Interface

To make it as easy as possible to create custom chains, we've implemented a "Runnable" protocol. The `Runnable` protocol is implemented for most components. This is a standard interface, which makes it easy to define custom chains as well as invoke them in a standard way. The standard interface includes:

- `stream`: stream back chunks of the response
- `invoke`: call the chain on an input
- `batch`: call the chain on a list of inputs

These also have corresponding async methods:

- `astream`: stream back chunks of the response async
- `ainvoke`: call the chain on an input async
- `abatch`: call the chain on a list of inputs async
- `astream_log`: stream back intermediate steps as they happen, in addition to the final response
- `astream_events`: **beta** stream events as they happen in the chain (introduced in `langchain-core` 0.1.14)

The **input type** and **output type** varies by component:

Component	Input Type	Output Type
Prompt	Dictionary	PromptValue
ChatModel	Single string, list of chat messages or a PromptValue	ChatMessage
LLM	Single string, list of chat messages or a PromptValue	String
OutputParser	The output of an LLM or ChatModel	Depends on the parser
Retriever	Single string	List of Documents
Tool	Single string or dictionary, depending on the tool	Depends on the tool

All runnables expose input and output **schemas** to inspect the inputs and outputs: - `input_schema`: an input Pydantic model auto-generated from the structure of the Runnable - `output_schema`: an output Pydantic model auto-generated from the structure of the Runnable

Let's take a look at these methods. To do so, we'll create a super simple PromptTemplate + ChatModel chain.

%pip install –upgrade –quiet langchain-core langchain-community langchain-openai

```
from langchain_core.prompts import
ChatPromptTemplate
from langchain_openai import ChatOpenAI

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

Input Schema

A description of the inputs accepted by a Runnable. This is a Pydantic model dynamically generated from the structure of any Runnable. You can call `.schema()` on it to obtain a JSONSchema representation.

```python
# The input schema of the chain is the input schema of its first part, the prompt.
chain.input_schema.schema()
```

```
{'title': 'PromptInput',
 'type': 'object',
 'properties': {'topic': {'title': 'Topic', 'type': 'string'}}}
```

```python
prompt.input_schema.schema()
```

```
{'title': 'PromptInput',
 'type': 'object',
 'properties': {'topic': {'title': 'Topic', 'type': 'string'}}}
```

```python
model.input_schema.schema()
```

```
{'title': 'ChatOpenAIInput',
 'anyOf': [{'type': 'string'},
  {'$ref': '#/definitions/StringPromptValue'},
  {'$ref': '#/definitions/ChatPromptValueConcrete'},
   {'type': 'array',
    'items': {'anyOf': [{'$ref':
 '#/definitions/AIMessage'},
      {'$ref': '#/definitions/HumanMessage'},
      {'$ref': '#/definitions/ChatMessage'},
      {'$ref': '#/definitions/SystemMessage'},
      {'$ref':
 '#/definitions/FunctionMessage'},
      {'$ref':
 '#/definitions/ToolMessage'}]}}],
  'definitions': {'StringPromptValue':
 {'title': 'StringPromptValue',
    'description': 'String prompt value.',
    'type': 'object',
    'properties': {'text': {'title': 'Text',
 'type': 'string'},
     'type': {'title': 'Type',
      'default': 'StringPromptValue',
      'enum': ['StringPromptValue'],
      'type': 'string'}},
    'required': ['text']},
   'AIMessage': {'title': 'AIMessage',
    'description': 'A Message from an AI.',
    'type': 'object',
    'properties': {'content': {'title':
 'Content',
      'anyOf': [{'type': 'string'},
       {'type': 'array',
        'items': {'anyOf': [{'type':
 'string'}, {'type': 'object'}]}}]},
     'additional_kwargs': {'title':
```

