Hide Search Matches

FakeChatModel

class
langchain_core.language_models.fake_chat_models.FakeChatModel # [source]

Bases: SimpleChatModel

Fake Chat Model wrapper for testing purposes.

> **Note**
>
> FakeChatModel implements the standard Runnable Interface. 🏃
>
> The Runnable Interface has additional methods that are available on runnables, such as with_types, with_retry, assign, bind, get_graph, and more.

param cache: BaseCache | bool | None = None #

Whether to cache the response.

- If true, will use the global cache.
- If false, will not use a cache
- If None, will use the global cache if it's set, otherwise no cache.
- If instance of BaseCache, will use the provided cache.

Caching is not currently supported for streaming methods of models.

param callback_manager: BaseCallbackManager | None = None #

> **Deprecated since version 0.1.7:** Use callbacks() instead.

Callback manager to add to the run trace.

param callbacks: Callbacks = None #

Callbacks to add to the run trace.

param custom_get_token_ids: Callable[[str], list[int]] | None = None #

Optional encoder to use for counting tokens.

param disable_streaming: bool | Literal['tool_calling'] = False #

Whether to disable streaming for this model.

If streaming is bypassed, then `stream()/astream()` will defer to `invoke()/ainvoke()`.

- If True, will always bypass streaming case.
- If "tool_calling", will bypass streaming case only when the model is called with a `tools` keyword argument.
- If False (default), will always use streaming case if available.

param metadata: dict[str, Any] | None = None #

Metadata to add to the run trace.

param rate_limiter: BaseRateLimiter | None = None #

An optional rate limiter to use for limiting the number of requests.

param tags: list[str] | None = None #

Tags to add to the run trace.

param verbose: bool [Optional] #

Whether to print out response text.

__call__(messages: list[BaseMessage], stop: list[str] | None = None, callbacks: list[BaseCallbackHandler] | BaseCallbackManager | None = None, **kwargs: Any) → BaseMessage #

> **Deprecated since version langchain-core==0.1.7:** Use `invoke()` instead.

Parameters:

- **messages** (list[BaseMessage])
- **stop** (list[str] | None)
- **callbacks** (list[BaseCallbackHandler] | BaseCallbackManager | None)
- **kwargs** (Any)

Return type:

BaseMessage

async abatch(inputs: list[Input], config: RunnableConfig | list[RunnableConfig] | None = None, *, return_exceptions: bool = False, **kwargs: Any | None) → list[Output] #

Default implementation runs ainvoke in parallel using asyncio.gather.

The default implementation of batch works well for IO bound runnables.

Subclasses should override this method if they can batch more efficiently; e.g., if the underlying Runnable uses an API which supports a batch mode.

Parameters:

- **inputs** (list[Input]) – A list of inputs to the Runnable.
- **config** (RunnableConfig | list[RunnableConfig] | None) – A config to use when invoking the Runnable. The config supports standard keys like 'tags', 'metadata' for tracing purposes, 'max_concurrency' for controlling how much work to do in parallel, and other keys. Please refer to the RunnableConfig for more details. Defaults to None.
- **return_exceptions** (bool) – Whether to return exceptions instead of raising them. Defaults to False.
- **kwargs** (Any | None) – Additional keyword arguments to pass to the Runnable.

Returns:

A list of outputs from the Runnable.

Return type:

list[Output]

async abatch_as_completed(inputs: Sequence[Input], config: RunnableConfig | Sequence[RunnableConfig] | None = None, *, return_exceptions: bool = False, **kwargs: Any | None) → AsyncIterator[tuple[int, Output | Exception]] #

Run ainvoke in parallel on a list of inputs, yielding results as they complete.

Parameters:

- **inputs** (Sequence[Input]) – A list of inputs to the Runnable.
- **config** (RunnableConfig | Sequence[RunnableConfig] | None) – A config to use when

invoking the Runnable. The config supports standard keys like 'tags', 'metadata' for tracing purposes, 'max_concurrency' for controlling how much work to do in parallel, and other keys. Please refer to the RunnableConfig for more details. Defaults to None. Defaults to None.

