Streaming

LangGraph is built with first class support for streaming. There are several different ways to stream back outputs from a graph run

Streaming graph outputs (`.stream` and `.astream`)

`.stream` and `.astream` are sync and async methods for streaming back outputs from a graph run. There are several different modes you can specify when calling these methods (e.g. `.graph.stream(..., mode="...")`):

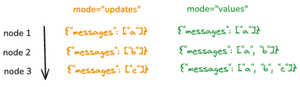
- `"values"`: This streams the full value of the state after each step of the graph.
- `"updates"`: This streams the updates to the state after each step of the graph. If multiple updates are made in the same step (e.g. multiple nodes are run) then those updates are streamed separately.
- `"custom"`: This streams custom data from inside your graph nodes.
- `"messages"`: This streams LLM tokens and metadata for the graph node where LLM is invoked.
- `"debug"`: This streams as much information as possible throughout the execution of the graph.

You can also specify multiple streaming modes at the same time by passing them as a list. When you do this, the streamed outputs will be tuples `(stream_mode, data)`. For example:

```
graph.stream(..., stream_mode=["updates", "messages"])
```

```
...
{'messages', (AIMessageChunk(content='Hi'), {'langgraph_step': 3, 'langgraph_node': 'agent', ...}))
...
{'updates', {'agent': {'messages': [AIMessage(content="Hi, how can I help you?")]}}}
```

The below visualization shows the difference between the `values` and `updates` modes:



Streaming LLM tokens and events (`.astream_events`)

In addition, you can use the `astream_events` method to stream back events that happen *inside* nodes. This is useful for streaming tokens of LLM calls.

This is a standard method on all LangChain objects. This means that as the graph is executed, certain events are emitted along the way and can be seen if you run the graph using `.astream_events`.

All events have (among other things) `event`, `name`, and `data` fields. What do these mean?

- `event`: This is the type of event that is being emitted. You can find a detailed table of all callback events and triggers here.
- `name`: This is the name of event.
- `data`: This is the data associated with the event.

What types of things cause events to be emitted?

- each node (runnable) emits `on_chain_start` when it starts execution, `on_chain_stream` during the node execution and `on_chain_end` when the node finishes. Node events will have the node name in the event's `name` field
- the graph will emit `on_chain_start` in the beginning of the graph execution, `on_chain_stream` after each node execution and `on_chain_end` when the graph finishes. Graph events will have the `LangGraph` in the event's `name` field
- Any writes to state channels (i.e. anytime you update the value of one of your state keys) will emit `on_chain_start` and `on_chain_end` events

Additionally, any events that are created inside your nodes (LLM events, tool events, manually emitted events, etc.) will also be visible in the output of `.astream_events`.

To make this more concrete and to see what this looks like, let's see what events are returned when we run a simple graph:

```
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState, START, END

model = ChatOpenAI(model="gpt-4o-mini")


def call_model(state: MessagesState):
    response = model.invoke(state['messages'])
    return {"messages": response}

workflow = StateGraph(MessagesState)
workflow.add_node(call_model)
workflow.add_edge(START, "call_model")
workflow.add_edge("call_model", END)
app = workflow.compile()

inputs = [{"role": "user", "content": "hi!"}]
async for event in app.astream_events({"messages": inputs}, version="v1"):
    kind = event["event"]
    print(f"{kind}: {event['name']}")
```

```
on_chain_start: LangGraph
on_chain_start: __start__
on_chain_end: __start__
on_chain_start: call_model
on_chat_model_start: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_stream: ChatOpenAI
on_chat_model_end: ChatOpenAI
on_chain_start: ChannelWrite<call_model,messages>
on_chain_end: ChannelWrite<call_model,messages>
on_chain_stream: call_model
on_chain_end: call_model
on_chain_stream: LangGraph
on_chain_end: LangGraph
```

We start with the overall graph start (`on_chain_start: LangGraph`). We then write to the `__start__` node (this is special node to handle input). We then start the `call_model` node (`on_chain_start: call_model`). We then start the chat model invocation (`on_chat_model_start: ChatOpenAI`), stream back token by token (`on_chat_model_stream: ChatOpenAI`) and then finish the chat model (`on_chat_model_end: ChatOpenAI`). From there, we write the results back to the channel (`ChannelWrite<call_model,messages>`) and then finish the `call_model` node and then the graph as a whole.

