ChatOllama

class langchain_community.chat_models.ollama.**ChatOllama** [source

Bases: `BaseChatModel`, `_OllamaCommon`

Ollama locally runs large language models.

To use, follow the instructions at https://ollama.ai/.

Example

```
from langchain_community.chat_models import ChatOllama
ollama = ChatOllama(model="llama2")
```

Note

ChatOllama 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 **auth**: *Callable | Tuple | None = None*

Additional auth tuple or callable to enable Basic/Digest/Custom HTTP Auth. Expects the same format, type and values as requests.request auth parameter.

param **base_url**: *str = 'http://localhost:11434'*

Base url the model is hosted under.

param **cache**: *BaseCache | bool | None = None*

Whether to cache the response.

- If true, will use the global ca
- 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 **format**: *str* | *None* = *None*

Specify the format of the output (e.g., json)

param **headers**: *dict* | *None* = *None*

Additional headers to pass to endpoint (e.g. Authorization, Referer). This is useful when Ollama is hosted on cloud services that require tokens for authentication.

param **keep_alive**: *int | str | None = None*

How long the model will stay loaded into memory.

The parameter (Default: 5 minutes) can be set to: 1. a duration string in Golang (such as “10m” or “24h”); 2. a number in seconds (such as 3600); 3. any negative number which will keep the model loaded in memory (e.g. -1 or “-1m”); 4. 0 which will unload the model immediately after generating a response;

See the [Ollama documents](ollama/ollama)

param **metadata**: *Dict[str, Any] | None = None*

Metadata to add to the run trace.

param **mirostat**: *int | None = None*

Enable Mirostat sampling for controlling perplexity. (default: 0, 0 = disabled, 1 = Mirostat, 2 = Mirostat 2.0)

param **mirostat_eta**: *float | None = None*

Influences how quickly the algorithm responds to feedback from the generated text. A lower learning rate will result in slower adjustments, while a higher learning rate will make the algorithm more responsive. (Default: 0.1)

param **mirostat_tau**: *float | None = None*

Controls the balance between coherence and diversity of the output. A lower value will result in more focused and coherent text. (Default: 5.0)

param **model**: *str = 'llama2'*

Model name to use.

param **num_ctx**: *int | None = None*

Sets the size of the context window used to generate the next token. (Default: 2048)

param **num_gpu**: *int | None = None*

The number of GPUs to use. On macOS it defaults to 1 to enable metal support, 0 to disable.

param **num_predict**: *int | None = None*

Maximum number of tokens to predict when generating text. (Default: 128, -1 = infinite generation, -2 = fill context)

param **num_thread**: *int | None = None*

Sets the number of threads to use during computation. By default, Ollama will detect this for optimal performance. It is recommended to set this value to the number of physical CPU cores your system has (as opposed to the logical number of cores).

param **rate_limiter**: *BaseRateLimiter | None = None*

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

param **raw**: *bool | None = None*

raw or not.

param **repeat_last_n**: *int | None = None*

Sets how far back for the model to look back to prevent repetition. (Default: 64, 0 = disabled, -1 = num_ctx)

param **repeat_penalty**: *float | None = None*

Sets how strongly to penalize repetitions. A higher value (e.g., 1.5) will pena
more strongly, while a lower value (e.g., 0.9) will be more lenient. (Default: 1.1)

param **stop**: *List[str] | None = None*

Sets the stop tokens to use.

param **system**: *str | None = None*

system prompt (overrides what is defined in the Modelfile)

param **tags**: *List[str] | None = None*

Tags to add to the run trace.

param **temperature**: *float | None = None*

The temperature of the model. Increasing the temperature will make the model answer more creatively. (Default: 0.8)

param **template**: *str | None = None*

full prompt or prompt template (overrides what is defined in the Modelfile)

param **tfs_z**: *float | None = None*

Tail free sampling is used to reduce the impact of less probable tokens from the output. A higher value (e.g., 2.0) will reduce the impact more, while a value of 1.0 disables this setting (default: 1)

param **timeout**: *int | None = None*

Timeout for the request stream

param **top_k**: *int | None = None*

Reduces the probability of generating nonsense. A higher value (e.g. 100) will give more diverse answers, while a lower value (e.g. 10) will be more conservative. (De

param **top_p**: *float | None = None*

Works together with top-k. A higher value (e.g., 0.95) will lead to more diverse text, while a lower value (e.g., 0.5) will generate more focused and conservative text. (Default: 0.9)

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.

latest

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' fo 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' fo 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 **agenerate**(***messages: List[List[BaseMessage]], stop: List[str] | None = None, callbacks: List[BaseCallbackHandler] | BaseCallbackManager | None = None, *, tags: List[str] | None = None, metadata: Dict[str, Any] | None = None, run_name: str | None = None, run_id: UUID | None = None, **kwargs: Any***) → LLMResult

Asynchronously pass a sequence of prompts to a model and return generations.