- **return_exceptions** (bool) – Whether to return exceptions instead of raising them. Defaults to False.
- **kwargs** (Any | None) – Additional keyword arguments to pass to the Runnable.

Yields:

A tuple of the index of the input and the output from the Runnable.

Return type:

AsyncIterator[tuple[int, Output | Exception]]

`async ainvoke(input: LanguageModelInput, config: RunnableConfig | None = None, *, stop: list[str] | None = None, **kwargs: Any) → BaseMessage #`

Default implementation of ainvoke, calls invoke from a thread.

The default implementation allows usage of async code even if the Runnable did not implement a native async version of invoke.

Subclasses should override this method if they can run asynchronously.

Parameters:

- **input** (LanguageModelInput)
- **config** (Optional[RunnableConfig])
- **stop** (Optional[list[str]])
- **kwargs** (Any)

Return type:

BaseMessage

`async astream(input: LanguageModelInput, config: RunnableConfig | None = None, *, stop: list[str] | None = None, **kwargs: Any) → AsyncIterator[BaseMessageChunk] #`

Default implementation of astream, which calls ainvoke. Subclasses should override this method if they support streaming output.

Parameters:

Parameters:

- **input** (LanguageModelInput) – The input to the Runnable.
- **config** (Optional[RunnableConfig]) – The config to use for the Runnable. Defaults to None.
- **kwargs** (Any) – Additional keyword arguments to pass to the Runnable.
- **stop** (Optional[list[str]])

Yields:

The output of the Runnable.

Return type:

AsyncIterator[BaseMessageChunk]

```
async astream_events(input: Any, config: RunnableConfig | None = None, *,
version: Literal['v1', 'v2'], include_names: Sequence[str] | None = None,
include_types: Sequence[str] | None = None, include_tags: Sequence[str] |
None = None, exclude_names: Sequence[str] | None = None, exclude_types:
Sequence[str] | None = None, exclude_tags: Sequence[str] | None = None,
**kwargs: Any) → AsyncIterator[StandardStreamEvent | CustomStreamEvent] #
```

Generate a stream of events.

Use to create an iterator over StreamEvents that provide real-time information about the progress of the Runnable, including StreamEvents from intermediate results.

A StreamEvent is a dictionary with the following schema:

- `event`**: str - Event names are of the**
  format: on_[runnable_type]_(start|stream|end).
- `name`: **str** - The name of the Runnable that generated the event.
- `run_id`**: str - randomly generated ID associated with the given execution of**
  the Runnable that emitted the event. A child Runnable that gets invoked as part of the execution of a parent Runnable is assigned its own unique ID.
- `parent_ids`**: List[str] - The IDs of the parent runnables that**
  generated the event. The root Runnable will have an empty list. The order of the parent IDs is from the root to the immediate parent. Only available for v2 version of the API. The v1 version of the API will return an empty list.
- `tags`**: Optional[List[str]] - The tags of the Runnable that generated**
  the event.

- `metadata`: **Optional[Dict[str, Any]] - The metadata of the Runnable** that generated the event.
- `data`: **Dict[str, Any]**

Below is a table that illustrates some events that might be emitted by various chains. Metadata fields have been omitted from the table for brevity. Chain definitions have been included after the table.

ATTENTION This reference table is for the V2 version of the schema.

event	name	chunk	input
on_chat_model_start	[model name]		{“message [[SystemM HumanMe
on_chat_model_stream	[model name]	AIMessageChunk(content=”hello”)	
on_chat_model_end	[model name]		{“message [[SystemM HumanMe
on_llm_start	[model name]		{‘input’: ‘he
on_llm_stream	[model name]	‘Hello’	
on_llm_end	[model name]		‘Hello hum
on_chain_start	format_docs		
on_chain_stream	format_docs	“hello world!, goodbye world!”	
on_chain_end	format_docs		[Documen
on_tool_start	some_tool		{“x”: 1, “y”:
on_tool_end	some_tool		
on_retriever_start	[retriever name]		{“query”: “
on_retriever_end	[retriever name]		{“query”: “
on_prompt_start	[template_name]		{“question

on_prompt_start	[template_name]		{ question “hello”}
on_prompt_end	[template_name]		{“question “hello”}

In addition to the standard events, users can also dispatch custom events (see example below).

