THE FREE GUIDE TO

CROSS-FUNCTIONAL PROMPT ENGINEERING

The $400,000 skill nobody explains.

A practical walkthrough of the skill behind Anthropic's highest-paid prompt engineering roles — plus a fill-in template to build your own.

Welcome

Most people use Claude the same way they use a search engine. They type a question, read the answer, and close the tab. That works fine for one-off tasks — but it leaves the most valuable part of Claude completely untouched.

This guide is about the skill that separates casual Claude users from the people getting hired into roles that pay up to $400,000 a year to do this full-time: cross-functional prompt engineering.

> ***What you'll get:*** *A plain-English breakdown of the skill, plus a fill-in-the-blank template to build your first instructions file today.*

One-off prompting vs. cross-functional prompt engineering

One-off prompting

This is normal prompting. You write a prompt, Claude gives you an answer, and the conversation ends there. The next time you open Claude, it has no idea what you talked about yesterday, what tone you like, or how you want things formatted. You're starting from zero, every single time.

Cross-functional prompt engineering

Instead of writing one-off prompts, you write a standing set of instructions — a kind of master prompt — that shapes how Claude behaves every single time, across many different tasks, for as long as that project exists. You're not asking a question. You're designing a role.

This is the actual skill Anthropic is hiring for in roles like Cross-functional Prompt Engineer, where the entire job is owning and shaping how Claude behaves across products — writing the system prompts and instruction sets that make AI consistent at scale. The posted salary for that role is $320,000–$405,000 a year.

Source: Anthropic careers listing, Cross-functional Prompt Engineer. Compensation reflects a full-time role with this skill at its core — not a freelance rate.

The good news: you don't need to work at Anthropic to practice the actual skill. You can start today, inside any Claude Project.

STEP 1

Build your instructions file

A fill-in-the-blank template for your first Claude Project

Open Claude and create a new Project. Inside the Project, go to the Instructions tab. Everything below maps directly to what you'll paste in there — fill in each blank, then copy the finished version into Claude.

1. Role & purpose

Why it matters: Without a defined role, Claude defaults to being a generic assistant — helpful, but not specialized. Naming the role focuses every response around a specific job.

What to include: One or two sentences. What is this project for, and what role is Claude playing inside it? (e.g. “You are my content strategist for short-form video scripts.”)

2. Tone & voice

Why it matters: Tone is the first thing people notice and the easiest thing to get wrong. Without guidance, Claude's default tone may not match your brand or the way you actually communicate.

What to include: Describe how responses should sound — formal or casual, direct or warm, short or detailed. Give 2–3 adjectives and one real example if you can.

3. Judgment & decision-making

Why it matters: Most real requests are ambiguous. This section tells Claude what to prioritize when instructions conflict or a request is unclear, instead of guessing or asking you every time.

What to include: What should Claude optimize for when there's a tradeoff? (e.g. “When in doubt, prioritize clarity over cleverness.”)

4. Do's

Why it matters: Explicit, positive instructions are followed far more reliably than implied ones. This is where you lock in the behaviors you always want.

What to include: List 3–6 specific behaviors Claude should always do in this project. Be concrete — vague rules get ignored under pressure.

5. Don'ts

Why it matters: Naming the things you don't want prevents the most common drift — the slow creep back into generic, default AI behavior over a long conversation.

What to include: List 3–6 specific behaviors to avoid. Include anything Claude has done in the past that you didn't like.

6. Knowledge & context

Why it matters: Claude can't read your mind or your business. This is where you hand over the background it would otherwise have to guess at — reducing follow-up questions and wrong assumptions.

What to include: Key facts about you, your business, your audience, or your project that Claude should always know without being told again.

7. Output format

Why it matters: Formatting preferences are easy to assume and easy to get wrong. Spelling this out once saves you from reformatting every single response.

What to include: How should answers usually be structured? Length, headers, bullet points vs. prose, whether to ask before writing long content.

8. Example situations

Why it matters: Examples anchor behavior faster than any rule can. Showing Claude 2–3 real situations and the ideal response teaches the pattern, not just the policy.

What to include: Write 2–3 short scenarios in an “if this happens, do this” format, based on situations that actually come up in your work.

> ***Next step:*** *Once your instructions file is filled in and pasted into your Project, move to the stress-test prompts in the companion document to find the gaps before they cost you in real work.*

STEP 2

Stress-test your instructions

Find where it breaks, then fix the principle, not the symptom

An instructions file is never finished on the first draft. The people who get good at this fast are the ones who stress-test constantly — deliberately trying to break their own setup, then improving it.

The loop:

- **1. Test it.** Throw a weird, ambiguous, or off-topic request at your Project on purpose.
- **2. Watch where it breaks.** Did the tone slip? Did it ignore a rule? Did it guess instead of asking?
- **3. Fix the principle, not the symptom.** Instead of adding a narrow new rule, ask what underlying guidance was missing — then clarify that.
- **4. Repeat.** Every weird response is a clue, not a failure.

> ***Companion resource:*** *A separate document — 100 Stress-Test Prompts — is built specifically to attack different parts of your instructions file (tone, edge cases, contradictions, scope creep, and more). Use it to put what you just built through its paces.*