M1.1 — Prompting & Structured Output

The discipline of making model output reliable — measured with numbers, not vibes.

Hiring signal: Prompt engineering (req-006) and structured output / JSON schema (req-026) each score **3/3** and appear at **high frequency** across harvested 2026 JDs. This module is where you stop guessing at prompts and start engineering them.

The Problem You Will Solve

Two engineers are debugging the same flaky AI feature.

- **Engineer A** adds exclamation marks to the system prompt — `"answer accurately!!!"` — re-runs it once, and says *"I think that helped."*
- **Engineer B** runs a **30-example labeled eval** before and after every change, and reports the delta in accuracy.

Engineer A is guessing. Engineer B is engineering.

Hiring signal: A prompt is a software artifact — it needs version control, regression tests, and a change log. That habit is what req-006 actually screens for.

Why This Module → Career Anchor

Requirement	JD frequency	M1.1 coverage
Prompt engineering (req-006)	**High**	Score 3 — practiced + measured in labs
Structured output / Pydantic / JSON schema (req-026)	**High**	Score 3 — schema as a reliability tool
Systematic LLM evaluation	**High**	Score 2 — A/B harness, confusion matrix

Production tip: Citing context outperforms adding exclamation marks because it exploits how transformer attention works — not because it is polite. Every rung of the ladder has a measurable effect on a labeled set.

Archetype strongest fit: AI Engineer — practical fluency across the LLM ecosystem (the GitLab Senior AI Engineer JD names exactly this).

Before We Begin — What You Need

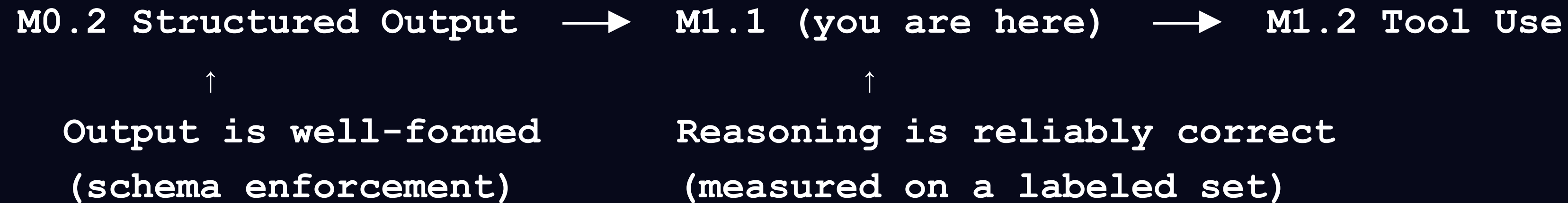
You MUST already own (from M0.2):

- **Structured output:** you can enforce a JSON schema with Pydantic + instructor, and you know the difference between prompt-only JSON and schema-enforced output.
- **Client SDK loop:** you can make a complete `litellm.completion()` call and inspect the response.
- `.env` **hygiene:** you have a working local environment with secrets loaded from a `.env` file — no hardcoded keys.
- **Tooling installed:** `litellm`, `pydantic`, `instructor`, plus basic `json` and `jsonlines` familiarity.

Gotcha — Missing these? Do not proceed. Return to M0.2 (Python SDKs & Structured Output) first — this module builds directly on its `llm.py` seam.

Bridge — Where M1.1 Fits

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



> *M0.2 showed you how to make the model's output structurally trustworthy. M1.1 teaches you to make the model's reasoning reliably correct — and to measure that reliability with numbers, not vibes.*

Connecting concept: Tool descriptions in M1.2 are prompts. The engineering discipline you build here applies directly to them.

