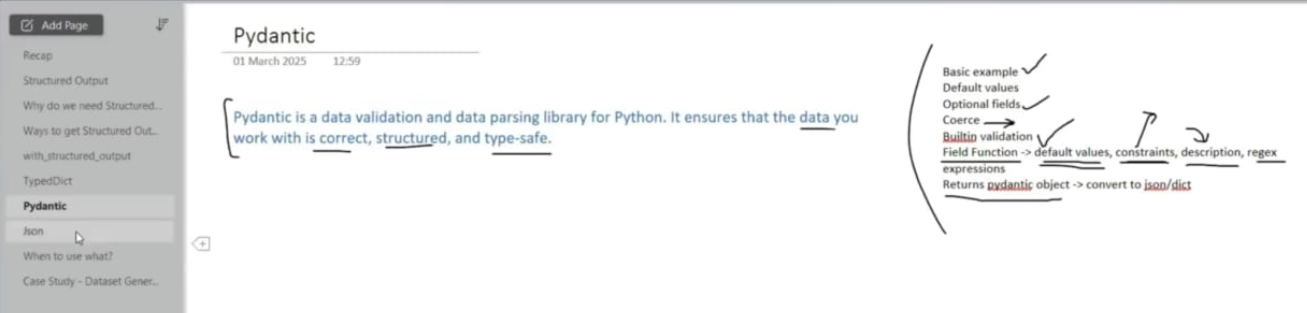
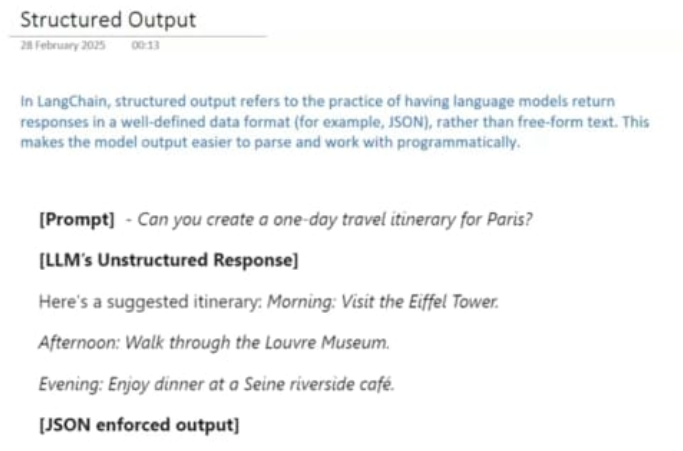
**Input output parsers**

<https://python.langchain.com/docs/how_to/structured_output/>

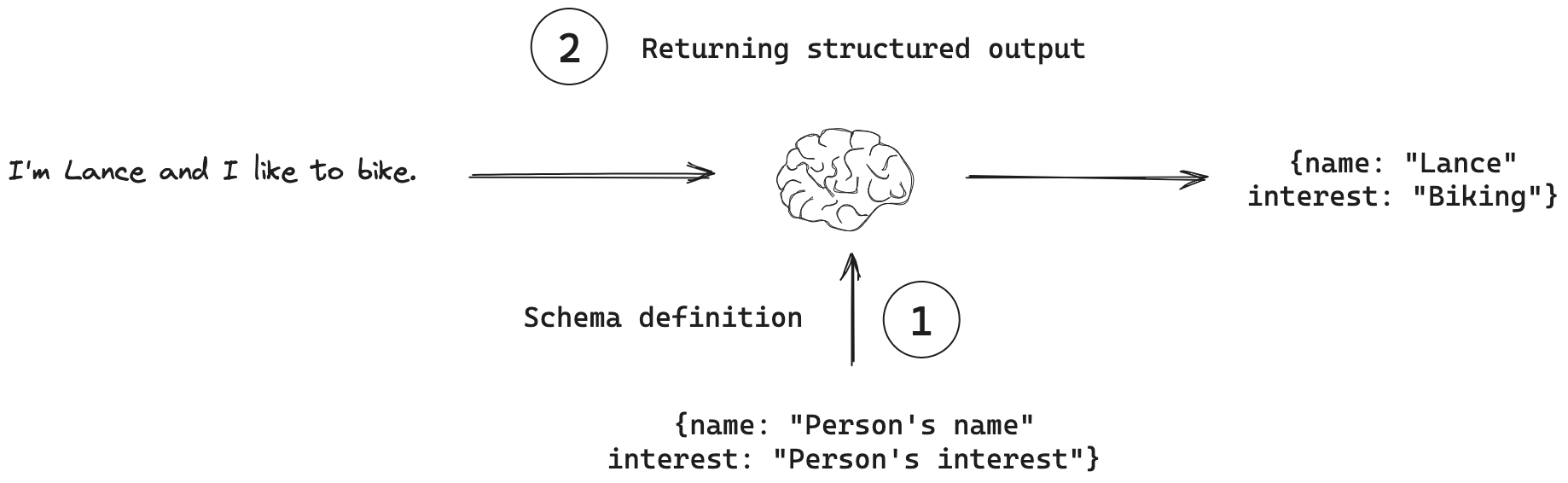




**Structured outputs**

**Overview**

For many applications, such as chatbots, models need to respond to users directly in natural language. However, there are scenarios where we need models to output in a *structured format*. For example, we might want to store the model output in a database and ensure that the output conforms to the database schema. This need motivates the concept of structured output, where models can be instructed to respond with a particular output structure.



**Key concepts**

1. **Schema definition:** The output structure is represented as a schema, which can be defined in several ways.
2. **Returning structured output:** The model is given this schema, and is instructed to return output that conforms to it.

**Recommended usage**

This pseudocode illustrates the recommended workflow when using structured output. LangChain provides a method, [with\_structured\_output()](https://python.langchain.com/docs/how_to/structured_output/#the-with_structured_output-method), that automates the process of binding the schema to the [model](https://python.langchain.com/docs/concepts/chat_models/) and parsing the output. This helper function is available for all model providers that support structured output.

# Define schema  
schema = {"foo": "bar"}  
# Bind schema to model  
model\_with\_structure = model.with\_structured\_output(schema)  
# Invoke the model to produce structured output that matches the schema  
structured\_output = model\_with\_structure.invoke(user\_input)

**Tool Order Matters**

When combining structured output with additional tools, bind tools **first**, then apply structured output:

# Correct  
model\_with\_tools = model.bind\_tools([tool1, tool2])  
structured\_model = model\_with\_tools.with\_structured\_output(schema)  
  
# Incorrect - will cause tool resolution errors  
structured\_model = model.with\_structured\_output(schema)  
broken\_model = structured\_model.bind\_tools([tool1, tool2])

**Schema definition**

The central concept is that the output structure of model responses needs to be represented in some way. While types of objects you can use depend on the model you're working with, there are common types of objects that are typically allowed or recommended for structured output in Python.

The simplest and most common format for structured output is a JSON-like structure, which in Python can be represented as a dictionary (dict) or list (list). JSON objects (or dicts in Python) are often used directly when the tool requires raw, flexible, and minimal-overhead structured data.

{  
 "answer": "The answer to the user's question",  
 "followup\_question": "A followup question the user could ask"  
}

As a second example, [Pydantic](https://docs.pydantic.dev/latest/) is particularly useful for defining structured output schemas because it offers type hints and validation. Here's an example of a Pydantic schema:

from pydantic import BaseModel, Field  
class ResponseFormatter(BaseModel):  
 """Always use this tool to structure your response to the user."""  
 answer: str = Field(description="The answer to the user's question")  
 followup\_question: str = Field(description="A followup question the user could ask")

**Returning structured output**

With a schema defined, we need a way to instruct the model to use it. While one approach is to include this schema in the prompt and *ask nicely* for the model to use it, this is not recommended. Several more powerful methods that utilizes native features in the model provider's API are available.

