Structured Output in LangChain

A Comprehensive Guide to TypedDict, Pydantic, and JSON Schema Integration

Definition

Structured Output in **LangChain** refers to the practice of making language models return responses in a **well-defined data format** (e.g., JSON or typed objects) instead of free-form text. This ensures outputs are *machine-readable, consistent, and easy to integrate* with applications.

Why Do We Need Structured Output?

Relying solely on free-form text from LLMs introduces ambiguity and inconsistency. Structured outputs address this by enforcing predefined schemas.

Key Benefits

- **Reliability:** Guarantees predictable response formats.
- **Integration:** Simplifies connection with APIs, databases, and UIs.
- **Error Reduction:** Minimizes parsing and formatting errors.
- **Automation:** Enables fully automated pipelines and agents.

Example

Free-form:

```
The capital of France is Paris.
```

Structured Output (JSON):

```
{
  "country": "France",
  "capital": "Paris"
}
```

Applications of Structured Output

Data Extraction

LLMs can extract structured facts from unstructured text for analytics or automation.

Example

Input: "The iPhone 15 was released in 2023 and features the A17 chip."

Output:

```
{
  "product": "iPhone 15",
  "release_year": 2023,
  "chip": "A17"
}
```

API Building

Structured output allows LLMs to return JSON or Pydantic-like objects that can be directly used as API responses.

Agents

LangChain agents rely on structured outputs to pass parameters between tools:

```
{
  "action": "search_web",
  "query": "latest AI research papers"
}
```

Ways to Achieve Structured Output in LangChain

LangChain provides several strategies for defining and enforcing structured responses.

1. `with_structured_output()`

When the underlying LLM supports structured output (e.g., GPT-4, Gemini), you can attach a schema directly.

Example

```
from typing import TypedDict
from langchain_openai import ChatOpenAI

class PersonInfo(TypedDict):
    name: str
    age: int
    country: str

llm = ChatOpenAI(model="gpt-4o-mini")
structured_llm = llm.with_structured_output(PersonInfo)
response = structured_llm.invoke("John is a 25-year-old from Canada.")
print(response)
```

Output:

```
{'name': 'John', 'age': 25, 'country': 'Canada'}
```

2. Output Parsers

For models without structured-output support, LangChain provides parsers such as:

- `StructuredOutputParser`
- `PydanticOutputParser`
- `ResponseSchema`

3. Function Calling

Models can return structured arguments that map to function signatures, ideal for agents or tool use.

Advanced Structured Output Techniques

TypedDict: Lightweight Static Schemas

TypedDict (from `typing`) defines dictionary keys and types for static type checking.

Examples

Basic TypedDict:

```
class User(TypedDict):
    name: str
    age: int
```

Annotated TypedDict (optional fields):

```
class ExtendedUser(TypedDict, total=False):
    name: str
    age: int
    email: str
```

Using Literal:

```
from typing import Literal

class Task(TypedDict):
    status: Literal["pending", "done"]
```

Pros: Lightweight, IDE-friendly, no runtime overhead. **Cons:** No validation or coercion.

Pydantic: Runtime Data Validation

Pydantic enforces schema validation, automatic type conversion, and rich error reporting.

Example

```
from pydantic import BaseModel, Field

class Product(BaseModel):
    id: int = Field(..., description="Product ID")
    name: str
    price: float = Field(..., gt=0)
    category: str | None = None
```

Advantages:

- Type coercion (e.g., "22" → 22)
- Default and optional fields

- Built-in validation

Disadvantages:

- Slightly heavier runtime overhead
- Requires dependency on Pydantic

JSON Schema: Language-Agnostic Validation

Defines structure in pure JSON format, useful for multi-language interoperability.

When to Use What?

Use TypedDict if:

- You need only static type hints.
- No runtime validation is required.
- You trust the LLM's structure.

Use Pydantic if:

- Validation or coercion is required.
- You need default values or detailed error reporting.

Use JSON Schema if:

- You prefer a language-neutral schema.
- You want validation without Python dependencies.

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

LLMs Supporting `with_structured_output()`

Overview

Structured output support in LangChain varies across language models. Some models natively provide schema enforcement via **JSON mode** or **function calling**, while others require explicit parsing logic through LangChain's `OutputParser` utilities.

