Streaming With LangChain

Streaming is critical in making applications based on LLMs feel responsive to end-users.

Important LangChain primitives like LLMs, parsers, prompts, retrievers, and agents implement the LangChain Runnable Interface.

This interface provides two general approaches to stream content:

1. sync `stream` and async `astream`: a **default implementation** of streaming that streams the **final output** from the chain.
2. async `astream_events` and async `astream_log`: these provide a way to stream both **intermediate steps** and **final output** from the chain.

Let’s take a look at both approaches, and try to understand a how to use them. 🥷

Using Stream

All `Runnable` objects implement a sync method called `stream` and an async variant called `astream`.

These methods are designed to stream the final output in chunks, yielding each chunk as soon as it is available.

Streaming is only possible if all steps in the program know how to process an **input stream**; i.e., process an input chunk one at a time, and yield a corresponding output chunk.

The complexity of this processing can vary, from straightforward tasks like emitting tokens produced by an LLM, to more challenging ones like streaming parts of JSON results before the entire JSON is complete.

The best place to start exploring streaming is with the single most important components in LLMs apps– the LLMs themselves!

LLMs and Chat Models

Large language models and their chat variants are the primary bottleneck in LLM based apps. 🙊

Large language models can take **several seconds** to generate a complete response to a query. This is far slower than the **~200-300 ms** threshold at which an application feels responsive to an end user.

The key strategy to make the application feel more responsive is to show intermediate progress; e.g., to stream the output from the model **token by token**.

```
from langchain.chat_models import ChatAnthropic

model = ChatAnthropic()

chunks = []
async for chunk in model.astream("hello. tell me something about yourself"):
    chunks.append(chunk)
    print(chunk.content, end="|", flush=True)
```

```
 Hello|!| My| name| is| Claude|.| I|'m| an| AI| assistant| created| by| An|throp|ic| to| be| helpful|,| harmless|,| and| honest|.||
```

Let's inspect one of the chunks

```
chunks[0]
```

```
AIMessageChunk(content=' Hello')
```

We got back something called an `AIMessageChunk`. This chunk represents a part of an `AIMessage`.

Message chunks are additive by design – one can simply add them up to get the state of the response so far!

```
chunks[0] + chunks[1] + chunks[2] + chunks[3]
+ chunks[4]
```

```
AIMessageChunk(content=' Hello! My name is')
```

Chains

Virtually all LLM applications involve more steps than just a call to a language model.

Let’s build a simple chain using `LangChain Expression Language` (`LCEL`) that combines a prompt, model and a parser and verify that streaming works.

We will use `StrOutputParser` to parse the output from the model. This is a simple parser that extracts the `content` field from an `AIMessageChunk`, giving us the `token` returned by the model.

TIP

LCEL is a *declarative* way to specify a “program” by chainining together different LangChain primitives. Chains created using LCEL benefit from an automatic implementation of `stream`, and `astream` allowing streaming of the final output. In fact, chains created with LCEL implement the entire standard Runnable interface.

```python
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_template("tell me a joke about {topic}")
parser = StrOutputParser()
chain = prompt | model | parser

async for chunk in chain.astream({"topic": "parrot"}):
    print(chunk, end="|", flush=True)
```

```
 Sure|,| here|'s| a| funny| joke| about| a| par|rot|:|
Why| doesn|'t| a| par|rot| ever| get| hungry|
at| night|?| Because| it| has| a| light|
snack| before| bed|!||
```

NOTE

You do not have to use the `LangChain Expression Language` to use LangChain and can instead rely on a standard **imperative** programming approach by caling `invoke`, `batch` or `stream` on each component individually, assigning the results to variables and then using them downstream as you see fit.

If that works for your needs, then that's fine by us 👌!

Working with Input Streams

What if you wanted to stream JSON from the output as it was being generated?

If you were to rely on `json.loads` to parse the partial json, the parsing would fail as the partial json wouldn't be valid json.

You'd likely be at a complete loss of what to do and claim that it wasn't possible to stream JSON.

Well, turns out there is a way to do it – the parser needs to operate on the **input stream**, and attempt to “auto-complete” the partial json into a valid state.