**Using tool calling**

Many [model providers support](https://python.langchain.com/docs/integrations/chat/) tool calling, a concept discussed in more detail in our [tool calling guide](https://python.langchain.com/docs/concepts/tool_calling/). In short, tool calling involves binding a tool to a model and, when appropriate, the model can *decide* to call this tool and ensure its response conforms to the tool's schema. With this in mind, the central concept is straightforward: *simply bind our schema to a model as a tool!* Here is an example using the ResponseFormatter schema defined above:

from langchain\_openai import ChatOpenAI  
model = ChatOpenAI(model="gpt-4o", temperature=0)  
# Bind responseformatter schema as a tool to the model  
model\_with\_tools = model.bind\_tools([ResponseFormatter])  
# Invoke the model  
ai\_msg = model\_with\_tools.invoke("What is the powerhouse of the cell?")

The arguments of the tool call are already extracted as a dictionary. This dictionary can be optionally parsed into a Pydantic object, matching our original ResponseFormatter schema.

# Get the tool call arguments  
ai\_msg.tool\_calls[0]["args"]  
{'answer': "The powerhouse of the cell is the mitochondrion. Mitochondria are organelles that generate most of the cell's supply of adenosine triphosphate (ATP), which is used as a source of chemical energy.",  
 'followup\_question': 'What is the function of ATP in the cell?'}  
# Parse the dictionary into a pydantic object  
pydantic\_object = ResponseFormatter.model\_validate(ai\_msg.tool\_calls[0]["args"])

**JSON mode**

In addition to tool calling, some model providers support a feature called JSON mode. This supports JSON schema definition as input and enforces the model to produce a conforming JSON output. You can find a table of model providers that support JSON mode [here](https://python.langchain.com/docs/integrations/chat/). Here is an example of how to use JSON mode with OpenAI:

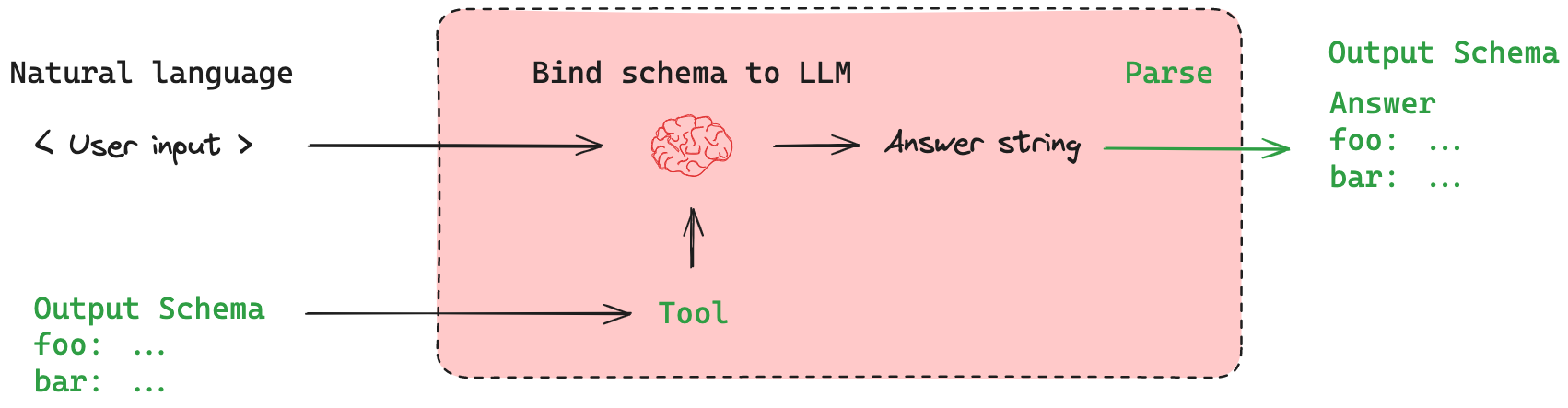
from langchain\_openai import ChatOpenAI  
model = ChatOpenAI(model="gpt-4o").with\_structured\_output(method="json\_mode")  
ai\_msg = model.invoke("Return a JSON object with key 'random\_ints' and a value of 10 random ints in [0-99]")  
ai\_msg  
{'random\_ints': [45, 67, 12, 34, 89, 23, 78, 56, 90, 11]}

**Structured output method**

There are a few challenges when producing structured output with the above methods:

1. When tool calling is used, tool call arguments needs to be parsed from a dictionary back to the original schema.
2. In addition, the model needs to be instructed to *always* use the tool when we want to enforce structured output, which is a provider specific setting.
3. When JSON mode is used, the output needs to be parsed into a JSON object.

With these challenges in mind, LangChain provides a helper function (with\_structured\_output()) to streamline the process.



This both binds the schema to the model as a tool and parses the output to the specified output schema.

# Bind the schema to the model  
model\_with\_structure = model.with\_structured\_output(ResponseFormatter)  
# Invoke the model  
structured\_output = model\_with\_structure.invoke("What is the powerhouse of the cell?")  
# Get back the pydantic object  
structured\_output  
ResponseFormatter(answer="The powerhouse of the cell is the mitochondrion. Mitochondria are organelles that generate most of the cell's supply of adenosine triphosphate (ATP), which is used as a source of chemical energy.", followup\_question='What is the function of ATP in the cell?')

**Further reading**

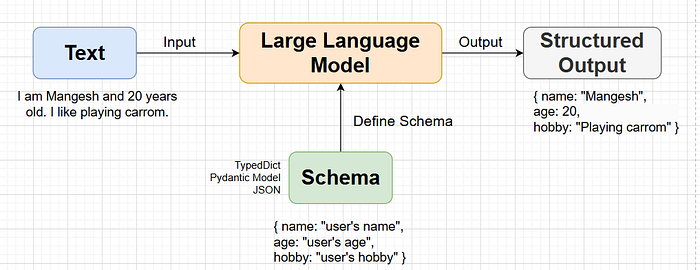
**What You’ll Learn**

* What is Structured Output?
* Why Structured Output is important in LLM applications
* Techniques to get Structured Output:
* *with\_structured\_output*
* *Output Parsers*
* Deep Dive: Structured Output in LangChain
* TypedDict
* Pydantic
* JSON Schema

**❓ What is Structured Output?**

In LangChain, structured output refers to the practice of having language models return responses in a well-defined data format (for example, JSON) rather than free-form text. This makes the model output easier to parse and work with programmatically.

Press enter or click to view image in full size



**❓ Why Structured Output is important in LLM applications**

LLMs by default return **unstructured text** — plain language that may vary every time. While this is fine for human reading, it’s problematic when:

* You want to feed LLM outputs into APIs, databases, or other systems
* You need to display results in a structured UI or dashboard
* You’re chaining the LLM output into another task or agent

**Example :**

🧠 Prompt:

**“Can you create a one-day travel itinerary for Paris?”**

❌ LLM’s Unstructured Response:

Here’s a suggested itinerary:   
Morning: Visit the Eiffel Tower.   
Afternoon: Walk through the Louvre Museum.   
Evening: Enjoy dinner at a Seine riverside café.

* While readable, this is **not machine-friendly**.
* You cannot easily extract, display, or reuse this information in code.