```
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'ai',
     'enum': ['ai'],
     'type': 'string'},
    'example': {'title': 'Example',
'default': False, 'type': 'boolean'}},
   'required': ['content']},
  'HumanMessage': {'title': 'HumanMessage',
   'description': 'A Message from a human.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'human',
     'enum': ['human'],
     'type': 'string'},
    'example': {'title': 'Example',
'default': False, 'type': 'boolean'}},
   'required': ['content']},
  'ChatMessage': {'title': 'ChatMessage',
   'description': 'A Message that can be
assigned an arbitrary speaker (i.e. role).',
   'type': 'object',
   'properties': {'content': {'title':
```

```
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'chat',
     'enum': ['chat'],
     'type': 'string'},
    'role': {'title': 'Role', 'type':
'string'}},
   'required': ['content', 'role']},
  'SystemMessage': {'title': 'SystemMessage',
   'description': 'A Message for priming AI
behavior, usually passed in as the first of a
sequence\nof input messages.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'system',
     'enum': ['system'],
     'type': 'string'}},
   'required': ['content']},
  'FunctionMessage': {'title':
'FunctionMessage',
   'description': 'A Message for passing the
result of executing a function back to a
model.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'function',
     'enum': ['function'],
     'type': 'string'},
    'name': {'title': 'Name', 'type':
'string'}},
   'required': ['content', 'name']},
  'ToolMessage': {'title': 'ToolMessage',
   'description': 'A Message for passing the
result of executing a tool back to a model.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
```

```
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'tool',
     'enum': ['tool'],
     'type': 'string'},
    'tool_call_id': {'title': 'Tool Call Id',
'type': 'string'}},
   'required': ['content', 'tool_call_id']},
  'ChatPromptValueConcrete': {'title':
'ChatPromptValueConcrete',
   'description': 'Chat prompt value which
explicitly lists out the message types it
accepts.\nFor use in external schemas.',
   'type': 'object',
   'properties': {'messages': {'title':
'Messages',
     'type': 'array',
     'items': {'anyOf': [{'$ref':
'#/definitions/AIMessage'},
       {'$ref':
'#/definitions/HumanMessage'},
       {'$ref': '#/definitions/ChatMessage'},
       {'$ref':
'#/definitions/SystemMessage'},
       {'$ref':
'#/definitions/FunctionMessage'},
       {'$ref':
'#/definitions/ToolMessage'}]}},
    'type': {'title': 'Type',
     'default': 'ChatPromptValueConcrete',
     'enum': ['ChatPromptValueConcrete'],
     'type': 'string'}},
   'required': ['messages']}}}
```

Output Schema

A description of the outputs produced by a Runnable. This is a Pydantic model dynamically generated from the structure of any Runnable. You can call `.schema()` on it to obtain a JSONSchema representation.

```
# The output schema of the chain is the
output schema of its last part, in this case
a ChatModel, which outputs a ChatMessage
chain.output_schema.schema()
```

```
{'title': 'ChatOpenAIOutput',
 'anyOf': [{'$ref':
'#/definitions/AIMessage'},
  {'$ref': '#/definitions/HumanMessage'},
  {'$ref': '#/definitions/ChatMessage'},
  {'$ref': '#/definitions/SystemMessage'},
  {'$ref': '#/definitions/FunctionMessage'},
  {'$ref': '#/definitions/ToolMessage'}],
 'definitions': {'AIMessage': {'title':
```