Let’s see such a parser in action to understand what this means.

```
from langchain_core.output_parsers import
JsonOutputParser
from langchain_openai.chat_models import
ChatOpenAI

model = ChatOpenAI()

chain = model | JsonOutputParser()  # This
parser only works with OpenAI right now
async for text in chain.astream(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries" which contains a list of
countries. Each country should have the key
`name` and `population`'
):
    print(text, flush=True)
```

```
{}
{'countries': []}
```

```
{'countries': [{}]}
{'countries': [{'name': ''}]}
{'countries': [{'name': 'France'}]}
{'countries': [{'name': 'France',
'population': ''}]}
{'countries': [{'name': 'France',
'population': '67'}]}
{'countries': [{'name': 'France',
'population': '67,'}]}
{'countries': [{'name': 'France',
'population': '67,022'}]}
{'countries': [{'name': 'France',
'population': '67,022,'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name': ''}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': ''}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,'}]}
```

```
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'}, {}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': ''}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan', 'population': ''}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan', 'population': '126'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan', 'population': '126,'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan', 'population': '126,860'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan', 'population': '126,860,'}]}
{'countries': [{'name': 'France',
'population': '67,022,000'}, {'name':
'Spain', 'population': '46,754,784'},
{'name': 'Japan', 'population':
'126,860,301'}]}
```

Now, let's **break** streaming. We'll use the previous example and append an extraction function at the end that extracts the country names from the finalized JSON.

DANGER

Any steps in the chain that operate on **finalized inputs** rather than on **input streams** can break streaming functionality via `stream` or `astream`.

TIP

Later, we will discuss the `astream_events` API which streams results from intermediate steps. This API will stream results from intermediate steps even if the chain contains steps that only operate on **finalized inputs**.

```
from langchain_core.output_parsers import
JsonOutputParser


# A function that operates on finalized
inputs
# rather than on an input_stream
def _extract_country_names(inputs):
    """A function that does not operates on
input streams and breaks streaming."""
    if not isinstance(inputs, dict):
        return ""

    if "countries" not in inputs:
        return ""

    countries = inputs["countries"]

    if not isinstance(countries, list):
        return ""

    country_names = [
        country.get("name") for country in
countries if isinstance(country, dict)
```

```
    ]
    return country_names


chain = model | JsonOutputParser() |
_extract_country_names

async for text in chain.astream(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries"'
):
    print(text, end="|", flush=True)
```

```
[None, None, None]|
```

Generator Functions

Le’ts fix the streaming using a generator function that can operate on the **input stream**.

TIP

A generator function (a function that uses `yield`) allows writing code that operators on **input streams**

```
from langchain_core.output_parsers import
JsonOutputParser


async def
_extract_country_names_streaming(input_stream):
    """A function that operates on input
streams."""
    country_names_so_far = set()

    async for input in input_stream:
        if not isinstance(input, dict):
            continue

        if "countries" not in input:
            continue

        countries = input["countries"]

        if not isinstance(countries, list):
            continue

        for country in countries:
            name = country.get("name")
            if not name:
                continue
            if name not in
country_names_so_far:
                yield name
                country_names_so_far.add(name)
```

```
chain = model | JsonOutputParser() |
_extract_country_names_streaming

async for text in chain.astream(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries"'
):
    print(text, end="|", flush=True)
```

```
France|Spain|Japan|
```

Non-streaming components

Some built-in components like Retrievers do not offer any `streaming`. What happens if we try to `stream` them? 🤨

```
from langchain_community.vectorstores import
FAISS
from langchain_core.output_parsers import
StrOutputParser
from langchain_core.prompts import
ChatPromptTemplate
from langchain_core.runnables import
RunnablePassthrough
```

```
from langchain_openai import OpenAIEmbeddings

template = """Answer the question based only
on the following context:
{context}

Question: {question}
"""
prompt =
ChatPromptTemplate.from_template(template)

vectorstore = FAISS.from_texts(
    ["harrison worked at kensho", "harrison
likes spicy food"],
    embedding=OpenAIEmbeddings(),
)
retriever = vectorstore.as_retriever()

chunks = [chunk for chunk in
retriever.stream("where did harrison work?")]
chunks
```

```
[[Document(page_content='harrison worked at
kensho'),
  Document(page_content='harrison likes spicy
food')]]
```

Stream just yielded the final result from that component.

