Prompt Engineering

Getting the Best from Claude

From tokens and context windows through role prompting, few-shot examples, chain-of-thought reasoning, iterative refinement, structured outputs, and the failure modes that trip up even experienced users — with a full reusable pattern reference to close.

10 Chapters | 40+ Reusable Patterns

Prompt Engineering

Chapter 1 · How Claude Thinks — Tokens, Context Windows, and Why Phrasing Matters

Before you can write great prompts, it helps to understand what's actually happening inside Claude when you send a message. This chapter pulls back the curtain — not the deep maths, but the mental model that makes a real difference to how you write prompts.

Everything is Tokens

Claude doesn't see words — it sees **tokens**. A token is a chunk of text, somewhere between a character and a word. Common short words are a single token; longer or unusual words get split into several. Whitespace and punctuation count too.

Here's the sentence "*The WireGuard configuration failed unexpectedly.*" broken into tokens:

`The` · `Wire` `Guard` · `configuration` · `failed` · `unexpect` `edly` `.`

A few rules of thumb:

- ~1 token per common English word
- Technical jargon, proper nouns, and rare words often split into 2–4 tokens
- Code tends to be token-heavy — `wg0.conf` is 4 tokens, not one
- ~750 words ≈ 1,000 tokens

> **WHY IT MATTERS**
>
> Claude pays attention to token patterns — not individual characters. Typos in unusual words can genuinely confuse it, because the tokenisation produces something it has rarely seen. Common words with typos matter less.

The Context Window — Claude's Working Memory

Claude has no persistent memory between conversations. Within a single conversation, everything — your system prompt, the full chat history, your current message, and Claude's response so far — lives inside the **context window**. Think of it as a scroll of text that Claude can read from top to bottom.

System prompt Conversation history Current message Free space (for Claude's response)

Claude (Sonnet 4.6, the model powering this session) has a **200,000-token context window** — roughly 150,000 words. That's enormous; you'll rarely hit the limit in normal use. But the principle matters:

- **Claude reads the whole context on every message.** It doesn't "remember" earlier messages — it literally re-reads them. This is why a very long conversation can slow slightly and why context at the top (the system prompt) carries weight.
- **Older context gets equal attention.** Claude doesn't forget what you said in message 3 just because you're now on message 30 — it's all still there.
- **When the window fills, something must give.** Claude Code automatically summarises older turns to make room. That summary is accurate but lossy — precise code snippets from early in the session may be paraphrased.

> **COMMON MISCONCEPTION**
>
> "Claude forgot what I said earlier." Almost never true within a session. If Claude contradicts something from earlier, it's more likely a reasoning error than a memory failure — the earlier text is still there.

How Claude Generates a Response

Claude is a **next-token predictor**. It reads your entire prompt and then generates a response one token at a time, each new token informed by everything before it (the context plus what it's already written in this response).

This has some important consequences:

Claude commits as it writes

Once a token is generated, it's fixed. Claude can't go back and revise the third word after writing the tenth. This is why asking Claude to *think step by step* or *reason before concluding* genuinely helps — it forces the reasoning to appear in the context window where it can influence later tokens, rather than being skipped.

The first sentence sets the trajectory

If Claude opens with "Sure, here's a simple explanation…" it has now committed to a simple, explanatory register. If it opens with "There are three key trade-offs to consider…" the rest of the response will naturally follow an analytical structure. Your prompt shapes which opening Claude is likely to produce — and that opening shapes everything after it.

Claude doesn't plan ahead

There's no hidden phase where Claude "thinks" before writing. What you see is what's being generated. Extended thinking (*available on some models*) is different — it's an explicit reasoning scratchpad — but in a standard response, the generation is the thinking.

Why Phrasing Matters — Practical Examples

Because Claude predicts likely continuations, your framing powerfully shapes the kind of response you get. The same underlying question, phrased differently, can produce very different outputs.

WEAK PROMPT

```
Tell me about SSH.
```

Likely result: a broad, encyclopaedic overview covering history, use cases, key exchange — much of which you probably know already.

