FunctionAgent / AgentWorkflow Basic Introduction

The `AgentWorkflow` is an orchestrator for running a system of one or more agents. In this example, we'll create a simple workflow with a single `FunctionAgent`, and use that to cover the basic functionality.

```
%pip install llama-index
```

Setup

In this example, we will use `OpenAI` as our LLM. For all LLMs, check out the examples documentation or LlamaHub for a list of all supported LLMs and how to install/use them.

```
from llama_index.llms.openai import OpenAI

llm = OpenAI(model="gpt-4o-mini", api_key="sk-...")
```

To make our agent more useful, we can give it tools/actions to use. In this case, we'll use Tavily to implement a tool that can search the web for information. You can get a free API key from Tavily.

```
%pip install tavily-python
```

When creating a tool, its very important to:

- give the tool a proper name and docstring/description. The LLM uses this to understand what the tool does.
- annotate the types. This helps the LLM understand the expected input and output types.
- use async when possible, since this will make the workflow more efficient.

```
from tavily import AsyncTavilyClient
```

```
async def search_web(query: str) -> str:
    """Useful for using the web to answer questions."""
    client = AsyncTavilyClient(api_key="tvly-...")
    return str(await client.search(query))
```

With the tool and and LLM defined, we can create an `AgentWorkflow` that uses the tool.

```
from llama_index.core.agent.workflow import FunctionAgent

agent = FunctionAgent(
    tools=[search_web],
    llm=llm,
    system_prompt="You are a helpful assistant that can search the web for information.",
)
```

Running the Agent

Now that our agent is created, we can run it!

```
response = await agent.run(user_msg="What is the weather in San Francisco?")
print(str(response))
```

```
The current weather in San Francisco is as follows:

- **Temperature**: 16.1°C (61°F)
- **Condition**: Partly cloudy
- **Wind**: 13.6 mph (22.0 kph) from the west
- **Humidity**: 64%
- **Visibility**: 16 km (9 miles)
- **Pressure**: 1017 mb (30.04 in)

For more details, you can check the full report [here](https://www.weatherapi.com/).
```

The above is the equivalent of the following of using `AgentWorkflow` with a single `FunctionAgent`:

```
from llama_index.core.agent.workflow import AgentWorkflow

workflow = AgentWorkflow(agents=[agent])

response = await workflow.run(user_msg="What is the weather in
San Francisco?")
```

If you were creating a workflow with multiple agents, you can pass in a list of agents to the `AgentWorkflow` constructor. Learn more in our multi-agent workflow example.

Maintaining State

By default, the `FunctionAgent` will maintain stateless between runs. This means that the agent will not have any memory of previous runs.

To maintain state, we need to keep track of the previous state. Since the `FunctionAgent` is running in a `Workflow`, the state is stored in the `Context`. This can be passed between runs to maintain state and history.

```
from llama_index.core.workflow import Context

ctx = Context(agent)
```

```
response = await agent.run(
    user_msg="My name is Logan, nice to meet you!", ctx=ctx
)
print(str(response))
```

```
Nice to meet you, Logan! How can I assist you today?
```

```
response = await agent.run(user_msg="What is my name?", ctx=ctx)
print(str(response))
```

```
Your name is Logan.
```

The context is serializable, so it can be saved to a database, file, etc. and loaded back in later.

The `JsonSerializer` is a simple serializer that uses `json.dumps` and `json.loads` to serialize and deserialize the context.

The `JsonPickleSerializer` is a serializer that uses `pickle` to serialize and deserialize the context. If you have objects in your context that are not serializable, you can use this serializer.

```
from llama_index.core.workflow import JsonPickleSerializer,
JsonSerializer

ctx_dict = ctx.to_dict(serializer=JsonSerializer())

restored_ctx = Context.from_dict(agent, ctx_dict,
serializer=JsonSerializer())
```

```
response = await agent.run(
    user_msg="Do you still remember my name?", ctx=restored_ctx
)
print(str(response))
```

```
Yes, I remember your name is Logan.
```

Streaming

The `AgentWorkflow` / `FunctionAgent` also supports streaming. Since the `AgentWorkflow` is a `Workflow`, it can be streamed like any other `Workflow`. This works by using the handler that is returned from the workflow. There are a few key events that are streamed, feel free to explore below.