This is OK 🥹! Not all components have to implement streaming – in some cases streaming is either unnecessary, difficult or just doesn't make sense.

> **TIP**
>
> An LCEL chain constructed using using non-streaming components, will still be able to stream in a lot of cases, with streaming of partial output starting after the last non-streaming step in the chain.

```
retrieval_chain = (
    {
        "context":
retriever.with_config(run_name="Docs"),
        "question": RunnablePassthrough(),
    }
    | prompt
    | model
    | StrOutputParser()
)
```

```
for chunk in retrieval_chain.stream(
    "Where did harrison work? " "Write 3 made
up sentences about this place."
):
    print(chunk, end="|", flush=True)
```

```
|H|arrison| worked| at| Kens|ho|,| a|
renowned| technology| company| known| for|
revolution|izing| the| artificial|
intelligence| industry|.
|K|ens|ho|,| located| in| the| heart| of|
Silicon| Valley|,| is| famous| for| its|
cutting|-edge| research| and| development|
in| machine| learning|.
|With| its| state|-of|-the|-art| facilities|
and| talented| team|,| Kens|ho| has| become|
a| hub| for| innovation| and| a| sought|-
after| workplace| for| tech| enthusiasts|
like| Harrison|.||
```

Now that we've seen how `stream` and `astream` work, let's venture into the world of streaming events. 🏞️

Using Stream Events

Event Streaming is a **beta** API. This API may change a bit based on feedback.

ⓘ NOTE

Introduced in langchain-core **0.1.14**.

```
import langchain_core

langchain_core.__version__
```

```
'0.1.14'
```

For the `astream_events` API to work properly:

- Use `async` throughout the code to the extent possible (e.g., async tools etc)
- Propagate callbacks if defining custom functions / runnables
- Whenever using runnables without LCEL, make sure to call `.astream()` on LLMs rather than `.ainvoke` to force the LLM to stream tokens.
- Let us know if anything doesn't work as expected! :)

Event Reference

Below is a reference table that shows some events that might be emitted by the various Runnable objects.

NOTE

When streaming is implemented properly, the inputs to a runnable will not be known until after the input stream has been entirely consumed. This means that `inputs` will often be included only for `end` events and rather than for `start` events.

event	name	
on_chat_model_start	[model name]	
on_chat_model_stream	[model name]	AIMessageCh
on_chat_model_end	[model name]	
on_llm_start	[model name]	
on_llm_stream	[model name]	‘Hello’
on_llm_end	[model name]	

event	name	
on_chain_start	format_docs	
on_chain_stream	format_docs	“hello world!
on_chain_end	format_docs	
on_tool_start	some_tool	
on_tool_stream	some_tool	{“x”: 1, “y”: “2
on_tool_end	some_tool	
on_retriever_start	[retriever name]	
on_retriever_chunk	[retriever name]	{documents:
on_retriever_end	[retriever name]	
on_prompt_start	[template_name]	
on_prompt_end	[template_name]	

Chat Model

Let’s start off by looking at the events produced by a chat model.

```
events = []
async for event in
model.astream_events("hello", version="v1"):
    events.append(event)
```

```
/home/eugene/.pyenv/versions/3.11.4/envs/langch
packages/langchain_core/_api/beta_decorator.py:
API is in beta and may change in the future.
  warn_beta(
```

ⓘ NOTE

Hey what’s that funny version=“v1” parameter in the API?! 😾

This is a **beta API**, and we’re almost certainly going to make some changes to it.

This version parameter will allow us to mimimize such breaking changes to your code.

> In short, we are annoying you now, so we don’t have to annoy you later.