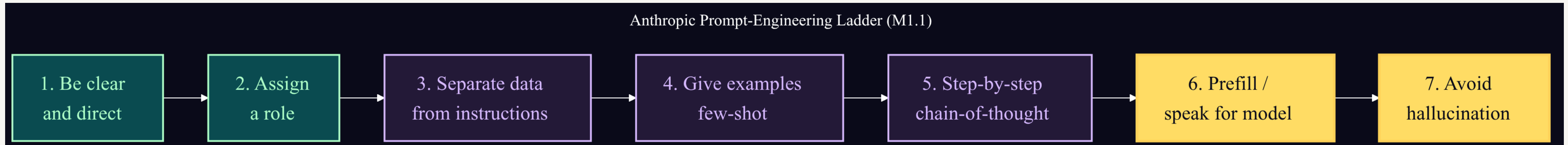
New Terms in This Module

Term	Definition
Prompt ladder	Anthropic's ordered hierarchy of prompt techniques, from clear-and-direct up to grounding constraints — applied lowest-rung-first.
A/B eval	Comparing two prompt versions against the same labeled dataset to measure which performs better.
Prompt registry	A versioned store of prompt templates with metadata: version, author, eval score, date.
Confusion matrix	For classification: true/false positives and negatives per class — assesses quality beyond a single accuracy number.

Pre-training principle: These four terms recur in every slide that follows. Meeting them now keeps them from adding cognitive load mid-explanation.

Concept 1 — Anthropic's Prompt-Engineering Ladder

Ordered by impact-per-effort. Apply from the bottom up — only add complexity when lower rungs fail.



Concept 1 (cont.) — Reading the Ladder

The rungs are **levers**, not tips — each has a measurable effect on a labeled eval set:

- **Rungs 1–3 (clear, role, delimited data):** cheapest, highest impact. Most reliability problems are solved here.
- **Rungs 4–5 (few-shot, chain-of-thought):** add tokens and latency; reserve for hard or ambiguous cases.
- **Rungs 6–7 (prefill/schema, grounding constraint):** lock the format and constrain invention — rung 6 is the M0.2 schema technique, re-cast as a reliability lever.

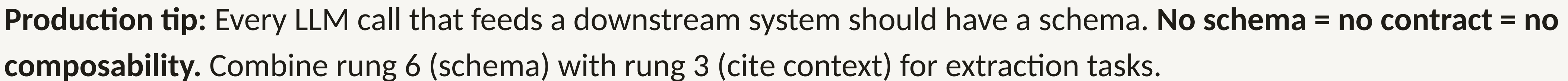
Production tip: Each rung adds cost and latency. Use the lowest rung that achieves your quality target — and prove it on the labeled set.

Gotcha — Enthusiasm-as-Signal: `"answer accurately!!!"` adds no structural signal. The model attends to *content*, not exclamation marks.

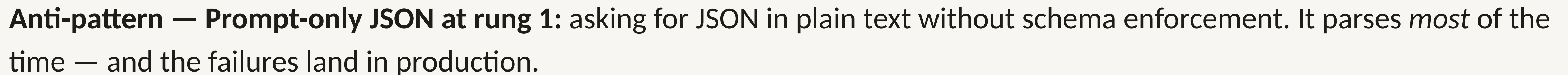
Concept 2 — Structured Output as a Reliability Tool

Structured output is not just machine-parseability — it is a **reliability** technique. Forcing `{"category": "...", "reason": "..."}` eliminates whole failure modes:

- **Hedging** — "well, it depends..." disappears when a schema demands a category.
- **Format drift** — every call returns the same shape, so downstream code never branches on prose.
- **Decision-dodging** — the model cannot "wriggle out" of committing to a value.



Production tip: Every LLM call that feeds a downstream system should have a schema. **No schema = no contract = no composability.** Combine rung 6 (schema) with rung 3 (cite context) for extraction tasks.



Anti-pattern — Prompt-only JSON at rung 1: asking for JSON in plain text without schema enforcement. It parses *most* of the time — and the failures land in production.

Concept 2 (cont.) — Schema-Enforced Classification

Rung 6 in code: a tool schema makes `category` a typed enum the model **must** fill.

```
TOOL = {
    "type": "function",
    "function": {
        "name": "classify",
        "parameters": {
            "type": "object",
            "properties": {
                "category": {"type": "string",
                             "enum": ["REFUND_ELIGIBLE", "EXCHANGE_ONLY",
                                      "INSUFFICIENT_INFO"]},
                "reason": {"type": "string"},
            },
            "required": ["category", "reason"],
        },
    },
}
```

Concept: The `enum` constraint is the reliability lever — the model cannot return a category outside your three valid values.