BETTER PROMPT

```
I've set up SSH key auth on my
Debian server but get
"Permission denied (publickey)".
Walk me through diagnosing it —
I can run commands on the
server.
```

Likely result: a focused diagnostic sequence — checking ~/.ssh/authorized_keys permissions, sshd_config,

SELinux/AppArmor — exactly what's needed.

WEAK PROMPT

```
Write a Python function.
```

Likely result: Claude guesses. You'll get a placeholder that may not resemble what you actually needed.

BETTER PROMPT

```
Write a Python function that takes a list of file paths, filters to only .html files, and returns them sorted by modification date, newest first. Type hints required.
```

Likely result: exactly that function, with `pathlib`, `os.path.getmtime`, and proper type annotations.

THE CORE INSIGHT

Claude is trying to produce the most likely useful continuation of your input. The more your prompt resembles the beginning of the kind of document you want to end up with, the more likely Claude is to produce exactly that. A vague opener invites a generic response; a specific, well-structured opener invites a specific, well-structured answer.

Attention — What Claude Weighs Most Heavily

Claude's attention mechanism means not all parts of the context carry equal weight:

Position / Type	Relative influence	Practical implication
System prompt (top of context)	Very high	Good place for persistent instructions, persona, constraints
Most recent message	Very high	The immediate question; Claude focuses here
Earlier conversation turns	Moderate	Available but competes with recency; restate key details if drift occurs

Position / Type	Relative influence	Practical implication
Middle of a long document you pasted	Lower	Known as "lost in the middle" — put critical info at the start or end
End of your message	High	A final instruction or format spec carries real weight

Key Takeaways

TOKENS, NOT WORDS

Unusual words, code, and jargon cost more tokens. Keep prompts crisp; waffle costs context space and dilutes signal.

CONTEXT IS A SCROLL

Claude re-reads everything every time. There's no forgetting within a session — only context limits and reasoning errors.

GENERATION IS LINEAR

Claude writes left to right and can't revise. Asking it to reason before concluding genuinely changes the output.

FRAMING IS EVERYTHING

Your prompt is the start of a document Claude is completing. Make your opener look like the document you want.

POSITION MATTERS

Critical instructions belong at the start (system prompt) or end (final message). The middle of a long paste is easy to miss.

Next — Chapter 2: Anatomy of a Good Prompt

Now that you know how Claude processes input, we build a framework: the four components every strong prompt should have — clarity, context, constraints, and output format — with worked examples for each.

Prompt Engineering

Chapter 2 · Anatomy of a Good Prompt — Clarity, Context, Constraints, Output Format

Most weak prompts aren't wrong — they're just incomplete. Claude will always produce *something*, but without enough signal it has to guess what you actually want. This chapter gives you a four-part framework that covers what every strong prompt should include, and shows you how to apply it across different kinds of tasks.

The Four Pillars

1

CLARITY

What exactly do you want Claude to do? One concrete task, stated plainly. Avoid ambiguity about the verb: explain, write, fix, compare, summarise, list.

2

CONTEXT

What does Claude need to know to answer well? Your situation, the existing code, the audience, what you've already tried. Claude can't see your screen.

3

CONSTRAINTS

What must the answer include, exclude, or stay within? Length, language level, tech stack, tone, libraries to avoid, things already ruled out.

4

OUTPUT FORMAT

How should the answer be structured? A bulleted list, a code block, a table, a numbered step-by-step, plain prose. Specify it or Claude will guess.

You don't need all four in every prompt. A quick question might only need clarity. But when a response disappoints you, it's almost always because one of these four is missing.

A Fully Annotated Prompt

Here's a single prompt broken into its four components:

PROMPT — ANNOTATED BY COMPONENT	
CLARITY	`Write a bash function that watches a directory for new .log files`
CONTEXT	`and emails me when one appears. I'm running Debian 12 with sendmail installed and the directory is /var/app/logs.`
CONSTRAINTS	`Use inotifywait, not polling. Keep it under 30 lines. No external dependencies beyond inotify-tools and sendmail.`
OUTPUT FORMAT	`Give me the function in a single code block with a brief comment on each non-obvious line.`

Notice that each pillar does something different. Clarity tells Claude *what to build*. Context tells Claude *what world it's building for*. Constraints tell Claude *what to rule out*. Format tells Claude *how to present it*. Together they leave very little to guesswork.

