How to return structured data from a model

> **PREREQUISITES**
>
> This guide assumes familiarity with the following concepts:
>
> - Chat models
> - Function/tool calling

It is often useful to have a model return output that matches a specific schema. One common use-case is extracting data from text to insert into a database or use with some other downstream system. This guide covers a few strategies for getting structured outputs from a model.

The `.with_structured_output()` method

> **SUPPORTED MODELS**
>
> You can find a list of models that support this method here.

This is the easiest and most reliable way to get structured outputs. `with_structured_output()` is implemented for models that provide native APIs for structuring outputs, like tool/function calling or JSON mode, and makes use of these capabilities under the hood.

This method takes a schema as input which specifies the names, types, and descriptions of the desired output attributes. The method returns a model-like Runnable, except that instead of outputting strings or messages it outputs objects corresponding to the given

schema. The schema can be specified as a TypedDict class, JSON Schema or a Pydantic class. If TypedDict or JSON Schema are used then a dictionary will be returned by the Runnable, and if a Pydantic class is used then a Pydantic object will be returned.

As an example, let's get a model to generate a joke and separate the setup from the punchline:

Select chat model: OpenAI ▾

```
pip install -qU langchain-openai
```

```
import getpass
import os

if not os.environ.get("OPENAI_API_KEY"):
  os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter API key for OpenAI:
")

from langchain_openai import ChatOpenAI

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

Pydantic class

If we want the model to return a Pydantic object, we just need to pass in the desired Pydantic class. The key advantage of using Pydantic is that the model-generated output will be validated. Pydantic will raise an error if any required fields are missing or if any fields are of the wrong type.

```
from typing import Optional

from pydantic import BaseModel, Field


# Pydantic
class Joke(BaseModel):
    """Joke to tell user."""
```

```
    setup: str = Field(description="The setup of the joke")
    punchline: str = Field(description="The punchline to the joke")
    rating: Optional[int] = Field(
        default=None, description="How funny the joke is, from 1 to 10"
    )


structured_llm = llm.with_structured_output(Joke)

structured_llm.invoke("Tell me a joke about cats")
```

```
Joke(setup='Why was the cat sitting on the computer?', punchline='Because it wanted to keep an eye on the mouse!', rating=7)
```

> **TIP**
>
> Beyond just the structure of the Pydantic class, the name of the Pydantic class, the docstring, and the names and provided descriptions of parameters are very important. Most of the time `with_structured_output` is using a model's function/tool calling API, and you can effectively think of all of this information as being added to the model prompt.

TypedDict or JSON Schema

If you don't want to use Pydantic, explicitly don't want validation of the arguments, or want to be able to stream the model outputs, you can define your schema using a TypedDict class. We can optionally use a special `Annotated` syntax supported by LangChain that allows you to specify the default value and description of a field. Note, the default value is *not* filled in automatically if the model doesn't generate it, it is only used in defining the schema that is passed to the model.

> **REQUIREMENTS**
>
> - Core: `langchain-core>=0.2.26`
> - Typing extensions: It is highly recommended to import `Annotated` and `TypedDict` from `typing_extensions` instead of `typing` to ensure consistent

> behavior across Python versions.

```
from typing_extensions import Annotated, TypedDict


# TypedDict
class Joke(TypedDict):
    """Joke to tell user."""

    setup: Annotated[str, ..., "The setup of the joke"]

    # Alternatively, we could have specified setup as:

    # setup: str                    # no default, no description
    # setup: Annotated[str, ...]    # no default, no description
    # setup: Annotated[str, "foo"]  # default, no description

    punchline: Annotated[str, ..., "The punchline of the joke"]
    rating: Annotated[Optional[int], None, "How funny the joke is, from 1
to 10"]


structured_llm = llm.with_structured_output(Joke)

structured_llm.invoke("Tell me a joke about cats")
```

```
{'setup': 'Why was the cat sitting on the computer?',
 'punchline': 'Because it wanted to keep an eye on the mouse!',
 'rating': 7}
```

Equivalently, we can pass in a JSON Schema dict. This requires no imports or classes and makes it very clear exactly how each parameter is documented, at the cost of being a bit more verbose.