Let’s take a look at the few of the start event and a few of the end events.

```
events[:3]
```

```
[{'event': 'on_chat_model_start',
  'run_id': 'd78b4ffb-0eb1-499c-8a90-
8e4a4aa2edae',
  'name': 'ChatOpenAI',
  'tags': [],
  'metadata': {},
  'data': {'input': 'hello'}},
 {'event': 'on_chat_model_stream',
  'run_id': 'd78b4ffb-0eb1-499c-8a90-
8e4a4aa2edae',
  'tags': [],
  'metadata': {},
  'name': 'ChatOpenAI',
  'data': {'chunk':
AIMessageChunk(content='')}},
 {'event': 'on_chat_model_stream',
  'run_id': 'd78b4ffb-0eb1-499c-8a90-
8e4a4aa2edae',
  'tags': [],
  'metadata': {},
  'name': 'ChatOpenAI',
  'data': {'chunk':
AIMessageChunk(content='Hello')}}]
```

```
events[-2:]
```

```
[{'event': 'on_chat_model_stream',
  'run_id': 'd78b4ffb-0eb1-499c-8a90-
8e4a4aa2edae',
  'tags': [],
  'metadata': {},
  'name': 'ChatOpenAI',
  'data': {'chunk':
AIMessageChunk(content='')}},
 {'event': 'on_chat_model_end',
  'name': 'ChatOpenAI',
  'run_id': 'd78b4ffb-0eb1-499c-8a90-
8e4a4aa2edae',
  'tags': [],
  'metadata': {},
  'data': {'output':
AIMessageChunk(content='Hello! How can I
assist you today?')}}]
```

Chain

Let’s revisit the example chain that parsed streaming JSON to explore the streaming events API.

```
chain = model | JsonOutputParser()  # This parser only works with OpenAI right now

events = [
    event
    async for event in chain.astream_events(
        'output a list of the countries france, spain and japan and their populations in JSON format. Use a dict with an outer key of "countries" which contains a list of countries. Each country should have the key `name` and `population`',
        version="v1",
    )
]
```

If you examine at the first few events, you’ll notice that there are **3** different start events rather than **2** start events.

The three start events correspond to:

1. The chain (model + parser)
2. The model
3. The parser

```
events[:3]
```

```
[{'event': 'on_chain_start',
  'run_id': 'aa992fb9-d79f-46f3-a857-
ae4acad841c4',
  'name': 'RunnableSequence',
  'tags': [],
  'metadata': {},
  'data': {'input': 'output a list of the
countries france, spain and japan and their
populations in JSON format. Use a dict with
an outer key of "countries" which contains a
list of countries. Each country should have
the key `name` and `population`'}},
 {'event': 'on_chat_model_start',
  'name': 'ChatOpenAI',
  'run_id': 'c5406de5-0880-4829-ae26-
bb565b404e27',
  'tags': ['seq:step:1'],
  'metadata': {},
  'data': {'input': {'messages':
[[HumanMessage(content='output a list of the
countries france, spain and japan and their
populations in JSON format. Use a dict with
an outer key of "countries" which contains a
list of countries. Each country should have
the key `name` and `population`')]]}}},
 {'event': 'on_parser_start',
  'name': 'JsonOutputParser',
  'run_id': '32b47794-8fb6-4ef4-8800-
23ed6c3f4519',
  'tags': ['seq:step:2'],
  'metadata': {},
  'data': {}}]
```

What do you think you'd see if you looked at the last 3 events? what about the middle?

Let's use this API to take output the stream events from the model and the parser. We're ignoring start events, end events and events from the chain.

```
num_events = 0

async for event in chain.astream_events(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries" which contains a list of
countries. Each country should have the key
`name` and `population`',
    version="v1",
):
    kind = event["event"]
    if kind == "on_chat_model_stream":
        print(
            f"Chat model chunk:
{repr(event['data']['chunk'].content)}",
            flush=True,
        )
    if kind == "on_parser_stream":
        print(f"Parser chunk: {event['data']
['chunk']}", flush=True)
    num_events += 1
    if num_events > 30:
        # Truncate the output
        print("...")
        break
```

```
Chat model chunk: ''
Parser chunk: {}
Chat model chunk: '{\n'
Chat model chunk: ' '
Chat model chunk: ' "'
Chat model chunk: 'countries'
Chat model chunk: '":'
Parser chunk: {'countries': []}
Chat model chunk: ' [\n'
Chat model chunk: '   '
Parser chunk: {'countries': [{}]}
Chat model chunk: ' {\n'
Chat model chunk: '     '
Chat model chunk: ' "'
Chat model chunk: 'name'
Chat model chunk: '":'
Parser chunk: {'countries': [{'name': ''}]}
Chat model chunk: ' "'
Parser chunk: {'countries': [{'name':
```

```
'France'}]}
Chat model chunk: 'France'
Chat model chunk: '",\n'
Chat model chunk: '      '
Chat model chunk: ' "'
...
```