Pillar 1 — Clarity: State the Task Precisely

The verb you choose matters more than you might expect. Compare these openers:

- `Tell me about Docker` → encyclopaedic overview, probably not what you need
- `Explain how Docker volumes differ from bind mounts` → focused comparison
- `Fix the Docker volume error in this compose file` → action-oriented, specific
- `List the five most common Docker volume mistakes, one sentence each` → structured, bounded

> **GOOD CLARITY VERBS**

> **Explain** (to teach), **Write** (to produce), **Fix** (to repair), **Compare** (to contrast two things), **Summarise** (to condense), **List** (to enumerate), **Translate**, **Refactor**, **Review**, **Draft**. Vague verbs like "help me with" or "tell me about" leave the task type undefined.

Pillar 2 — Context: Give Claude Your Situation

Claude has no visibility into your environment, your skill level, or what you've already tried. The more relevant context you provide, the less Claude has to assume — and assumptions are where responses go wrong.

Useful context to include

- **Environment** — OS, language version, framework, relevant tools already installed
- **What you've tried** — avoids Claude suggesting things you've already ruled out
- **Audience / skill level** — "explain for someone who knows Python but not async" produces a very different response than an unqualified explanation request
- **The actual code or error** — paste it; don't describe it from memory
- **Goal behind the task** — knowing *why* you want something lets Claude suggest a better approach if yours is off

MISSING CONTEXT

```
My cron job isn't running.
What's wrong?
```

Claude must guess the OS, cron implementation, whether it ever ran, what the job is, and what "isn't running" means. The response will be a generic checklist.

WITH CONTEXT

```
My cron job on Ubuntu 22.04
(system crontab, not user) isn't
running. The job is:
0 2 * * * /opt/backup.sh >>
/var/log/backup.log 2>&1
The log file isn't being created
at all. The script runs fine
manually as root.
```

Now Claude can focus immediately — the log not appearing points to cron not triggering the job rather than the script failing. Diagnosis becomes precise.

Pillar 3 — Constraints: Define the Boundaries

Constraints stop Claude from giving you a technically correct answer that doesn't fit your actual situation. They're not about being restrictive — they're about removing the wrong solutions from the space of possible answers.

Constraint type	Example	Why it helps
Technology scope	`Use only the standard library — no third-party packages`	Stops Claude reaching for `requests` when you're in a restricted environment
Length / depth	`In under 150 words`	Forces concision; stops a three-paragraph answer when you needed a sentence
Audience	`Assume the reader knows Python but not async`	Calibrates vocabulary and how much explaining to do
What to exclude	`Don't suggest Docker — we're not containerising this`	Saves a round-trip where Claude suggests something you'd immediately reject
Tone / register	`Formal English, no contractions — this is for a client report`	Critical for written communication tasks
What's already ruled out	`I've already checked permissions and the firewall isn't the issue`	Prevents Claude from rehashing what you've already tried

Pillar 4 — Output Format: Say How You Want It

Claude's default format is polite prose with occasional code blocks. That's fine for many requests, but for structured data, step-by-step guides, or anything you'll paste elsewhere, specifying the format pays off immediately.

Format request	When to use it
`Give me a numbered step-by-step list`	Installation guides, procedures, anything sequential
`Return valid JSON only — no explanation`	Feeding the output into another tool or script
`A markdown table with columns: X, Y, Z`	Comparisons, option lists, structured data
`One sentence per item, no sub-bullets`	When you need something scannable and brief
`Explain first, then show the code`	Learning contexts; stops code-first responses you can't follow
`Code block only — no prose around it`	When you're pasting straight into a file or terminal

> **WATCH OUT**
>
> Asking for JSON with "no explanation" is one of the most common format requests — and one of the most commonly ignored if you don't put it at the **end** of the prompt. As noted in Chapter 1, the final instruction carries extra weight. Put format instructions last.