Concept 3 — Systematic A/B Eval on a Labeled Set

Intuition-driven prompting is a dead end: you iterate once, it *seems* better, you ship — then discover the gain was on the three cases you tested while untested ones regressed.

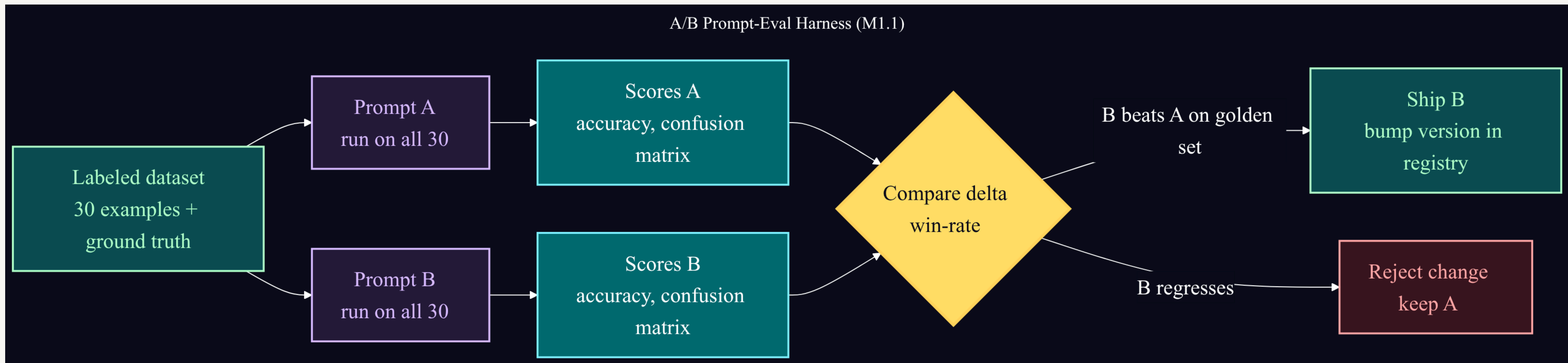
The workflow (same rigor as software testing):

1. **Build a dataset** — 20–50 examples with ground-truth labels (start at 30).
2. **Define a metric** — exact-match for classification; a confusion matrix for *which* class fails.
3. **Run variants** — execute Prompt A and Prompt B against the full set.
4. **Report win-rates** — "B beats A on 15/30, ties on 10, loses on 5."

Gotcha — Sample size: 30 examples catch gross failures; **300+** catch subtle regressions. Start at 30 and expand — never treat 30 as a production guarantee.

Concept 3 (cont.) — The A/B Harness

Same dataset, two prompts, one comparison. This harness is the **MP-1 seed**.



Concept 4 — Prompt Versioning & Regression

Prompts are code. They have versions, they have tests, and they can regress. A change that fixes three cases while breaking one *existing* case is a regression — you only catch it against a fixed labeled set.

Minimal implementation:

- Store prompts as **versioned constants or YAML**, not inline strings. Each has an ID, a version, an eval score, and a change description.
- **Regression gate:** if a new version scores lower than the previous one on the golden set → reject.
- **CI integration:** run the eval harness on every prompt change in the repository.

Production tip: Treat prompt changes like code changes — they deserve PR review and automated tests. GitHub Copilot's team maintains a benchmark suite of completions that prompt changes must not regress on.

Check Your Reasoning — Before We Code

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

This is not a grade. It is a mirror. Answer confidently and correctly, and you are ready for the worked example.

> **Q:** *Your system prompt currently says* `"Please answer accurately and helpfully!!!"` *A teammate suggests replacing it with* `"You are a senior support specialist. Cite the policy provided, then think step by step."` ***Without running an eval****, which change is more likely to improve reliability — and why?*

- **A)** The exclamation-mark version — more emphasis signals importance.
- **B)** The teammate's version — it applies rungs 2, 3, and 5, which change *how the model attends* to the input.
- **C)** They are equivalent; only the model version matters.

Take the full check inside the platform — it grades you live and explains every distractor.

>
> *Open Check 1 (Understand) — multiple-choice + short answer · ~8 min*

Worked Example — Climbing the Ladder on a Refund Classifier

The task: classify a customer return request as `REFUND_ELIGIBLE`, `EXCHANGE_ONLY`, or `INSUFFICIENT_INFO`, scored against a labeled set.