✅ JSON Enforced Output (Structured):

[  
 { "time": "Morning", "activity": "Visit the Eiffel Tower" },  
 { "time": "Afternoon", "activity": "Walk through the Louvre Museum" },  
 { "time": "Evening", "activity": "Enjoy dinner at a Seine riverside café" }  
]

* Clearly defined fields (time, activity)
* Easy to parse, store, display, or chain with another system
* Suitable for APIs, UI rendering, or databases

**🔄 Techniques to get Structured Output**

Different LLMs (Large Language Models) have **different capabilities** when it comes to returning structured output. LangChain provides two distinct approaches based on what your LLM supports.

**✅ 1. Using**with\_structured\_output()**(For Models That Support It)**

Some advanced LLMs like **OpenAI’s GPT-4**, **Anthropic Claude**, or **Gemini Pro** can directly generate **JSON-style structured outputs** if properly instructed.

LangChain’s with\_structured\_output() method allows you to define your desired schema using TypedDict, Pydantic, or JSON Schema, and the LLM will try to follow it strictly.

**🔹 Best For:**

* Direct structured JSON output
* APIs, dashboards, pipelines
* Fast, model-native formatting

**❌ 2. Using Output Parsers (For Models That Can’t Output Structured JSON Natively)**

Some open-source or less capable LLMs **can’t natively return structured formats**, even with clear prompting. They often return plain text or loosely structured responses.

In such cases, LangChain provides **Output Parsers** — post-processing tools that parse raw LLM output into structured formats.

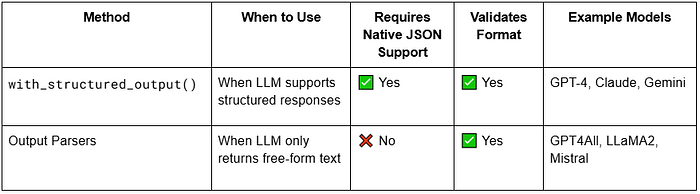
* Output parsers can extract fields using regex, JSON deserialization, or even another LLM.
* You define a schema (Pydantic, JSON, etc.) and the parser attempts to align the LLM’s output accordingly.

**🔹 Best For:**

* Local models (e.g., LLaMA2, Mistral, GPT4All)
* Use cases where structured output is critical but unsupported natively
* Adding structure after receiving unstructured text

**🔁 Summary Table:**

Press enter or click to view image in full size



**🧠 Deep Dive: Structured Output in LangChain**

Structured output is one of the most powerful features in LangChain. It allows us to instruct the LLM to return responses in a well-defined format, such as a dictionary, JSON object, or even a validated model, instead of free-form text.

**🚀 Step 1: Defining Format with**with\_structured\_output()

Before calling the LLM with .invoke(), we can define the **expected data format** using with\_structured\_output(). This ensures LangChain instructs the LLM to respond in that format.

There are **three ways** to define structured output:

**1️⃣ TypedDict (for structure only, not validation)**

**💡 What is TypedDict?**

TypedDict is a way to define a dictionary in Python where you specify what keys and values should exist. It helps ensure that your dictionary follows a specific structure.

Why use TypedDict?

* It tells Python what keys are required and what types of values they should have.
* It does not validate data at runtime (it just helps with type hints for better coding)

from typing import TypedDict  
  
class Person(TypedDict):  
 name: str  
 age: int  
  
new\_person: Person = {'name':'Rahul', 'age':35}  
  
print(new\_person)

But here’s the catch:

**⚠️ Limitations of TypedDict**

* ❌ **No runtime validation**: You can define age: int But still assign "35" (string) without error.
* ❌ LLM may **not return data in the expected type**, and you won’t know.
* ❌ No support for constraints(data validation) (e.g.“rating must be > 3”).

In short: **TypedDict is useful for shaping output**, not for validating it.

**with\_structured\_output example:**

import os  
from typing import TypedDict, Annotated, Optional, Literal  
from langchain\_community.llms import Ollama  
  
model = Ollama(model="deepseek")  
  
# schema if we want to annotate the output basicaly to tell llm what to return and create prompt for us  
class Review(TypedDict):  
  
 key\_themes: Annotated[list[str], "Write down all the key themes discussed in the review in a list"]  
 summary: Annotated[str, "A brief summary of the review"]  
 sentiment: Annotated[Literal["pos", "neg"], "Return sentiment of the review either negative, positive or neutral"]  
 pros: Annotated[Optional[list[str]], "Write down all the pros inside a list"]  
 cons: Annotated[Optional[list[str]], "Write down all the cons inside a list"]  
 name: Annotated[Optional[str], "Write the name of the reviewer"]  
  
structured\_model = model.with\_structured\_output(Review)  
  
result = structured\_model.invoke("""I recently upgraded to the Samsung Galaxy S24 Ultra, and I must say, it’s an absolute powerhouse! The Snapdragon 8 Gen 3 processor makes everything lightning fast—whether I’m gaming, multitasking, or editing photos. The 5000mAh battery easily lasts a full day even with heavy use, and the 45W fast charging is a lifesaver.  
  
The S-Pen integration is a great touch for note-taking and quick sketches, though I don't use it often. What really blew me away is the 200MP camera—the night mode is stunning, capturing crisp, vibrant images even in low light. Zooming up to 100x actually works well for distant objects, but anything beyond 30x loses quality.  
  
However, the weight and size make it a bit uncomfortable for one-handed use. Also, Samsung’s One UI still comes with bloatware—why do I need five different Samsung apps for things Google already provides? The $1,300 price tag is also a hard pill to swallow.  
  
Pros:  
Insanely powerful processor (great for gaming and productivity)  
Stunning 200MP camera with incredible zoom capabilities  
Long battery life with fast charging  
S-Pen support is unique and useful  
   
Review by Nitish Singh  
""")  
  
print(result['sentiment'])  
print(result['summary'])

**2️⃣ Pydantic (structure + validation ✅)**

**💡 What is Pydantic?**

**Pydantic** is a data validation and data parsing library for Python. It ensures that the data you work with is correct, structured, and type-safe.

from pydantic import BaseModel  
  
class Student(BaseModel):  
  
 name: str  
  
new\_student = {'name':'Lucky'}  
  
student = Student(\*\*new\_student)  
  
print(type(student))

**Default Values in Pydantic:**For assigning default values to a variable.

from pydantic import BaseModel  
  
class Student(BaseModel):  
  
 name: str = 'Lucky'  
  
new\_student = {}  
  
student = Student(\*\*new\_student)  
  
print(student.name)

output

Lucky

**Optional in Pydantic:**if some variables are optional, we can use them. It will not raise an error if values not assign to them

from pydantic import BaseModel  
  
class Student(BaseModel):  
  
 name: str = 'Lucky'  
 age: Optional[int] = None  
  
new\_student = {}  
  
student = Student(\*\*new\_student)  
  
print(student.age)

Output :

None

**Coerce**: Pydantic checks the data type of the variable, and if something else comes it tries to convert it internally

from pydantic import BaseModel  
from typing import Optional   
  
class Student(BaseModel):  
  
 name: str = 'Lucky'  
 age: Optional[int] = None  
  
new\_student = {'age' : '32'}  
  
student = Student(\*\*new\_student)  
  
print(student.age)

Output :

32

**Built-in validation:** like email

from pydantic import BaseModel, EmailStr  
  
class Student(BaseModel):  
  
 name: str = 'Lucky'  
 age: Optional[int] = None  
 email: EmailStr  
  
new\_student = {'age' : '32', 'email' : 'abc'}  
  
student = Student(\*\*new\_student)  
  
print(student.age)

