M0.2 — Python SDKs & Structured Output

From "it works 90% of the time" to a typed function your code can trust — schema enforcement is the line between a prototype and a production component.

Hiring signal: Structured output / Pydantic / JSON schema (req-026, Score 3) shows up at **High** frequency across harvested 2026 JDs. This module installs the first reliability tool of the curriculum.

The 2 AM Parser Crash

You shipped an AI feature. It calls the model, expects a clean object, and hands it downstream:

```
{"name": "Alice", "email": "alice@co.com"}
```

Instead, one call in ten returns a *paragraph of prose* — or JSON wrapped in markdown fences. Your downstream parser throws `JSONDecodeError`, and the pager goes off at 2 AM.

- You shipped a feature. It works **90%** of the time.
- The 10% failures are **not model errors** — they are format errors your code cannot parse.
- Structured output is the difference between a prototype and a production component.

Career anchor: req-026 (structured output / Pydantic / JSON schema) appears at **High** frequency in harvested 2026 JDs — schema fluency is a baseline expectation, not a specialty.

Why This Module → Career Anchor

Requirement	Score	JD frequency	M0.2 coverage
Structured output / Pydantic / JSON schema (req-026)	**3**	**High**	Direct — the homework artifact
LLM API integration / production deployment	2	28/30 JDs	Practiced in every lab
Python proficiency	1	30/30 JDs	Reproducible env + typed code

Production tip: Treat every LLM call that feeds downstream code as a typed function — a schema, a validator, and an explicit failure mode. Free-form output is only acceptable at the user-facing surface.

Archetype fit: AI Engineer — Generalist · Agentic Systems Engineer. M0.2 is the first module that produces a portfolio artifact: a validated `extract_contact` function.

Before We Begin — What M0.1 Gave You

This module assumes you finished **M0.1** and can do all four:

- **Tokens & context window** — you can explain what a token is and why context length drives cost.
- **Temperature & sampling** — you can set `temperature=0` and explain the reliability trade-off.
- **Statelessness** — you know every API call is independent; the `messages` array IS the memory.
- `finish_reason` / `stop_reason` — you can inspect why generation stopped and read `usage`.

Gotcha — Missing one of these? Do not proceed. Re-run the M0.1 worked example first; the labs here build directly on that wrapper. Planned remedial decks: `python/env-vars-secrets`, `python/http-json-basics` (status: planned, tracked in the backlog).

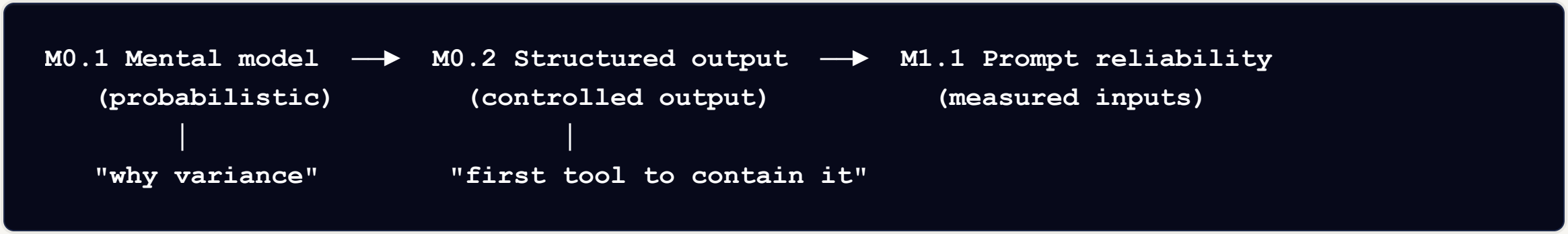
New Terms in This Module

Term	Definition
Structured output	An LLM response constrained to a specific schema (JSON, typed object) instead of free prose.
Tool schema	A JSON Schema definition passed to the model that constrains its response to the schema's shape.
Pydantic	A Python library for runtime type validation; defines typed models and raises on violations.
`instructor`	A library wrapping the client (OpenAI / Anthropic / litellm) to add Pydantic validation and auto-retry.
Client SDK vs Agent SDK	Client SDK = you own the request/response loop; Agent SDK = the framework owns the loop and calls your functions.