```
json_schema = {
    "title": "joke",
    "description": "Joke to tell user.",
    "type": "object",
    "properties": {
        "setup": {
```

```
            "type": "string",
            "description": "The setup of the joke",
        },
        "punchline": {
            "type": "string",
            "description": "The punchline to the joke",
        },
        "rating": {
            "type": "integer",
            "description": "How funny the joke is, from 1 to 10",
            "default": None,
        },
    },
    "required": ["setup", "punchline"],
}
structured_llm = llm.with_structured_output(json_schema)

structured_llm.invoke("Tell me a joke about cats")
```

```
{'setup': 'Why was the cat sitting on the computer?',
 'punchline': 'Because it wanted to keep an eye on the mouse!'}
```

Choosing between multiple schemas

The simplest way to let the model choose from multiple schemas is to create a parent schema that has a Union-typed attribute.

Using Pydantic

```
from typing import Union


class Joke(BaseModel):
    """Joke to tell user."""

    setup: str = Field(description="The setup of the joke")
    punchline: str = Field(description="The punchline to the joke")
    rating: Optional[int] = Field(
        default=None, description="How funny the joke is, from 1 to 10"
    )


class ConversationalResponse(BaseModel):
```

```
    """Respond in a conversational manner. Be kind and helpful."""

    response: str = Field(description="A conversational response to the
user's query")


class FinalResponse(BaseModel):
    final_output: Union[Joke, ConversationalResponse]


structured_llm = llm.with_structured_output(FinalResponse)

structured_llm.invoke("Tell me a joke about cats")
```

```
FinalResponse(final_output=Joke(setup='Why was the cat sitting on the
computer?', punchline='Because it wanted to keep an eye on the mouse!',
rating=7))
```

```
structured_llm.invoke("How are you today?")
```

```
FinalResponse(final_output=ConversationalResponse(response="I'm just a
computer program, so I don't have feelings, but I'm here and ready to help
you with whatever you need!"))
```

Using TypedDict

```
from typing import Optional, Union

from typing_extensions import Annotated, TypedDict


class Joke(TypedDict):
    """Joke to tell user."""

    setup: Annotated[str, ..., "The setup of the joke"]
    punchline: Annotated[str, ..., "The punchline of the joke"]
    rating: Annotated[Optional[int], None, "How funny the joke is, from 1
to 10"]


class ConversationalResponse(TypedDict):
```

```
    """Respond in a conversational manner. Be kind and helpful."""

    response: Annotated[str, ..., "A conversational response to the user's
query"]


class FinalResponse(TypedDict):
    final_output: Union[Joke, ConversationalResponse]


structured_llm = llm.with_structured_output(FinalResponse)

structured_llm.invoke("Tell me a joke about cats")
```

```
{'final_output': {'setup': 'Why was the cat sitting on the computer?',
  'punchline': 'Because it wanted to keep an eye on the mouse!',
  'rating': 7}}
```

```
structured_llm.invoke("How are you today?")
```

```
{'final_output': {'response': "I'm just a computer program, so I don't have
feelings, but I'm here and ready to help you with whatever you need!"}}
```

Responses shall be identical to the ones shown in the Pydantic example.

Alternatively, you can use tool calling directly to allow the model to choose between options, if your chosen model supports it. This involves a bit more parsing and setup but in some instances leads to better performance because you don't have to use nested schemas. See this how-to guide for more details.

Streaming

We can stream outputs from our structured model when the output type is a dict (i.e., when the schema is specified as a TypedDict class or JSON Schema dict).

> **INFO**
>
> Note that what's yielded is already aggregated chunks, not deltas.

```
from typing_extensions import Annotated, TypedDict


# TypedDict
class Joke(TypedDict):
    """Joke to tell user."""

    setup: Annotated[str, ..., "The setup of the joke"]
    punchline: Annotated[str, ..., "The punchline of the joke"]
    rating: Annotated[Optional[int], None, "How funny the joke is, from 1
to 10"]


structured_llm = llm.with_structured_output(Joke)

for chunk in structured_llm.stream("Tell me a joke about cats"):
    print(chunk)
```