Output:

pydantic\_core.\_pydantic\_core.ValidationError: 1 validation error for Student  
email  
 value is not a valid email address: An email address must have an @-sign. [type=value\_error, input\_value='abc', input\_type=str]

**Field function:**Let’s say we want student's CGPA should be between 0 and 10, then we can use Field function.

from pydantic import BaseModel, EmailStr, Field  
from typing import Optional  
  
class Student(BaseModel):  
  
 name: str = 'nitish'  
 age: Optional[int] = None  
 email: EmailStr  
 cgpa: float = Field(gt=0, lt=10, default=5, description='A decimal value representing the cgpa of the student')  
  
  
new\_student = {'age':'32', 'email':'abc@gmail.com'}  
  
student = Student(\*\*new\_student)  
  
print(student.age)

**Pydantic object to JSON or dict :**

from pydantic import BaseModel, EmailStr, Field  
from typing import Optional  
  
class Student(BaseModel):  
  
 name: str = 'nitish'  
 age: Optional[int] = None  
 email: EmailStr  
 cgpa: float = Field(gt=0, lt=10, default=5, description='A decimal value representing the cgpa of the student')  
  
  
new\_student = {'age':'32', 'email':'abc@gmail.com'}  
  
student = Student(\*\*new\_student)  
  
student\_dict = dict(student)  
  
print(student\_dict['age'])  
  
student\_json = student.model\_dump\_json()

**3️⃣ JSON Schema**

If you’re working in environments where you want to define schema in JSON (not Python), or use third-party APIs that rely on JSON Schemas, LangChain also supports:

{  
 "title": "student",  
 "description": "schema about students",  
 "type": "object",  
 "properties":{  
 "name":"string",  
 "age":"integer"  
 },  
 "required":["name"]  
}

**with\_structured\_output example:**

from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
model = ChatOpenAI()  
  
# schema  
json\_schema = {  
 "title": "Review",  
 "type": "object",  
 "properties": {  
 "key\_themes": {  
 "type": "array",  
 "items": {  
 "type": "string"  
 },  
 "description": "Write down all the key themes discussed in the review in a list"  
 },  
 "summary": {  
 "type": "string",  
 "description": "A brief summary of the review"  
 },  
 "sentiment": {  
 "type": "string",  
 "enum": ["pos", "neg"],  
 "description": "Return sentiment of the review either negative, positive or neutral"  
 },  
 "pros": {  
 "type": ["array", "null"],  
 "items": {  
 "type": "string"  
 },  
 "description": "Write down all the pros inside a list"  
 },  
 "cons": {  
 "type": ["array", "null"],  
 "items": {  
 "type": "string"  
 },  
 "description": "Write down all the cons inside a list"  
 },  
 "name": {  
 "type": ["string", "null"],  
 "description": "Write the name of the reviewer"  
 }  
 },  
 "required": ["key\_themes", "summary", "sentiment"]  
}  
  
  
structured\_model = model.with\_structured\_output(json\_schema)  
  
result = structured\_model.invoke("""I recently upgraded to the Samsung Galaxy S24 Ultra, and I must say, it’s an absolute powerhouse! The Snapdragon 8 Gen 3 processor makes everything lightning fast—whether I’m gaming, multitasking, or editing photos. The 5000mAh battery easily lasts a full day even with heavy use, and the 45W fast charging is a lifesaver.  
  
The S-Pen integration is a great touch for note-taking and quick sketches, though I don't use it often. What really blew me away is the 200MP camera—the night mode is stunning, capturing crisp, vibrant images even in low light. Zooming up to 100x actually works well for distant objects, but anything beyond 30x loses quality.  
  
However, the weight and size make it a bit uncomfortable for one-handed use. Also, Samsung’s One UI still comes with bloatware—why do I need five different Samsung apps for things Google already provides? The $1,300 price tag is also a hard pill to swallow.  
  
Pros:  
Insanely powerful processor (great for gaming and productivity)  
Stunning 200MP camera with incredible zoom capabilities  
Long battery life with fast charging  
S-Pen support is unique and useful  
   
Review by Nitish Singh  
""")  
  
print(result)

**Ideal for:**

* External tools
* Cross-language setups
* Web-based schema validation

**When to Use What?**

**✅ Use**TypedDict**if:**

* You **only need type hints** (basic structure enforcement).
* You **don’t need validation** (e.g., checking if numbers are positive).
* You **trust the LLM** to return correct data.

**✅ Use**Pydantic**if:**

* You need **data validation** (e.g., sentiment must be "positive", "neutral", or "negative").
* You need **default values** if the LLM misses fields.
* You want **automatic type conversion** (e.g., "100" → 100).

**✅ Use**JSON Schema**if:**

* You **don’t want to import extra Python libraries** (like Pydantic).
* You need **validation, but don’t need Python objects**.
* You want to define structure **in a standard JSON format**.

**What is Structured Output in LangChain**

Structured output refers to **predictable, machine-readable outputs from LLMs** rather than free-form text.

* LLMs naturally generate **text**, but for applications like **databases, APIs, and ML pipelines**, you need **consistent formats**.
* Structured output allows you to **extract fields reliably**, **validate types**, and **avoid parsing errors**.

**Example:**

Instead of:

"Alice is 25 years old and lives in New York."

Structured output could be:

{

"name": "Alice",

"age": 25,

"city": "New York"

}

**2️⃣ Why Structured Output is Important**

1. **Data Consistency:** Ensures all responses have the same fields and types.
2. **Automation:** Easier to feed into databases, APIs, or pipelines.
3. **Validation:** Catch errors early using type checking.
4. **Integration:** Works well with Python objects, JSON schemas, and typed dicts.
5. **Downstream Processing:** Simplifies analytics, ML data prep, and dataset generation.

**3️⃣ Ways to Get Structured Output in LangChain**

LangChain provides **multiple approaches**:

**A. Structured Output with LangChain Schema**

* Use StructuredOutputParser or OutputFixingParser.
* Define expected fields, types, and constraints.
* Example with JSON output parser:

from langchain.output\_parsers import StructuredOutputParser

from langchain.prompts import PromptTemplate

from langchain.chat\_models import ChatOpenAI

# Define schema

schema = {

"name": "str",

"age": "int",

"city": "str"

}

parser = StructuredOutputParser.from\_dict(schema)

prompt = PromptTemplate(

input\_variables=["text"],

template="Extract structured info from the text:\n{text}\nReturn JSON."

)

llm = ChatOpenAI(model="gpt-4", temperature=0)

text = "Alice is 25 years old and lives in New York."

output = llm(prompt.format(text=text))

structured = parser.parse(output)

print(structured)

# {'name': 'Alice', 'age': 25, 'city': 'New York'}

**B. Typed Dictionaries**

* Python TypedDict provides **type hints** for structured data.
* Works well with **pydantic** or static type checking.

from typing import TypedDict

class Person(TypedDict):

name: str

age: int

city: str

data: Person = {"name": "Alice", "age": 25, "city": "New York"}

* Can integrate with LLM outputs using **parsers** that validate types.

**C. Pydantic Models**

* **Pydantic** validates both types and constraints.
* Useful for **production pipelines** where you need strict validation.

from pydantic import BaseModel

class PersonModel(BaseModel):

name: str

age: int

city: str

# Convert LLM output to Pydantic model

person = PersonModel.parse\_raw('{"name": "Alice", "age": 25, "city": "New York"}')

print(person.name) # Alice

* Advantage: automatically raises **validation errors** for wrong types.