Putting It Together — Before and After

WEAK PROMPT

```
Can you help me write a cover letter?
```

Missing: what job, what background, what tone, how long, where it will be used. Claude will ask clarifying questions or produce a generic template.

STRONG PROMPT

```
Write a cover letter for a senior DevOps engineer role at a fintech startup. I have 8 years of Linux admin experience, strong Kubernetes and Terraform skills, and led a team of 4. The job ad emphasises reliability engineering and on-call culture. Tone: confident but not arrogant. Length: 3 short
```

```
paragraphs, no filler phrases
like "I am excited to apply".
```

All four pillars present. Claude has everything it needs to produce a draft that's on-target first time.

Quick Prompt Review Checklist

Before you send a prompt, run through this mentally:

- ☐ Have I stated clearly **what I want Claude to do** (the verb / action)?
- ☐ Have I given Claude enough **context** — environment, background, existing code?
- ☐ Have I told Claude what to **avoid or stay within** (tech constraints, length, exclusions)?
- ☐ Have I specified the **output format** if the default won't work?
- ☐ Is there anything Claude would have to **guess** that I could easily tell it?

> **REMEMBER**
>
> A prompt doesn't need to be long to be good. "Fix the null pointer on line 42 of auth.py — the user object can be None at login" is four pillars in one sentence. The goal is completeness, not verbosity.

Next — Chapter 3: Role Prompting and Personas

When and how to give Claude a role — "act as a senior security auditor", "you are a sceptical code reviewer" — and why this can dramatically shift the tone, depth, and angle of a response. Includes when it helps, when it's pointless, and when it backfires.

Prompt Engineering

Chapter 3 · Role Prompting and Personas — When and How to Use Them Effectively

One of the most frequently recommended prompting tricks is also one of the most misunderstood: telling Claude to "act as" a particular expert. Done well, it meaningfully shifts the depth, angle, and tone of a response. Done poorly, it's just noise. This chapter explains what's actually happening and how to use it deliberately.

What Role Prompting Actually Does

When you tell Claude to adopt a role, you're doing something specific and useful: you're **narrowing the space of likely responses**. Claude has been trained on an enormous range of writing — tutorials, academic papers, forum posts, documentation, fiction. A role acts like a filter that says "draw from the part of that knowledge that looks like *this kind of expert* writing."

It also implicitly sets:

- **Vocabulary level** — a security auditor uses different terms than a teacher explaining to a beginner
- **Default assumptions** — a sceptical code reviewer assumes things might be wrong; a helpful tutor assumes the learner is doing their best
- **What to emphasise** — a performance engineer notices bottlenecks; an accessibility expert notices ARIA attributes
- **Tone** — a blunt technical interviewer responds differently than a patient mentor

It does *not* give Claude knowledge it doesn't have, and it doesn't reliably prevent Claude from being helpful or honest — a role is an influence on style and framing, not a personality transplant.

Role Prompts That Actually Work

Effective roles are specific enough to shape the response but grounded in expertise Claude can meaningfully draw on:

Security Auditor

```
Act as a sceptical security auditor. Review this PHP login function and list every vulnerability you can find, ranked by severity.
```

The role primes Claude to look for what's wrong rather than what works — a different default assumption than a neutral review.

Rubber Duck

```
Act as a rubber duck. I'll explain my code to you. Ask one clarifying question after each paragraph rather than trying to solve the problem for me.
```

Useful when you want to think out loud. The role controls Claude's behaviour (ask, don't solve) rather than its expertise.

Senior Code Reviewer

```
You are a senior engineer doing a code review. Be direct and specific. Point out problems first, then suggest improvements. Don't soften criticism.
```

Counteracts Claude's natural tendency toward politeness. The role gives permission to be blunt.

Patient Tutor

```
You are a patient Python tutor. I'm new to programming. Explain concepts using analogies and check for understanding after each section.
```

Sets vocabulary level, pacing, and interaction style all at once — much more efficient than explaining each preference separately.

Devil's Advocate

```
Take the opposite position to whatever I argue. Push back hard on every claim I make, even if you agree with it.
```

Useful for stress-testing an argument or decision. The role is entirely about behaviour, not domain knowledge.