```
{}
{'setup': ''}
{'setup': 'Why'}
{'setup': 'Why was'}
{'setup': 'Why was the'}
{'setup': 'Why was the cat'}
{'setup': 'Why was the cat sitting'}
{'setup': 'Why was the cat sitting on'}
{'setup': 'Why was the cat sitting on the'}
{'setup': 'Why was the cat sitting on the computer'}
{'setup': 'Why was the cat sitting on the computer?'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': ''}
{'setup': 'Why was the cat sitting on the computer?', 'punchline':
'Because'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
```

```
it wanted to keep'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an eye'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an eye on'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an eye on the'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an eye on the mouse'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an eye on the mouse!'}
{'setup': 'Why was the cat sitting on the computer?', 'punchline': 'Because
it wanted to keep an eye on the mouse!', 'rating': 7}
```

Few-shot prompting

For more complex schemas it's very useful to add few-shot examples to the prompt. This can be done in a few ways.

The simplest and most universal way is to add examples to a system message in the prompt:

```
from langchain_core.prompts import ChatPromptTemplate

system = """You are a hilarious comedian. Your specialty is knock-knock
jokes. \
Return a joke which has the setup (the response to "Who's there?") and the
final punchline (the response to "<setup> who?").

Here are some examples of jokes:

example_user: Tell me a joke about planes
example_assistant: {{"setup": "Why don't planes ever get tired?",
"punchline": "Because they have rest wings!", "rating": 2}}

example_user: Tell me another joke about planes
example_assistant: {{"setup": "Cargo", "punchline": "Cargo 'vroom vroom',
but planes go 'zoom zoom'!", "rating": 10}}

example_user: Now about caterpillars
example_assistant: {{"setup": "Caterpillar", "punchline": "Caterpillar
```

```
really slow, but watch me turn into a butterfly and steal the show!",
"rating": 5}}"""

prompt = ChatPromptTemplate.from_messages([("system", system), ("human",
"{input}")])

few_shot_structured_llm = prompt | structured_llm
few_shot_structured_llm.invoke("what's something funny about woodpeckers")
```

API Reference: ChatPromptTemplate

```
{'setup': 'Woodpecker',
 'punchline': "Woodpecker you a joke, but I'm afraid it might be too 'hole-
some'!",
 'rating': 7}
```

When the underlying method for structuring outputs is tool calling, we can pass in our examples as explicit tool calls. You can check if the model you're using makes use of tool calling in its API reference.

```
from langchain_core.messages import AIMessage, HumanMessage, ToolMessage

examples = [
    HumanMessage("Tell me a joke about planes", name="example_user"),
    AIMessage(
        "",
        name="example_assistant",
        tool_calls=[
            {
                "name": "joke",
                "args": {
                    "setup": "Why don't planes ever get tired?",
                    "punchline": "Because they have rest wings!",
                    "rating": 2,
                },
                "id": "1",
            }
        ],
    ),
    # Most tool-calling models expect a ToolMessage(s) to follow an
AIMessage with tool calls.
    ToolMessage("", tool_call_id="1"),
```

```
    # Some models also expect an AIMessage to follow any ToolMessages,
    # so you may need to add an AIMessage here.
    HumanMessage("Tell me another joke about planes", name="example_user"),
    AIMessage(
        "",
        name="example_assistant",
        tool_calls=[
            {
                "name": "joke",
                "args": {
                    "setup": "Cargo",
                    "punchline": "Cargo 'vroom vroom', but planes go 'zoom
zoom'!",
                    "rating": 10,
                },
                "id": "2",
            }
        ],
    ),
    ToolMessage("", tool_call_id="2"),
    HumanMessage("Now about caterpillars", name="example_user"),
    AIMessage(
        "",
        tool_calls=[
            {
                "name": "joke",
                "args": {
                    "setup": "Caterpillar",
                    "punchline": "Caterpillar really slow, but watch me
turn into a butterfly and steal the show!",
                    "rating": 5,
                },
                "id": "3",
            }
        ],
    ),
    ToolMessage("", tool_call_id="3"),
]
system = """You are a hilarious comedian. Your specialty is knock-knock
jokes. \
Return a joke which has the setup (the response to "Who's there?") \
and the final punchline (the response to "<setup> who?")."""

prompt = ChatPromptTemplate.from_messages(
    [("system", system), ("placeholder", "{examples}"), ("human",
"{input}")]
```

