Workflows

A `Workflow` in LlamaIndex is an event-driven abstraction used to chain together several events. Workflows are made up of `steps`, with each step responsible for handling certain event types and emitting new events.

`Workflow`s in LlamaIndex work by decorating function with a `@step` decorator. This is used to infer the input and output types of each workflow for validation, and ensures each step only runs when an accepted event is ready.

You can create a `Workflow` to do anything! Build an agent, a RAG flow, an extraction flow, or anything else you want.

Workflows are also automatically instrumented, so you get observability into each step using tools like Arize Pheonix. (**NOTE:** Observability works for integrations that take advantage of the newer instrumentation system. Usage may vary.)

Tip

Workflows make async a first-class citizen, and this page assumes you are running in an async environment. What this means for you is setting up your code for async properly. If you are already running in a server like FastAPI, or in a notebook, you can freely use await already!

If you are running your own python scripts, its best practice to have a single async entry point.

```
async def main():
    w = MyWorkflow(...)
    result = await w.run(...)
    print(result)


if __name__ == "__main__":
    import asyncio

    asyncio.run(main())
```

Getting Started

As an illustrative example, let's consider a naive workflow where a joke is generated and then critiqued.

```
from llama_index.core.workflow import (
    Event,
    StartEvent,
    StopEvent,
    Workflow,
    step,
)

# `pip install llama-index-llms-openai` if you don't already have it
from llama_index.llms.openai import OpenAI


class JokeEvent(Event):
    joke: str


class JokeFlow(Workflow):
    llm = OpenAI()

    @step
    async def generate_joke(self, ev: StartEvent) -> JokeEvent:
        topic = ev.topic

        prompt = f"Write your best joke about {topic}."
        response = await self.llm.acomplete(prompt)
        return JokeEvent(joke=str(response))

    @step
    async def critique_joke(self, ev: JokeEvent) -> StopEvent:
        joke = ev.joke

        prompt = f"Give a thorough analysis and critique of the following
joke: {joke}"
        response = await self.llm.acomplete(prompt)
        return StopEvent(result=str(response))


w = JokeFlow(timeout=60, verbose=False)
result = await w.run(topic="pirates")
print(str(result))
```

There's a few moving pieces here, so let's go through this piece by piece.

Defining Workflow Events

```
class JokeEvent(Event):
    joke: str
```

Events are user-defined pydantic objects. You control the attributes and any other auxiliary methods. In this case, our workflow relies on a single user-defined event, the `JokeEvent`.

Setting up the Workflow Class

```
class JokeFlow(Workflow):
    llm = OpenAI(model="gpt-4o-mini")
    ...
```

Our workflow is implemented by subclassing the `Workflow` class. For simplicity, we attached a static `OpenAI` llm instance.

Workflow Entry Points

```
class JokeFlow(Workflow):
    ...

    @step
    async def generate_joke(self, ev: StartEvent) -> JokeEvent:
        topic = ev.topic

        prompt = f"Write your best joke about {topic}."
        response = await self.llm.acomplete(prompt)
        return JokeEvent(joke=str(response))

    ...
```

Here, we come to the entry-point of our workflow. While most events are use-defined, there are two special-case events, the `StartEvent` and the `StopEvent` that the framework provides out of the box. Here, the `StartEvent` signifies where to send the initial workflow input.

The `StartEvent` is a bit of a special object since it can hold arbitrary attributes. Here, we accessed the topic with `ev.topic`, which would raise an error if it wasn't there. You could also do `ev.get("topic")` to handle the case where the attribute might not be there without raising an error.

At this point, you may have noticed that we haven't explicitly told the workflow what events are handled by which steps. Instead, the `@step` decorator is used to infer the input and output types of each step. Furthermore, these inferred input and output types are also used to verify for you that the workflow is valid before running!