```
'AIMessage',
   'description': 'A Message from an AI.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'ai',
     'enum': ['ai'],
     'type': 'string'},
    'example': {'title': 'Example',
'default': False, 'type': 'boolean'}},
   'required': ['content']},
  'HumanMessage': {'title': 'HumanMessage',
   'description': 'A Message from a human.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'human',
     'enum': ['human'],
     'type': 'string'},
    'example': {'title': 'Example',
'default': False, 'type': 'boolean'}},
   'required': ['content']},
  'ChatMessage': {'title': 'ChatMessage',
   'description': 'A Message that can be
assigned an arbitrary speaker (i.e. role).',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'chat',
     'enum': ['chat'],
     'type': 'string'},
    'role': {'title': 'Role', 'type':
'string'}},
   'required': ['content', 'role']},
  'SystemMessage': {'title': 'SystemMessage',
   'description': 'A Message for priming AI
behavior, usually passed in as the first of a
sequence\nof input messages.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'system',
     'enum': ['system'],
     'type': 'string'}},
   'required': ['content']},
  'FunctionMessage': {'title':
'FunctionMessage',
   'description': 'A Message for passing the
result of executing a function back to a
model.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'function',
     'enum': ['function'],
     'type': 'string'},
    'name': {'title': 'Name', 'type':
'string'}},
   'required': ['content', 'name']},
  'ToolMessage': {'title': 'ToolMessage',
   'description': 'A Message for passing the
result of executing a tool back to a model.',
   'type': 'object',
   'properties': {'content': {'title':
'Content',
     'anyOf': [{'type': 'string'},
      {'type': 'array',
       'items': {'anyOf': [{'type':
'string'}, {'type': 'object'}]}}]},
    'additional_kwargs': {'title':
'Additional Kwargs', 'type': 'object'},
    'type': {'title': 'Type',
     'default': 'tool',
     'enum': ['tool'],
     'type': 'string'},
    'tool_call_id': {'title': 'Tool Call Id',
'type': 'string'}},
   'required': ['content', 'tool_call_id']}}}
```

Stream

```
for s in chain.stream({"topic": "bears"}):
    print(s.content, end="", flush=True)
```

```
Sure, here's a bear-themed joke for you:

Why don't bears wear shoes?

Because they already have bear feet!
```

Invoke

```
chain.invoke({"topic": "bears"})
```

```
AIMessage(content="Why don't bears wear
shoes? \n\nBecause they have bear feet!")
```

Batch

```
chain.batch([{"topic": "bears"}, {"topic":
"cats"}])
```

```
[AIMessage(content="Sure, here's a bear joke
for you:\n\nWhy don't bears wear shoes?
\n\nBecause they already have bear feet!"),
 AIMessage(content="Why don't cats play poker
in the wild?\n\nToo many cheetahs!")]
```

You can set the number of concurrent requests by using the `max_concurrency` parameter

```
chain.batch([{"topic": "bears"}, {"topic":
"cats"}], config={"max_concurrency": 5})
```

```
[AIMessage(content="Why don't bears wear
shoes?\n\nBecause they have bear feet!"),
 AIMessage(content="Why don't cats play poker
in the wild? Too many cheetahs!")]
```

Async Stream

```
async for s in chain.astream({"topic":
"bears"}):
    print(s.content, end="", flush=True)
```

```
Why don't bears wear shoes?

Because they have bear feet!
```

Async Invoke

```
await chain.ainvoke({"topic": "bears"})
```

```
AIMessage(content="Why don't bears ever wear
shoes?\n\nBecause they already have bear
feet!")
```

Async Batch

```
await chain.abatch([{"topic": "bears"}])
```

```
[AIMessage(content="Why don't bears wear
shoes?\n\nBecause they have bear feet!")]
```

Async Stream Events (beta)

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

Note: Introduced in langchain-core 0.2.0

For now, when using the astream_events API, for everything to work properly please:

- Use `async` throughout the code (including 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.

Event Reference

Here is a reference table that shows some events that might be emitted by the various Runnable objects. Definitions for some of the Runnable are included after the table.

⚠️ When streaming the inputs for the runnable will not be available until the input stream has been entirely consumed This means that the inputs will be available at for the corresponding `end` hook rather than `start` event.

event	name	
on_chat_model_start	[model name]	

event	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]	
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

event	name	
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]	

Here are declarations associated with the events shown above:

`format_docs`:

```
def format_docs(docs: List[Document]) -> str:
    '''Format the docs.'''
    return ", ".join([doc.page_content for
doc in docs])

format_docs = RunnableLambda(format_docs)
```

`some_tool`:

```
@tool
def some_tool(x: int, y: str) -> dict:
    '''Some_tool.'''
    return {"x": x, "y": y}
```

`prompt`:

```
template = ChatPromptTemplate.from_messages(
    [("system", "You are Cat Agent 007"),
("human", "{question}")]
).with_config({"run_name": "my_template",
"tags": ["my_template"]})
```

Let's define a new chain to make it more interesting to show off the `astream_events` interface (and later the `astream_log` interface).