This method should make use of batched calls for models that expose a batched API.

Use this method when you want to:

1. take advantage of batched calls,
2. need more output from the model than just the top generated value,
3. **are building chains that are agnostic to the underlying language model** type (e.g., pure text completion models vs chat models).

Parameters::

- **messages** (*List[List[BaseMessage]]*) – List of list of messages.
- **stop** (*List[str] | None*) – Stop words to use when generating. Model output is cut of at the first occurrence of any of these substrings.
- **callbacks** (*List[BaseCallbackHandler] | BaseCallbackManager | None*) – Callbacks to pass through. Used for executing additional functionality, such as logging or streaming, throughout generation.
- ****kwargs** (*Any*) – Arbitrary additional keyword arguments. These are usually passe to the model provider API call.
- **tags** (*List[str] | None*) –
- **metadata** (*Dict[str, Any] | None*) –
- **run_name** (*str | None*) –
- **run_id** (*UUID | None*) –
- ****kwargs** –

Returns::

An LLMResult, which contains a list of candidate Generations for each input prompt and additional model provider-specific output.

Return type::

LLMResult

async **agenerate_prompt**(*prompts: List[PromptValue], stop: List[str] | None* = *None, callbacks: List[BaseCallbackHandler] | BaseCallbackManager | None* = *None, **kwargs: Any*) → LLMResult

Asynchronously pass a sequence of prompts and return model generations.

This method should make use of batched calls for models that expose a batched API.

Use this method when you want to:

1. take advantage of batched calls,
2. need more output from the model than just the top generated value,
3. **are building chains that are agnostic to the underlying language model** type (e.g., pure text completion models vs chat models).

Parameters::

- **prompts** (*List[PromptValue]*) – List of PromptValues. A PromptValue is an object that can be converted to match the format of any language model (string for pure text generation models and BaseMessages for chat models).
- **stop** (*List[str]* | *None*) – Stop words to use when generating. Model output is cut of at the first occurrence of any of these substrings.
- **callbacks** (*List[BaseCallbackHandler]* | *BaseCallbackManager* | *None*) – Callbacks to pass through. Used for executing additional functionality, such as logging or streaming, throughout generation.
- ****kwargs** (*Any*) – Arbitrary additional keyword arguments. These are usually passe to the model provider API call.

Returns::

An LLMResult, which contains a list of candidate Generations for each input prompt and additional model provider-specific output.

Return type::

LLMResult

async **ainvoke**(*input: LanguageModelInput, config: RunnableConfig | None* = *None, *, stop: List[str] | None* = *None, **kwargs: Any*) → BaseMe latest

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 **apredict**(***text: str***, *******, ***stop: Sequence[str] | None*** = *None*, ****kwargs: Any***) → **str**

> **Deprecated since version langchain-core==0.1.7:** Use `ainvoke` instead.

Parameters::

- **text** (*str*) –
- **stop** (*Sequence[str] | None*) –
- **kwargs** (*Any*) –

Return type::

str

async **apredict_messages**(***messages: List[BaseMessage]***, *******, ***stop: Sequence[str] | None*** = *None*, ****kwargs: Any***) → **BaseMessage**

> **Deprecated since version langchain-core==0.1.7:** Use `ainvoke` instead.

latest

Parameters::

- **messages** (*List[BaseMessage]*) –

- **stop** (*Sequence[str] | None*) –
- **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 a**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 t 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]

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]**

Beta

This API is in beta and may change in the future.

latest

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 **invoke**d as part c
  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 evens 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]		{"messa

event	name	chunk	input
on_chat_model_stream	[model name]	AIMessageChunk(content="hello")	
on_chat_model_end	[model name]		{"messa [[Systen Human
on_llm_start	[model name]		{'input':
on_llm_stream	[model name]	'Hello'	
on_llm_end	[model name]		'Hello h
on_chain_start	format_docs		
on_chain_stream	format_docs	"hello world!, goodbye world!"	
on_chain_end	format_docs		[Docum
on_tool_start	some_tool		{"x": 1, "
on_tool_end	some_tool		
on_retriever_start	[retriever name]		{"query'
on_retriever_end	[retriever name]		{"query'
on_prompt_start	[template_name]		{"questi "hello"}
on_prompt_end	[template_name]		{"questi "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:

```python
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

```python
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
    )
    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 matchin 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*]

latest

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

Parameters::

