

Structured Output

- The output from an LLM is **unstructured (Text)**
- **Structured output** is output in **well-defined data format** (eg. JSON)

```
{  
  "name": "John Doe",  
  "email": "john@example.com"  
}
```

- LLMs can have a conversation with other systems with the help of structured output



`with_structured_output` → Use this to generate structured output

LLMs who cannot generate structured output → Use **Output parsers**

✨ 1. Structured Output

3 Ways:

1. Typed Dictionary
2. Pydantic
3. JSON Schema

1. Typed Dictionary

- You define the key, values and their type
 - Name: str

- Age: int



It does not give you any error if you pass `str` instead of `int`.

- This is just a way to define a dictionary

```
from typing import TypedDict
```

```
class Person(TypedDict):
```

```
    name: str
```

```
    age: int
```

- You define a dictionary

Problem Statement:

- We have reviews
- We want columns: **summary & sentiments**

```
from langchain_groq import ChatGroq
```

```
from dotenv import load_dotenv
```

```
from typing import TypedDict
```

```
load_dotenv()
```

```
model= ChatGroq(model="gemma2-9b-it")
```

Create a class:

```
# Create a class
```

```
class Review (TypedDict):
    summary: str
    sentiment: str
```

- The class inherits **TypedDict**

Pass the dictionary into the model:

```
structured_model = model.with_structured_output(Review)
```

Invoke the model:

```
structured_model.invoke("""
    I recently upgraded to the Samsung Galaxy S24 Ultra, and I must s

The S-Pen integration is a great touch for note-taking and quick sketches, thoug

However, the weight and size make it a bit uncomfortable for one-handed use. Al

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
""")
```

```
{'sentiment': 'mixed',
 'summary': 'The Samsung Galaxy S24 Ultra is a powerful phone with a great camera, but it's also heavy, expensive, and has bloatware.'}
```

You can add a rating column as well:

```
# Create a class
```

```
class Review (TypedDict):
    summary: str
```

```
sentiment: str
rating: int
```

```
structured_model = model.with_structured_output(Review)
```

```
structured_model.invoke("""
Good mobile but facing heating issue 😞😞
""")
```

```
{'rating': 3,
 'sentiment': 'negative',
 'summary': 'Good mobile but facing heating issue'}
```



This predicts the exact ratings.

Annotation, Literal, Optional:

```
from typing import Annotated, Literal, Optional
```

- The model could not know what to do, in this case, we give it specific instructions.

```
from typing import Annotated
```

```
class Review (TypedDict):
```

```
    key_themes: Annotated[list[str], "Write down all the key themes discussed in the review"]
    summary: Annotated[str, "A brief summary of the review"]
    sentiment: Annotated[Literal["positive", "negative", "mixed"], "Return sentiment"]
    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"]
```

```
rating: int
```

```
structured_model = model.with_structured_output(Review)
```

```
structured_model.invoke("""
```

```
Very ok quality product. Very tasty and crispy. Good packing. Please keep it up.  
""
```

```
)
```

```
{'cons': [],  
 'pros': ['Very ok quality product', 'Very tasty and crispy', 'Good packing'],  
 'rating': 5,  
 'sentiment': 'positive',  
 'summary': "A positive review praising the product's quality, taste, and packaging."}
```

`Literal["positive", "negative", "mixed"]` : You can specify the values for a specific key

`Optional [list[str]]` → You can set an optional key

Pass Multiple reviews:

```
from typing import Annotated, Literal, Optional
```

```
class Review (TypedDict):
```

```
    key_themes: Annotated[list[str], "Write down all the key themes discussed i  
n the review in a list"]
```

```
    summary: Annotated[str, "A brief summary of the review"]
```

```
    sentiment: Annotated[Literal["positive", "negative", "mixed"], "Return sentim  
ent 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"]
```

```
    rating: int
```

```
Reviews = ["Completely waste , fully disappointed about quality , strongly non  
refer , please don't purchase anyone , please visit local offline stores and pur
```

```
chase , 1299 and 4 days waiting time all waste", "These pistachios are of top  
quality—crunchy, fresh, and packed with flavor. Perfect as a healthy snack or  
to add to desserts and dishes. The packaging keeps them fresh for longer, an  
d the taste is absolutely delightful. A great value for the price! Highly recomm  
end!"]
```

```
structured_model = model.with_structured_output(Review)
```

```
for r in Reviews:  
    data1= structured_model.invoke(r)  
    print(data1)
```

```
{'rating': 1, 'sentiment': 'negative', 'summary': 'The product is completely disapp  
{'cons': [], 'key_themes': ['quality', 'taste', 'freshness', 'versatility', 'value'
```



Con: You cannot do data validation




2. Pydantic

```
pip install pydantic
```

What is Pydantic?

