToolCallEvent` is emitted. Otherwise, we say the workflow is done an emit a `StopEvent`

`handle_tool_calls()` : Triggered by `ToolCallEvent` , it calls tools with error handling and returns tool outputs. This event triggers a **loop** since it emits an `InputEvent` , which takes us back to `handle_llm_input()`

Run the Workflow!

NOTE: With loops, we need to be mindful of runtime. Here, we set a timeout of 120s.

```
from llama_index.core.tools import FunctionTool
from llama_index.llms.openai import OpenAI


def add(x: int, y: int) -> int:
    """Useful function to add two numbers."""
    return x + y


def multiply(x: int, y: int) -> int:
    """Useful function to multiply two numbers."""
    return x * y


tools = [
    FunctionTool.from_defaults(add),
    FunctionTool.from_defaults(multiply),
]

agent = FuncationCallingAgent(
    llm=OpenAI(model="gpt-4o-mini"), tools=tools, timeout=120,
verbose=True
)

ret = await agent.run(input="Hello!")
```

```
Running step prepare_chat_history
Step prepare_chat_history produced event InputEvent
Running step handle_llm_input
Step handle_llm_input produced event StopEvent
```

```
print(ret["response"])
```

```
assistant: Hello! How can I assist you today?
```

```
ret = await agent.run(input="What is (2123 + 2321) * 312?")
```

```
Running step prepare_chat_history
Step prepare_chat_history produced event InputEvent
Running step handle_llm_input
Step handle_llm_input produced event ToolCallEvent
Running step handle_tool_calls
Step handle_tool_calls produced event InputEvent
Running step handle_llm_input
Step handle_llm_input produced event ToolCallEvent
Running step handle_tool_calls
Step handle_tool_calls produced event InputEvent
Running step handle_llm_input
Step handle_llm_input produced event StopEvent
```

Chat History

By default, the workflow is creating a fresh `Context` for each run. This means that the chat history is not preserved between runs. However, we can pass our own `Context` to the workflow to preserve chat history.

```
from llama_index.core.workflow import Context

ctx = Context(agent)

ret = await agent.run(input="Hello! My name is Logan.", ctx=ctx)
print(ret["response"])

ret = await agent.run(input="What is my name?", ctx=ctx)
print(ret["response"])
```

```
Running step prepare_chat_history
Step prepare_chat_history produced event InputEvent
Running step handle_llm_input
Step handle_llm_input produced event StopEvent
assistant: Hello, Logan! How can I assist you today?
Running step prepare_chat_history
Step prepare_chat_history produced event InputEvent
Running step handle_llm_input
```

```
Step handle_llm_input produced event StopEvent
assistant: Your name is Logan.
```

Streaming

Using the `handler` returned from the `.run()` method, we can also access the streaming events.

```python
agent = FuncationCallingAgent(
    llm=OpenAI(model="gpt-4o-mini"), tools=tools, timeout=120,
verbose=False
)

handler = agent.run(input="Hello! Write me a short story about a
cat.")

async for event in handler.stream_events():
    if isinstance(event, StreamEvent):
        print(event.delta, end="", flush=True)

response = await handler
# print(ret["response"])
```

```
Once upon a time in a quaint little village, there lived a curious
cat named Whiskers. Whiskers was no ordinary cat; he had a beautif
ul coat of orange and white fur that shimmered in the sunlight, an
d his emerald green eyes sparkled with mischief.

Every day, Whiskers would explore the village, visiting the bakery
for a whiff of freshly baked bread and the flower shop to sniff th
e colorful blooms. The villagers adored him, often leaving out lit
tle treats for their favorite feline.

One sunny afternoon, while wandering near the edge of the village,
Whiskers stumbled upon a hidden path that led into the woods. His
curiosity piqued, he decided to follow the path, which was lined w
ith tall trees and vibrant wildflowers. As he ventured deeper, he
heard a soft, melodic sound that seemed to beckon him.

Following the enchanting music, Whiskers soon found himself in a c
learing where a group of woodland creatures had gathered. They wer
e having a grand celebration, complete with dancing, singing, and
a feast of berries and nuts. The animals welcomed Whiskers with op
en paws, inviting him to join their festivities.

Whiskers, delighted by the warmth and joy of his new friends, danc
ed and played until the sun began to set. As the sky turned shades
of pink and orange, he realized it was time to return home. The wo
```

odland creatures gifted him a small, sparkling acorn as a token of their friendship.

From that day on, Whiskers would often visit the clearing, sharing stories of the village and enjoying the company of his woodland fr iends. He learned that adventure and friendship could be found in the most unexpected places, and he cherished every moment spent in the magical woods.

And so, Whiskers continued to live his life filled with curiosity, laughter, and the warmth of friendship, reminding everyone that so metimes, the best adventures are just a whisker away.