**D. JSON Output**

* Simplest approach: ask LLM to produce **JSON** directly.
* Then parse using json.loads:

import json

json\_str = '{"name": "Alice", "age": 25, "city": "New York"}'

data = json.loads(json\_str)

* Less strict than Pydantic; can combine with validation logic.

**4️⃣ When to Use Which Method**

| **Method** | **When to Use** |
| --- | --- |
| **StructuredOutputParser** | Quick integration in LangChain, flexible, works with any LLM |
| **TypedDict** | Static type checking in Python, light-weight, no runtime validation |
| **Pydantic** | Production pipelines, strict validation, ensures type and constraint safety |
| **JSON parsing** | Simple scripts, rapid prototyping, less strict, may need manual validation |

**Rule of thumb:**

* **Experiment / prototype** → JSON
* **Type safety / Pythonic code** → TypedDict
* **Production / strict validation** → Pydantic
* **LangChain integration** → StructuredOutputParser

**5️⃣ Case Study: Dataset Generation**

Suppose you want to generate a dataset of **people’s profiles** from unstructured text for training a model.

**Steps:**

1. **Unstructured text input:**

"Alice is 25 years old and lives in New York. Bob is 30 and lives in San Francisco."

1. **Prompt LLM for structured output:**

prompt\_text = """

Extract structured info from the following text. Return a JSON list of people:

Text: {text}

"""

1. **Parse LLM output using Pydantic / LangChain parser:**

from pydantic import BaseModel

from typing import List

import json

class Person(BaseModel):

name: str

age: int

city: str

output = '[{"name": "Alice", "age": 25, "city": "New York"}, {"name": "Bob", "age": 30, "city": "San Francisco"}]'

people: List[Person] = [Person.parse\_obj(d) for d in json.loads(output)]

1. **Use dataset for ML / Analytics / DB insert.**

**6️⃣ Best Practices**

1. Always define **expected fields and types**.
2. Use **LLM output constraints** (StructuredOutputParser) instead of free-form text.
3. Validate with **Pydantic** for production.
4. For quick scripts or prototyping, **JSON parsing** is fine.
5. Consider **multi-step extraction**: extract raw JSON → validate → normalize.
6. Combine **SystemMessage instructions** with **structured output schema** for reliability.

**✅ Summary**

* Structured output is critical for **predictable, machine-readable LLM responses**.
* **Methods:** LangChain parsers, TypedDict, Pydantic, JSON.
* **When to use:** Prototyping → JSON, Python type safety → TypedDict, Production → Pydantic.
* **Case study:** Dataset generation for ML models is a classic use case.
* **LangChain support:** makes schema + structured outputs easy and integrates directly with chat models
* **Why structured outputs are needed**
* **Different approaches**
* **Code demos with prompts + chains**

**1️⃣ Why Structured Outputs?**

LLMs often return free-form text:

Alice is 25 years old and lives in New York.

Problems with free-form text:

* Hard to extract fields automatically
* Error-prone parsing
* Difficult to use in downstream pipelines

Structured outputs solve this by producing predictable data:

{

"name": "Alice",

"age": 25,

"city": "New York"

}

LangChain provides multiple ways to **enforce structure**:

* JSON
* Pydantic models
* TypedDict
* JSON Schema

**2️⃣ JSON Structured Output with WithStructuredOutputParser**

**Purpose:**

* Ask LLM to produce JSON and **parse it automatically**.

**Code Demo with PromptTemplate + Chain:**

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain.output\_parsers import JSONOutputParser, WithStructuredOutputParser

# 1️⃣ LLM

llm = ChatOpenAI(model="gpt-4", temperature=0)

# 2️⃣ JSON parser

json\_parser = JSONOutputParser()

structured\_parser = WithStructuredOutputParser(parser=json\_parser)

# 3️⃣ Prompt

prompt\_template = PromptTemplate(

input\_variables=["person\_description"],

template="Return a JSON with name, age, city for: {person\_description}"

)

# 4️⃣ Chain

chain = LLMChain(llm=llm, prompt=prompt\_template)

# 5️⃣ Run

raw\_output = chain.run(person\_description="Alice, 25, New York")

parsed\_output = structured\_parser.parse(raw\_output)

print(parsed\_output)

✅ Returns: {'name': 'Alice', 'age': 25, 'city': 'New York'}

**3️⃣ Pydantic Structured Output**

**Purpose:**

* Strict type enforcement and validation using **Pydantic models**.

**Code Demo with ChatPromptTemplate + Chain:**

from pydantic import BaseModel

from langchain.chat\_models import ChatOpenAI

from langchain.prompts.chat import ChatPromptTemplate, HumanMessagePromptTemplate

from langchain.chains import LLMChain

from langchain.output\_parsers import PydanticOutputParser, WithStructuredOutputParser

# 1️⃣ Define Pydantic model

class Person(BaseModel):

name: str

age: int

city: str

# 2️⃣ LLM

llm = ChatOpenAI(model="gpt-4", temperature=0)

# 3️⃣ Pydantic parser + WithStructuredOutputParser

pydantic\_parser = PydanticOutputParser(pydantic\_object=Person)

structured\_parser = WithStructuredOutputParser(parser=pydantic\_parser)

# 4️⃣ ChatPromptTemplate

human\_prompt = HumanMessagePromptTemplate.from\_template("Return JSON for: {person\_description}")

chat\_prompt = ChatPromptTemplate.from\_messages([human\_prompt])

# 5️⃣ Chain

chain = LLMChain(llm=llm, prompt=chat\_prompt)

# 6️⃣ Run

raw\_output = chain.run(person\_description="Alice, 25, New York")

person\_obj = structured\_parser.parse(raw\_output)

print(person\_obj.name, person\_obj.age, person\_obj.city)

✅ Returns a **Pydantic model instance** with validation.

**4️⃣ TypedDict / StructuredOutputParser**

**Purpose:**

* Use **TypedDict** for **static type hints**.
* StructuredOutputParser ensures the fields exist.

**Code Demo with Chain:**

from typing import TypedDict

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain.output\_parsers import StructuredOutputParser, WithStructuredOutputParser

# 1️⃣ Define TypedDict

class PersonDict(TypedDict):

name: str

age: int

city: str

# 2️⃣ Schema for parser

schema = {"name": "str", "age": "int", "city": "str"}

parser = StructuredOutputParser.from\_dict(schema)

structured\_parser = WithStructuredOutputParser(parser=parser)

# 3️⃣ LLM

llm = ChatOpenAI(model="gpt-4", temperature=0)

# 4️⃣ Prompt

prompt\_template = PromptTemplate(

input\_variables=["person\_description"],

template="Return JSON for: {person\_description}"

)

# 5️⃣ Chain

chain = LLMChain(llm=llm, prompt=prompt\_template)

# 6️⃣ Run

raw\_output = chain.run(person\_description="Alice, 25, New York")

parsed\_dict: PersonDict = structured\_parser.parse(raw\_output)

print(parsed\_dict)

✅ Returns dictionary with type hints.

**5️⃣ JSON Schema**

**Purpose:**

* Define **formal JSON schema** for structured output.