Custom events will be only be surfaced with in the v2 version of the API!

A custom event has following format:

Attribute	Type	Description
name	str	A user defined name for the event.
data	Any	The data associated with the event. This can be anything, though we suggest making it JSON serializable.

Here are declarations associated with the standard 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"]})
```

Example:

```
from langchain_core.runnables import RunnableLambda

async def reverse(s: str) -> str:
    return s[::-1]

chain = RunnableLambda(func=reverse)

events = [
    event async for event in chain.astream_events("hello", version="v2")
]

# will produce the following events (run_id, and parent_ids
# has been omitted for brevity):
[
    {
        "data": {"input": "hello"},
        "event": "on_chain_start",
        "metadata": {},
        "name": "reverse",
        "tags": [],
    },
    {
        "data": {"chunk": "olleh"},
        "event": "on_chain_stream",
        "metadata": {},
        "name": "reverse",
        "tags": [],
    },
    {
        "data": {"output": "olleh"},
        "event": "on_chain_end",
        "metadata": {},
        "name": "reverse",
        "tags": [],
    },
]
```

Example: Dispatch Custom Event

```
from langchain_core.callbacks.manager import (
    adispatch_custom_event,
)
from langchain_core.runnables import RunnableLambda, RunnableConfig
import asyncio


async def slow_thing(some_input: str, config: RunnableConfig) -> str:
    """Do something that takes a long time."""
    await asyncio.sleep(1) # Placeholder for some slow operation
    await adispatch_custom_event(
        "progress_event",
        {"message": "Finished step 1 of 3"},
        config=config # Must be included for python < 3.10
```

```
        config=config # Must be included for python < 3.10
    )
    await asyncio.sleep(1) # Placeholder for some slow operation
    await adispatch_custom_event(
        "progress_event",
        {"message": "Finished step 2 of 3"},
        config=config # Must be included for python < 3.10
    )
    await asyncio.sleep(1) # Placeholder for some slow operation
    return "Done"

slow_thing = RunnableLambda(slow_thing)

async for event in slow_thing.astream_events("some_input", version="v2"):
    print(event)
```

Parameters:

- **input** (Any) – The input to the Runnable.
- **config** (RunnableConfig | None) – The config to use for the Runnable.
- **version** (Literal['v1', 'v2']) – The version of the schema to use either v2 or v1. Users should use v2. v1 is for backwards compatibility and will be deprecated in 0.4.0. No default will be assigned until the API is stabilized. custom events will only be surfaced in v2.
- **include_names** (Sequence[str] | None) – Only include events from runnables with matching names.
- **include_types** (Sequence[str] | None) – Only include events from runnables with matching types.
- **include_tags** (Sequence[str] | None) – Only include events from runnables with matching tags.
- **exclude_names** (Sequence[str] | None) – Exclude events from runnables with matching names.
- **exclude_types** (Sequence[str] | None) – Exclude events from runnables with matching types.
- **exclude_tags** (Sequence[str] | None) – Exclude events from runnables with matching tags.
- **kwargs** (Any) – Additional keyword arguments to pass to the Runnable. These will be passed to astream_log as this implementation of astream_events is built on top of astream_log.

Yields:

An async stream of StreamEvents.

Raises:

NotImplementedError – If the version is not v1 or v2.

Return type:

AsyncIterator[StandardStreamEvent | CustomStreamEvent]

```
batch(inputs: list[Input], config: RunnableConfig | list[RunnableConfig]
| None = None, *, return_exceptions: bool = False, **kwargs: Any | None)
→ list[Output] #
```

Default implementation runs invoke in parallel using a thread pool executor.

The default implementation of batch works well for IO bound runnables.

Subclasses should override this method if they can batch more efficiently; e.g., if the underlying Runnable uses an API which supports a batch mode.

Parameters:

- **inputs** (list[Input])
- **config** (RunnableConfig | list[RunnableConfig] | None)
- **return_exceptions** (bool)
- **kwargs** (Any | None)

Return type:

list[Output]