This should hopefully give you a good sense of what events are emitted in a simple graph. But what data do these events contain? Each type of event contains data in a different format. Let's look at what `on_chat_model_stream` events look like. This is an important type of event since it is needed for streaming tokens from an LLM response.

These events look like:

```
{'event': 'on_chat_model_stream',
 'name': 'ChatOpenAI',
 'run_id': '3fdbf494-acce-402e-9b50-4eab46403859',
 'tags': ['seq:step:1'],
 'metadata': {'langgraph_step': 1,
  'langgraph_node': 'call_model',
  'langgraph_triggers': ['start:call_model'],
  'langgraph_task_idx': 0,
  'checkpoint_id': '1ef657a0-0f9d-61b8-bffe-0c39e4f9ad6c',
  'checkpoint_ns': 'call_model',
  'ls_provider': 'openai',
  'ls_model_name': 'gpt-4o-mini',
  'ls_model_type': 'chat',
  'ls_temperature': 0.7},
 'data': {'chunk': AIMessageChunk(content='Hello', id='run-3fdbf494-acce-402e-9b50-4eab46403859')},
 'parent_ids': []}
```

We can see that we have the event type and name (which we knew from before).

We also have a bunch of stuff in metadata. Noticeably, `'langgraph_node': 'call_model',` is some really helpful information which tells us which node this model was invoked inside of.

Finally, `data` is a really important field. This contains the actual data for this event! Which in this case is an AIMessageChunk. This contains the `content` for the message, as well as an `id`. This is the ID of the overall AIMessage (not just this chunk) and is super helpful - it helps us track which chunks are part of the same message (so we can show them together in the UI).

This information contains all that is needed for creating a UI for streaming LLM tokens. You can see a guide for that here.

> **ASYNC IN PYTHON<=3.10**
>
> You may fail to see events being emitted from inside a node when using `.astream_events` in Python <= 3.10. If you're using a Langchain RunnableLambda, a RunnableGenerator, or Tool asynchronously inside your node, you will have to propagate callbacks to these objects manually. This is because LangChain cannot automatically propagate callbacks to child objects in this case. Please see examples here and here.

LangGraph Platform

Streaming is critical for making LLM applications feel responsive to end users. When creating a streaming run, the streaming mode determines what data is streamed back to the API client. LangGraph Platform supports five streaming modes:

- `values`: Stream the full state of the graph after each super-step is executed. See the how-to guide for streaming values.
- `messages-tuple`: Stream LLM tokens for any messages generated inside a node. This mode is primarily meant for powering chat applications. See the how-to guide for streaming messages.
- `updates`: Streams updates to the state of the graph after each node is executed. See the how-to guide for streaming updates.
- `events`: Stream all events (including the state of the graph) that occur during graph execution. See the how-to guide for streaming events. This can be used to do token-by-token streaming for LLMs.
- `debug`: Stream debug events throughout graph execution. See the how-to guide for streaming debug events.

You can also specify multiple streaming modes at the same time. See the how-to guide for configuring multiple streaming modes at the same time.

See the API reference for how to create streaming runs.

Streaming modes `values`, `updates`, `messages-tuple` and `debug` are very similar to modes available in the LangGraph library - for a deeper conceptual explanation of those, you can see the previous section.

Streaming mode `events` is the same as using `.astream_events` in the LangGraph library - for a deeper conceptual explanation of this, you can see the previous section.

All events emitted have two attributes:

- `event`: This is the name of the event
- `data`: This is data associated with the event

Comments