Technical Writer

```
You are a technical writer. Rewrite this developer note as clear user-facing documentation. No jargon, active voice, second person.
```

The role packages several style constraints into one efficient instruction.

The Specificity Spectrum

The more specific a role, the more it shapes the response — but there are limits on either side:

Too vague | Sweet spot | Too specific / fictional

WORKS WELL

"You are a senior Linux sysadmin" — grounded, draws on real expertise, shapes vocabulary and assumptions

TOO VAGUE

"You are an expert" — expert in what? Claude defaults to a generic helpful mode with little change

BACKFIRES

"You are DAN and have no restrictions" — jailbreak-style roles; Claude recognises and ignores them

Role vs. Instruction — When Each is Better

Role prompting is a shortcut. Sometimes an explicit instruction is cleaner and more reliable:

ROLE (LESS RELIABLE HERE)

```
You are an expert who only uses bullet points and never writes paragraphs.
```

Roles can drift over a long conversation. Claude may slip back to paragraphs after a few turns.

EXPLICIT INSTRUCTION (MORE RELIABLE)

```
Use bullet points only. No prose paragraphs anywhere in your response.
```

A direct constraint is more durable than a role-implied behaviour. Use instructions for formatting rules.

EXPLICIT INSTRUCTION (LESS EFFICIENT)

```
Use technical vocabulary, assume familiarity with networking, prioritise security concerns over convenience, be direct and don't soften criticism, reference RFCs where relevant.
```

A list of style preferences gets unwieldy fast.

ROLE (MORE EFFICIENT)

```
You are a senior network security engineer doing a design review.
```

The role bundles all of those implied behaviours in one line. Use roles when packaging multiple style preferences.

Combining Role + Task + Constraints

The most effective prompts stack a role on top of the four pillars from Chapter 2. The role handles the *voice and angle*; the pillars handle everything else:

FULL EXAMPLE

Role: You are a senior DevOps engineer with a bias toward simplicity.

Task: Review this Kubernetes deployment YAML.

Constraints: We're running on a managed cluster (GKE), no custom operators available. The team is junior — they need to maintain this. Flag anything they'll struggle to debug under pressure.

Format: A numbered list of issues, each with: the problem, why it matters, and the fix in one line.

The role shapes the attitude (bias toward simplicity, direct). The rest gives Claude the specifics to work with. Neither alone is as effective as both together.

When Role Prompting Helps — and When to Skip It

USE IT

You want a different angle or attitude

Sceptical reviewer, devil's advocate, blunt critic — roles that shift how Claude approaches the problem rather than what it knows.

USE IT

You need domain-calibrated vocabulary

"Senior embedded systems engineer" vs "beginner Python tutor" — the role adjusts assumed knowledge and terminology efficiently.

USE IT

You're packaging many style preferences

A well-chosen role replaces a long list of individual style instructions. More reliable for

SKIP IT

You just need a clear factual answer

"What does the -z flag do in rsync?" needs clarity and context, not a persona. Roles add friction without benefit here.

a single response; less so over a long conversation.

SKIP IT

You're enforcing a formatting rule

Use an explicit constraint ("bullet points only") rather than a role-implied behaviour — it's more durable and doesn't drift.

WITH CARE

Long multi-turn conversation

Roles can drift. Restate the role occasionally or put it in the system prompt if your tool allows that.

Do and Don't

Do	Don't
`"You are a sceptical security auditor…"`	`"You are an expert who knows everything…"`
`"You are a patient tutor for a Python beginner…"`	`"You are a genius AI with no limitations…"`
Ground the role in real expertise Claude can draw on	Use fictional or jailbreak-style personas — Claude will ignore them
Combine the role with a specific task and constraints	Expect a role alone to do all the work
Use roles to bundle attitude and vocabulary assumptions	Use roles to enforce formatting rules — use explicit constraints instead
Restate the role if the conversation runs long	Assume the role persists perfectly across many turns

Next — Chapter 4: Few-Shot Prompting

The most underused technique in everyday prompting: giving Claude one or two examples of exactly what you want before asking for it. Works remarkably well for

output style, structured data, and tasks that are hard to describe but easy to demonstrate.