If you only want to stream the LLM output, you can use the `AgentStream` events.

```
from llama_index.core.agent.workflow import (
    AgentInput,
    AgentOutput,
    ToolCall,
    ToolCallResult,
    AgentStream,
)

handler = agent.run(user_msg="What is the weather in
Saskatoon?")

async for event in handler.stream_events():
    if isinstance(event, AgentStream):
        print(event.delta, end="", flush=True)
        # print(event.response)  # the current full response
        # print(event.raw)  # the raw llm api response
        # print(event.current_agent_name)  # the current agent
name
    # elif isinstance(event, AgentInput):
    #    print(event.input)  # the current input messages
    #    print(event.current_agent_name)  # the current agent
name
```

```
    # elif isinstance(event, AgentOutput):
    #    print(event.response)  # the current full response
    #    print(event.tool_calls)  # the selected tool calls, if
any
    #    print(event.raw)  # the raw llm api response
    # elif isinstance(event, ToolCallResult):
    #    print(event.tool_name)  # the tool name
    #    print(event.tool_kwargs)  # the tool kwargs
    #    print(event.tool_output)  # the tool output
    # elif isinstance(event, ToolCall):
    #     print(event.tool_name)  # the tool name
    #     print(event.tool_kwargs)  # the tool kwargs
```

```
The current weather in Saskatoon is as follows:

- **Temperature**: 22.2°C (72°F)
- **Condition**: Overcast
- **Humidity**: 25%
- **Wind Speed**: 6.0 mph (9.7 kph) from the northwest
- **Visibility**: 4.8 km
- **Pressure**: 1018 mb

For more details, you can check the full report [here](https://ww
w.weatherapi.com/).
```

Tools and State

Tools can also be defined that have access to the workflow context. This means you can set and retrieve variables from the context and use them in the tool or between tools.

Note: The `Context` parameter should be the first parameter of the tool.

```
from llama_index.core.workflow import Context


async def set_name(ctx: Context, name: str) -> str:
    state = await ctx.get("state")
    state["name"] = name
    await ctx.set("state", state)
    return f"Name set to {name}"


agent = FunctionAgent(
    tools=[set_name],
    llm=llm,
    system_prompt="You are a helpful assistant that can set a
name.",
    initial_state={"name": "unset"},
```

```
)

ctx = Context(agent)

response = await agent.run(user_msg="My name is Logan", ctx=ctx)
print(str(response))

state = await ctx.get("state")
print(state["name"])
```

```
Your name has been set to Logan.
Logan
```

Human in the Loop

Tools can also be defined that involve a human in the loop. This is useful for tasks that require human input, such as confirming a tool call or providing feedback.

Using workflow events, we can emit events that require a response from the user. Here, we use the built-in `InputRequiredEvent` and `HumanResponseEvent` to handle the human in the loop, but you can also define your own events.

`wait_for_event` will emit the `waiter_event` and wait until it sees the `HumanResponseEvent` with the specified `requirements`. The `waiter_id` is used to ensure that we only send one `waiter_event` for each `waiter_id`.

```
from llama_index.core.workflow import (
    Context,
    InputRequiredEvent,
    HumanResponseEvent,
)


async def dangerous_task(ctx: Context) -> str:
    """A dangerous task that requires human confirmation."""

    question = "Are you sure you want to proceed?"
    response = await ctx.wait_for_event(
        HumanResponseEvent,
        waiter_id=question,
        waiter_event=InputRequiredEvent(
            prefix=question,
            user_name="Logan",
        ),
        requirements={"user_name": "Logan"},
    )
    if response.response == "yes":
        return "Dangerous task completed successfully."
```

```
    else:
        return "Dangerous task aborted."


agent = FunctionAgent(
    tools=[dangerous_task],
    llm=llm,
    system_prompt="You are a helpful assistant that can perform
dangerous tasks.",
)
```

```
handler = agent.run(user_msg="I want to proceed with the
dangerous task.")

async for event in handler.stream_events():
    if isinstance(event, InputRequiredEvent):
        response = input(event.prefix).strip().lower()
        handler.ctx.send_event(
            HumanResponseEvent(
                response=response,
                user_name=event.user_name,
            )
        )

response = await handler
print(str(response))
```

```
The dangerous task has been completed successfully. If you need an
ything else, feel free to ask!
```

In production scenarios, you might handle human-in-the-loop over a websocket or multiple API requests.

As mentioned before, the `Context` object is serializable, and this means we can also save the workflow mid-run and restore it later.

NOTE: Any functions/steps that were in-progress will start from the beginning when the workflow is restored.

```
from llama_index.core.workflow import JsonSerializer

handler = agent.run(user_msg="I want to proceed with the
dangerous task.")

input_ev = None
async for event in handler.stream_events():
    if isinstance(event, InputRequiredEvent):
        input_ev = event
        break
```

```
# save the context somewhere for later
ctx_dict = handler.ctx.to_dict(serializer=JsonSerializer())

# get the response from the user
response_str = input(input_ev.prefix).strip().lower()

# restore the workflow
restored_ctx = Context.from_dict(agent, ctx_dict,
serializer=JsonSerializer())

handler = agent.run(ctx=restored_ctx)
handler.ctx.send_event(
    HumanResponseEvent(
        response=response_str,
        user_name=input_ev.user_name,
    )
)
response = await handler
print(str(response))
```

```
The dangerous task has been completed successfully. If you need an
ything else, feel free to ask!
```