1) What is "Structured Output" (LangChain context)

- **Definition:** Getting an LLM to return results in a **well-defined data shape** (e.g., JSON that matches a schema) instead of free-form text.
- **Why it matters:** Structured results are easy to **parse, validate, and use in code** (store in DBs, feed into UIs, call downstream tools).

Quick contrast

- **Unstructured (plain text):** "Morning: Eiffel Tower. Afternoon: Louvre. Evening: Seine..."
- **Structured (JSON):**

```json
[
  {"time": "morning", "activity": "Visit the Eiffel Tower"},
  {"time": "afternoon", "activity": "Walk through the Louvre Museum"},
  {"time": "evening", "activity": "Dinner by the Seine"}
]
```

2) Why we need it (common use cases)

- **Data extraction:** Pull entities, attributes, and relationships from documents.
- **API building:** Turn user prompts into **typed request objects** you can post to APIs.
- **Agents / tool use:** Force valid arguments for tool/function calls (e.g., "book_flight({...})").

3) Two ways to get structured output

1. **Native structured output** (preferred)
    - Tell the model the exact schema and have it **emit JSON that conforms**.
    - In LangChain this is the with_structured_output(...) method on chat models.
2. **Output parsers** (fallback)
    - Parse plain text using regex, JSON detection + repair, or custom logic.
    - Works with any model but is **more brittle**.

Rule of thumb: **Use native structured output first**; keep a lightweight parser as a safety net.

4) with_structured_output: what it returns & schema options

You supply a **schema**, and the model is wrapped to return **validated Python objects** (not just strings).

You can provide the schema as:

- **TypedDict** (Python typing; structure hints, no runtime validation by itself)
- **pydantic BaseModel** (adds **runtime validation**, defaults, coercion)
- **JSON Schema** (plain dict schema; validation depends on the wrapper/your code)

Conceptually:

chat_model -> with_structured_output(schema) -> structured, validated Python object

Two underlying mechanisms (models differ)

- **JSON mode:** The LLM is forced to output strict JSON only.
- **Function/tool calling:** The LLM "calls" a function with **arguments** matching your schema.

LangChain usually picks the right mechanism automatically for the model you're using; many wrappers allow overriding (e.g., method="json_mode" or method="function_calling") and enabling strictness. If you change providers, keep this in mind.

TypedDict—define the shape (type hints only)

What it is: A way to declare **required/optional keys and value types** for dictionaries. Improves IDE help and static checks; doesn't validate at runtime by itself.

Minimal example (itinerary item)

```python
from typing import TypedDict, Literal, List, Optional

class ItineraryItem(TypedDict):
    time: Literal["morning", "afternoon", "evening"]   # narrow values
    activity: str
    notes: Optional[str]

class ParisPlan(TypedDict):
    city: Literal["Paris"]
    day_plan: List[ItineraryItem]
```

Using with LangChain (Python, sketch)

```python
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o-mini")  # example model id
structured_llm = llm.with_structured_output(ParisPlan)  # returns Python dicts

result = structured_llm.invoke("Plan a one-day itinerary for Paris.")
# result: {"city": "Paris", "day_plan": [...]}
```

Pros: Simple, no third-party dependency, great dev ergonomics.
Cons: No runtime validation; you rely on the model to behave.

6) Pydantic—validation & parsing

What it is: A **data parsing/validation** library. Ensures data is **correct, structured, and type-safe** at runtime. Can **coerce** types and **fill defaults**.

Example with constraints & defaults

```python
from pydantic import BaseModel, Field, conint
from typing import List, Literal, Optional

class ItineraryItem(BaseModel):
    time: Literal["morning", "afternoon", "evening"]
    activity: str = Field(min_length=3)
    duration_hours: Optional[conint(ge=0, le=12)] = 2  # default, 0..12
    notes: Optional[str] = None

class ParisPlan(BaseModel):
    city: Literal["Paris"] = "Paris"
    day_plan: List[ItineraryItem]

# LangChain
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o-mini")
structured_llm = llm.with_structured_output(ParisPlan)

plan = structured_llm.invoke("Plan a one-day itinerary for Paris.")
# plan is a ParisPlan object; invalid data raises a ValidationError
```

Pros: Real validation, defaults, coercion (“100” → 100).
Cons: Extra dependency; slightly more boilerplate.