Pydantic is a Python library that helps you **define data shapes** (schemas) using simple Python classes.

You tell Pydantic what kind of data you expect, and it will:

-  Validate the data
-  Enforce the types
-  Throw errors if the data is wrong

```
from pydantic import BaseModel

class Student(BaseModel):
    name: str

new_student= {'name': 'John'}

student1= Student(**new_student)

print(student1)
```

```
name= 'John '
```

- `**new_student` unpacks `{'name': 'John'}` into `name='John'` .

Why Use `*` ?

The `**` operator allows you to dynamically pass fields to a class or function when the values are already stored in a dictionary. It's especially useful when:

- You don't know the keys ahead of time.
- You're working with dynamic data, like parsing JSON.

If we replace John with a number → It will throw an error

```
from pydantic import BaseModel

class Student(BaseModel):
    name: str

new_student= {'name': 25}

student1= Student(**new_student)
```

```
print(student1)
```

```
260 )  
  
ValidationError: 1 validation error for Student  
name  
  Input should be a valid string [type=string_type, input_value=25, input_type=int]  
  For further information visit https://errors.pydantic.dev/2.11/v/string\_type
```

- It's expecting a string

Set default value:

```
from pydantic import BaseModel  
  
class Student(BaseModel):  
    name: str = 'John'  
  
new_student= {}  
  
student1= Student(**new_student)  
  
print(student1)
```

```
name= 'John '
```

You can fetch the object with `.name` as well:

```
student1.name  
✓ 0.0s  
'John '
```


Optional Fields:



If **no value** → `value=None`

```
from pydantic import BaseModel
from typing import Optional

class Student(BaseModel):
    name: str = 'John'
    age: Optional[int]=None

new_student= {}

student1= Student(**new_student)

print(student1)
```

```
name='John' age=None
```

```
new_student= {'age': 4}

student1= Student(**new_student)

print(student1)
```

```
name='John' age=4
```

Type Coercing:

Even if you pass number like `'45'` , Pydantic will convert it into Int

```
new_student= {'age': '45'}

student1= Student(**new_student)

print(student1)
```

```
name='John' age=45
```

Validate Emails

```
pip install pydantic[email]
```

```
from pydantic import BaseModel, EmailStr
```

EmailStr

- It's a built-in datatype inside Pydantic that validates email

```
from pydantic import BaseModel, EmailStr
from typing import Optional

class Student(BaseModel):
    name: str = 'John'
    age: Optional[int]=None
    email: EmailStr

new_student= {'age': 45, 'email': "abc@gmail.com"}

student1= Student(**new_student)
student1
```

```
Student(name='John', age=45, email='abc@gmail.com')
```

✗ Wrong email:

```
from pydantic import BaseModel, EmailStr
from typing import Optional
```

```
class Student(BaseModel):
    name: str = 'John'
    age: Optional[int]=None
    email: EmailStr
```

```
new_student= {'age': 45, 'email': "abc"}
```

```
student1= Student(**new_student)
student1
```

```
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

```
from pydantic import Field
```

Purpose	Example
Set default values	<code>Field(default = "India")</code>
Add description/help text	<code>Field(..., description = "User name")</code>
Set constraints	<code>Field(gt = 0)</code> (greater than 0)
Make things optional/required	<code>Field(default=None)</code> or <code>...</code>

Field Validations (Constraints)

Constraint	Meaning	Example
<code>gt=0</code>	Greater than 0	<code>Field(gt=0)</code>
<code>lt=100</code>	Less than 100	<code>Field(lt=100)</code>

Constraint	Meaning	Example
<code>ge=1</code>	Greater than or equal to 1	<code>Field(ge=1)</code>
<code>le=10</code>	Less than or equal to 10	<code>Field(le=10)</code>
<code>min_length=5</code>	Minimum length for a string	<code>Field(min_length=5)</code>
<code>max_length=50</code>	Maximum length for a string	<code>Field(max_length=50)</code>
<code>regex=...</code>	Pattern matching using Regex	<code>Field(regex="^[a-z]+\$")</code>

So basically:

`Field()` gives more control and meaning to your fields.