Workflow Exit Points

```
class JokeFlow(Workflow):
    ...

    @step
    async def critique_joke(self, ev: JokeEvent) -> StopEvent:
        joke = ev.joke

        prompt = f"Give a thorough analysis and critique of the following
joke: {joke}"
        response = await self.llm.acomplete(prompt)
```

```
        return StopEvent(result=str(response))

    ...
```

Here, we have our second, and last step, in the workflow. We know its the last step because the special `StopEvent` is returned. When the workflow encounters a returned `StopEvent`, it immediately stops the workflow and returns whatever we passed in the `result` parameter.

In this case, the result is a string, but it could be a dictionary, list, or any other object.

Running the Workflow

```
w = JokeFlow(timeout=60, verbose=False)
result = await w.run(topic="pirates")
print(str(result))
```

Lastly, we create and run the workflow. There are some settings like timeouts (in seconds) and verbosity to help with debugging.

The `.run()` method is async, so we use await here to wait for the result. The keyword arguments passed to `run()` will become fields of the special `StartEvent` that will be automatically emitted and start the workflow. As we have seen, in this case `topic` will be accessed from the step with `ev.topic`.

Customizing entry and exit points

Most of the times, relying on the default entry and exit points we have seen in the [Getting Started] section is enough. However, workflows support custom events where you normally would expect `StartEvent` and `StopEvent`, let's see how.

Using a custom `StartEvent`

When we call the `run()` method on a workflow instance, the keyword arguments passed become fields of a `StartEvent` instance that's automatically created under the hood. In case we want to pass complex data to start a workflow, this approach might become cumbersome, and it's when we can introduce a custom start event.

To be able to use a custom start event, the first step is creating a custom class that inherits from `StartEvent`:

```
from pathlib import Path

from llama_index.core.workflow import StartEvent
from llama_index.indices.managed.llama_cloud import LlamaCloudIndex
from llama_index.llms.openai import OpenAI
```

```
class MyCustomStartEvent(StartEvent):
    a_string_field: str
    a_path_to_somewhere: Path
    an_index: LlamaCloudIndex
    an_llm: OpenAI
```

All we have to do now is using `MyCustomStartEvent` as event type in the steps that act as entry points. Take this artificially complex step for example:

```
class JokeFlow(Workflow):
    ...

    @step
    async def generate_joke_from_index(
        self, ev: MyCustomStartEvent
    ) -> JokeEvent:
        # Build a query engine using the index and the llm from the start
event
        query_engine = ev.an_index.as_query_engine(llm=ev.an_llm)
        topic = query_engine.query(
            f"What is the closest topic to {a_string_field}"
        )
        # Use the llm attached to the start event to instruct the model
        prompt = f"Write your best joke about {topic}."
        response = await ev.an_llm.acomplete(prompt)
        # Dump the response on disk using the Path object from the event
        ev.a_path_to_somewhere.write_text(str(response))
        # Finally, pass the JokeEvent along
        return JokeEvent(joke=str(response))
```

We could still pass the fields of `MyCustomStartEvent` as keyword arguments to the `run` method of our workflow, but that would be, again, cumbersome. A better approach is to use pass the event instance through the `start_event` keyword argument like this:

```
custom_start_event = MyCustomStartEvent(...)
w = JokeFlow(timeout=60, verbose=False)
result = await w.run(start_event=custom_start_event)
print(str(result))
```

This approach makes the code cleaner and more explicit and allows autocompletion in IDEs to work properly.

Using a custom `StopEvent`

Similarly to `StartEvent`, relying on the built-in `StopEvent` works most of the times but not always. In fact, when we use `StopEvent`, the result of a workflow must be set to the `result` field of the event instance. Since a result can be any Python object, the `result` field of `StopEvent` is typed as `Any`, losing any advantage from the typing system. Additionally,

returning more than one object is cumbersome: we usually stuff a bunch of unrelated objects into a dictionary that we then assign to `StopEvent.result`.