We climb the ladder rung by rung, recording the score at each step:

1. **Rung 1 — bare prompt.** Baseline: does it even commit to a category?
2. **Rung 3 — delimited policy + output format.** Ground the decision in the policy.
3. **Rung 5 — chain-of-thought.** "Think step by step before the final JSON."
4. **Rung 6 — schema enforcement.** Force the typed `enum` so the category is always valid.

What to observe: each rung produces more reliable, more structured output — and the labeled set tells you *how much* more.

Code Walk — Rung 1 → Rung 3

```
MODEL = os.getenv("LLM_MODEL", "claude-sonnet-4-6")

def complete(content):
    r = litellm.completion(
        model=MODEL, temperature=0, max_tokens=512,
        messages=[{"role": "user", "content": content}],
    )
    return r.choices[0].message.content.strip()

# Rung 1: bare prompt — no policy, no format
print(complete(CUSTOMER_MSG))

# Rung 3: delimited policy + explicit output format
prompt3 = f"""
<policy>{POLICY}</policy>
<customer_message>{CUSTOMER_MSG}</customer_message>
Reply as JSON, based only on the policy: {{"category": "...", "reason": "..."}}
"""
print(complete(prompt3))
```

Concept: rung 3's delimiters mark where policy ends and the request begins — a boundary that also guards against injection.

Code Walk — Rung 5 → Rung 6

```python
# Rung 5: chain-of-thought — append a reasoning instruction
prompt5 = prompt3 + "\n\nThink step by step before giving your final JSON."
print(complete(prompt5))

# Rung 6: schema enforcement — the category MUST be a valid enum
r = litellm.completion(
    model=MODEL,
    messages=[{"role": "user", "content": prompt3}],
    tools=[TOOL],
    tool_choice={"type": "function", "function": {"name": "classify"}},
    temperature=0, max_tokens=512,
)
args = r.choices[0].message.tool_calls[0].function.arguments
print(json.dumps(json.loads(args), indent=2))
```

Production tip: rung 6 with `tool_choice` forced is the cross-provider equivalent of Anthropic prefill — the model *must* call your tool and match the schema.

Results — The Trade-off Is Visible

Climbing the ladder on the labeled refund set (illustrative — your numbers come from your own run):

Rung	Technique	Accuracy	Relative cost/call
1	Bare prompt	2/5	1.0×
3	Delimited policy + format	4/5	1.1×
5	+ Chain-of-thought	4/5	1.8×
6	+ Schema enforcement	5/5	1.2×

Concept: rung 3 bought the biggest jump for almost no cost. Rung 5 added cost without moving accuracy here — so on *this* task, you would not ship it. Rung 6 locked the format for a small cost.

Gotcha: these numbers are from one small set. Re-run on 30+ before drawing conclusions — the win-rate on five cases is not significant.

Apply It — Plan Your Next Three Changes

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

You have a prompt that classifies customer support tickets into 5 categories. Current accuracy on your 30-example set is **73%**.

> ***Q:*** *Describe the exact sequence of **3 changes** you would try next, in ladder order, and how you would determine whether each one actually improved things.*

A strong answer:

- Apply rungs in order — try rung 3 (cite/ delimit) before reaching for rung 5 or 7.
- For each change: run the A/B harness, compute the delta, and read the **confusion matrix** to see which category improved.
- If the delta is below your threshold → discard the change and keep the prior version.

Type your plan below — the platform grades it against a root-cause / method / measurement rubric.

>
> *Open Check 2 (Apply) — natural-language plan · up to 3 attempts · ~10 min*

Lab Brief — Your Turn

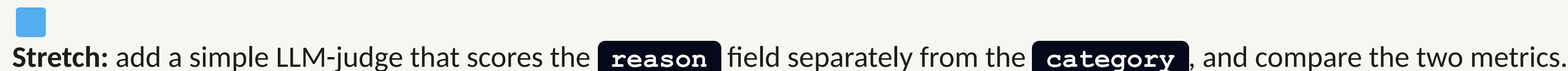
HW-1.1: Take the provided 30-example task; write **3 prompt variants**; report win-rates with a **confusion matrix**.