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

Return type::

Runnable[LanguageModelInput, BaseMessage]

call_as_llm(*message: str, stop: List[str] | None = None, **kwargs: Any*) → str

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

Parameters::

- **message** (*str*) –
- **stop** (*List[str]* | *None*) –
- **kwargs** (*Any*) –

Return type::

str

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

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[*Input*, *Output*]

```
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[Input, Output]

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[*Input*, *Output*]

```
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
)
```

generate(*messages: List[List[BaseMessage]], stop: List[str] | None = None, callbacks: List[BaseCallbackHandler] | BaseCallbackManager | No... tags: List[str] | None = None, metadata: Dict[str, Any] | None =*

run_name: str | None *= None*, ***run_id: UUID | None*** *= None*, *****kwargs: Any*****)** → **LLMResult**

Pass a sequence of prompts to the model and return model generations.

This method should make use of batched calls for models that expose a batched API.

Use this method when you want to:

1. take advantage of batched calls,
2. need more output from the model than just the top generated value,
3. **are building chains that are agnostic to the underlying language model** type (e.g., pure text completion models vs chat models).

Parameters::

- **messages** (*List[List[BaseMessage]]*) – List of list of messages.
- **stop** (*List[str]* | *None*) – Stop words to use when generating. Model output is cut of at the first occurrence of any of these substrings.
- **callbacks** (*List[BaseCallbackHandler]* | *BaseCallbackManager* | *None*) – Callbacks to pass through. Used for executing additional functionality, such as logging or streaming, throughout generation.
- ****kwargs** (*Any*) – Arbitrary additional keyword arguments. These are usually passe to the model provider API call.
- **tags** (*List[str]* | *None*) –
- **metadata** (*Dict[str, Any]* | *None*) –
- **run_name** (*str* | *None*) –
- **run_id** (*UUID* | *None*) –
- ****kwargs** –

Returns::

An LLMResult, which contains a list of candidate Generations for each input prompt and additional model provider-specific output.

Return type::

LLMResult

generate_prompt(*prompts: List[PromptValue], stop: List[str] | None = None, callbacks: List[BaseCallbackHandler] | BaseCallbackManager | None = None, **kwargs: Any*) → LLMResult

Pass a sequence of prompts to the model and return model generations.

This method should make use of batched calls for models that expose a batched API.

Use this method when you want to:

1. take advantage of batched calls,
2. need more output from the model than just the top generated value,
3. **are building chains that are agnostic to the underlying language model** type (e.g., pure text completion models vs chat models).

Parameters::

- **prompts** (*List[PromptValue]*) – List of PromptValues. A PromptValue is an object that can be converted to match the format of any language model (string for pure text generation models and BaseMessages for chat models).
- **stop** (*List[str]* | *None*) – Stop words to use when generating. Model output is cut of at the first occurrence of any of these substrings.
- **callbacks** (*List[BaseCallbackHandler]* | *BaseCallbackManager* | *None*) – Callbacks to pass through. Used for executing additional functionality, such as logging or streaming, throughout generation.
- ****kwargs** (*Any*) – Arbitrary additional keyword arguments. These are usually passe to the model provider API call.

Returns::

An LLMResult, which contains a list of candidate Generations for each input prompt and additional model provider-specific output.

Return type::

LLMResult

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]) → int`

Get the number of tokens in the messages.

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

Parameters::

messages (*List[BaseMessage]*) – The message inputs to tokenize.

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]

latest

`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

predict(*text: str*, ***, *stop: Sequence[str] | None = None*, ***kwargs: Any*) → str

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

Parameters::

- **text** (*str*) –
- **stop** (*Sequence[str] | None*) –
- **kwargs** (*Any*) –

Return type::

str

predict_messages(*messages: List[BaseMessage]*, ***, *stop: Sequence[str] | None = None*, ***kwargs: Any*) → BaseMessage

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

Parameters::

- **messages** (*List[BaseMessage]*) –
- **stop** (*Sequence[str] | None*) –
- **kwargs** (*Any*) –

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 t 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]

to_json() → SerializedConstructor | SerializedNotImplemented

Serialize the Runnable to JSON.

Returns::

A JSON-serializable representation of the Runnable.

Return type::

SerializedConstructor | *SerializedNotImplemented*

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.
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 langchain_core.pydantic_v1 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
feathers")

# -> AnswerWithJustification(
#     answer='They weigh the same',
#     justification='Both a pound of bricks and a pound of feathers weigh
one pound. The weight is the same, but the volume or density of the
objects may differ.'
# )
```

Example: Pydantic schema (include_raw=True):