```
batch_as_completed(inputs: Sequence[Input], config: RunnableConfig |
Sequence[RunnableConfig] | None = None, *, return_exceptions: bool =
False, **kwargs: Any | None) → Iterator[tuple[int, Output | Exception]] #
```

Run invoke in parallel on a list of inputs, yielding results as they complete.

Parameters:

- **inputs** (Sequence[Input])
- **config** (RunnableConfig | Sequence[RunnableConfig] | None)
- **return_exceptions** (bool)
- **kwargs** (Any | None)

Return type:

Iterator[tuple[int, Output | Exception]]

bind(**kwargs: Any) → Runnable[Input, Output] #

Bind arguments to a Runnable, returning a new Runnable.

Useful when a Runnable in a chain requires an argument that is not in the output of the previous Runnable or included in the user input.

Parameters:

kwargs (Any) – The arguments to bind to the Runnable.

Returns:

A new Runnable with the arguments bound.

Return type:

Runnable[Input, Output]

Example:

```
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import StrOutputParser

llm = ChatOllama(model='llama2')

# Without bind.
chain = (
    llm
    | StrOutputParser()
)

chain.invoke("Repeat quoted words exactly: 'One two three four five.'")
# Output is 'One two three four five.'

# With bind.
chain = (
    llm.bind(stop=["three"])
    | StrOutputParser()
)

chain.invoke("Repeat quoted words exactly: 'One two three four five.'")
# Output is 'One two'
```

bind_tools(tools: Sequence[Dict[str, Any] | type | Callable | BaseTool], **kwargs: Any) → Runnable[LanguageModelInput, BaseMessage] #

Parameters:

- **tools** (Sequence[Union[Dict[str, Any], type, Callable, BaseTool]])

- **tools** (Sequence[Union[Dict[str, Any], type, Callable, BaseTool]])
- **kwargs** (Any)

Return type:

Runnable[LanguageModelInput, BaseMessage]

configurable_alternatives(which: ConfigurableField, *, default_key: str = 'default', prefix_keys: bool = False, **kwargs: Runnable[Input, Output] | Callable[[], Runnable[Input, Output]]) → RunnableSerializable #

Configure alternatives for Runnables that can be set at runtime.

Parameters:

- **which** (ConfigurableField) – The ConfigurableField instance that will be used to select the alternative.
- **default_key** (str) – The default key to use if no alternative is selected. Defaults to “default”.
- **prefix_keys** (bool) – Whether to prefix the keys with the ConfigurableField id. Defaults to False.
- ****kwargs** (Runnable[Input, Output] | Callable[[], Runnable[Input, Output]]) – A dictionary of keys to Runnable instances or callables that return Runnable instances.

Returns:

A new Runnable with the alternatives configured.

Return type:

RunnableSerializable

```
from langchain_anthropic import ChatAnthropic
from langchain_core.runnables.utils import ConfigurableField
from langchain_openai import ChatOpenAI

model = ChatAnthropic(
    model_name="claude-3-sonnet-20240229"
).configurable_alternatives(
    ConfigurableField(id="llm"),
    default_key="anthropic",
    openai=ChatOpenAI()
)

# uses the default model ChatAnthropic
print(model.invoke("which organization created you?").content)
```

```
# uses ChatOpenAI
print(
    model.with_config(
        configurable={"llm": "openai"}
    ).invoke("which organization created you?").content
)
```

configurable_fields(**kwargs: ConfigurableField | ConfigurableFieldSingleOption | ConfigurableFieldMultiOption) → RunnableSerializable #

Configure particular Runnable fields at runtime.

Parameters:

kwargs (ConfigurableField | ConfigurableFieldSingleOption | ConfigurableFieldMultiOption) – A dictionary of ConfigurableField instances to configure.

Returns:

A new Runnable with the fields configured.

Return type:

RunnableSerializable