**Code Demo with Chain:**

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain.output\_parsers import StructuredOutputParser, WithStructuredOutputParser

# 1️⃣ JSON Schema

json\_schema = {

"type": "object",

"properties": {

"name": {"type": "string"},

"age": {"type": "integer"},

"city": {"type": "string"}

},

"required": ["name", "age", "city"]

}

# 2️⃣ Parser

parser = StructuredOutputParser.from\_json\_schema(json\_schema)

structured\_parser = WithStructuredOutputParser(parser=parser)

# 3️⃣ LLM

llm = ChatOpenAI(model="gpt-4", temperature=0)

# 4️⃣ Prompt

prompt\_template = PromptTemplate(

input\_variables=["person\_description"],

template="Return JSON adhering to schema for: {person\_description}"

)

# 5️⃣ Chain

chain = LLMChain(llm=llm, prompt=prompt\_template)

# 6️⃣ Run

raw\_output = chain.run(person\_description="Alice, 25, New York")

parsed = structured\_parser.parse(raw\_output)

print(parsed)

✅ Returns a dictionary conforming to **JSON Schema**.

**6️⃣ Summary Table**

| **Method** | **Output Type** | **Validation** | **Prompt Integration** | **Best Use Case** |
| --- | --- | --- | --- | --- |
| with\_structured\_output\_json | dict | Minimal | ✅ PromptTemplate | Quick JSON parsing |
| with\_structured\_output\_pydantic | Pydantic model | Strong, strict | ✅ ChatPromptTemplate | Production pipelines |
| with\_structured\_output\_typed | TypedDict/dict | IDE/static type | ✅ PromptTemplate | Type hints, lightweight |
| JSON Schema | dict | Schema enforced | ✅ PromptTemplate | API integration, formal schema |
| WithStructuredOutputParser | Wraps parser | Depends on parser | ✅ Both | Auto parse + enforce structured output |

**✅ Key Takeaways**

1. **Always combine structured parser + prompt template** to guide LLM.
2. WithStructuredOutputParser **wraps any parser** and automatically parses LLM output.
3. **JSON parser** → simple, minimal validation.
4. **Pydantic parser** → strict type enforcement, production-ready.
5. **TypedDict / StructuredOutputParser** → lightweight, type hints.
6. **JSON Schema** → formal schema enforcement, API pipelines.

[**https://www.analyticsvidhya.com/blog/2024/11/output-parsers/**](https://www.analyticsvidhya.com/blog/2024/11/output-parsers/)

**Quick Recap**

Output parsing is nothing but, organizing the output of a Large Language Model(LLM) in a predefined structure or schema. The [output of LLM is generally plain text](https://medium.com/data-and-beyond/langchain-to-interact-with-llms-2fe2dec92e02) and does not follow any concrete schema. So, using output parsers, we can enforce some structure onto the output of LLM. Different types of output parsers can be a Pydantic object, a simple dictionary, a JSON object, etc.

This helps in ensuring the consistency of the responses of the model. This plays a crucial role in API development or database integration where a consistent data format is needed.

There are two kinds of LLMs. One that supports output parsing by default which can be implemented simply by with\_structured\_output(schema) method. [Read this article](https://medium.com/data-and-beyond/getting-structured-outputs-from-llms-2709b82f96f3)to understand the implementation properly. The second kind of LLM is those that do not support output parsing by default. So LangChain provides different Output Parser classes to implement this.

Press enter or click to view image in full size

**What are Output Parsers?**

Output Parser in LangChain is like a translator that takes the raw text generated by a language model and turns it into a format your program can actually work with — like a list, dictionary, or any custom structure. This is useful when working with tools, chains, or agents where the response needs to be parsed before further processing.

When a language model generates text, the output is just a string. But if your application expects structured data (e.g., a JSON object, dictionary, Pydantic model, etc.), you need to parse it using a proper parsing class of LangChain.

Let’s see different types of output parsers with their implementation in Python.

**Types of Output Parsers in LangChain**

**1. StringOutputParser**

This is the most basic parser, used to extract clean string output from the LLM’s raw response.

**Use case:** When you only want the final answer text (without any extra formatting or metadata).

import os  
from langchain\_huggingface import ChatHuggingFace, HuggingFacePipeline  
from dotenv import load\_dotenv  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_community.llms import Ollama  
import warnings  
warnings.filterwarnings("ignore")  
  
load\_dotenv()  
  
llm = HuggingFacePipeline.from\_model\_id(  
 model\_id='google/gemma-2-2b-it',  
 task='text-generation',  
 pipeline\_kwargs=dict(  
 temperature=0.5  
 )  
)  
model = ChatHuggingFace(llm=llm)  
  
# 1st prompt -> detailed report  
template1 = PromptTemplate(  
 template='Write a detailed report on {topic}',  
 input\_variables=['topic']  
)  
  
# 2nd prompt -> summary  
template2 = PromptTemplate(  
 template='Write a 5 line summary on the following text. /n {text}',  
 input\_variables=['text']  
)  
  
#without str output parser  
prompt1 = template1.invoke({'topic':'black hole'})  
result = model.invoke(prompt1)  
print(result)  
prompt2 = template2.invoke({'text':result})  
result2 = model.invoke(prompt2)  
print("="\*200)  
print(result2)  
  
# with str output parser we can use the chain as well  
parser = StrOutputParser()  
  
chain = template1 | model | parser | template2 | model | parser  
  
result = chain.invoke({'topic':'black hole'})  
  
print(result)

With the help of a parser, we can parse a string to produce proper output, like removing metadata and all, and pass it to the next template.

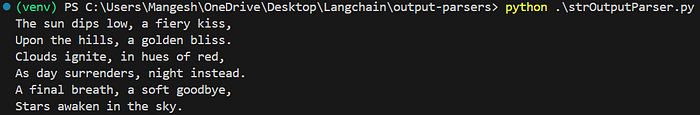
Without a parser, we need to create two separate chains.

**Str Output Parser**

[StrOutputParse](https://python.langchain.com/api_reference/core/output_parsers/langchain_core.output_parsers.string.StrOutputParser.html)is the most basic output parser in LangChain. It takes the raw output from the language model and returns it as a plain string, without making any changes or trying to structure it. The example below shows the implementation of this parser.

from langchain\_google\_genai import GoogleGenerativeAI  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import StrOutputParser  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
llm = GoogleGenerativeAI(model="gemini-1.5-pro")  
parser = StrOutputParser()  
  
prompt = PromptTemplate(  
 template="Write a short, creative poem about {topic}.",  
 input\_variables=["topic"]  
)  
  
chain = prompt | llm | parser  
  
result = chain.invoke({"topic": "sunset"})  
  
print(result)

Press enter or click to view image in full size



Output of strOutputParser

It returns just plain text. If we need specific pieces of information (like a name, age, or status), we would have to manually extract them. This problem can be solved using JsonOuputParser.

