M0.1 — How LLMs Actually Behave

Tokens, sampling, context, and why output is probabilistic — the mental model everything else rests on.

Hiring signal: LLM API fluency appears in **28/30** harvested 2026 JDs. This module builds the bedrock mental model that separates engineers who debug LLM failures from those who blame the model.

How to Read These Slides — Conventions

Your cohort spans bootcamp grads, mid-career engineers, and PhDs. To keep us aligned, every deck uses the same conventions:

Color legend (in callouts and tags):

- **Learning outcome** — what you can do after this slide
- **Career anchor** — links concept to a real 2026 JD requirement
- **Warning / misconception** — common error to avoid
- **Lab / hands-on** — code you will write or run
- **Production gotcha** — something that bites you at scale
- **In development** — feature exists in roadmap; offline path provided

Speaker notes are part of the curriculum. The notes block under each slide (`KEY CONCEPTS / EMPHASIZE / MISCONCEPTION / REFERENCES`) is the canonical source for the *why*, the *common errors*, and the *cited evidence*. Read them. They are graded against a 5-gate visual rubric.

References pattern: Every claim that maps to a job-market requirement, an empirical study, or a vendor doc has an inline citation [N] mapped to the REFERENCES block in the notes. Click through them — they are the receipts.

Code blocks use a high-contrast palette (cream `#F8F4E3` on deep navy `#0A0E1A`, with cyan keywords, gold strings) optimized for projector and dark-mode viewing. If anything is hard to read, flag it in the cohort channel — we patch the theme, not the slides.

The Problem You Will Solve

You are debugging a customer-support agent at 2 a.m. The agent is issuing refunds it should not issue — but only sometimes, not reliably. Same prompt, different decision.

- You set `temperature=0` to "make it deterministic." The bug persists.
- You search for a race condition. There is none.
- You stare at the code. The code is fine.

The bug is in your mental model, not your code.

By the end of this module you can: Call a chat model via API, control determinism with precision, read `stop_reason`/usage, and explain *why* LLM output is probabilistic — in language a skeptical colleague will accept.

Why This Module → Career Anchor

Requirement	JD frequency	M0.1 coverage
LLM API integration / production deployment	**28/30 JDs**	Score 2 — practiced in labs
Python proficiency	**30/30 JDs**	Score 1 — foundational setup
Cost, latency, and token optimization	**16/30 JDs**	Score 1 — token counting lab
Prompt engineering	**24/30 JDs**	Score 1 — temperature/sampling

Production tip: Engineers who understand `finish_reason` and token budgets from day one write cheaper, more reliable agents from the start. The cost-awareness habit you form in Lab 1 pays dividends across every subsequent module.

Archetype strongest fit: AI Engineer — Generalist (2.8/3.0 avg coverage). M0.1 is the foundation all archetypes share.

Before We Begin — What You Need

You MUST already own:

- **Python 3.11+** [3] installed with `uv` or `venv`: you can create a virtual environment and install packages without looking up the commands.
- **An API key** for Anthropic or OpenAI in a `.env` file: you can make an authenticated HTTP request from Python.
- **HTTP/JSON fluency**: you understand what a REST endpoint is, what JSON looks like, and can read an API response as a Python dict.
- `litellm` **≥ 1.50.0** [4] — the provider-neutral LLM client this curriculum uses (we pin a known-good version per module).

Gotcha — Missing these? Do not proceed. See the prereq remedial decks:

- `python/http-json-basics` (status: planned, REMEDIAL_BACKLOG.md)
- `python/env-vars-secrets` (status: planned, REMEDIAL_BACKLOG.md)

[3] Python 3.11+ required because litellm and the type-hint patterns we use (`X | None`, generic `dict[str, Any]`) target 3.10+ and several optional deps require 3.11. — python.org/downloads

[4] litellm 1.50.0 is the version pinned for M0.x labs. It provides a unified `completion()` interface across OpenAI, Anthropic, Bedrock, Vertex, and 100+ providers, so swapping models means changing one env var. — github.com/BerriAI/litellm

Bridge — Where M0.1 Fits in the Curriculum

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

M0.1 plants the seed: **"LLMs are probabilistic next-token predictors."**
Every module from M0.2 through M3.7 is an engineering response to this single fact.

Connecting concept: M0.2 will add structured output — the first direct engineering response to LLM probabilism (schema enforcement means variance no longer flows into invalid JSON).