```
from pydantic import BaseModel, EmailStr, Field
from typing import Optional

class Student(BaseModel):

    name: str = 'john'
    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'}

student1 = Student(**new_student)
student1
```

```
Student(name='john', age=32, email='abc@gmail.com', cgpa=5)
```

- This is Pydantic object

Convert this into python dictionary:

```
student_dict = dict(student1)
student_dict
```

```
{'name': 'john', 'age': 32, 'email': 'abc@gmail.com', 'cgpa': 6.3}
```

Convert to JSON:

```
#Convert to JSON:
student1_json= student1.model_dump_json()
student1_json
```

```
'{"name":"john","age":32,"email":"abc@gmail.com","cgpa":6.3}'
```

💡 Pydantic + with_structured_output (Mostly Used)

```
from langchain_groq import ChatGroq
from dotenv import load_dotenv
from typing import TypedDict, Annotated, Optional, Literal
from pydantic import BaseModel, Field
```

- Class will inherit the `BaseModel` from Pydantic

```
class Review(BaseModel):
    key_themes : list[str] = Field(description= "Write down all the key themes discussed")
    summary: str = Field(description= "A brief summary of the review")
    sentiment: Literal['Positive', 'Negative', "Mixed"] = Field(description= "Return sentiment")
    pros: Optional[list[str]] = Field(default=None, description="Write down all the pros")
    cons: Optional[list[str]] = Field(default=None, description="Write down all the cons")
```

```
name: Optional[str] = Field(default=None, description="Write the name of the
```

```
#default=None required for Optional value
```

```
structured_model = model.with_structured_output(Review)
```

- name, pros, cons → Optional

Invoke:

```
result = structured_model.invoke("""I recently upgraded to the Samsung Galaxy S24 Ultra. The S-Pen integration is a great touch for note-taking and quick sketches, though the weight and size make it a bit uncomfortable for one-handed use. Overall, it's a fantastic upgrade for productivity and creativity."")
```

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 Randy Orton

```
""")
```

```
print(result)
```

Output:

```
key_themes=['Performance', 'Camera Quality', 'Battery Life', 'Size and Weight', 'Bloatware', 'Price']
```

```
summary='The Samsung Galaxy S24 Ultra is a powerful smartphone with a fantastic camera and long battery life, but its size, weight, bloatware, and high price are notable drawbacks.'
```

rice are drawbacks.'

sentiment='Mixed'

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']

cons=['The weight and size make it a bit uncomfortable for one-handed use.', '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.']

name='Randy Orton'

3. JSON Schema

- Used when project is made in multiple languages

JSON Schema is a way to **describe the structure and rules** for a JSON document.

It's like saying:

"This JSON must have these fields, and those fields must follow these rules."

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

```

    },
    "required":["name"] #Name of the student must be present
}

```

- **title** → Title of schema
- **description** → Optional
- **type** → datatype of schema
- **properties** → You write all the attributes in this
- **required** → These fields must be present.

Pydantic Schema → JSON Schema

Pydantic Schema:

```

class Review(BaseModel):
    key_themes : list[str] = Field(description= "Write down all the key themes discussed")
    summary: str = Field(description= "A brief summary of the review")
    sentiment: Literal['Positive', 'Negative', "Mixed"] = Field(description= "Return sentiment")
    pros: Optional[list[str]] = Field(default=None, description="Write down all the pros")
    cons: Optional[list[str]] = Field(default=None, description="Write down all the cons")
    name: Optional[str] = Field(default=None, description="Write the name of the reviewer")

```

JSON Schema:

```

json_schema = {
    "title": "Review",
    "type": "object",
    "properties": {
        "key_themes": {
            "type": "array", #Array = List
            "items": {
                "type": "string" # Type of items inside the list= 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"],    #"enum"= Literal
    "description": "Return sentiment of the review either negative, positive or neu"
  },
  "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"]
}

```

- `array` = List

```
{
  "key_themes": {
    "type": "array", #Array = List
    "items": {
      "type": "string" # Type of items inside the list= string
    }
  }
}
```

- **"type": "array" → key_themes is a list**
 - **"type": "string" → Type of items inside the list= string**
 - In short, **list[string]**
- **"description" → Optional**



"enum" = Literal

- **"type": ["array", "null"] → Null=Optional Field**

Remaining steps are exactly same:

```
structured_model = model.with_structured_output(json_schema)
```

```
result = structured_model.invoke("""I recently upgraded to the Samsung Galaxy S23 Ultra.