**2. JsonOutputParser**

Use this when you expect the LLM to return a well-formed JSON object.

from langchain\_huggingface import ChatHuggingFace, HuggingFaceEndpoint, HuggingFacePipeline  
from dotenv import load\_dotenv  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import JsonOutputParser  
import warnings  
warnings.filterwarnings("ignore")  
  
load\_dotenv()  
  
# Define the model  
llm = HuggingFacePipeline.from\_model\_id(  
 model\_id='google/gemma-2-2b-it',  
 task='text-generation'  
)  
  
model = ChatHuggingFace(llm=llm)  
  
parser = JsonOutputParser()  
  
template = PromptTemplate(  
 template='Give me 5 facts about {topic} \n {format\_instruction}',  
 input\_variables=['topic'],  
 partial\_variables={'format\_instruction': parser.get\_format\_instructions()}  
)  
  
# without chain  
# prompt = template.invoke({'topic': 'black hole'})  
# result = model.invoke(prompt)  
# final\_result = parser.parse(result.content)  
# print(final\_result)  
  
# with chain  
chain = template | model | parser  
  
result = chain.invoke({'topic':'black hole'})  
# if don't have any input variable then we have to pass empty dictionary chain.invoke({})  
  
print(result)

Here we call this format\_instruction a partial\_variables because it is getting filled before runtime with the help of the parser.get\_format\_instructions

Limitation : With jsonoutput parser, we can get JSON output, but we can’t enforce any schema For this, we can use structured output parser

**JSON Output Parser**

[The JsonOutputParser](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/types/json/) helps in formatting the plain text output of LLM into a JSON object. Consider an example, where the output of LLM is plain text with details about a book. But you are only interested in details like author name, publishing year, genre, and publisher. And you want this to be in a proper format like JSON. This is implemented below.

from langchain\_google\_genai import GoogleGenerativeAI  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import JsonOutputParser  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
llm = GoogleGenerativeAI(model="gemini-1.5-pro")  
parser = JsonOutputParser()  
  
  
prompt = PromptTemplate(  
 template="""Give me book name, author name, publisher name, genre   
 and its publishing year of the book\n\n{book\_name} \n{format\_instructions}""",  
  
 input\_variables=["book\_name"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()}  
)  
  
chain = prompt | llm | parser  
  
result = chain.invoke({"book\_name": "Wings of Fire"})  
  
print(type(result))  
print(result)

Press enter or click to view image in full size



Output of JsonOutputparser

By using JsonOutputParser, we can ask for the details we want to be included in JSON, but we cannot enforce certain schema over the LLM. The LLM might forget a field, change a key, or return the wrong data type.

**Example: Using JsonOutputParser with Pydantic**

Here’s an example of how to combine JsonOutputParser with [Pydantic](https://www.analyticsvidhya.com/blog/2024/10/structuring-inputs-and-outputs-in-multi-agent-systems/) to parse a language model’s output into a structured format.

Example 2 –

from langchain\_core.output\_parsers import JsonOutputParser

from langchain\_core.prompts import PromptTemplate

from langchain\_openai import ChatOpenAI

from pydantic import BaseModel, Field

# Initialize the model

model = ChatOpenAI(temperature=0)

# Define the desired data structure

class MovieQuote(BaseModel):

character: str = Field(description="The character who said the quote")

quote: str = Field(description="The quote itself")

# Create the parser with the schema

parser = JsonOutputParser(pydantic\_object=MovieQuote)

# Define a query

quote\_query = "Give me a famous movie quote with the character name."

# Set up the prompt with formatting instructions

prompt = PromptTemplate(

template="Answer the user query.\n{format\_instructions}\n{query}\n",

input\_variables=["query"],

partial\_variables={"format\_instructions": parser.get\_format\_instructions()},

)

# Combine the prompt, model, and parser

chain = prompt | model | parser

# Invoke the chai

response = chain.invoke({"query": quote\_query})

print(response)

**Using JsonOutputParser Without Pydantic**

For scenarios where strict schema validation is not necessary, you can use JsonOutputParser without defining a Pydantic schema. This approach is simpler but offers less control over the output structure.

Example:

# Define a simple query

quote\_query = "Tell me a fun fact about movies."

# Initialize the parser without a Pydantic object

parser = JsonOutputParser()

prompt = PromptTemplate(

template="Answer the user query.\n{format\_instructions}\n{query}\n",

input\_variables=["query"],

partial\_variables={"format\_instructions": parser.get\_format\_instructions()},

)

chain = prompt | model | parser

response = chain.invoke({"query": quote\_query})

print(response)

**3. StructuredOutputParser**

This is a an output parser in LangChain that helps extract structured JSON data from LLM response based on predefined field schema.

It works by defining a list of fields (ResponseSchema) that the model should return, ensuring the output follows a structured format.

from langchain\_huggingface import ChatHuggingFace, HuggingFaceEndpoint,HuggingFacePipeline  
from dotenv import load\_dotenv  
from langchain\_core.prompts import PromptTemplate  
from langchain.output\_parsers import StructuredOutputParser, ResponseSchema  
from langchain\_community.llms import Ollama  
load\_dotenv()  
  
model = Ollama(model="mistral")  
  
schema = [  
 ResponseSchema(name='fact\_1', description='Fact 1 about the topic'),  
 ResponseSchema(name='fact\_2', description='Fact 2 about the topic'),  
 ResponseSchema(name='fact\_3', description='Fact 3 about the topic'),  
]  
  
parser = StructuredOutputParser.from\_response\_schemas(schema)  
  
format\_instructions = parser.get\_format\_instructions()  
  
# Prompt Template  
template = PromptTemplate(  
 template="""  
You are an expert science writer. Provide exactly 3 facts about the topic below.  
  
Topic: {topic}  
  
Respond ONLY in the following JSON format:  
{format\_instruction}  
""",  
 input\_variables=['topic'],  
 partial\_variables={'format\_instruction': format\_instructions}  
)  
  
# without chain  
# prompt = template.invoke({'topic': 'black hole'})  
# result = model.invoke(prompt)  
# final\_result = parser.parse(result.content)  
# print(final\_result)  
  
  
# # using chains  
chain = template | model | parser  
  
result = chain.invoke({'topic':'black hole'})  
  
print(result)

Limitation: is that we can’t provide data validation

To resolve this, we have a Pydantic output parser

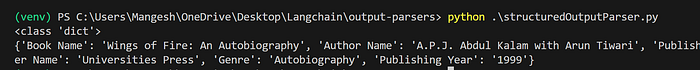
This can be solved by using yet another kind of parser class — StructuredOutputParser.

**Structured Output Parser**

[StructuredOutputParser](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/types/structured/)helps you enforce schemas, guide the LLM, and safely extract structured data. It helps in defining a schema, that LLM has to follow under any circumstances. The same above example can be coded using StructuredOutputParser as below —

from langchain\_google\_genai import GoogleGenerativeAI  
from langchain\_core.prompts import PromptTemplate  
from langchain.output\_parsers import StructuredOutputParser, ResponseSchema  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
llm = GoogleGenerativeAI(model="gemini-1.5-pro")  
  
schema = [  
 ResponseSchema(name="Book Name", description="Name of the book."),  
 ResponseSchema(name="Author Name", description="Author of the book."),  
 ResponseSchema(name="Publisher Name", description="Who is the publisher of this book?"),  
 ResponseSchema(name="Genre", description="What is the genre of this book?"),  
 ResponseSchema(name="Publishing Year", description="Which year was this book published?")  
]  
  
parser = StructuredOutputParser.from\_response\_schemas(schema)  
  
prompt = PromptTemplate(  
 template="""Give me book name, author name, publisher name, genre   
 and its publishing year of the book\n\n{book\_name} \n{format\_instructions}""",  
  
 input\_variables=["book\_name"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()}  
)  
  
chain = prompt | llm | parser  
  
result = chain.invoke({"book\_name": "Wings of Fire"})  
  
print(type(result))  
print(result)

Press enter or click to view image in full size



Output of StructuredOutputParser

**4. PydanticOutputParser**

**PydanticOutputParser** is a **structured output parser** in LangChain that uses **Pydantic models** to enforce **schema validation** when processing LLM responses.

This is the most robust and production-grade option. It uses **Pydantic models** to define schemas and performs real **data validation**.