```
)
few_shot_structured_llm = prompt | structured_llm
few_shot_structured_llm.invoke({"input": "crocodiles", "examples":
examples})
```

API Reference: AIMessage | HumanMessage | ToolMessage

```
{'setup': 'Crocodile',
 'punchline': 'Crocodile be seeing you later, alligator!',
 'rating': 6}
```

For more on few shot prompting when using tool calling, see here.

(Advanced) Specifying the method for structuring outputs

For models that support more than one means of structuring outputs (i.e., they support both tool calling and JSON mode), you can specify which method to use with the `method=` argument.

> **JSON MODE**
>
> If using JSON mode you'll have to still specify the desired schema in the model prompt. The schema you pass to `with_structured_output` will only be used for parsing the model outputs, it will not be passed to the model the way it is with tool calling.
>
> To see if the model you're using supports JSON mode, check its entry in the API reference.

```
structured_llm = llm.with_structured_output(None, method="json_mode")

structured_llm.invoke(
    "Tell me a joke about cats, respond in JSON with `setup` and
`punchline` keys"
)
```

```
{'setup': 'Why was the cat sitting on the computer?',
 'punchline': 'Because it wanted to keep an eye on the mouse!'}
```

complex. You can avoid raising exceptions and handle the raw output yourself by passing `include_raw=True`. This changes the output format to contain the raw message output, the `parsed` value (if successful), and any resulting errors:

```
structured_llm = llm.with_structured_output(Joke, include_raw=True)

structured_llm.invoke("Tell me a joke about cats")
```

```
{'raw': AIMessage(content='', additional_kwargs={'tool_calls': [{'id':
'call_f25ZRmh8u5vHlOWfTUw8sJFZ', 'function': {'arguments': '{"setup":"Why
was the cat sitting on the computer?","punchline":"Because it wanted to keep
an eye on the mouse!","rating":7}', 'name': 'Joke'}, 'type': 'function'}]},
response_metadata={'token_usage': {'completion_tokens': 33, 'prompt_tokens':
93, 'total_tokens': 126}, 'model_name': 'gpt-4o-2024-05-13',
'system_fingerprint': 'fp_4e2b2da518', 'finish_reason': 'stop', 'logprobs':
None}, id='run-d880d7e2-df08-4e9e-ad92-dfc29f2fd52f-0', tool_calls=[{'name':
'Joke', 'args': {'setup': 'Why was the cat sitting on the computer?',
'punchline': 'Because it wanted to keep an eye on the mouse!', 'rating': 7},
'id': 'call_f25ZRmh8u5vHlOWfTUw8sJFZ', 'type': 'tool_call'}],
usage_metadata={'input_tokens': 93, 'output_tokens': 33, 'total_tokens':
126}),
 'parsed': {'setup': 'Why was the cat sitting on the computer?',
  'punchline': 'Because it wanted to keep an eye on the mouse!',
  'rating': 7},
 'parsing_error': None}
```

Prompting and parsing model outputs directly

Not all models support `.with_structured_output()`, since not all models have tool calling or JSON mode support. For such models you'll need to directly prompt the model to use a specific format, and use an output parser to extract the structured response from

the raw model output.

Using `PydanticOutputParser`

The following example uses the built-in `PydanticOutputParser` to parse the output of a chat model prompted to match the given Pydantic schema. Note that we are adding `format_instructions` directly to the prompt from a method on the parser:

```
from typing import List

from langchain_core.output_parsers import PydanticOutputParser
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field


class Person(BaseModel):
    """Information about a person."""

    name: str = Field(..., description="The name of the person")
    height_in_meters: float = Field(
        ..., description="The height of the person expressed in meters."
    )


class People(BaseModel):
    """Identifying information about all people in a text."""

    people: List[Person]


# Set up a parser
parser = PydanticOutputParser(pydantic_object=People)

# Prompt
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "Answer the user query. Wrap the output in `json`
tags\n{format_instructions}",
        ),
        ("human", "{query}"),
    ]
```

```
).partial(format_instructions=parser.get_format_instructions())
```

API Reference: PydanticOutputParser | ChatPromptTemplate

Let's take a look at what information is sent to the model:

```
query = "Anna is 23 years old and she is 6 feet tall"

print(prompt.invoke({"query": query}).to_string())
```

```
System: Answer the user query. Wrap the output in `json` tags
The output should be formatted as a JSON instance that conforms to the JSON
schema below.