New Terms in This Module

Term	Definition
Token	The atomic unit of text an LLM processes — roughly 3–4 English characters. Not words, not characters.
Context window	The maximum number of tokens (input + output) a model can process in one call.
Temperature	A scalar that controls how much the model samples from vs. concentrates on the top-probability tokens.
`finish_reason` / `stop_reason`	The API field that tells your code *why* generation stopped — essential for loop control.
Hallucination	A confident-sounding but factually wrong or invented response — a structural property of next-token prediction, not a bug.

Pre-training principle (Mayer #7): These terms appear in every slide that follows. Learning them now prevents them from introducing cognitive load mid-explanation.

Concept 1 — What Is a Token?

A language model does not see words or characters. It sees **tokens** — subword units produced by the model's tokenizer.

- **~3–4 English characters** per token on average
- `"unbelievable"` → 3–4 tokens; `"a"` → 1 token
- **1,000 words ≈ 1,300 tokens** (rough English rule)

```
"Hello, world!" → ["Hello", ",", " world", "!"]   # 4 tokens
```

Why it matters for you:

- Cost = input tokens × input rate + output tokens × output rate
- Context window is a token budget, not a word limit
- Long system prompts eat budget before the user speaks

Concept: Every token costs money and consumes context budget. Token awareness is a first-class engineering discipline.

Concept 2 — The Context Window Budget

Every API call has a **fixed token budget** — the context window. Every token consumed by input reduces what is available for output.

```
total_budget = input_tokens + output_tokens ≤ context_window

input_tokens = system_prompt
             + conversation_history
             + retrieved_documents   ← RAG will add many tokens here
             + user_message
```

Model	Context window
Claude Sonnet 4	200,000 tokens
GPT-4o	128,000 tokens
GPT-4o mini	128,000 tokens

Gotcha — The Unlimited-Context Illusion: A 200K-token window sounds like "infinite." A multi-agent loop with retrieved documents, tool results, and conversation history can fill it in minutes.

Concept 3 — Sampling and Temperature

When generating the next token, the model produces a **probability distribution** over its vocabulary. Temperature scales that distribution.

```
temperature = 0.0 → near-deterministic (concentrates on the top token)
temperature = 0.7 → balanced variance (good for creative tasks)
temperature = 1.0 → raw distribution (more variance, sometimes incoherent)
temperature > 1.0 → flattened distribution (rarely useful)
```

Concept: The model is not *looking up* an answer. It is *sampling* from a distribution at every step.

Gotcha — Temperature ≠ quality knob: "Higher temperature = better output" is false. For a binary policy decision (approve refund YES/NO), high temperature introduces dangerous variance. For a creative tagline, it is desirable.

Concept 4 — Training Cutoff and What the Model Cannot Know

The model's weights encode patterns from training data with a **knowledge cutoff**. It has no access to:

- Events after the cutoff date
- Your company's internal documents
- Live prices, order statuses, real-time inventory
- Anything that was not in its training corpus

Anti-pattern — Treating the model as a database: When asked about your customer's order history, the model will generate a plausible-sounding but completely fabricated answer. It has no database. It has pattern weights.

Production tip: This is why RAG and tools exist. Ground the model in retrieved, current documents (M1.x) or callable functions (M1.2). Never assume the model "knows" enterprise-specific facts.

Concept 5 — Hallucination as a Structural Property

Hallucination is the expected behavior of a next-token predictor operating outside its knowledge distribution. The model has no mechanism to distinguish *"I know this factually"* from *"I am generating a plausible-sounding completion."*

```
Question: What was Acme Corp's refund rate in Q3 2024?
Model (to itself): "Most probable token after '...Q3 2024 was'... is '23.4%'"
Model (to you): "Acme Corp's refund rate in Q3 2024 was 23.4%."
```

🟥

Anti-pattern — Believing confident output: Confidence in tone is independent of factual accuracy. The model has no "I don't know" signal — it generates what is plausible.

🟦

Mitigations you will build later — architect around hallucination, do not prompt it away:

- **Guardrails** (M2.4) — output validators that reject responses violating schema, policy, or factuality rules.
- **RAG / grounding** (M1.1–M1.3) — inject authoritative source text so the next-token distribution is conditioned on real evidence.
- **Tool use & function calling** (M2.1–M2.2) — the model defers factual lookups to a database query instead of inventing a number.
- **Confidence scoring + abstention** (M2.5) — explicit "I cannot answer" paths when retrieved evidence is missing or weak.

Concept 6 — Statelessness and Memory

Each API call is **completely stateless**. The model has no memory of previous calls.

```
# Call 1 — tells model learner's name
call_1 = [{"role": "user", "content": "My name is Alex."}]
reply_1 = client.messages.create(...)  # model says "Nice to meet you, Alex"

# Call 2 — NEW call, no history passed
call_2 = [{"role": "user", "content": "What is my name?"}]
reply_2 = client.messages.create(...)  # ← model says "I don't know your name"
```

The `messages` array IS the memory. Include all relevant history or the model has none.