```
from langchain_community.vectorstores import
FAISS
from langchain_core.output_parsers import
StrOutputParser
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"],
embedding=OpenAIEmbeddings()
)
retriever = vectorstore.as_retriever()

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

Now let’s use `astream_events` to get events from the retriever and the LLM.

```
async for event in
retrieval_chain.astream_events(
    "where did harrison work?", version="v1",
include_names=["Docs", "my_llm"]
):
    kind = event["event"]
    if kind == "on_chat_model_stream":
        print(event["data"]["chunk"].content,
end="|")
    elif kind in {"on_chat_model_start"}:
        print()
        print("Streaming LLM:")
    elif kind in {"on_chat_model_end"}:
        print()
        print("Done streaming LLM.")
    elif kind == "on_retriever_end":
        print("--")
        print("Retrieved the following
documents:")
        print(event["data"]["output"]
["documents"])
    elif kind == "on_tool_end":
        print(f"Ended tool: {event['name']}")
    else:
        pass
```

```
/home/eugene/src/langchain/libs/core/langchain_
LangChainBetaWarning: This API is in beta and m
```

```
  warn_beta(
```

```
--
Retrieved the following documents:
[Document(page_content='harrison worked at
kensho')]

Streaming LLM:
|H|arrison| worked| at| Kens|ho|.||
Done streaming LLM.
```

Async Stream Intermediate Steps

All runnables also have a method `.astream_log()` which is used to stream (as they happen) all or part of the intermediate steps of your chain/sequence.

This is useful to show progress to the user, to use intermediate results, or to debug your chain.

You can stream all steps (default) or include/exclude steps by name, tags or metadata.

This method yields JSONPatch ops that when applied in the same order as received build up the RunState.

```
class LogEntry(TypedDict):
    id: str
    """ID of the sub-run."""
    name: str
    """Name of the object being run."""
    type: str
    """Type of the object being run, eg.
prompt, chain, llm, etc."""
    tags: List[str]
    """List of tags for the run."""
    metadata: Dict[str, Any]
    """Key-value pairs of metadata for the
run."""
    start_time: str
    """ISO-8601 timestamp of when the run
started."""

    streamed_output_str: List[str]
    """List of LLM tokens streamed by this
run, if applicable."""
    final_output: Optional[Any]
    """Final output of this run.
    Only available after the run has finished
successfully."""
    end_time: Optional[str]
    """ISO-8601 timestamp of when the run
ended.
    Only available after the run has
finished."""
```

```
class RunState(TypedDict):
    id: str
    """ID of the run."""
    streamed_output: List[Any]
    """List of output chunks streamed by
Runnable.stream()"""
    final_output: Optional[Any]
    """Final output of the run, usually the
result of aggregating (`+`) streamed_output.
    Only available after the run has finished
successfully."""

    logs: Dict[str, LogEntry]
    """Map of run names to sub-runs. If
filters were supplied, this list will
    contain only the runs that matched the
filters."""
```