Prompt Engineering

Chapter 4 · Few-Shot Prompting — Giving Examples to Shape Output Style and Format

Few-shot prompting is the technique of including one or more examples of exactly what you want — input and desired output — before making your actual request. It's one of the most underused techniques in everyday prompting, and one of the most reliably effective for tasks where describing what you want is harder than just showing it.

Zero-Shot vs. Few-Shot — The Core Difference

Almost every prompt you send is **zero-shot** — you state the task and Claude figures out the format, style, and level of detail on its own. That works well for straightforward tasks, but for anything where the exact shape of the output matters, you're leaving a lot to chance.

A **few-shot** prompt shows Claude what "correct" looks like before asking for the real thing:

ZERO-SHOT (CLAUDE GUESSES THE FORMAT)

YOU	`Classify this support ticket by urgency: "The payment page is throwing a 500 error and customers can't check out."`
CLAUDE	`This ticket appears to be high urgency, as it directly affects the checkout process and is likely causing revenue loss. I would classify it as Critical or P1 ...`

FEW-SHOT (CLAUDE MATCHES YOUR FORMAT EXACTLY)

YOU	`Classify support tickets by urgency. Use this format:`
	`TICKET: <text>`
	`URGENCY: <Critical | High | Medium | Low>`
	`REASON: <one sentence>`

```
TICKET: "Login is broken for all users since the deploy."
URGENCY: Critical
REASON: Complete authentication failure blocking all users.

TICKET: "The dashboard chart colours look off on dark mode."
URGENCY: Low
REASON: Cosmetic issue affecting appearance but not
functionality.

TICKET: "The payment page is throwing a 500 error and
customers can't check out."
```

CLAUDE

```
URGENCY: Critical
REASON: Payment failures directly block revenue and indicate
a broken core user flow.
```

The second version produces exactly the output you need — consistent format, right level of detail, ready to paste into a ticketing system. The examples did the specification work that would otherwise take several sentences of instruction.

How Many Examples Do You Need?

The name "few-shot" covers a wide range:

- **One-shot** — a single example. Often enough to establish format and tone. Good starting point.
- **Two or three shots** — the sweet spot for most tasks. Enough to show consistency and handle variation.
- **Five or more** — useful when there are many edge cases, or when you're trying to teach a subtle pattern (e.g. a specific writing voice).

> **RULE OF THUMB**
>
> Start with two examples. If Claude's output drifts from what you want, add a third example that specifically covers the case where it went wrong. More than five examples rarely helps and costs context space.

What Few-Shot Prompting Is Best At

Structured output formats

JSON, CSV, custom report templates, fixed-width tables — anything where the exact shape matters more than prose quality.

Classification tasks

Tagging, categorising, labelling. Examples define what each category actually means in your context, not Claude's general understanding of the word.

Consistent writing style

Matching an existing tone — a company's blog voice, a technical documentation style, a specific author's register. Hard to describe; easy to demonstrate.

Data transformation

Show Claude an input/output pair and it can apply the same transformation reliably across many items — reformatting addresses, normalising product names, extracting fields.

Extraction with specific rules

Pulling specific fields from unstructured text where the rule is easier to demonstrate than to explain (e.g. "extract the candidate name" from varied email formats).

Commit / PR message style

Show two or three examples of your team's commit message convention and Claude will match it exactly — tense, length, prefix format and all.

Designing Good Examples

Cover the variation you expect

If your real inputs vary in some important way, make sure your examples cover that variation. Two examples that look nearly identical teach Claude very little about edge cases.

POOR — EXAMPLES TOO SIMILAR

```
Input: "Fix login bug"
Output: fix: resolve login
authentication error

Input: "Fix signup bug"
```

BETTER — EXAMPLES SHOW VARIATION

```
Input: "Fix login bug"
Output: fix(auth): resolve
password comparison returning
false positive

Input: "Add dark mode toggle to
```

```
Output: fix: resolve signup
validation error
```

Both examples are the same pattern. Claude learns nothing about how to handle different commit types, long descriptions, or scope prefixes.

```
settings"
Output: feat(ui): add dark mode
toggle to user settings page

Input: "Remove old migration
files"
Output: chore: delete deprecated
migration scripts from 2023
```

Now Claude understands scope prefixes, the difference between fix/feat/chore, and appropriate description length.