Because both the model and the parser support streaming, we see sreaming events from both components in real time! Kind of cool isn't it? 🦜

Filtering Events

Because this API produces so many events, it is useful to be able to filter on events.

You can filter by either component `name`, component `tags` or component `type`.

By Name

```
chain = model.with_config({"run_name":
"model"}) | JsonOutputParser().with_config(
    {"run_name": "my_parser"}
)

max_events = 0
async for event in chain.astream_events(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries" which contains a list of
countries. Each country should have the key
`name` and `population`',
    version="v1",
    include_names=["my_parser"],
):
    print(event)
    max_events += 1
    if max_events > 10:
        # Truncate output
        print("...")
        break
```

```
{'event': 'on_parser_start', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk': {}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': []}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{}]}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': ''}]}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': 'France'}]}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': 'France',
'population': 670}]}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': 'France',
'population': 670600}]}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': 'France',
'population': 67060000}]}}}
```

```
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': 'France',
'population': 67060000}, {}]}}}
{'event': 'on_parser_stream', 'name':
'my_parser', 'run_id': '450011c0-6f3b-4ec8-
92d4-6603d9d1d603', 'tags': ['seq:step:2'],
'metadata': {}, 'data': {'chunk':
{'countries': [{'name': 'France',
'population': 67060000}, {'name': ''}]}}}
...
```

By Type

```
chain = model.with_config({"run_name":
"model"}) | JsonOutputParser().with_config(
    {"run_name": "my_parser"}
)

max_events = 0
async for event in chain.astream_events(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries" which contains a list of
countries. Each country should have the key
`name` and `population`',
    version="v1",
    include_types=["chat_model"],
):
    print(event)
    max_events += 1
    if max_events > 10:
        # Truncate output
        print("...")
        break
```

```
{'event': 'on_chat_model_start', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'input':
{'messages': [[HumanMessage(content='output a
list of the countries france, spain and japan
and their populations in JSON format. Use a
dict with an outer key of "countries" which
contains a list of countries. Each country
should have the key `name` and
`population`')]]}}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content='')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content='{\n')}}
{'event': 'on_chat_model_stream', 'name':
```

```
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content=' ')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content=' "')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content='countries')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content='":')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content=' [\n')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content='   ')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
```

```
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content=' {\n')}}
{'event': 'on_chat_model_stream', 'name':
'model', 'run_id': '9ba1ef9f-5954-4649-b3da-
1171b6abb000', 'tags': ['seq:step:1'],
'metadata': {}, 'data': {'chunk':
AIMessageChunk(content='     ')}}
...
```

By Tags

CAUTION

Tags are inherited by child components of a given runnable.