---

7) When to use which (decision guide)

- ✅ **Use TypedDict if:**
    - You only need **type hints** and basic structure.
    - You **don’t need validation** (e.g., you trust the LLM and will eyeball results).
    - You want **zero extra deps**.
- ✅ **Use Pydantic if:**
    - You need **runtime validation** (e.g., enums like "positive" | "neutral" | "negative").
    - You want **defaults**, **type conversion**, and **clear errors**.
    - You may get **partial/missing fields** from the LLM and need safe handling.
- ✅ **Use JSON Schema if:**
    - You prefer a **language-neutral** schema (interop with non-Python systems).
    - You want validation but **don’t need Python objects**.
    - You dislike adding Pydantic but still want a formal spec.

JSON Schema example

python Copy Edi

```python
schema = {
  "title": "ParisPlan",
  "type": "object",
  "properties": {
    "city": {"const": "Paris"},
    "day_plan": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "time": {"enum": ["morning", "afternoon", "evening"]},
          "activity": {"type": "string"},
          "notes": {"type": ["string", "null"]}
        },
        "required": ["time", "activity"],
        "additionalProperties": False
      }
    }
  },
  "required": ["city", "day_plan"],
  "additionalProperties": False
}
# structured_llm = llm.with_structured_output(schema)  # many wrappers accept JSON Schema dicts
```

8) A few things to remember about with_structured_output

- **Mechanisms:** It uses **JSON-only mode** (Claude/Gemini/others) **or function/tool calling** (common in OpenAI). Same goal; different plumbing.
- **Strictness:** Prefer **strict schemas** (enums, required fields, min/max, regex). Models behave better with tighter specs.
- **Descriptions help:** Add field descriptions; models follow them.
- **Small, flat shapes win:** Deeply nested or very long schemas are harder for models.
- **Include examples in the prompt** when the schema is complex.
- **Fallback:** Wrap calls with a try/except; if validation fails, retry with a clarifying instruction or run a repair step.

End-to-end mini patterns

A) Strongly-typed extraction (Pydantic)

```python
from pydantic import BaseModel, Field
from typing import List, Literal

class ReviewInsight(BaseModel):
    aspect: Literal["price", "quality", "delivery", "support"]
    sentiment: Literal["positive", "neutral", "negative"]
    evidence: str = Field(description="Short quote from the text")

class Extraction(BaseModel):
    product: str
    insights: List[ReviewInsight]

prompt = """Extract insights from the reviews below.
Return only what fits the schema.
Reviews:
- "Great delivery, price was ok"
- "Support never replied, really bad experience"
"""

structured_llm = llm.with_structured_output(Extraction)
data = structured_llm.invoke(prompt)
```

B) API request builder (TypedDict)

```
from typing import TypedDict, Literal, List

class FlightQuery(TypedDict):
    origin: str
    destination: str
    date: str           # ISO date
    passengers: int
    cabin: Literal["economy", "premium_economy", "business", "first"]
    filters: List[str]

rq = llm.with_structured_output(FlightQuery).invoke(
  "Book a business class flight from JED to DXB next Friday for 2 adults, window seats."
)
# -> Post rq to your flight API
```

C) JSON Schema for cross-service interop

Send the schema (above) and the user prompt to any provider that supports JSON mode/tool calling, then validate with your favorite JSON Schema validator before using the data.

Common pitfalls & fixes

- **Model adds prose around JSON** → Enable JSON-only mode, or say: "Return **only** valid JSON. No prose."
- **Extra/unexpected keys** → Set additionalProperties: false (JSON Schema) or use strict Pydantic models.
- **Hallucinated enums** → Use **Literal/Enum** and add **descriptions + examples**.
- **Numbers as strings** → Let Pydantic coerce or post-process types.
- **Partial results** → Make some fields optional, fill with defaults, then re-ask for missing parts.