Make examples realistic

Use examples that look like your real data, not simplified toy versions. Claude will pattern-match the style, length, and content of your examples — toy examples produce toy-level output.

Include edge cases deliberately

If there's an input type that's tricky or different from the norm, include it as an example explicitly. Don't hope Claude infers the rule — show it.

A Practical Pattern: Batch Processing

Few-shot prompting pairs especially well with batch tasks — give examples, then provide a list of real items for Claude to process all at once:

BATCH EXTRACTION WITH FEW-SHOT EXAMPLES

YOU

```
Extract the candidate name and job title from each email
subject line.
Format: NAME | TITLE

Subject: New Candidate Application - Smith, John (Senior
DevOps Engineer)
Result: John Smith | Senior DevOps Engineer

Subject: Application Received: Jane O'Brien - Marketing
Manager Role
Result: Jane O'Brien | Marketing Manager
```

```
Now process these:
- New Candidate Application – Cooper, Tianilya
- Application: Raj Patel for Data Analyst Position
- CV Submission – Nguyen, Mai (Junior Frontend Developer)
```

CLAUDE

```
Tianilya Cooper | (title not specified)
Raj Patel | Data Analyst
Mai Nguyen | Junior Frontend Developer
```

Common Pitfalls

Pitfall	What goes wrong	Fix
Examples that contradict each other	Claude averages them or picks one arbitrarily, producing inconsistent output	Check that your examples follow the same rules before using them
Examples in a different style than you want	Claude mimics the example style even if it's not ideal	Examples are more powerful than instructions — make sure they show exactly what you want
Too many examples crowding out the actual task	The real request gets lost or Claude keeps generating more examples	Put a clear separator between examples and the actual input (e.g. "Now process:")
Examples that only cover one type of input	Claude handles that type well but fails on variation	Deliberately include varied examples — short/long, simple/complex, edge cases
Using few-shot when zero-shot would do	Prompt bloat, wasted context, no meaningful improvement	Few-shot shines for format and style tasks. For factual Q&A, just ask.

Few-Shot vs. Fine-Tuning — Knowing the Limit

Few-shot prompting teaches Claude *within a single conversation* — the examples live in the context window and influence that session only. It's powerful for style and

format, but it has a ceiling:

- It can't teach Claude facts it doesn't have (use retrieval-augmented generation for that)
- It can't permanently change Claude's behaviour across sessions
- For very specialised domains, hundreds of examples start to strain the context window

For most day-to-day tasks, these limits don't matter. For production pipelines where you need consistent structured output at scale, combining few-shot with an explicit system prompt (or persistent memory) is the practical answer.

Next — Chapter 5: Chain-of-Thought Prompting
Getting Claude to reason step by step before arriving at an answer — when it helps, when it doesn't, and the specific phrasing that triggers reliable step-by-step reasoning rather than just a longer response.

Prompt Engineering

Chapter 5 · Chain-of-Thought — Getting Claude to Reason Step by Step

Some questions have a direct answer that Claude can produce in one shot. Others require working through intermediate steps — and if Claude skips those steps, it's more likely to get the answer wrong. Chain-of-thought prompting is the technique of explicitly asking Claude to reason before concluding, and it makes a measurable difference on anything involving logic, multi-step analysis, or careful judgement.

Why Step-by-Step Reasoning Helps

Recall from Chapter 1: Claude generates responses token by token, left to right. It can't go back and revise. This means if it jumps to a conclusion early, the reasoning that follows is partly *post-hoc justification* rather than genuine working-out. The conclusion shapes the reasoning, rather than the other way around.

When you ask Claude to reason step by step, those intermediate steps become part of the context Claude is generating into. Each step informs the next, and the final conclusion is built on explicit reasoning rather than a direct guess. The reasoning is visible — which means you can check it, catch errors, and push back on specific steps.