**🧪 Why Use**PydanticOutputParser**?**

✅ **Strict Schema Enforcement** — Ensures that LLM responses follow a well-defined structure.  
✅ **Type Safety** — Automatically converts LLM outputs into Python objects.  
✅ **Easy Validation** — Uses Pydantic’s built-in validation to catch incorrect or missing data.  
✅ **Seamless Integration** — Works well with other LangChain components.

from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import PydanticOutputParser  
from pydantic import BaseModel, Field  
from langchain\_community.llms import Ollama  
  
model = Ollama(model="mistral")  
  
class Person(BaseModel):  
  
 name: str = Field(description='Name of the person')  
 age: int = Field(gt=18, description='Age of the person')  
 city: str = Field(description='Name of the city the person belongs to')  
  
parser = PydanticOutputParser(pydantic\_object=Person)  
  
template = PromptTemplate(  
 template='Generate the name, age and city of a fictional {place} person \n {format\_instruction}',  
 input\_variables=['place'],  
 partial\_variables={'format\_instruction':parser.get\_format\_instructions()}  
)  
  
# without chain  
prompt = template.invoke({'place': 'Vietnam'})  
result = model.invoke(prompt)  
final\_result = parser.parse(result)  
print(final\_result)  
  
  
# using chain  
# chain = template | model | parser  
  
# final\_result = chain.invoke({'place':'sri lankan'})  
  
# print(final\_result)

This helps your LLM understand **exactly what format** is expected — reducing hallucinations and improving reliability.

This type of parser also faces some kind of limitations. It cannot perform any kind of validation. Many times, we will have to perform validations like data types, values, etc. This can be achieved using PydanticOutputParser.

**Pydantic Output Parser**

[PydanticOutputParser](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/types/pydantic/) parses the output of LLM into a Pydantic model. Pydantic is a library for data validation and settings management in Python.

It allows you to define data models with built-in validation, type checking, and error handling. When you combine it with LangChain, the PydanticOutputParser can convert the raw output from an LLM into well-validated and structured Python objects based on the schema you define using Pydantic. Example —

from langchain\_google\_genai import GoogleGenerativeAI  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import PydanticOutputParser  
from pydantic import BaseModel, Field  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
llm = GoogleGenerativeAI(model="gemini-1.5-pro")  
  
class Book(BaseModel):  
 name: str = Field(description="The name of the book")  
 author: str = Field(description="The author of the book")  
 publisher: str = Field(description="The publisher of the book")  
 genre: str = Field(description="The genre of the book")  
 publishing\_year: int = Field(description="The publishing year of the book")  
  
parser = PydanticOutputParser(pydantic\_object=Book)  
  
prompt = PromptTemplate(  
 template="""Give me book name, author name, publisher name, genre   
 and its publishing year of the book\n\n{book\_name}\n\n{format\_instructions}""",  
 input\_variables=["book\_name"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()}  
)  
  
# Chain prompt -> LLM -> parser  
chain = prompt | llm | parser  
  
result = chain.invoke({"book\_name": "Wings of Fire"})  
  
print(type(result))  
print(result)

Press enter or click to view image in full size



Output of PydanticOutputParser

**Other Parsers**

Other parsers include the ones which are rarely used. A few of them are discussed below:

Example 2 –

import os

from langchain\_core.output\_parsers import PydanticOutputParser

from langchain\_core.prompts import PromptTemplate

from langchain\_openai import ChatOpenAI # Use ChatOpenAI for GPT-4

from pydantic import BaseModel, Field, model\_validator

# Ensure OpenAI API key is set correctly

os.environ['OPENAI\_API\_KEY'] = userdata.get('OPENAI\_API\_KEY') # Set key from userdata

openai\_key = os.getenv('OPENAI\_API\_KEY')

# Define the model using ChatOpenAI

model = ChatOpenAI(api\_key=openai\_key, model\_name="gpt-4o", temperature=0.0)

# Define a Pydantic model for the output

class Joke(BaseModel):

setup: str = Field(description="The question in the joke")

punchline: str = Field(description="The answer in the joke")

@model\_validator(mode="before")

@classmethod

def validate\_setup(cls, values: dict) -> dict:

setup = values.get("setup")

if setup and setup[-1] != "?":

raise ValueError("Setup must end with a question mark!"

return values

# Create the parser

parser = PydanticOutputParser(pydantic\_object=Joke)

# Define the prompt template

prompt = PromptTemplate(

template="Answer the user query.\n{format\_instructions}\n{query}\n",

input\_variables=["query"],

partial\_variables={"format\_instructions": parser.get\_format\_instructions()},

)

# Combine the model and parser

prompt\_and\_model = prompt | model

# Generate output

output = prompt\_and\_model.invoke({"query": "Tell me a joke."})

# Parse the output

parsed\_output = parser.invoke(output)

# Print the parsed output

print(parsed\_output)

**CSV Parser**

[CommaSeparatedListOutputParser](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/types/csv/) allows us to parse the output of LLM into a list separated by commas. This is generally used when dealing with pandas dataframe or series. The same use case of book info can be implemented as follows —

from langchain\_google\_genai import GoogleGenerativeAI  
from langchain\_core.prompts import PromptTemplate  
from langchain\_core.output\_parsers import CommaSeparatedListOutputParser  
  
from dotenv import load\_dotenv  
load\_dotenv()  
  
llm = GoogleGenerativeAI(model="gemini-1.5-pro")  
parser = CommaSeparatedListOutputParser()  
  
prompt = PromptTemplate(  
 template="""Give me book name, author name, publisher name, genre   
 and its publishing year of the book\n\n{book\_name}\n\n{format\_instructions}""",  
 input\_variables=["book\_name"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()}  
)  
  
# Chain prompt -> LLM -> parser  
chain = prompt | llm | parser  
  
result = chain.invoke({"book\_name": "Wings of Fire"})  
  
print(type(result))  
print(result)

Press enter or click to view image in full size



Output of CSVParser

**DateTime Parser**

[This type of parser converts](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/types/datetime/)the output of LLM into a date-time format. This becomes clear with the example below —

from langchain\_google\_genai import GoogleGenerativeAI  
from langchain\_core.prompts import PromptTemplate  
from langchain.output\_parsers import DatetimeOutputParser  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
llm = GoogleGenerativeAI(model="gemini-1.5-pro")  
parser = DatetimeOutputParser()  
  
prompt = PromptTemplate(  
 template="When is {festival} celebrated? \n {format\_instructions}",  
 input\_variables=["festival"],  
 partial\_variables={"format\_instructions": parser.get\_format\_instructions()}  
)  
  
chain = prompt | llm | parser  
  
result = chain.invoke({"festival": "Diwali"})  
  
print(type(result))  
print(result)

Press enter or click to view image in full size



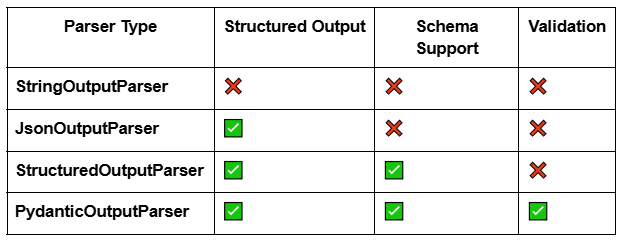
Output of DateTime Parser

Similarly, we can parse the output of the model into different formats such as Pandas DataFrame, Enum, etc. You can [refer to this documentation](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/) to understand more about these other output parsers.

**Summary**

Here’s how the four parsers compare:

Press enter or click to view image in full size



**📌 Conclusion**

Output parsers are the **glue** between your LLM’s raw output and the real-world systems you’re connecting to — APIs, UIs, data pipelines, and more.

Choosing the right parser ensures that your AI apps are **modular, reliable, and production-ready**.