Pre-training principle: These five terms recur on every slide that follows. Meeting them now keeps them from becoming cognitive load mid-explanation later.

Bridge — Where M0.2 Fits

The curriculum arc: **Foundation → Retrieval → Orchestration → Production**



M0.1 explained *why* LLM output varies. M0.2 gives you the first **reliability tool**: forcing the model's output into a shape your code can trust.

Connecting concept: Schema enforcement means variance no longer flows into invalid JSON — it is caught at the validation boundary and either repaired or raised, never silently passed downstream.

Concept 1 — Reproducible Env & Secret Hygiene

The #1 reason learners stall is environment friction. Solve it once, correctly:

- **Isolated environment** — `uv` (fastest) or `venv`. Never install AI deps globally; `litellm` / `langchain` transitive conflicts are a constant source of pain.
- **Secrets in `.env`, never in code** — `.env` is gitignored (real keys); `.env.example` is committed (placeholders).
- **Pin dependencies** — `requirements.txt` or `pyproject.toml`. `litellm==1.30.2` today may break on `1.31.0` tomorrow.
- **The `llm.py` seam** — one file routes every model call, so you swap providers by changing one env var.

```
# .env.example — copy to .env and fill in real values. NEVER commit .env.
ANTHROPIC_API_KEY=sk-ant-REPLACE_ME
LLM_MODEL=claude-sonnet-4-6
```

Anti-pattern — hardcoding API keys in source: Committed keys are found by automated scanners within minutes and your billing reflects it. Use `.env` from day one.

Concept 1b — The First Authenticated Call

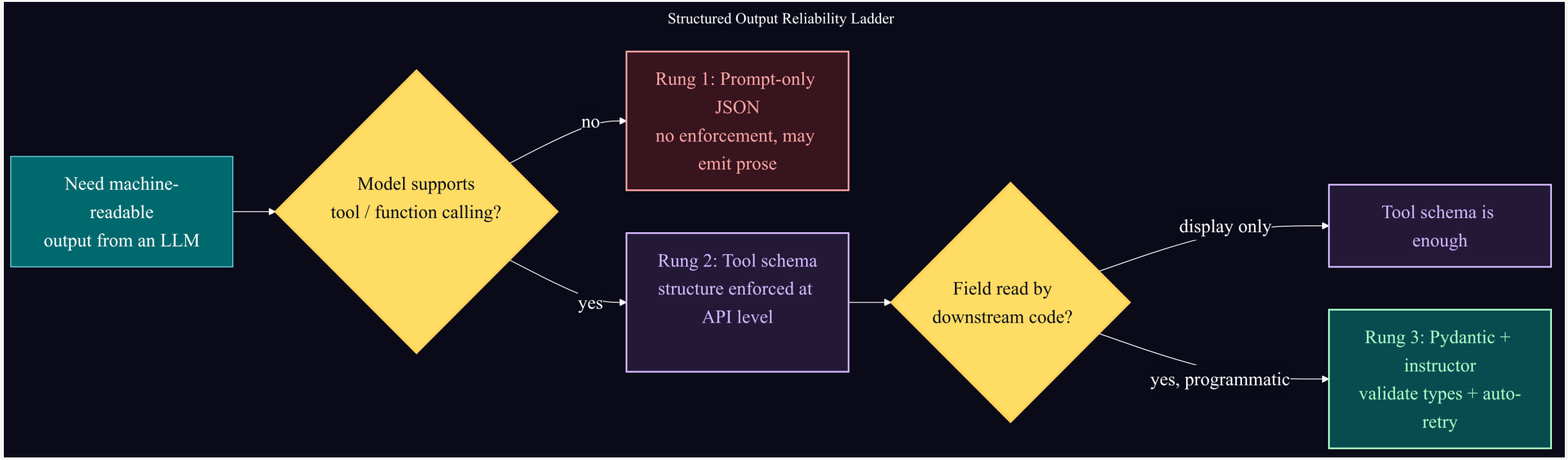
Lab 1's whole job: stand up the scaffold and make one authenticated call that **fails loudly** if the key is missing.

```python
import os
from dotenv import load_dotenv
load_dotenv()  # reads .env in the current directory

# Fail loudly if the key isn't set — better than a cryptic 401
api_key = os.getenv("ANTHROPIC_API_KEY") or os.getenv("OPENAI_API_KEY")
if not api_key:
    raise EnvironmentError("No API key found. Copy .env.example to .env.")

import litellm
model = os.getenv("LLM_MODEL", "claude-sonnet-4-6")
resp = litellm.completion(model=model, max_tokens=10, temperature=0,
    messages=[{"role": "user", "content": "Reply with exactly: READY"}])
print(resp.choices[0].message.content.strip())
```