If you're using tags to filter, make sure that this is what you want.

```
chain = (model |
JsonOutputParser()).with_config({"tags":
["my_chain"]})

max_events = 0
async for event in chain.astream_events(
    'output a list of the countries france,
spain and japan and their populations in JSON
format. Use a dict with an outer key of
"countries" which contains a list of
countries. Each country should have the key
`name` and `population`',
    version="v1",
    include_tags=["my_chain"],
):
    print(event)
    max_events += 1
    if max_events > 10:
        # Truncate output
        print("...")
        break
```

```
{'event': 'on_chain_start', 'run_id':
'd4c78db8-be20-4fa0-87d6-cb317822967a',
'name': 'RunnableSequence', 'tags':
['my_chain'], 'metadata': {}, 'data':
{'input': 'output a list of the countries
france, spain and japan and their populations
in JSON format. Use a dict with an outer key
of "countries" which contains a list of
countries. Each country should have the key
`name` and `population`'}}
{'event': 'on_chat_model_start', 'name':
'ChatOpenAI', 'run_id': '15e46d9f-ccf5-4da2-
b9e3-b2a85873ba4c', 'tags': ['seq:step:1',
'my_chain'], 'metadata': {}, 'data':
{'input': {'messages':
[[HumanMessage(content='output a list of the
countries france, spain and japan and their
populations in JSON format. Use a dict with
an outer key of "countries" which contains a
list of countries. Each country should have
the key `name` and `population`')]]}}}
{'event': 'on_parser_start', 'name':
'JsonOutputParser', 'run_id': '91945f4f-0deb-
4999-acf0-f6d191c89b34', 'tags':
['seq:step:2', 'my_chain'], 'metadata': {},
'data': {}}
{'event': 'on_chat_model_stream', 'name':
'ChatOpenAI', 'run_id': '15e46d9f-ccf5-4da2-
b9e3-b2a85873ba4c', 'tags': ['seq:step:1',
'my_chain'], 'metadata': {}, 'data':
{'chunk': AIMessageChunk(content='')}}
{'event': 'on_parser_stream', 'name':
'JsonOutputParser', 'run_id': '91945f4f-0deb-
4999-acf0-f6d191c89b34', 'tags':
['seq:step:2', 'my_chain'], 'metadata': {},
'data': {'chunk': {}}}
{'event': 'on_chain_stream', 'run_id':
'd4c78db8-be20-4fa0-87d6-cb317822967a',
'tags': ['my_chain'], 'metadata': {}, 'name':
'RunnableSequence', 'data': {'chunk': {}}}
{'event': 'on_chat_model_stream', 'name':
'ChatOpenAI', 'run_id': '15e46d9f-ccf5-4da2-
b9e3-b2a85873ba4c', 'tags': ['seq:step:1',
'my_chain'], 'metadata': {}, 'data':
{'chunk': AIMessageChunk(content='{"')}}
{'event': 'on_chat_model_stream', 'name':
'ChatOpenAI', 'run_id': '15e46d9f-ccf5-4da2-
b9e3-b2a85873ba4c', 'tags': ['seq:step:1',
'my_chain'], 'metadata': {}, 'data':
{'chunk':
AIMessageChunk(content='countries')}}
{'event': 'on_chat_model_stream', 'name':
'ChatOpenAI', 'run_id': '15e46d9f-ccf5-4da2-
b9e3-b2a85873ba4c', 'tags': ['seq:step:1',
'my_chain'], 'metadata': {}, 'data':
{'chunk': AIMessageChunk(content='":')}}
{'event': 'on_parser_stream', 'name':
'JsonOutputParser', 'run_id': '91945f4f-0deb-
4999-acf0-f6d191c89b34', 'tags':
['seq:step:2', 'my_chain'], 'metadata': {},
'data': {'chunk': {'countries': []}}}
{'event': 'on_chain_stream', 'run_id':
'd4c78db8-be20-4fa0-87d6-cb317822967a',
'tags': ['my_chain'], 'metadata': {}, 'name':
'RunnableSequence', 'data': {'chunk':
{'countries': []}}}
...
```

Non-streaming components

Remember how some components don't stream well because they don't operate on **input streams**?

While such components can break streaming of the final output when using `astream`, `astream_events` will still yield streaming events from intermediate steps that support streaming!

```
# Function that does not support streaming.
# It operates on the finalizes inputs rather
than
# operating on the input stream.
def _extract_country_names(inputs):
    """A function that does not operates on
input streams and breaks streaming."""
    if not isinstance(inputs, dict):
        return ""

    if "countries" not in inputs:
        return ""

    countries = inputs["countries"]

    if not isinstance(countries, list):
        return ""

    country_names = [
        country.get("name") for country in
countries if isinstance(country, dict)
    ]
    return country_names


chain = (
    model | JsonOutputParser() |
_extract_country_names
)  # This parser only works with OpenAI right
now
```