What you build:

- **3 prompt variants** in your registry (may start from the lab variants, but must differ meaningfully).
- A **3×3 confusion matrix** per variant — REFUND_ELIGIBLE × EXCHANGE_ONLY × INSUFFICIENT_INFO.
- A results table: variant | accuracy | most-common error type.

"Done" looks like:

1. All 3 variants run on the full 30-example set — not just 5.
2. Confusion matrices are numerically correct.
3. The best variant reaches ≥ **75% accuracy**, and your write-up explains *why* it wins.



Stretch: add a simple LLM-judge that scores the `reason` field separately from the `category`, and compare the two metrics.

How to Submit

Two supported paths — pick one:

1. Platform (primary): submit through the in-page lab UI; the AI grader scores against the rubric and returns line-level feedback in about 30 seconds.

Submit your lab — AI-graded

2. Production track (GitHub fork): fork drdgreed/career-foundry, add your solution under `submissions/<your-handle>/m1-1/`, and open a PR. The **same grader runs in CI** as a PR comment.

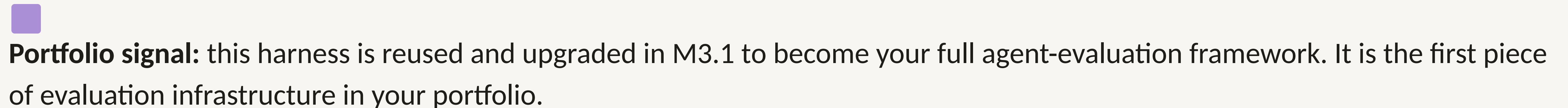
Production tip: The CI grader on your fork is the same harness the platform runs — your prompt eval is gated exactly the way a production LLM team would gate it.

Roadmap: see ROADMAP.md for cohort support channels and the CI-grader rollout.

What You Build — MP-1 Prompt-Eval Harness

By the end of the lab you have a standalone CLI tool — the **MP-1 seed**:

- Reads a JSONL dataset (one `{id, input, expected}` per line).
- Runs **≥ 3 prompt variants** from a versioned registry.
- Outputs a scored comparison table (exact-match, optional LLM-judge).
- Saves results to JSONL for **regression tracking**.



```
python mp1_harness.py --dataset data/returns_eval.jsonl --output results/run_001.jsonl
```

Portfolio signal: this harness is reused and upgraded in M3.1 to become your full agent-evaluation framework. It is the first piece of evaluation infrastructure in your portfolio.

Reflection Prompt

Take **5 minutes** before you start HW-1.1. Write your answers in a text file.

1. Which rung produced the biggest accuracy jump — and what does that reveal about your baseline's errors?

> *If rung 3 (cite context) moved the number most, your baseline was likely hallucinating policy details it was never given.*

2. In your current or target role, which AI features would benefit most from a versioned prompt registry and regression tests — and who would own it?

> *Name a concrete feature and a concrete owner.*

3. What is the minimum eval-set size you would accept before deploying a prompt to production? Write your reasoning.

> *Defend the number — 30 is a floor, not a finish line.*

Full reflection page: Open on career-forge.org (offline mirror).

Self-Assessment Rubric

Score yourself 0–3 on each criterion. **Threshold to advance to M1.2: 11/15.**

Criterion	0 — Not yet	1 — Emerging	2 — Proficient	3 — Exemplary
Ladder application	Can name rungs	Applies one or two	Climbs in order on a real task	Justifies each rung by measured impact
Structured output as reliability	Sees it as formatting	Uses a schema	Combines schema + cite-context	Explains failure modes it removes
A/B eval discipline	Eyeballs outputs	Runs on a few cases	Runs full 30 + win-rate	Reports confusion matrix per class
Versioning & regression	Inline prompt strings	Stores versions	Regression gate on golden set	Gate wired into CI
Reasoning quality	"It felt better"	States a delta	Attributes delta to a rung	Explains *why* via attention/grounding

Self-assessment page: Open on career-forge.org (offline mirror).

Next module: M1.2 — Tool Use & Function Calling. You will give the model *actions*, not just text. Tool descriptions are prompts — the discipline you just built decides whether the model calls the right tool at the right time.