```
from langchain_core.runnables import ConfigurableField
from langchain_openai import ChatOpenAI

model = ChatOpenAI(max_tokens=20).configurable_fields(
    max_tokens=ConfigurableField(
        id="output_token_number",
        name="Max tokens in the output",
        description="The maximum number of tokens in the output",
    )
)

# max_tokens = 20
print(
    "max_tokens_20: ",
    model.invoke("tell me something about chess").content
)

# max_tokens = 200
print("max_tokens_200: ", model.with_config(
    configurable={"output_token_number": 200}
    ).invoke("tell me something about chess").content
)
```

`get_num_tokens(text: str) → int` #

Get the number of tokens present in the text.

Useful for checking if an input fits in a model's context window.

Parameters:

text (str) – The string input to tokenize.

Returns:

The integer number of tokens in the text.

Return type:

int

`get_num_tokens_from_messages(messages: list[BaseMessage], tools: Sequence | None = None) → int` #

Get the number of tokens in the messages.

Useful for checking if an input fits in a model's context window.

Note: the base implementation of get_num_tokens_from_messages ignores tool schemas.

Parameters:

- **messages** (list[BaseMessage]) – The message inputs to tokenize.
- **tools** (Sequence | None) – If provided, sequence of dict, BaseModel, function, or BaseTools to be converted to tool schemas.

Returns:

The sum of the number of tokens across the messages.

Return type:

int

`get_token_ids(text: str) → list[int]` #

Return the ordered ids of the tokens in a text.

Parameters:

text (str) – The string input to tokenize.

Returns:

A list of ids corresponding to the tokens in the text, in order they occur

in the text.

Return type:

list[int]

`invoke(input: LanguageModelInput, config: RunnableConfig | None = None, *, stop: list[str] | None = None, **kwargs: Any) → BaseMessage #`

Transform a single input into an output. Override to implement.

Parameters:

- **input** (LanguageModelInput) – The input to the Runnable.
- **config** (Optional[RunnableConfig]) – A config to use when invoking the Runnable. The config supports standard keys like 'tags', 'metadata' for tracing purposes, 'max_concurrency' for controlling how much work to do in parallel, and other keys. Please refer to the RunnableConfig for more details.
- **stop** (Optional[list[str]])
- **kwargs** (Any)

Returns:

The output of the Runnable.

Return type:

BaseMessage

`stream(input: LanguageModelInput, config: RunnableConfig | None = None, *, stop: list[str] | None = None, **kwargs: Any) → Iterator[BaseMessageChunk] #`

Default implementation of stream, which calls invoke. Subclasses should override this method if they support streaming output.

Parameters:

- **input** (LanguageModelInput) – The input to the Runnable.
- **config** (Optional[RunnableConfig]) – The config to use for the Runnable. Defaults to

None.

- **kwargs** (Any) – Additional keyword arguments to pass to the Runnable.
- **stop** (Optional[list[str]])

Yields:

The output of the Runnable.

Return type:

Iterator[BaseMessageChunk]

`with_alisteners(*, on_start: AsyncListener | None = None, on_end: AsyncListener | None = None, on_error: AsyncListener | None = None) → Runnable[Input, Output]` #

Bind asynchronous lifecycle listeners to a Runnable, returning a new Runnable.

on_start: Asynchronously called before the Runnable starts running. on_end: Asynchronously called after the Runnable finishes running. on_error: Asynchronously called if the Runnable throws an error.

The Run object contains information about the run, including its id, type, input, output, error, start_time, end_time, and any tags or metadata added to the run.

Parameters:

- **on_start** (Optional[AsyncListener]) – Asynchronously called before the Runnable starts running. Defaults to None.
- **on_end** (Optional[AsyncListener]) – Asynchronously called after the Runnable finishes running. Defaults to None.
- **on_error** (Optional[AsyncListener]) – Asynchronously called if the Runnable throws an error. Defaults to None.

Returns:

A new Runnable with the listeners bound.

Return type:

Runnable[Input, Output]

Example:

```
from langchain_core.runnables import RunnableLambda
```

```
from langchain_core.runnables import RunnableLambda
import time

async def test_runnable(time_to_sleep : int):
    print(f"Runnable[{time_to_sleep}s]: starts at {format_t(time.time())}")
    await asyncio.sleep(time_to_sleep)
    print(f"Runnable[{time_to_sleep}s]: ends at {format_t(time.time())}")

async def fn_start(run_obj : Runnable):
    print(f"on start callback starts at {format_t(time.time())}
    await asyncio.sleep(3)
    print(f"on start callback ends at {format_t(time.time())}")

async def fn_end(run_obj : Runnable):
    print(f"on end callback starts at {format_t(time.time())}
    await asyncio.sleep(2)
    print(f"on end callback ends at {format_t(time.time())}")

runnable = RunnableLambda(test_runnable).with_alisteners(
    on_start=fn_start,
    on_end=fn_end
)
async def concurrent_runs():
    await asyncio.gather(runnable.ainvoke(2), runnable.ainvoke(3))

asyncio.run(concurrent_runs())
Result:
on start callback starts at 2024-05-16T14:20:29.637053+00:00
on start callback starts at 2024-05-16T14:20:29.637150+00:00
on start callback ends at 2024-05-16T14:20:32.638305+00:00
on start callback ends at 2024-05-16T14:20:32.638383+00:00
Runnable[3s]: starts at 2024-05-16T14:20:32.638849+00:00
Runnable[5s]: starts at 2024-05-16T14:20:32.638999+00:00
Runnable[3s]: ends at 2024-05-16T14:20:35.640016+00:00
on end callback starts at 2024-05-16T14:20:35.640534+00:00
Runnable[5s]: ends at 2024-05-16T14:20:37.640169+00:00
on end callback starts at 2024-05-16T14:20:37.640574+00:00
on end callback ends at 2024-05-16T14:20:37.640654+00:00
on end callback ends at 2024-05-16T14:20:39.641751+00:00
```

with_config(**config: RunnableConfig | None = None**, ****kwargs: Any**) → **Runnable[Input, Output]** #

Bind config to a Runnable, returning a new Runnable.

Parameters:

- **config** (RunnableConfig | None) – The config to bind to the Runnable.
- **kwargs** (Any) – Additional keyword arguments to pass to the Runnable.

Returns:

A new Runnable with the config bound.

Return type:

Runnable[Input, Output]

with_fallbacks(fallbacks: Sequence[Runnable[Input, Output]], *, exceptions_to_handle: tuple[type[BaseException], ...] = (<class 'Exception'>,), exception_key: Optional[str] = None) → RunnableWithFallbacksT[Input, Output] #

Add fallbacks to a Runnable, returning a new Runnable.

The new Runnable will try the original Runnable, and then each fallback in order, upon failures.

Parameters:

- **fallbacks** (Sequence[Runnable[Input, Output]]) – A sequence of runnables to try if the original Runnable fails.
- **exceptions_to_handle** (tuple[type[BaseException], ...]) – A tuple of exception types to handle. Defaults to (Exception,).
- **exception_key** (Optional[str]) – If string is specified then handled exceptions will be passed to fallbacks as part of the input under the specified key. If None, exceptions will not be passed to fallbacks. If used, the base Runnable and its fallbacks must accept a dictionary as input. Defaults to None.

Returns:

A new Runnable that will try the original Runnable, and then each fallback in order, upon failures.

Return type:

RunnableWithFallbacksT[Input, Output]

Example

```
from typing import Iterator

from langchain_core.runnables import RunnableGenerator


def _generate_immediate_error(input: Iterator) -> Iterator[str]:
    raise ValueError()
    yield ""
```