Production tip: Failing loudly at startup on a missing key beats a buried HTTP 401 three frames deep in a stack trace.

Concept 2 — The Reliability Ladder

Three approaches to structured output, in increasing order of guarantee. Climb only as high as your task needs.



Production tip: Use Rung 3 (Pydantic + instructor) for any field your code reads programmatically.

Concept 2a — Rung 1: Prompt-Only JSON (Fragile)

Ask the model to "reply with only a JSON object." It usually works — and that *usually* is the problem.

```
r1 = complete([{"role": "user", "content":
    f"Extract name, email, order_id, issue. Reply with ONLY JSON.\n\n{CUSTOMER_TEXT}"}],
    temperature=0, max_tokens=256)
raw = r1.choices[0].message.content.strip()
if raw.startswith("```"):          # strip markdown fences the model added anyway
    raw = raw.split("```")[1]
    if raw.startswith("json"):
        raw = raw[4:]
data = json.loads(raw)             # may still raise JSONDecodeError
```

- No enforcement: the model can wrap JSON in prose or markdown, or emit invalid JSON.
- You write defensive parsing code — and still get surprised in production.

Gotcha: Every line of fence-stripping and try/except here is a symptom that the format guarantee lives in the wrong place — the prompt, not the API.

Concept 2b — Rung 2: Tool Schema (Reliable)

Define a typed function schema and force the model to "call" it. The response matches the schema's shape — you get a dict.

```
EXTRACT_TOOL = {"type": "function", "function": {
    "name": "extract_customer_request",
    "parameters": {"type": "object",
        "properties": {
            "customer_name":   {"type": "string"},
            "customer_email":  {"type": "string"},
            "refund_requested":{"type": "boolean"}},
        "required": ["customer_name", "customer_email", "refund_requested"]}}}
r2 = complete([{"role": "user", "content": CUSTOMER_TEXT}], tools=[EXTRACT_TOOL],
    tool_choice={"type": "function", "function": {"name": "extract_customer_request"}})
data = json.loads(r2.choices[0].message.tool_calls[0].function.arguments)
```

Concept: Tool schema guarantees JSON **syntax** and field **presence** — but not semantic validity. `customer_email: "not provided"` is syntactically valid and semantically wrong.

Concept 2c — Rung 3: Pydantic + instructor

Add the **semantic** layer: typed fields, validators, and automatic retry on violation.

```python
from pydantic import BaseModel, field_validator
import instructor, litellm

class CustomerRequest(BaseModel):
    customer_name: str
    customer_email: str
    refund_requested: bool

    @field_validator("customer_email")
    @classmethod
    def looks_like_email(cls, v: str) -> str:
        if "@" not in v:
            raise ValueError(f"'{v}' is not an email")
        return v.lower().strip()

client = instructor.from_litellm(litellm.completion)    # returns Pydantic objects
```

Production tip: A validator that rejects `"not provided"` turns a silent bad value into a retried — or loudly raised — failure.

Concept 3 — Client SDK vs Agent SDK

The distinction the rest of the course is built on — *who owns the loop*.