The S-Pen integration is a great touch for note-taking and quick sketches, though it's a bit bulky.

However, the weight and size make it a bit uncomfortable for one-handed use. Any tips for making the most of it?

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

Overall, it's a fantastic upgrade, but the size is a bit of a trade-off. I'm looking forward to exploring all the features it has to offer.

Thanks for your help!""")
```

```
structured_output = result['structured_output']
print(structured_output)
```

```
structured_output = result['structured_output']
print(structured_output)
```

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 Randy Orton

```
"""
```

```
print(result)
```

Output:

```
{'cons': ['The weight and size make it a bit uncomfortable for one-handed use.', '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.'],
```

```
'key_themes': ['Performance', 'Camera', 'Battery Life', 'S-Pen', 'Bloatware', 'Price'],
```

```
'name': 'Randy Orton',
```

```
'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'],
```

```
'sentiment': 'pos',
```

```
'summary': 'The Samsung Galaxy S24 Ultra is a powerful and feature-rich phone with a stunning camera and long battery life, but its size, bloatware, and high price are drawbacks.'}
```



The type you get is a **dictionary**.

JSON Schema Types You Can Use

Type	Description	Example
string	Text	"Alice"
number	Any number (int or float)	10.5
integer	Whole numbers	25
boolean	True or False	true
array	List of items	[1, 2, 3]
object	JSON object (dictionary)	{ "key": "value" }
null	Empty / null value	null

Convert Pydantic to JSON Schema

- Pydantic models automatically **generate JSON Schema**.

```
Class.model_json_schema()
```

Ex.

```
class Review(BaseModel):
    key_themes : list[str] = Field(description= "Write down all the key themes discussed in the review in a list")
    summary: str = Field(description= "A brief summary of the review")
    sentiment: Literal['Positive', 'Negative', "Mixed"] = Field(description= "Return sentiment of the review either negative, positive or Mixed")
    pros: Optional[list[str]] = Field(default=None, description="Write down all the pros inside a list")
    cons: Optional[list[str]] = Field(default=None, description="Write down all the cons inside a list")
    name: Optional[str] = Field(default=None, description="Write the name of the reviewer")
```

```
Review.model_json_schema()
```

Output:

```
{'properties': {'key_themes': {'description': 'Write down all the key themes discussed in the review in a list',  
  'items': {'type': 'string'},  
  'title': 'Key Themes',  
  'type': 'array'},  
  'summary': {'description': 'A brief summary of the review',  
    'title': 'Summary',  
    'type': 'string'},  
  'sentiment': {'description': 'Return sentiment of the review either negative, positive or Mixed',  
    'enum': ['Positive', 'Negative', 'Mixed'],  
    'title': 'Sentiment',  
    'type': 'string'},  
  'pros': {'anyOf': [{'items': {'type': 'string'}, 'type': 'array'}, {'type': 'null'}],  
    'default': None,  
    'description': 'Write down all the pros inside a list',  
    'title': 'Pros'},  
  'cons': {'anyOf': [{'items': {'type': 'string'}, 'type': 'array'}, {'type': 'null'}],  
    'default': None,  
    'description': 'Write down all the cons inside a list',  
    'title': 'Cons'},  
  'name': {'anyOf': [{'type': 'string'}, {'type': 'null'}],  
    'default': None,  
    'description': 'Write the name of the reviewer',  
    'title': 'Name'}}},  
  'required': ['key_themes', 'summary', 'sentiment'],  
  'title': 'Review',  
  'type': 'object'}
```

- **Type = Dictionary**

When to use what?

Use TypedDict if:



















- You only need type hints (basic structure enforcement).
- You don't need validation (e.g., checking 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 (Pydantic).
- You need validation but don't need Python objects.
- You want to define structure in a standard JSON format.

Feature	TypedDict 	Pydantic 	JSON Schema 
Basic structure			
Type enforcement			
Data validation			
Default values			
Automatic conversion			
Cross-language compatibility	