As expected, the `astream` API doesn't work correctly because `_extract_country_names` doesn't operate on streams.

```
async for chunk in chain.astream(
    'output a list of the countries france, spain and japan and their populations in JSON format. Use a dict with an outer key of "countries" which contains a list of countries. Each country should have the key `name` and `population`',
):
    print(chunk, flush=True)
```

Now, let's confirm that with astream_events we're still seeing streaming output from the model and the parser.

```
num_events = 0

async for event in chain.astream_events(
    'output a list of the countries france, spain and japan and their populations in JSON format. Use a dict with an outer key of "countries" which contains a list of countries. Each country should have the key `name` and `population`',
    version="v1",
):
    kind = event["event"]
    if kind == "on_chat_model_stream":
        print(
            f"Chat model chunk:
{repr(event['data']['chunk'].content)}",
            flush=True,
        )
    if kind == "on_parser_stream":
        print(f"Parser chunk: {event['data']
['chunk']}", flush=True)
    num_events += 1
    if num_events > 30:
        # Truncate the output
        print("...")
        break
```

```
Chat model chunk: ''
Parser chunk: {}
Chat model chunk: '{"'
Chat model chunk: 'countries'
Chat model chunk: '":'
Parser chunk: {'countries': []}
Chat model chunk: ' [\n'
Chat model chunk: '    '
Parser chunk: {'countries': [{}]}
Chat model chunk: ' {"'
Chat model chunk: 'name'
Chat model chunk: '":'
Parser chunk: {'countries': [{'name': ''}]}
Chat model chunk: ' "'
Parser chunk: {'countries': [{'name':
'France'}]}
Chat model chunk: 'France'
Chat model chunk: '",'
Chat model chunk: ' "'
Chat model chunk: 'population'
Chat model chunk: '":'
Parser chunk: {'countries': [{'name':
'France', 'population': ''}]}
Chat model chunk: ' "'
Parser chunk: {'countries': [{'name':
'France', 'population': '67'}]}
Chat model chunk: '67'
Parser chunk: {'countries': [{'name':
'France', 'population': '67 million'}]}
Chat model chunk: ' million'
Chat model chunk: '"},\n'
...
```

Propagating Callbacks

⚠ **CAUTION**

If you’re using invoking runnables inside your tools, you need to propagate callbacks to the runnable; otherwise, no stream events will be generated.

NOTE

When using RunnableLambdas or @chain decorator, callbacks are propagated automatically behind the scenes.

```
from langchain_core.runnables import
RunnableLambda
from langchain_core.tools import tool


def reverse_word(word: str):
    return word[::-1]


reverse_word = RunnableLambda(reverse_word)


@tool
def bad_tool(word: str):
    """Custom tool that doesn't propagate
callbacks."""
    return reverse_word.invoke(word)


async for event in
bad_tool.astream_events("hello",
version="v1"):
    print(event)
```

```
{'event': 'on_tool_start', 'run_id':
'39e4a7eb-c13d-46f0-99e7-75c2fa4aa6a6',
'name': 'bad_tool', 'tags': [], 'metadata':
{}, 'data': {'input': 'hello'}}
{'event': 'on_tool_stream', 'run_id':
'39e4a7eb-c13d-46f0-99e7-75c2fa4aa6a6',
'tags': [], 'metadata': {}, 'name':
'bad_tool', 'data': {'chunk': 'olleh'}}
{'event': 'on_tool_end', 'name': 'bad_tool',
'run_id': '39e4a7eb-c13d-46f0-99e7-
75c2fa4aa6a6', 'tags': [], 'metadata': {},
'data': {'output': 'olleh'}}
```