As an example, for the schema {"properties": {"foo": {"title": "Foo",
"description": "a list of strings", "type": "array", "items": {"type":
"string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well-formatted instance of the
schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-
formatted.

Here is the output schema:
\`\`\`
{"description": "Identifying information about all people in a text.",
"properties": {"people": {"title": "People", "type": "array", "items":
{"$ref": "#/definitions/Person"}}}, "required": ["people"], "definitions":
{"Person": {"title": "Person", "description": "Information about a person.",
"type": "object", "properties": {"name": {"title": "Name", "description":
"The name of the person", "type": "string"}, "height_in_meters": {"title":
"Height In Meters", "description": "The height of the person expressed in
meters.", "type": "number"}}, "required": ["name", "height_in_meters"]}}}
\`\`\`
Human: Anna is 23 years old and she is 6 feet tall
```

And now let's invoke it:

```
chain = prompt | llm | parser

chain.invoke({"query": query})
```

```
People(people=[Person(name='Anna', height_in_meters=1.8288)])
```

Custom Parsing

You can also create a custom prompt and parser with LangChain Expression Language (LCEL), using a plain function to parse the output from the model:

```
import json
import re
from typing import List

from langchain_core.messages import AIMessage
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field


class Person(BaseModel):
    """Information about a person."""

    name: str = Field(..., description="The name of the person")
    height_in_meters: float = Field(
        ..., description="The height of the person expressed in meters."
    )


class People(BaseModel):
    """Identifying information about all people in a text."""

    people: List[Person]


# Prompt
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "Answer the user query. Output your answer as JSON that  "
            "matches the given schema: \`\`\`json\n{schema}\n\`\`\`. "
            "Make sure to wrap the answer in \`\`\`json and \`\`\` tags",
        ),
        ("human", "{query}"),
```

```
    ]
).partial(schema=People.schema())


# Custom parser
def extract_json(message: AIMessage) -> List[dict]:
    """Extracts JSON content from a string where JSON is embedded between
\`\`\`json and \`\`\` tags.

    Parameters:
        text (str): The text containing the JSON content.

    Returns:
        list: A list of extracted JSON strings.
    """
    text = message.content
    # Define the regular expression pattern to match JSON blocks
    pattern = r"\`\`\`json(.*?)\`\`\`"

    # Find all non-overlapping matches of the pattern in the string
    matches = re.findall(pattern, text, re.DOTALL)

    # Return the list of matched JSON strings, stripping any leading or
trailing whitespace
    try:
        return [json.loads(match.strip()) for match in matches]
    except Exception:
        raise ValueError(f"Failed to parse: {message}")
```

API Reference: AIMessage | ChatPromptTemplate

Here is the prompt sent to the model:

```
query = "Anna is 23 years old and she is 6 feet tall"

print(prompt.format_prompt(query=query).to_string())
```

```
System: Answer the user query. Output your answer as JSON that  matches the
given schema: \`\`\`json
{'title': 'People', 'description': 'Identifying information about all people
in a text.', 'type': 'object', 'properties': {'people': {'title': 'People',
'type': 'array', 'items': {'$ref': '#/definitions/Person'}}}, 'required':
['people'], 'definitions': {'Person': {'title': 'Person', 'description':
```

```
'Information about a person.', 'type': 'object', 'properties': {'name':
{'title': 'Name', 'description': 'The name of the person', 'type':
'string'}, 'height_in_meters': {'title': 'Height In Meters', 'description':
'The height of the person expressed in meters.', 'type': 'number'}},
'required': ['name', 'height_in_meters']}}}
\`\`\`. Make sure to wrap the answer in \`\`\`json and \`\`\` tags
Human: Anna is 23 years old and she is 6 feet tall
```

And here's what it looks like when we invoke it:

```python
chain = prompt | llm | extract_json

chain.invoke({"query": query})
```

```
[{'people': [{'name': 'Anna', 'height_in_meters': 1.8288}]}]
```

Edit this page

Was this page helpful?