First step to support custom stop events, we need to create a subclass of `StopEvent`:

```
from llama_index.core.workflow import StopEvent


class MyStopEvent(StopEvent):
    critique: CompletionResponse
```

We can now replace `StopEvent` with `MyStopEvent` in our workflow:

```
class JokeFlow(Workflow):
    ...

    @step
    async def critique_joke(self, ev: JokeEvent) -> MyStopEvent:
        joke = ev.joke

        prompt = f"Give a thorough analysis and critique of the following
joke: {joke}"
        response = await self.llm.acomplete(prompt)
        return MyStopEvent(response)

    ...
```

The one important thing we need to remember when using a custom stop events, is that the result of a workflow run will be the instance of the event:

```
w = JokeFlow(timeout=60, verbose=False)
# Warning! `result` now contains an instance of MyStopEvent!
result = await w.run(topic="pirates")
# We can now access the event fields as any normal Event
print(result.critique.text)
```

This approach takes advantage of the Python typing system, is friendly to autocompletion in IDEs and allows introspection from outer applications that now know exactly what a workflow run will return.

Drawing the Workflow

Workflows can be visualized, using the power of type annotations in your step definitions. You can either draw all possible paths through the workflow, or the most recent execution, to help with debugging.

First install:

```
pip install llama-index-utils-workflow
```

Then import and use:

```
from llama_index.utils.workflow import (
    draw_all_possible_flows,
    draw_most_recent_execution,
)

# Draw all
draw_all_possible_flows(JokeFlow, filename="joke_flow_all.html")

# Draw an execution
w = JokeFlow()
await w.run(topic="Pirates")
draw_most_recent_execution(w, filename="joke_flow_recent.html")
```

Working with Global Context/State

Optionally, you can choose to use global context between steps. For example, maybe multiple steps access the original `query` input from the user. You can store this in global context so that every step has access.

```
from llama_index.core.workflow import Context


@step
async def query(self, ctx: Context, ev: MyEvent) -> StopEvent:
    # retrieve from context
    query = await ctx.get("query")

    # do something with context and event
    val = ...
    result = ...

    # store in context
    await ctx.set("key", val)

    return StopEvent(result=result)
```

Waiting for Multiple Events

The context does more than just hold data, it also provides utilities to buffer and wait for multiple events.

For example, you might have a step that waits for a query and retrieved nodes before synthesizing a response:

```
from llama_index.core import get_response_synthesizer


@step
async def synthesize(
    self, ctx: Context, ev: QueryEvent | RetrieveEvent
) -> StopEvent | None:
    data = ctx.collect_events(ev, [QueryEvent, RetrieveEvent])
    # check if we can run
    if data is None:
        return None

    # unpack -- data is returned in order
    query_event, retrieve_event = data

    # run response synthesis
    synthesizer = get_response_synthesizer()
    response = synthesizer.synthesize(
        query_event.query, nodes=retrieve_event.nodes
    )

    return StopEvent(result=response)
```

Using `ctx.collect_events()` we can buffer and wait for ALL expected events to arrive. This function will only return data (in the requested order) once all events have arrived.

Manually Triggering Events

Normally, events are triggered by returning another event during a step. However, events can also be manually dispatched using the `ctx.send_event(event)` method within a workflow.

Here is a short toy example showing how this would be used:

```
from llama_index.core.workflow import step, Context, Event, Workflow


class MyEvent(Event):
    pass


class MyEventResult(Event):
    result: str


class GatherEvent(Event):
    pass


class MyWorkflow(Workflow):
    @step
    async def dispatch_step(
        self, ctx: Context, ev: StartEvent
    ) -> MyEvent | GatherEvent:
```

```
        ctx.send_event(MyEvent())
        ctx.send_event(MyEvent())

        return GatherEvent()

    @step
    async def handle_my_event(self, ev: MyEvent) -> MyEventResult:
        return MyEventResult(result="result")

    @step
    async def gather(
        self, ctx: Context, ev: GatherEvent | MyEventResult
    ) -> StopEvent | None:
        # wait for events to finish
        events = ctx.collect_events(ev, [MyEventResult, MyEventResult])
        if not events:
            return None

        return StopEvent(result=events)
```