Concept: Apparent "conversation memory" is an engineering layer built on top of stateless API calls — not a native model capability.

Concept 7 — Reading `finish_reason` / `stop_reason`

Every API response includes a `finish_reason` (OpenAI) or `stop_reason` (Anthropic) field. **Agent loops must read this field.**

Value	Meaning	What your code must do
`"stop"` / `"end_turn"`	Model finished naturally	Proceed normally
`"length"`	Hit `max_tokens` budget	**Error: output may be truncated**
`"tool_calls"` / `"tool_use"`	Model wants to call a tool	Execute the tool, continue loop
`"content_filter"`	Safety filter blocked output	Handle gracefully, escalate if needed

Anti-pattern — Ignoring `finish_reason`: Code that always reads `response.choices[0].message.content` without checking `finish_reason` will silently process truncated JSON as valid output. This is a common source of hard-to-debug agent failures.

Check Your Mental Model — Before We Code

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

This is not a grade. It is a mirror. If you answer confidently and correctly, you are ready for the worked example. If you stumble, re-read the concept slide and come back.

Question 1 preview: Which of the following best describes why the same prompt at temperature=1 produces different outputs on different calls?

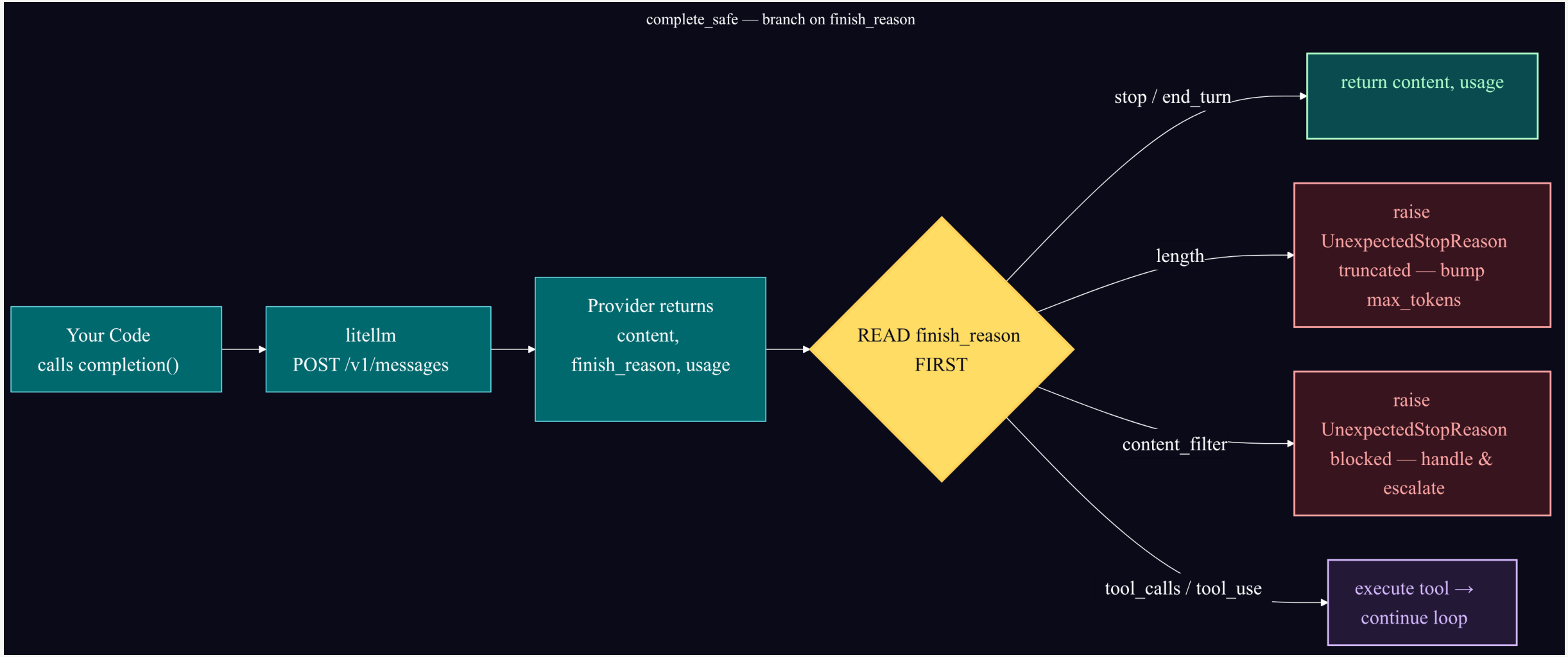
- A) The API randomly shuffles the prompt before sending it to the model
- B) The model samples from a probability distribution over tokens at every generation step
- C) The model uses a different context window size on each call
- D) The provider introduces randomness to prevent caching