Here's a re-implementation that does propagate callbacks correctly. You'll notice that now we're getting events from the `reverse_word` runnable as well.

```
@tool
def correct_tool(word: str, callbacks):
    """A tool that correctly propagates
callbacks."""
    return reverse_word.invoke(word,
{"callbacks": callbacks})


async for event in
correct_tool.astream_events("hello",
version="v1"):
    print(event)
```

```
{'event': 'on_tool_start', 'run_id':
'4263aca5-f221-4eb7-b07e-60a89fb76c5c',
'name': 'correct_tool', 'tags': [],
'metadata': {}, 'data': {'input': 'hello'}}
{'event': 'on_chain_start', 'name':
'reverse_word', 'run_id': '65e3679b-e238-
47ce-a875-ee74480e696e', 'tags': [],
'metadata': {}, 'data': {'input': 'hello'}}
{'event': 'on_chain_end', 'name':
'reverse_word', 'run_id': '65e3679b-e238-
47ce-a875-ee74480e696e', 'tags': [],
'metadata': {}, 'data': {'input': 'hello',
'output': 'olleh'}}
{'event': 'on_tool_stream', 'run_id':
'4263aca5-f221-4eb7-b07e-60a89fb76c5c',
'tags': [], 'metadata': {}, 'name':
'correct_tool', 'data': {'chunk': 'olleh'}}
{'event': 'on_tool_end', 'name':
'correct_tool', 'run_id': '4263aca5-f221-
4eb7-b07e-60a89fb76c5c', 'tags': [],
'metadata': {}, 'data': {'output': 'olleh'}}
```

If you’re invoking runnables from within Runnable Lambdas or @chains, then callbacks will be passed automatically on your behalf.

```
from langchain_core.runnables import
RunnableLambda


async def reverse_and_double(word: str):
    return await reverse_word.ainvoke(word) *
2


reverse_and_double =
RunnableLambda(reverse_and_double)

await reverse_and_double.ainvoke("1234")

async for event in
reverse_and_double.astream_events("1234",
version="v1"):
    print(event)
```

```
{'event': 'on_chain_start', 'run_id':
'714d22d4-a3c3-45fc-b2f1-913aa7f0fc22',
'name': 'reverse_and_double', 'tags': [],
'metadata': {}, 'data': {'input': '1234'}}
{'event': 'on_chain_start', 'name':
'reverse_word', 'run_id': '35a6470c-db65-
4fe1-8dff-4e3418601d2f', 'tags': [],
'metadata': {}, 'data': {'input': '1234'}}
{'event': 'on_chain_end', 'name':
'reverse_word', 'run_id': '35a6470c-db65-
4fe1-8dff-4e3418601d2f', 'tags': [],
'metadata': {}, 'data': {'input': '1234',
'output': '4321'}}
{'event': 'on_chain_stream', 'run_id':
'714d22d4-a3c3-45fc-b2f1-913aa7f0fc22',
'tags': [], 'metadata': {}, 'name':
'reverse_and_double', 'data': {'chunk':
'43214321'}}
{'event': 'on_chain_end', 'name':
'reverse_and_double', 'run_id': '714d22d4-
a3c3-45fc-b2f1-913aa7f0fc22', 'tags': [],
'metadata': {}, 'data': {'output':
'43214321'}}
```

And with the @chain decorator:

```
from langchain_core.runnables import chain


@chain
async def reverse_and_double(word: str):
    return await reverse_word.ainvoke(word) *
2


await reverse_and_double.ainvoke("1234")

async for event in
reverse_and_double.astream_events("1234",
version="v1"):
    print(event)
```

```
{'event': 'on_chain_start', 'run_id':
'17c89289-9c71-406d-90de-86f76b5e798b',
'name': 'reverse_and_double', 'tags': [],
'metadata': {}, 'data': {'input': '1234'}}
{'event': 'on_chain_start', 'name':
'reverse_word', 'run_id': 'b1105188-9196-
43c1-9603-4f2f58e51de4', 'tags': [],
'metadata': {}, 'data': {'input': '1234'}}
{'event': 'on_chain_end', 'name':
'reverse_word', 'run_id': 'b1105188-9196-
43c1-9603-4f2f58e51de4', 'tags': [],
'metadata': {}, 'data': {'input': '1234',
'output': '4321'}}
{'event': 'on_chain_stream', 'run_id':
'17c89289-9c71-406d-90de-86f76b5e798b',
'tags': [], 'metadata': {}, 'name':
'reverse_and_double', 'data': {'chunk':
'43214321'}}
{'event': 'on_chain_end', 'name':
'reverse_and_double', 'run_id': '17c89289-
9c71-406d-90de-86f76b5e798b', 'tags': [],
'metadata': {}, 'data': {'output':
'43214321'}}
```