Streaming Events

You can also iterate over events as they come in. This is useful for streaming purposes, showing progress, or for debugging. The handler object will emit events that are explicitly written to the stream using `ctx.write_event_to_stream()`:

```
class ProgressEvent(Event):
    msg: str


class MyWorkflow(Workflow):
    @step
    async def step_one(self, ctx: Context, ev: StartEvent) -> FirstEvent:
        ctx.write_event_to_stream(ProgressEvent(msg="Step one is happening"))
        return FirstEvent(first_output="First step complete.")
```

You can then pick up the events like this:

```
w = MyWorkflow(...)

handler = w.run(topic="Pirates")

async for event in handler.stream_events():
    print(event)

result = await handler
```

Retry steps execution in case of failures

A step that fails its execution might result in the failure of the entire workflow, but oftentimes errors are expected and the execution can be safely retried. Think of a HTTP request that times out because of a transient congestion of the network, or an external API call that hits a rate limiter.

For all those situation where you want the step to try again, you can use a "Retry Policy". A retry policy is an object that instructs the workflow to execute a step multiple times, dictating how much time has to pass before a new attempt. Policies take into consideration how much time passed since the first failure, how many consecutive failures happened and which was the last error occurred.

To set a policy for a specific step, all you have to do is passing a policy object to the `@step` decorator:

```
from llama_index.core.workflow.retry_policy import ConstantDelayRetryPolicy


class MyWorkflow(Workflow):
    # ...more workflow definition...

    # This policy will retry this step on failure every 5 seconds for at most
10 times
    @step(retry_policy=ConstantDelayRetryPolicy(delay=5, maximum_attempts=10))
    async def flaky_step(self, ctx: Context, ev: StartEvent) -> StopEvent:
        result = flaky_call()  # this might raise
        return StopEvent(result=result)
```

You can see the API docs for a detailed description of the policies available in the framework. If you can't find a policy that's suitable for your use case, you can easily write a custom one. The only requirement for custom policies is to write a Python class that respects the `RetryPolicy` protocol. In other words, your custom policy class must have a method with the following signature:

```
def next(
    self, elapsed_time: float, attempts: int, error: Exception
) -> Optional[float]:
    ...
```

For example, this is a retry policy that's excited about the weekend and only retries a step if it's Friday:

```
from datetime import datetime


class RetryOnFridayPolicy:
    def next(
        self, elapsed_time: float, attempts: int, error: Exception
    ) -> Optional[float]:
        if datetime.today().strftime("%A") == "Friday":
```

```
            # retry in 5 seconds
            return 5
        # tell the workflow we don't want to retry
        return None
```

Human-in-the-loop

Since workflows are so flexible, there are many possible ways to implement human-in-the-loop patterns.

The easiest way to implement a human-in-the-loop is to use the `InputRequiredEvent` and `HumanResponseEvent` events during event streaming.

```
from llama_index.core.workflow import InputRequiredEvent, HumanResponseEvent


class HumanInTheLoopWorkflow(Workflow):
    @step
    async def step1(self, ev: StartEvent) -> InputRequiredEvent:
        return InputRequiredEvent(prefix="Enter a number: ")

    @step
    async def step2(self, ev: HumanResponseEvent) -> StopEvent:
        return StopEvent(result=ev.response)


# workflow should work with streaming
workflow = HumanInTheLoopWorkflow()

handler = workflow.run()
async for event in handler.stream_events():
    if isinstance(event, InputRequiredEvent):
        # here, we can handle human input however you want
        # this means using input(), websockets, accessing async state, etc.
        # here, we just use input()
        response = input(event.prefix)
        handler.ctx.send_event(HumanResponseEvent(response=response))

final_result = await handler
```

Here, the workflow will wait until the `HumanResponseEvent` is emitted.