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

> *Open Check 1 (Understand) — 3 multiple-choice + 1 short answer · ~8 min*
>
> *Live grading. Immediate feedback on every distractor. Mastery is recorded toward the M0.1 advance gate.*

Worked Example — The `complete_safe` Wrapper

Goal: Write a wrapper that raises loudly on bad `stop_reason` instead of silently producing broken output. — Watch 17-second animation



Code Walk — `complete_safe`, Part 1: Setup

```python
# lab4_stop_reason.py  — install: pip install litellm==1.50.0 python-dotenv
import os
from dotenv import load_dotenv          # load OPENAI_API_KEY from .env
import litellm

MODEL = os.getenv("LLM_MODEL", "gpt-4o-mini")    # env-var with safe default

class UnexpectedStopReason(Exception):
    """Raised when the LLM stops for an unexpected reason (truncation, filter)."""
    pass
```

Line 3: `load_dotenv()` reads your `.env` file and sets environment variables. Never hardcode API keys in source files — they end up in git history.

Line 6: `os.getenv(key, default)` means the code works in local dev (with `.env`) and in CI (where `LLM_MODEL` is set as an env var in the runner).

Production tip: The `LLM_MODEL` env var pattern is the `llm.py` seam — a single place to swap model providers without touching agent logic.

Code Walk — `complete_safe`, Part 2: The Wrapper

```
def complete_safe(messages: list[dict], max_tokens: int = 512) -> tuple[str, object]:
    """Raises on truncation or content filter; never silently returns bad output."""
    response = litellm.completion(
        model=MODEL, messages=messages, max_tokens=max_tokens
    )
    choice = response.choices[0]
    reason = choice.finish_reason          # READ THIS FIRST

    if reason == "length":                  # truncation: output is incomplete
        raise UnexpectedStopReason(f"Truncated at {max_tokens} tokens.")
    if reason == "content_filter":
        raise UnexpectedStopReason("Blocked by content filter.")
    return choice.message.content, response.usage
```

Line 7: Extract `finish_reason` *before* reading content — the content may be garbage if reason is `"length"`.

Line 10: `"length"` means truncation — raise immediately, do not process the partial output.

What Could Break — Production Gotchas

Three failure modes you will encounter in the wild:

1. Silent truncation at `finish_reason="length"`

- **Symptom:** Agent produces partially-formed JSON or mid-sentence responses
- **Why:** `max_tokens` is too low for the actual output length; you did not check `finish_reason`
- **Diagnosis:** Log `finish_reason` on every call; set an alert when it is `"length"`

2. Temperature variance producing wrong policy decisions

- **Symptom:** A binary decision (approve/deny) gives different answers on retries
- **Why:** `temperature` is set above 0 for a deterministic task
- **Diagnosis:** For all policy decisions, set `temperature=0`; verify with 5-run consistency test

3. Context overflow silently truncating input

- **Symptom:** Agent "forgets" early parts of the conversation; RAG results ignored
- **Why:** Token budget exhausted; provider silently truncates oldest messages
- **Diagnosis:** Pre-count tokens with `litellm.token_counter()`; log `usage.prompt_tokens`

Debug This Code — Apply Your Mental Model

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

The following code has a bug. What is wrong, and how do you fix it?

```
def call_model(prompt: str) -> str:
    messages = [{"role": "user", "content": prompt}]
    response = litellm.completion(
        model=MODEL,
        messages=messages,
        max_tokens=50,        # deliberately small for cost control
    )
    content = response.choices[0].message.content   # ← the bug lives here
    return content                                   # ← and propagates from here
```

The symptom: On long outputs, the function returns truncated text mid-sentence. On short outputs, it works fine.

🟩

Type your diagnosis below. The platform grades your natural-language explanation against a 3-dimension rubric (root cause, mechanism, fix) and returns specific feedback — not just "right/wrong."

> ▶️
> *Open Check 2 (Apply) — natural-language debug · up to 3 attempts · ~10 min*
>
> *LLM-graded against root-cause, mechanism, and fix dimensions. Immediate misconception tags. Skip-and-come-back unlocks at attempt 3.*

Lab Brief — Your Turn

HW-0.1: Write three prompts at `temp=0` and `temp=1`, run each 5×, tabulate the variance.