	Client SDK (litellm, openai, anthropic)	Agent SDK / framework (LangGraph, smolagents)
Who owns the loop	**You** write the `while` loop and tool dispatch	**Framework** runs perceive→reason→act→observe
Control vs code	Full control, maximum transparency, more code	Less code for common patterns, harder to debug
Good for	Single-turn calls, custom loops, learning	Multi-step loops, multi-agent, stateful workflows

Rule of thumb: loop logic simple and fully specified → Client SDK. Loop complex or needs persistence → framework.

Gotcha: Choosing a framework early locks you into its abstractions. Understand the client SDK first — every Milestone 0 and 1 lab uses litellm directly so you see every line.

Concept 3b — Same Task, Two Loops

```python
# Version A — Client SDK: you own every line of the loop
def answer_with_retry(task, max_retries=2):
    messages = [{"role": "user", "content": task}]
    for _ in range(max_retries):
        r = litellm.completion(model=MODEL, messages=messages,
                               temperature=0, max_tokens=512)
        content = r.choices[0].message.content.strip()
        if content:
            return content
        messages.append({"role": "user", "content": "Empty — answer again."})
    return "No answer after retries."

# Version B — Agent framework: the loop is managed for you
agent = CodeAgent(tools=[], model=LiteLLMModel(model_id=MODEL))
result = agent.run(task)    # the while-loop, message accumulation, dispatch — all hidden
```

Concept: Both produce the same answer. Version A is ~12 lines you can debug; Version B is 2 lines that hide the loop. When B misbehaves, you need A to debug it.

Check Your Mental Model — Before We Code

Formative Check 1 of 2 | Bloom level: Understand | ~8 minutes

This is a mirror, not a grade. Answer it before the worked example.

> ***Q:*** *A teammate says "I just added 'please respond in JSON' to the system prompt — that should be enough for production." What is wrong with this approach, and what would you recommend instead?*

Answer direction: Prompt-only JSON (Rung 1) has **no enforcement** — the model can still return prose or malformed JSON. For production, climb to a **tool schema** (Rung 2) or **Pydantic + instructor** (Rung 3), which enforce the schema at the API / validation layer and retry on violation.

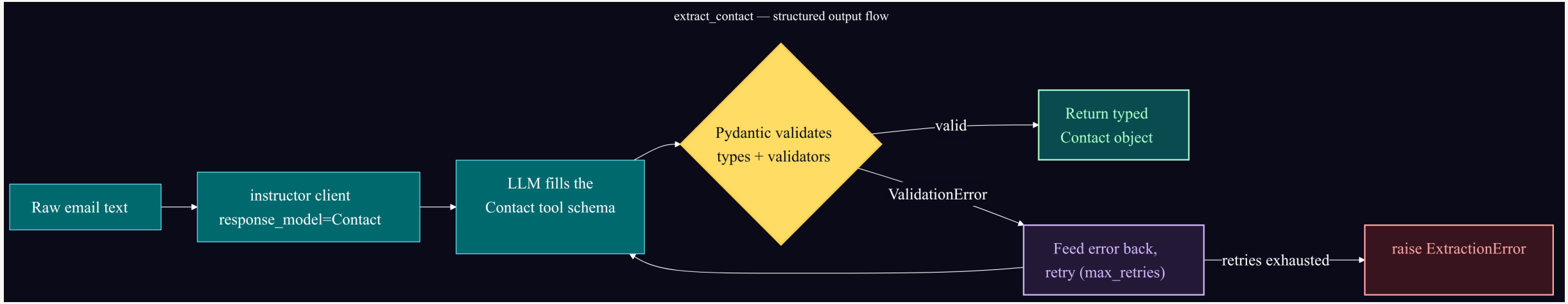
🟨

Misconception named — Prose-Prompt-JSON: believing that *instructing* the model to output JSON is equivalent to *enforcing* a schema.

> ▶️
> *Open Check 1 (Understand) — scenario correction · ~8 min*

Worked Example — `extract_contact`

Goal: Turn a raw customer email into a typed `Contact` object — the same task as the homework, walked through end to end. Raw text in, validated object out, loud failure on bad data.