Also note that you can break out of the loop, and resume it later. This is useful if you want to pause the workflow to wait for a human response, but continue the workflow later.

```
handler = workflow.run()
async for event in handler.stream_events():
    if isinstance(event, InputRequiredEvent):
        break

# now we handle the human response
response = input(event.prefix)
```

```
handler.ctx.send_event(HumanResponseEvent(response=response))

# now we resume the workflow streaming
async for event in handler.stream_events():
    continue

final_result = await handler
```

Stepwise Execution

Workflows have built-in utilities for stepwise execution, allowing you to control execution and debug state as things progress.

```
# Create a workflow, same as usual
workflow = JokeFlow()
# Get the handler. Passing `stepwise=True` will block execution, waiting for
manual intervention
handler = workflow.run(stepwise=True)
# Each time we call `run_step`, the workflow will advance and return all the
events
# that were produced in the last step. This events need to be manually
propagated
# for the workflow to keep going (we assign them to `produced_events` with the
:= operator).
while produced_events := await handler.run_step():
    # If we're here, it means there's at least an event we need to propagate,
    # let's do it with `send_event`
    for ev in produced_events:
        handler.ctx.send_event(ev)

# If we're here, it means the workflow execution completed, and
# we can now access the final result.
result = await handler
```

Decorating non-class Functions

You can also decorate and attach steps to a workflow without subclassing it.

Below is the `JokeFlow` from earlier, but defined without subclassing.

```
from llama_index.core.workflow import (
    Event,
    StartEvent,
    StopEvent,
    Workflow,
    step,
)
from llama_index.llms.openai import OpenAI


class JokeEvent(Event):
```

```
    joke: str


joke_flow = Workflow(timeout=60, verbose=True)


@step(workflow=joke_flow)
async def generate_joke(ev: StartEvent) -> JokeEvent:
    topic = ev.topic

    prompt = f"Write your best joke about {topic}."

    llm = OpenAI()
    response = await llm.acomplete(prompt)
    return JokeEvent(joke=str(response))


@step(workflow=joke_flow)
async def critique_joke(ev: JokeEvent) -> StopEvent:
    joke = ev.joke

    prompt = (
        f"Give a thorough analysis and critique of the following joke: {joke}"
    )
    response = await llm.acomplete(prompt)
    return StopEvent(result=str(response))
```

Maintaining Context Across Runs

As you have seen, workflows have a `Context` object that can be used to maintain state across steps.

If you want to maintain state across multiple runs of a workflow, you can pass a previous context into the `.run()` method.

```
handler = w.run()
result = await handler

# continue with next run
handler = w.run(ctx=handler.ctx)
result = await handler
```

Resources

Resources are external dependencies you can inject into the steps of a workflow.

As a simple example, look at `memory` in the following workflow:

```
from llama_index.core.workflow.resource import Resource
from llama_index.core.memory import Memory
```

```
def get_memory(*args, **kwargs):
    return Memory.from_defaults("user_id_123", token_limit=60000)


class SecondEvent(Event):
    msg: str


class WorkflowWithResource(Workflow):
    @step
    async def first_step(
        self,
        ev: StartEvent,
        memory: Annotated[Memory, Resource(get_memory)],
    ) -> SecondEvent:
        print("Memory before step 1", memory)
        await memory.aput(
            ChatMessage(role="user", content="This is the first step")
        )
        print("Memory after step 1", memory)
        return SecondEvent(msg="This is an input for step 2")

    @step
    async def second_step(
        self, ev: SecondEvent, memory: Annotated[Memory, Resource(get_memory)]
    ) -> StopEvent:
        print("Memory before step 2", memory)
        await memory.aput(ChatMessage(role="user", content=ev.msg))
        print("Memory after step 2", memory)
        return StopEvent(result="Messages put into memory")
```

To inject a resource into a workflow step, you have to add a parameter to the step signature and define its type, using `Annotated` and invoke the `Resource()` wrapper passing a function or callable returning the actual Resource object. The return type of the wrapped function must match the declared type, ensuring consistency between what's expected and what's provided during execution. In the example above, `memory: Annotated[Memory, Resource(get_memory)` defines a resource of type `Memory` that will be provided by the `get_memory()` function and passed to the step in the `memory` parameter when the workflow runs.