What you build:

- `hw01_results.csv` — 30 rows (3 prompts × 2 temps × 5 runs), columns: prompt_type, temperature, run, response
- `hw01_analysis.py` — the code that runs the experiment and writes the CSV
- A 100-word paragraph: *Why does variance differ across prompt types and temperatures?*

"Done" looks like:

1. All 30 API calls are logged with prompt + response + temperature
2. The factual prompt at temp=0 returns identical output across all 5 runs
3. The paragraph correctly attributes variance to temperature AND prompt ambiguity

Scaffold: `lessons/m01/lab2_temperature.py` — extend it to loop over 3 prompts.

Submit your lab — 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>/m0-1/`, open a PR — same grader runs in CI as a PR comment.

Assessment Rubric

Criterion	Proficient	Exemplary
API call correctness	All 30 calls succeed, usage fields logged	Calls use litellm seam; handles rate-limit retries gracefully
Token-cost calculation	Estimates cost for each call using input/output rates	Compares estimated vs. actual prompt_tokens; notes discrepancy
Variance analysis	CSV populated; paragraph attributes variance to temperature	Paragraph distinguishes prompt ambiguity as a second source of variance independent of temperature
stop_reason handling	Checks stop_reason; raises on "length"	Wrapper is reusable; raises with an actionable message that names the fix
Mental model accuracy	Paragraph avoids anthropomorphism ("model decided")	Paragraph uses "probability distribution" and "sampling" correctly

Production tip: Your `hw01_analysis.py` is reusable in M1.1's prompt-evaluation harness — write it cleanly.

Submit for AI-graded scoring — grader scores each criterion above. Production-track: PR against drdgreed/career-foundry under `submissions/<your-handle>/m0-1/` for CI feedback.

Time, Support, and Resources

Expected time to complete HW-0.1: 2–3 hours
(30 API calls + analysis + paragraph; longer if you are new to litellm setup)

Scaffold code: `lessons/m01/lab2_temperature.py` in the course repo

Setup check: Before starting, verify you can run `lessons/m01/lab1_tokens.py` successfully — if that works, your environment is ready for HW-0.1.

Where to get help:

- **AI Tutor** (live, in-page) — Claude-/OpenAI-backed inline assistant trained on this curriculum. Ask about the lab, the worked example, or your own code; it cites slides back to you. Chat bubble in the lower-right corner of every M0.1 page.
- **Report an issue / ask the cohort** — [Report Issue] button in the support widget files a pre-tagged GitHub Issue (auto-labels `module:m0-1`).
  🚧
  Cohort chat tracked on ROADMAP.md.
- **Reflection prompts** — M0.1 reflection page (offline mirror).

Catch-up path: Start with the factual prompt × 2 temps × 5 runs (10 calls). Build intuition, then extend to 30.

Reflection Prompt

Take **5 minutes** before you start HW-0.1. Write your answers in a text file.

1. Where will your implementation diverge from the worked example — and why?

> *Think about prompt length, model choice, whether you are calling Anthropic or OpenAI, and what temperature patterns you expect for your chosen prompts.*

2. How would you apply LLM probabilism in your current or target role?

> *Name a concrete decision in your work where output variance would matter — and what temperature policy you would set.*

3. What is still confusing? Write it down.

> *You will return to this after the lab. If the confusion persists, it is your study prompt for M0.2 and M1.1.*

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

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

What You Have Built + What Is Next

This lab produces:

- `hw01_results.csv` — empirical temperature variance data
- `hw01_analysis.py` — reusable experiment harness (feeds into M1.1's prompt-eval harness)
- A written mental model of probabilistic LLM behavior

Next module: M0.2 — Python-for-AI, SDKs & Structured Output

Adds structured output (JSON schema enforcement via Pydantic) — the first direct engineering response to the probabilism you just learned. M0.1 explains *why* variance is a problem; M0.2 teaches the first tool to contain it.

This Module → Your Portfolio → The Job Market

Skill demonstrated	JD frequency	Coverage score
LLM API integration / production deployment	**28/30 JDs**	2 — practiced in labs
Cost, latency, and token optimization	**16/30 JDs**	1 — token counting lab
Python proficiency	**30/30 JDs**	1 — foundational setup

Archetype strongest fit: AI Engineer — Generalist (2.8/3.0) · Agentic Systems Engineer (2.9/3.0) · AI Solutions/FDE (2.7/3.0)

Portfolio signal: M0.1 is not yet a portfolio artifact — it is the mental model that makes your subsequent portfolio credible. Hiring managers who interview you will probe LLM fundamentals; this module is your answer to "explain why LLM output is non-deterministic."

Data source: 30 live JDs, Greenhouse, 2026-06-08. Review annually.