Worked Example — Step 1: The Schema

Define the contract first. The schema *is* the specification of what "done" means.

```python
from pydantic import BaseModel
from typing import Optional

class Contact(BaseModel):
    name: str
    email: str
    phone: Optional[str] = None
    company: Optional[str] = None
```

- **Required** fields (`name`, `email`) must be present or validation fails.
- **Optional** fields default to `None` — `phone` not in the email is fine, not an error.

Concept: The Pydantic model is a machine-checkable contract. Anything that does not match it is rejected before it reaches your code.

Worked Example — Step 2: The instructor Call

Wrap the client once, then call it like a typed function.

```
import instructor, litellm, os

client = instructor.from_litellm(litellm.completion)

def extract_contact(text: str) -> Contact:
    return client.chat.completions.create(
        model=os.getenv("LLM_MODEL", "claude-sonnet-4-6"),
        response_model=Contact,
        messages=[{"role": "user",
                   "content": f"Extract contact info:\n\n{text}"}],
        max_retries=2,        # retry on ValidationError
        temperature=0,
    )
```

- `response_model=Contact` tells instructor what shape to enforce and return.
- `max_retries=2` runs the validate-and-repair loop automatically.

Production tip: Returning `Contact` (not `dict`) means every caller gets type-checking and autocomplete for free.

Worked Example — Step 3: Handle the Failure Mode

Wrap exhausted retries in a **named** exception, so callers can catch the right thing.

```python
class ExtractionError(Exception):
    """Raised when a valid Contact cannot be produced after retries."""

def extract_contact(text: str) -> Contact:
    try:
        return client.chat.completions.create(
            model=MODEL, response_model=Contact, max_retries=1,
            messages=[{"role": "user", "content": f"Extract contact:\n\n{text}"}],
            temperature=0)
    except Exception as e:            # instructor exhausted its retries
        raise ExtractionError(f"Could not extract a valid Contact: {e}") from e
```

Anti-pattern — swallowing the error: A bare `except: pass` returns a half-built or `None` contact downstream. Raise `ExtractionError` so the caller decides what to do.

Worked Example — Step 4: Side-by-Side

Dimension	Rung 1: Prompt JSON	Rung 2: Tool schema	Rung 3: Pydantic + instructor
Lines of glue code	High (fence-strip + try/except)	Low	Low
Enforcement	None	Syntax + presence	Syntax + presence + **semantics**
On bad output	Silent / `JSONDecodeError`	`KeyError` downstream	Retry, then `ExtractionError`
Cost of failure	Debugged at 2 AM	Debugged at parse site	Caught at call time
Use when	Never for production	Display-only fields	Any field code reads

Production tip: The three rungs cost roughly the same to write. The difference is *where* failure surfaces — and Rung 3 surfaces it at the call, where it is cheapest to handle.

Apply Your Mental Model — Design the Schema

Formative Check 2 of 2 | Bloom level: Apply | ~10 minutes

You need to extract `{product_name, quantity, unit_price}` from order-confirmation emails.

> *Q: Write the Pydantic model and the `instructor`-wrapped client call that enforces this schema. What happens if the model returns `quantity` as a string instead of an `int`?*

Answer direction: Define `OrderLine(BaseModel)` with typed fields (`quantity: int`, `unit_price: float`); wrap the client with `instructor.from_litellm`. If the model returns `quantity` as a string, Pydantic v2 **raises** by default — instructor catches it and retries; if retries are exhausted, a `ValidationError` propagates for the caller to handle.

🟨

Misconception named — Silent-Type-Coercion: assuming Pydantic will quietly coerce `"3"` into `3`. In strict mode it raises; do not rely on silent coercion.

> ▶️
> *Open Check 2 (Apply) — schema design + code completion · up to 3 attempts · ~10 min*

Lab Brief — Your Turn

Four labs build the muscle; the homework is the portfolio artifact.

- **Lab 1 — env scaffold:** `.env` / `.env.example`, load with `python-dotenv`, confirm no secret in `git diff`.
- **Lab 2 — three approaches:** prompt-only JSON vs tool schema; observe the failure surface of each.
- **Lab 3 — Pydantic + instructor:** validation + retry; confirm schema enforcement (and the manual loop under the hood).
- **Lab 4 — SDK comparison:** the same task with the client SDK vs a minimal agent framework; note what the framework hides.