Resources are shared among steps of a workflow, and the `Resource()` wrapper will invoke the factory function only once. In case this is not the desired behavior, passing `cache=False` to `Resource()` will inject different resource objects in different steps, invoking the factory function as many times.

Checkpointing Workflows

Workflow runs can also be made to create and store checkpoints upon every step completion via the `WorfklowCheckpointer` object. These checkpoints can be then be used as the starting

points for future runs, which can be a helpful feature during the development (and debugging) of your Workflow.

```
from llama_index.core.workflow import WorkflowCheckpointer

w = JokeFlow(...)
w_cptr = WorkflowCheckpointer(workflow=w)

# to checkpoint a run, use the `run` method from w_cptr
handler = w_cptr.run(topic="Pirates")
await handler

# to view the stored checkpoints of this run
w_cptr.checkpoints[handler.run_id]

# to run from one of the checkpoints, use `run_from` method
ckpt = w_cptr.checkpoints[handler.run_id][0]
handler = w_cptr.run_from(topic="Ships", checkpoint=ckpt)
await handler
```

Deploying a Workflow

You can deploy a workflow as a multi-agent service with llama_deploy (repo). Each agent service is orchestrated via a control plane and communicates via a message queue. Deploy locally or on Kubernetes.

Examples

To help you become more familiar with the workflow concept and its features, LlamaIndex documentation offers example notebooks that you can run for hands-on learning:

- Common Workflow Patterns walks you through common usage patterns like looping and state management using simple workflows. It's usually a great place to start.
- RAG + Reranking shows how to implement a real-world use case with a fairly simple workflow that performs both ingestion and querying.
- Citation Query Engine similar to RAG + Reranking, the notebook focuses on how to implement intermediate steps in between retrieval and generation. A good example of how to use the `Context` object in a workflow.
- Corrective RAG adds some more complexity on top of a RAG workflow, showcasing how to query a web search engine after an evaluation step.
- Utilizing Concurrency explains how to manage the parallel execution of steps in a workflow, something that's important to know as your workflows grow in complexity.

RAG applications are easy to understand and offer a great opportunity to learn the basics of workflows. However, more complex agentic scenarios involving tool calling, memory, and routing are where workflows excel.

The examples below highlight some of these use-cases.

- ReAct Agent is obviously the perfect example to show how to implement tools in a workflow.
- Function Calling Agent is a great example of how to use the LlamaIndex framework primitives in a workflow, keeping it small and tidy even in complex scenarios like function calling.
- CodeAct Agent is a great example of how to create a CodeAct Agent from scratch.
- Human In The Loop: Story Crafting is a powerful example showing how workflow runs can be interactive and stateful. In this case, to collect input from a human.
- Reliable Structured Generation shows how to implement loops in a workflow, in this case to improve structured output through reflection.
- Query Planning with Workflows is an example of a workflow that plans a query by breaking it down into smaller items, and executing those smaller items. It highlights how to stream events from a workflow, execute steps in parallel, and looping until a condition is met.
- Checkpointing Workflows is a more exhaustive demonstration of how to make full use of `WorkflowCheckpointer` to checkpoint Workflow runs.

Last but not least, a few more advanced use cases that demonstrate how workflows can be extremely handy if you need to quickly implement prototypes, for example from literature:

- Advanced Text-to-SQL
- JSON Query Engine
- Long RAG
- Multi-Step Query Engine
- Multi-Strategy Workflow
- Router Query Engine
- Self Discover Workflow
- Sub-Question Query Engine