```
def _generate(input: Iterator) -> Iterator[str]:
    yield from "foo bar"


runnable = RunnableGenerator(_generate_immediate_error).with_fallbacks(
    [RunnableGenerator(_generate)]
    )
print(''.join(runnable.stream({}))) #foo bar
```

Parameters:

- **fallbacks** (Sequence[Runnable[Input, Output]]) – A sequence of runnables to try if the original Runnable fails.
- **exceptions_to_handle** (tuple[type[BaseException], ...]) – A tuple of exception types to handle.
- **exception_key** (Optional[str]) – If string is specified then handled exceptions will be passed to fallbacks as part of the input under the specified key. If None, exceptions will not be passed to fallbacks. If used, the base Runnable and its fallbacks must accept a dictionary as input.

Returns:

A new Runnable that will try the original Runnable, and then each fallback in order, upon failures.

Return type:

RunnableWithFallbacksT[Input, Output]

with_listeners(*, on_start: Callable[[Run], None] | Callable[[Run, RunnableConfig], None] | None = None, on_end: Callable[[Run], None] | Callable[[Run, RunnableConfig], None] | None = None, on_error: Callable[[Run], None] | Callable[[Run, RunnableConfig], None] | None = None) → Runnable[Input, Output] #

Bind lifecycle listeners to a Runnable, returning a new Runnable.

on_start: Called before the Runnable starts running, with the Run object. on_end: Called after the Runnable finishes running, with the Run object. on_error: Called if the Runnable throws an error, with the Run object.

The Run object contains information about the run, including its id, type, input, output, error, start_time, end_time, and any tags or metadata added to the run.

Parameters:

- **on_start** (Optional[Union[Callable[[Run], None], Callable[[Run, RunnableConfig],

None]]]) – Called before the Runnable starts running. Defaults to None.

- **on_end** (Optional[Union[Callable[[Run], None], Callable[[Run, RunnableConfig], None]]]) – Called after the Runnable finishes running. Defaults to None.
- **on_error** (Optional[Union[Callable[[Run], None], Callable[[Run, RunnableConfig], None]]]) – Called if the Runnable throws an error. Defaults to None.

Returns:

A new Runnable with the listeners bound.

Return type:

Runnable[Input, Output]

Example:

```
from langchain_core.runnables import RunnableLambda
from langchain_core.tracers.schemas import Run

import time

def test_runnable(time_to_sleep : int):
    time.sleep(time_to_sleep)

def fn_start(run_obj: Run):
    print("start_time:", run_obj.start_time)

def fn_end(run_obj: Run):
    print("end_time:", run_obj.end_time)

chain = RunnableLambda(test_runnable).with_listeners(
    on_start=fn_start,
    on_end=fn_end
)
chain.invoke(2)
```

with_retry(*, retry_if_exception_type: tuple[type[BaseException], ...] = (<class 'Exception'>,), wait_exponential_jitter: bool = True, stop_after_attempt: int = 3) → Runnable[Input, Output] #

Create a new Runnable that retries the original Runnable on exceptions.

Parameters:

- **retry_if_exception_type** (tuple[type[BaseException], ...]) – A tuple of exception types to retry on. Defaults to (Exception,).
- **wait_exponential_jitter** (bool) – Whether to add jitter to the wait time between retries

- **wait_exponential_jitter** (bool) – Whether to add jitter to the wait time between retries. Defaults to True.
- **stop_after_attempt** (int) – The maximum number of attempts to make before giving up. Defaults to 3.

Returns:

A new Runnable that retries the original Runnable on exceptions.

Return type:

Runnable[Input, Output]

Example:

```
from langchain_core.runnables import RunnableLambda

count = 0


def _lambda(x: int) -> None:
    global count
    count = count + 1
    if x == 1:
        raise ValueError("x is 1")
    else:
         pass


runnable = RunnableLambda(_lambda)
try:
    runnable.with_retry(
        stop_after_attempt=2,
        retry_if_exception_type=(ValueError,),
    ).invoke(1)
except ValueError:
    pass