Homework — `extract_contact(text) -> Contact`**:** Pydantic schema + validation + one retry. Required `name`, `email`; optional `phone`, `company`. `email` is lowercase-stripped. Raise `ExtractionError` (not a generic exception) after retry.

▶ **Submit your homework — AI-graded in ~30s** *(primary)*. The grader scores against the rubric and returns line-level feedback.

Production-track: fork drdgreed/career-foundry, add your solution under `submissions/<your-handle>/m0-2/`, open a PR — the same grader runs in CI as a PR comment.

Time, Support, and Resources

Expected time for HW-0.2: 2–3 hours (schema design + extraction + the failure-path test; longer if new to `uv` / `.env` setup).

Setup check: If `lab1_env.py` prints `READY`, your environment is ready for the homework.

Where to get help:

- **AI Tutor** (live, in-page) — Claude-/OpenAI-backed inline assistant trained on this curriculum; it cites slides back to you. Chat bubble in the lower-right of every M0.2 page.
- **Report an issue / ask the cohort** — the support widget files a pre-tagged GitHub Issue (auto-labels `module:m0-2`).
  🚧
  Cohort chat is tracked on ROADMAP.md.
- **Reflection prompts** — M0.2 reflection page (offline mirror).

Catch-up path: Use `instructor` directly (Lab 3 style) and wrap its `ValidationError` in `ExtractionError` — get the artifact working first, then study the manual loop.

Reflection Prompt

Take **5 minutes** before you start HW-0.2. Write your answers in a text file.

1. How often did Rung 1 fail?

> *When you ran prompt-only JSON, how often did it return prose or malformed output? What does that failure rate reveal about relying on prose instructions for structural guarantees?*

2. Which systems receive your LLM output?

> *In your current or target role, what downstream systems consume model output? How would schema enforcement change the reliability contract with them?*

3. What still feels like a black box?

> *Name one thing about the `instructor` retry loop you cannot yet explain. Write it down before the lab — then check whether the lab resolves it.*

📖
Full reflection page: Open on career-forge.org — learner-friendly rendering with facilitator notes. (Offline mirror)

Self-Assessment Rubric

Score yourself **0–3** on each criterion. **Threshold to advance: 8/15.**

Criterion	0	1	2	3
Reproducible env & secrets	Keys in code	`.env` used	`.env.example` committed, key gitignored	Pinned deps + `llm.py` seam
Reliability ladder	Cannot name rungs	Names all three	Explains enforcement per rung	Chooses correct rung per task
Pydantic + instructor	Cannot write a model	Writes a model	Adds a validator	Validate + retry + named error
Failure handling	Swallows errors	Catches generically	Raises a named exception	Actionable message + retry policy
Client vs Agent SDK	Cannot distinguish	States the difference	Names what a framework hides	Justifies the choice per scenario

🟩

Production tip: Your `extract_contact` function carries forward into M1.1's prompt-evaluation harness — write it as production code, not throwaway.

📊

Self-assessment page: Open on career-forge.org — five-criterion self-scoring (threshold 8/15 to advance). (Offline mirror)

What You Built + What Is Next

This module produces:

- A reproducible project scaffold — `.env` secrets, pinned deps, the `llm.py` seam (→ `mLO-0.2a`).
- `extract_contact(text) -> Contact` — a validated, retrying, loudly-failing typed function (→ `mLO-0.2b`).
- A working command of the **client SDK vs agent SDK** distinction the rest of the course builds on (→ `mLO-0.2c`).

Next module: M1.1 — Prompting & Structured Output for Reliability.
M0.2 gave you the structured-output tool; M1.1 teaches you to systematically improve the *inputs* that produce those outputs — building a 30-example eval harness that measures prompt quality as an engineering metric. Your `extract_contact` function becomes the unit under test.

Portfolio signal: `extract_contact` is direct, demonstrable evidence for req-026 (structured output / Pydantic / JSON schema, Score 3) — the kind of artifact a hiring manager can read in 60 seconds and trust.

Module links: Slides · Module hub · Course repo · Roadmap