Streaming JSONPatch chunks

This is useful eg. to stream the `JSONPatch` in an HTTP server, and then apply the ops on the client to rebuild the run state there. See LangServe for tooling to make it easier to build a webserver from any Runnable.

```
async for chunk in
retrieval_chain.astream_log(
    "where did harrison work?",
include_names=["Docs"]
):
    print("-" * 40)
    print(chunk)
```

```
----------------------------------------
RunLogPatch({'op': 'replace',
  'path': '',
  'value': {'final_output': None,
            'id': '82e9b4b1-3dd6-4732-8db9-
90e79c4da48c',
            'logs': {},
            'name': 'RunnableSequence',
            'streamed_output': [],
            'type': 'chain'}})
----------------------------------------
RunLogPatch({'op': 'add',
  'path': '/logs/Docs',
  'value': {'end_time': None,
            'final_output': None,
            'id': '9206e94a-57bd-48ee-8c5e-
fdd1c52a6da2',
            'metadata': {},
            'name': 'Docs',
            'start_time': '2024-01-
19T22:33:55.902+00:00',
            'streamed_output': [],
            'streamed_output_str': [],
            'tags': ['map:key:context',
'FAISS', 'OpenAIEmbeddings'],
            'type': 'retriever'}})
----------------------------------------
RunLogPatch({'op': 'add',
  'path': '/logs/Docs/final_output',
  'value': {'documents':
[Document(page_content='harrison worked at
kensho')]}},
 {'op': 'add',
  'path': '/logs/Docs/end_time',
  'value': '2024-01-19T22:33:56.064+00:00'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': ''},
 {'op': 'replace', 'path': '/final_output',
'value': ''})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': 'H'},
 {'op': 'replace', 'path': '/final_output',
'value': 'H'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': 'arrison'},
 {'op': 'replace', 'path': '/final_output',
'value': 'Harrison'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': ' worked'},
 {'op': 'replace', 'path': '/final_output',
'value': 'Harrison worked'})
----------------------------------------
```

```
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': ' at'},
 {'op': 'replace', 'path': '/final_output',
'value': 'Harrison worked at'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': ' Kens'},
 {'op': 'replace', 'path': '/final_output',
'value': 'Harrison worked at Kens'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': 'ho'},
 {'op': 'replace',
  'path': '/final_output',
  'value': 'Harrison worked at Kensho'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': '.'},
 {'op': 'replace',
  'path': '/final_output',
  'value': 'Harrison worked at Kensho.'})
----------------------------------------
RunLogPatch({'op': 'add', 'path':
'/streamed_output/-', 'value': ''})
```

Streaming the incremental RunState

You can simply pass `diff=False` to get incremental values of `RunState`. You get more verbose output with more repetitive parts.

```
async for chunk in
retrieval_chain.astream_log(
    "where did harrison work?",
include_names=["Docs"], diff=False
):
    print("-" * 70)
    print(chunk)
```

```
---------------------------------------------
-------------------------
RunLog({'final_output': None,
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {},
 'name': 'RunnableSequence',
 'streamed_output': [],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': None,
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': None,
                   'final_output': None,
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
```

```
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': [],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': None,
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': [],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': '',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': [''],
 'type': 'chain'})
```

```
---------------------------------------------
-------------------------
RunLog({'final_output': 'H',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H'],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': 'Harrison',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-19T22:33:57.120+00:00',
                   'final_output': {'documents': [Document(page_content='harrison worked at kensho')]},
                   'id': '8de10b49-d6af-4cb7-a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags': ['map:key:context', 'FAISS', 'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H', 'arrison'],
 'type': 'chain'})
------------------------------------------------------------------------------
RunLog({'final_output': 'Harrison worked',
 'id': '431d1c55-7c50-48ac-b3a2-2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-19T22:33:57.120+00:00',
                   'final_output': {'documents': [Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H', 'arrison', '
worked'],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': 'Harrison worked at',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H', 'arrison', '
worked', ' at'],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': 'Harrison worked at
Kens',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H', 'arrison', '
worked', ' at', ' Kens'],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': 'Harrison worked at
Kensho',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
```

```
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H', 'arrison', '
worked', ' at', ' Kens', 'ho'],
 'type': 'chain'})
---------------------------------------------
-------------------------
RunLog({'final_output': 'Harrison worked at
Kensho.',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['', 'H', 'arrison', '
worked', ' at', ' Kens', 'ho', '.'],
 'type': 'chain'})
------------------------------------------------
--------------------------
RunLog({'final_output': 'Harrison worked at
Kensho.',
 'id': '431d1c55-7c50-48ac-b3a2-
2f5ba5f35172',
 'logs': {'Docs': {'end_time': '2024-01-
19T22:33:57.120+00:00',
                   'final_output':
{'documents':
[Document(page_content='harrison worked at
kensho')]},
                   'id': '8de10b49-d6af-4cb7-
a4e7-fbadf6efa01e',
                   'metadata': {},
                   'name': 'Docs',
                   'start_time': '2024-01-
19T22:33:56.939+00:00',
                   'streamed_output': [],
                   'streamed_output_str': [],
                   'tags':
['map:key:context', 'FAISS',
'OpenAIEmbeddings'],
                   'type': 'retriever'}},
 'name': 'RunnableSequence',
 'streamed_output': ['',
                     'H',
                     'arrison',
                     ' worked',
                     ' at',
                     ' Kens',
                     'ho',
                     '.',
                     ''],
 'type': 'chain'})
```