Support Classification

Model / Provider	Structured Output Support	Notes / Comments	Fallback Required
OpenAI (GPT-4, GPT-4o)	Supported (`JSON mode`, Function Calling)	Fully supports structured outputs via schema or TypedDict definitions. Widely used reference implementation.	No
Anthropic (Claude 3)	Supported	Supports structured responses using tool-calling / schema format. Works seamlessly with LangChain.	No
Google Gemini / Vertex AI	Partially Supported	Newer versions add TypedDict / schema support. Older Gemini API endpoints may fail with structured calls.	Partial
Groq (LLaMA-3 via Groq API)	Conditionally Supported	Works in many cases using `with_structured_output()` and TypedDicts, but may fail intermittently depending on schema complexity.	Partial
Fireworks	Supported	Dedicated structured output interface added in latest LangChain releases.	No
Deepseek	Supported (JSON Mode)	Official JSON mode added; can return validated schema outputs.	No
Ollama	Supported (Updated Defaults)	Structured output defaults were recently improved; compatible with TypedDict and Pydantic schemas.	No
Other / Custom LLMs	Not Supported	Most community or research models lack structured API support. Requires manual parsing.	Yes (`OutputParser`)

Practical Guidelines

- Always verify structured output support by running a small test with `model.with_structured_outp`
- If the model fails or returns free-form text, fall back to:
  - `PydanticOutputParser`
  - `StructuredOutputParser`
  - or a `RetryOutputParser` for robustness.
- For consistent results, prefer LLMs with native JSON / function-calling capabilities.
- Regularly check the LangChain Changelog for updated model support.

Summary

Rule of Thumb:

- Use `with_structured_output()` for OpenAI, Claude, Fireworks, Deepseek, Gemini, and Ollama.
- Use `OutputParser` when working with custom, open-source, or unsupported models.

JSON Mode vs Function Calling in LangChain

LangChain supports structured outputs via two primary mechanisms, depending on the LLM's capabilities: **JSON Mode** and **Function Calling**.

1. JSON Mode

Definition: JSON Mode enforces that the LLM generates strict JSON text that adheres to a predefined schema.

Key Features:

- The model's output is forced to be valid JSON.
- Works seamlessly with `with_structured_output()` when the LLM natively supports JSON.
- Reduces parsing errors caused by free-form text.
- Example supported models: OpenAI GPT-4-turbo, Claude 3, Gemini 1.5.

Example:

```
{
  "summary": "Excellent product, highly recommended.",
  "sentiment": "positive",
  "key_themes": ["performance", "battery", "camera"]
}
```

2. Function Calling

Definition: Function Calling is a tool-like mechanism where the LLM returns structured arguments that "call" a predefined schema or function.

Key Features:

- The LLM outputs parameters matching a function or Pydantic model schema.
- Supported by models that do not enforce strict JSON formatting but understand tool/schema calls.
- LangChain converts these outputs automatically into Python objects.
- Example supported models: Groq (openai/gpt-oss-20b), OpenAI GPT-4, Claude.

Example:

```
{
  "function_call": "Review",
  "arguments": {
    "summary": "Excellent product, highly recommended.",
    "sentiment": "positive",
    "key_themes": ["performance", "battery", "camera"]
  }
}
```

Guidelines for Choosing Between JSON Mode and Function Calling

- **Use JSON Mode** if the model natively supports strict JSON formatting — ensures minimal parsing errors.
- **Use Function Calling** if the model supports structured arguments but not enforced JSON — works well with Groq, some OpenAI, and other tool-friendly LLMs.
- **Fallback:** For models that do not support either mechanism, use `OutputParser` to convert free-form text into structured data.

Mechanism	**Explicit Choice?**	**LLMs Supporting**	**How LangChain Uses It**
JSON Mode	Auto based on LLM	GPT-4-turbo, Claude 3, Gemini	`with_structured_output(schema)` forces JSON if supported
Function Calling	Auto based on LLM	Groq, OpenAI, Claude	`with_structured_output(schema)` uses function-call internally
OutputParser fallback	Yes	Any LLM	Manually parse free-form text into schema

Summary

In summary:

- Use **TypedDict** for lightweight schema definitions.
- Use **Pydantic** for robust runtime validation and data safety.
- Use **JSON Schema** for cross-language or platform-agnostic validation.

LangChain's `with_structured_output()` acts as a bridge between LLMs and these schema tools, enabling predictable, type-safe responses ideal for API design, data extraction, and agent workflows.