

M02 — Prompt Engineering Lab

Lab — Generative AI Boot Camp for Developers — Exported 2026-08-03

Time budget: ~20 minutes

Section: Day 1 AM (shared with M01 lab)

2026-08-03

Learning Objectives

After completing this lab, you will: 1. Apply the four pillars (Context, Instruction, Constraints, Format) to craft effective prompts 2. Iterate and refine prompts to measurably improve output quality 3. Generate real development artifacts using prompt engineering across multiple tools

2026-08-03

Tool Reminders

You used these tools in M01. Quick refresher:

Tool	Best For	Command / Access
OpenCode CLI	Provider-flexible, scriptable, one-shot tasks	<code>opencode run "prompt"</code>
VS Code Chat	IDE-integrated, sees your open files	Ctrl+Shift+I in VS Code
Claude Code <i>(optional)</i>	Deep codebase analysis, complex refactoring	<code>claude -p "prompt" # Requires ANTHROPIC_API_KEY set</code>
Codex CLI <i>(optional)</i>	Fast prototyping, web dev	<code>codex exec "prompt"</code>

Today's focus: Prompt engineering patterns work the same regardless of which tool you use. Try at least two tools across the exercises to see how each handles the same prompt differently.

Concept Review: Prompt Engineering Essentials

The M02 presentation covered prompt patterns and the iteration loop. This review solidifies those concepts before you apply them.

The Four Pillars of Every Good Prompt

Pillar	What It Does	Weak Example	Strong Example
Context	Sets the scene — what the AI needs to know	"Write a function"	"You are a senior Python developer building a REST API. The codebase uses FastAPI with Pydantic models."
Instruction	The specific task to perform	"Add validation"	"Add input validation for the <code>create_user</code> endpoint. Validate email format, username length (3–32 chars, alphanumeric + underscore), and password minimum 8 characters."
Constraints	Boundaries and rules	"Make it good"	"Use only the Python standard library. Include type hints. Handle null and empty inputs. Raise ValueError with descriptive messages."
Format	How output should be structured	(not specified)	"Return a complete Python function with a Google-style docstring. Include 3 example usages in comments. Output as a single code block."

The Prompt Engineering Loop

Write → Evaluate → Identify Weakness → Refine → Repeat

Your first prompt is a draft — not a final product. Each iteration should produce measurably better output. If you're not seeing improvement after 2–3 iterations, change your approach, not just the wording.

Five Core Patterns

Pattern	When to Use	Example Key Phrase
Role-based	Always — it's the simplest, most effective pattern	"You are a senior {role} specializing in {domain}..."
Task Decomposition	Complex multi-step tasks	Break into sequential sub-prompts: schema → API → tests
Structured Outputs	Need parseable, predictable results	"Return a JSON object with keys: name (string), age (number)..."
Few-Shot	Nuanced or stylistic tasks	Include 2–5 examples of desired input→output pairs
Chain-of-Thought	Debugging, analysis, complex reasoning	"Think through this step by step. Explain your reasoning before giving the final answer."

Prompt Debugging: Symptom → Fix

Symptom	Likely Cause	Fix
Off-topic response	Too vague or ambiguous	Add context, specify the domain
Too short / shallow	No length or depth constraint	Add word count, required sections
Wrong format	Format not specified	Request JSON, table, YAML, or code block
Hallucinated APIs/ functions	No grounding	Provide reference docs; add "Use only standard library"
Refinement not helping	Changing the wrong variable	Identify the <i>specific</i> weakness before rewriting
Overly cautious / refuses	Triggered safety filter	Reframe without triggering language

Prerequisites

- LLM API access configured (from M01 lab)
 - OpenCode CLI and VS Code working (verified in M01)
-

2026-08-03

Exercise 1: Generate an API Specification

Use prompt iteration to go from a vague prompt to a production-quality API spec.

Step 1 — Baseline (1 minute)

Send this minimal prompt using **OpenCode**:

```
opencode run "Write an API spec for a task management system."
```

Evaluate: What's missing? Too generic? Wrong format? Make notes in your iteration tracker (`workshop/m02/iteration-tracker.md`).

Step 2 — Apply the Four Pillars (3 minutes)

Rewrite the prompt using all four pillars. Your refined prompt must include: - **Context:** Role, framework, audience - **Instruction:** Specific endpoints, resources, operations - **Constraints:** What to include and exclude - **Format:** Output format (OpenAPI YAML, Markdown table, etc.)

Try it with a **different tool** than Step 1 — if you used OpenCode, now use **VS Code Chat**.

Step 3 — Add a Pattern (3 minutes)

Choose at least one additional pattern (role-based, few-shot, structured output) and further refine. Try a **third tool** if available (Claude Code or Codex).

```
# Optional: Claude Code
claude -p "You are a senior API architect. Design a REST API..." # Requires ANTHROPIC_API_KEY set

# Optional: Codex
codex exec "You are a senior API architect. Design a REST API..."
```

Reflection

Fill out the Exercise 1 table in your iteration tracker. Which change produced the biggest quality improvement?

Exercise 2: Generate Function Stubs

Apply prompt engineering to generate production-ready code, not just a skeleton.

Scenario

You need a `calculate_shipping()` function for an e-commerce system. It takes package weight, destination country, and shipping method, then returns a cost estimate.

Step 1 — Baseline

```
Write a function to calculate shipping costs.
```

Step 2 — Refine with Constraints

Add: specific parameters with types, docstring format, error cases to handle, and an example of your team's coding style (or a popular style guide reference).

Try this in **VS Code Chat** — select any existing Python file first to give the AI context about your coding style.

Deliverable: A complete, well-documented function stub ready for implementation.

Exercise 3: Generate Documentation from Code

The highest-ROI AI use case. Use prompt engineering to document real code.

Setup

Copy the sample code from `workshop/m02/sample-code.py` into your working directory. This is a working `UserService` class that lacks proper documentation.

Step 1 — Baseline

Paste the code into your AI tool and say: "Document this code."

Step 2 — Refine

Add: target audience (other developers on your team), documentation format (docstrings? README? API reference?), level of detail, and whether to include usage examples.

Try with **OpenCode** — it can read the file directly:

```
opencode run "Read workshop/m02/sample-code.py. Generate comprehensive documentation following Google-style d
```

What to Compare

- Baseline vs. refined: which documentation would you actually want on your team?
 - Did the AI catch the hardcoded salt? The caching TTL? The password hashing approach?
-

Deliverables

1. Completed iteration tracker (`workshop/m02/iteration-tracker.md`)
 2. Function stub from Exercise 2 with type hints and docstring
 3. Documented `user_service.py` from Exercise 3
-

2026-08-03

Troubleshooting

Symptom	Likely Cause	Resolution
Output too generic	Missing context or constraints	Specify domain, audience, detail level
Wrong format	Format not specified	Add explicit instruction: "Return as OpenAPI YAML"
Hallucinated APIs	No grounding	Add "Use only standard library" or provide reference docs
Refinement not helping	Changing wrong variable	Identify the <i>specific</i> weakness before rewriting
Too verbose	No length constraint	Add word count or "be concise"
Model refuses task	Triggered safety filter	Reframe; break into smaller sub-prompts
VS Code Chat slow	Large context	Close unused editor tabs; reduce selected code
Claude/Codex errors	Missing API key	Use OpenCode instead — same patterns work

Fallback Path

If you're struggling: 1. Start with the simplest version of your task — one clear instruction 2. Add one improvement at a time — don't try to fix everything at once 3. Use the debugging table above to diagnose specific issues 4. Ask the facilitator for a live prompt engineering walkthrough

2026-08-03