Parallelism

Let's take a look at how LangChain Expression Language supports parallel requests. For example, when using a `RunnableParallel` (often written as a dictionary) it executes each element in parallel.

```
from langchain_core.runnables import
RunnableParallel

chain1 =
ChatPromptTemplate.from_template("tell me a
joke about {topic}") | model
chain2 = (
    ChatPromptTemplate.from_template("write a
short (2 line) poem about {topic}")
    | model
)
```

```
combined = RunnableParallel(joke=chain1,
poem=chain2)
```

```
%%time
chain1.invoke({"topic": "bears"})
```

```
CPU times: user 18 ms, sys: 1.27 ms, total:
19.3 ms
Wall time: 692 ms
```

```
AIMessage(content="Why don't bears wear
shoes?\n\nBecause they already have bear
feet!")
```

```
%%time
chain2.invoke({"topic": "bears"})
```

```
CPU times: user 10.5 ms, sys: 166 µs, total:
10.7 ms
Wall time: 579 ms
```

```
AIMessage(content="In forest's
embrace,\nMajestic bears pace.")
```

```
%%time
combined.invoke({"topic": "bears"})
```

```
CPU times: user 32 ms, sys: 2.59 ms, total:
34.6 ms
Wall time: 816 ms
```

```
{'joke': AIMessage(content="Sure, here's a
bear-related joke for you:\n\nWhy did the
bear bring a ladder to the bar?\n\nBecause he
heard the drinks were on the house!"),
 'poem': AIMessage(content="In wilderness
they roam,\nMajestic strength, nature's
throne.")}
```

Parallelism on batches

Parallelism can be combined with other runnables. Let's try to use parallelism with batches.

```
%%time
chain1.batch([{"topic": "bears"}, {"topic":
"cats"}])
```

```
CPU times: user 17.3 ms, sys: 4.84 ms, total:
22.2 ms
Wall time: 628 ms
```

```
[AIMessage(content="Why don't bears wear
shoes?\n\nBecause they have bear feet!"),
 AIMessage(content="Why don't cats play poker
in the wild?\n\nToo many cheetahs!")]
```

```
%%time
chain2.batch([{"topic": "bears"}, {"topic":
"cats"}])
```

```
CPU times: user 15.8 ms, sys: 3.83 ms, total:
19.7 ms
Wall time: 718 ms
```

```
[AIMessage(content='In the wild, bears
roam,\nMajestic guardians of ancient home.'),
 AIMessage(content='Whiskers grace, eyes
gleam,\nCats dance through the moonbeam.')]
```

```
%%time
combined.batch([{"topic": "bears"}, {"topic":
```

```
"cats"}])
```

```
CPU times: user 44.8 ms, sys: 3.17 ms, total: 48 ms
Wall time: 721 ms
```

```
[{'joke': AIMessage(content="Sure, here's a bear joke for you:\n\nWhy don't bears wear shoes?\n\nBecause they have bear feet!"),
  'poem': AIMessage(content="Majestic bears roam,\nNature's strength, beauty shown.")},
 {'joke': AIMessage(content="Why don't cats play poker in the wild?\n\nToo many cheetahs!"),
  'poem': AIMessage(content="Whiskers dance, eyes aglow,\nCats embrace the night's gentle flow.")}]
```