assert (count == 2)
```

Parameters:

- **retry_if_exception_type** (tuple[type[BaseException], ...]) – A tuple of exception types to retry on
- **wait_exponential_jitter** (bool) – Whether to add jitter to the wait time between retries
- **stop_after_attempt** (int) – The maximum number of attempts to make before giving up

Returns:

A new Runnable that retries the original Runnable on exceptions.

Return type:

Runnable[Input, Output]

`with_structured_output(schema: Dict | type, *, include_raw: bool = False, **kwargs: Any) → Runnable[LanguageModelInput, Dict | BaseModel]` #

Model wrapper that returns outputs formatted to match the given schema.

Parameters:

- **schema** (Union[Dict, type]) –

  The output schema. Can be passed in as:

  - an OpenAI function/tool schema,
  - a JSON Schema,
  - a TypedDict class (support added in 0.2.26),
  - or a Pydantic class.

  If `schema` is a Pydantic class then the model output will be a Pydantic instance of that class, and the model-generated fields will be validated by the Pydantic class. Otherwise the model output will be a dict and will not be validated. See `langchain_core.utils.function_calling.convert_to_openai_tool()` for more on how to properly specify types and descriptions of schema fields when specifying a Pydantic or TypedDict class.

  > **Changed in version 0.2.26:** Added support for TypedDict class.

- **include_raw** (bool) – If False then only the parsed structured output is returned. If an error occurs during model output parsing it will be raised. If True then both the raw model response (a BaseMessage) and the parsed model response will be returned. If an error occurs during output parsing it will be caught and returned as well. The final output is always a dict with keys “raw”, “parsed”, and “parsing_error”.
- **kwargs** (Any)

Returns:

A Runnable that takes same inputs as a `langchain_core.language_models.chat.BaseChatModel`.

If `include_raw` is False and `schema` is a Pydantic class, Runnable outputs an instance of `schema` (i.e., a Pydantic object).

Otherwise if `include_raw` is False then Runnable outputs a dict.

Otherwise, if `include_raw` is False then Runnable outputs a dict.

If `include_raw` is True, then Runnable outputs a dict with keys:

- `"raw"`: BaseMessage
- `"parsed"`: None if there was a parsing error, otherwise the type depends on the `schema` as described above.
- `"parsing_error"`: Optional[BaseException]

Return type:

Runnable[LanguageModelInput, Union[Dict, BaseModel]]

Example: Pydantic schema (include_raw=False):

```
from pydantic import BaseModel

class AnswerWithJustification(BaseModel):
    '''An answer to the user question along with justification for the answer.
    answer: str
    justification: str

llm = ChatModel(model="model-name", temperature=0)
structured_llm = llm.with_structured_output(AnswerWithJustification)

structured_llm.invoke("What weighs more a pound of bricks or a pound of feathe

# -> AnswerWithJustification(
#     answer='They weigh the same',
#     justification='Both a pound of bricks and a pound of feathers weigh one
# )
```

Example: Pydantic schema (include_raw=True):

```
from pydantic import BaseModel

class AnswerWithJustification(BaseModel):
    '''An answer to the user question along with justification for the answer.
    answer: str
    justification: str

llm = ChatModel(model="model-name", temperature=0)
structured_llm = llm.with_structured_output(AnswerWithJustification, include_r

structured_llm.invoke("What weighs more a pound of bricks or a pound of feathe
# -> {
#     'raw': AIMessage(content='', additional_kwargs={'tool_calls': [{'id': 'c
#     'parsed': AnswerWithJustification(answer='They weigh the same.', justifi
#     'parsing_error': None
# }
```

Example: Dict schema (include_raw=False):

```
from pydantic import BaseModel
from langchain_core.utils.function_calling import convert_to_openai_tool

class AnswerWithJustification(BaseModel):
    '''An answer to the user question along with justification for the answer.
    answer: str
    justification: str

dict_schema = convert_to_openai_tool(AnswerWithJustification)
llm = ChatModel(model="model-name", temperature=0)
structured_llm = llm.with_structured_output(dict_schema)

structured_llm.invoke("What weighs more a pound of bricks or a pound of feathe
# -> {
#     'answer': 'They weigh the same',
#     'justification': 'Both a pound of bricks and a pound of feathers weigh o
# }
```

with_types(*, input_type: type[Input] | None = None, output_type: type[Output] | None = None) → Runnable[Input, Output] #

Bind input and output types to a Runnable, returning a new Runnable.

Parameters:

- **input_type** (type[Input] | None) – The input type to bind to the Runnable. Defaults to None.
- **output_type** (type[Output] | None) – The output type to bind to the Runnable. Defaults to None.

Returns:

A new Runnable with the types bound.

Return type:

Runnable[Input, Output]