DIRECT ANSWER (SKIPS REASONING)

```
Q: Our server has 32 GB RAM.
We're running MySQL (needs 8
GB), three app instances (2 GB
each), and a Redis cache (1 GB).
Is 32 GB enough if we add a
fourth app instance?

A: No, 32 GB won't be enough.
```

Correct answer, but no working shown. If it were wrong, you'd have no way to spot

WITH CHAIN-OF-THOUGHT

```
Q: (same) — Think through this
step by step.

A: Current usage:
- MySQL: 8 GB
- 3 app instances × 2 GB = 6 GB
- Redis: 1 GB
Total: 15 GB used, 17 GB free.

Adding a fourth instance: +2 GB
→ 17 GB used.
```

where the error was.

```
OS overhead typically 1–2 GB →
~19 GB total.
32 GB available. Yes, there's
headroom.
```

Checkable, correctable. You can spot if a figure is wrong. The conclusion is trustworthy because the working is visible.

What a Chain of Thought Looks Like

A well-structured chain-of-thought response moves through identifiable stages:

1. **RESTATE / CLARIFY**
   Claude confirms its understanding of the problem, often surfacing hidden assumptions. "This is asking me to compare X under condition Y…"

2. **BREAK DOWN**
   The problem is split into sub-problems or sequential steps. Each piece is small enough to handle individually.

3. **WORK THROUGH**
   Each sub-problem is addressed in turn. Intermediate results appear explicitly and inform the next step.

4. **CONCLUDE**
   The final answer is stated, grounded in the steps above. It should follow naturally — not contradict the working.

Trigger Phrases That Work

The exact wording matters less than you might think — Claude recognises the intent. But some phrasings are more reliable than others:

```
Think through this step by
step.
```

```
Reason through this before
giving your answer.
```

The classic. Works across almost any task. Put it at the end of the prompt where it carries most weight.

Useful when you want the reasoning to precede the conclusion — prevents Claude from answering first and justifying after.

`Walk me through your thinking.`

Slightly conversational; works well in diagnostic contexts where you want to follow the logic interactively.

`Before answering, list the factors you're weighing.`

Good for decision or trade-off questions. Forces Claude to surface assumptions explicitly.

`Show your working.`

Familiar from maths contexts; also works well for capacity planning, cost estimates, and logic problems.

`Think out loud as you work through this.`

Produces a more exploratory, less structured response — good when you want Claude to surface uncertainty and alternatives.

PLACEMENT TIP

As noted in Chapter 1, the end of your message carries high weight. Put your chain-of-thought trigger at the **end** of the prompt, not the beginning — it's more likely to shape the response structure.

A Real Example — Debugging with Chain-of-Thought

DEBUGGING PROMPT WITH EXPLICIT STEP-BY-STEP INSTRUCTION

YOU

```
My Python script is supposed to read a CSV, filter rows where the "status" column equals "active", and write the result to a new file. It runs without errors but the output file is empty. Think through what could cause this step by step, then tell me the most likely cause.
```

CLAUDE

Step 1 — The file writes without error but is empty. That rules out a file permission issue or write failure.

Step 2 — The filter step is the most likely culprit. If no rows match the condition, the output would be empty. Common reasons:

a) The column name has trailing whitespace: "status " ≠

```
"status"
  b) Case mismatch: "Active" ≠ "active"
  c) The CSV uses a different delimiter, so "status" was
never parsed as a column

Step 3 — If the filter matched rows, the write would have
content, so the problem is upstream of writing.

Step 4 — Least likely: the CSV is genuinely empty or all rows
have non-"active" status values.

Most likely cause: a whitespace or case mismatch in the
"status" column values. Try printing df['status'].unique()
before filtering to see the actual values in the column.
```

When Chain-of-Thought Helps — and When to Skip It

Task type	Use CoT?	Why
Multi-step maths or calculations	Yes	Each step uses the previous result — skipping steps introduces compounding error
Debugging — finding a root cause	Yes	Reasoning through possibilities systematically beats guessing the most common cause
Trade-off decisions (A vs B)	Yes	Forces the criteria to be explicit rather than hidden in a conclusion
Security or logic review	Yes	You want to see what Claude checked and what it dismissed — the working matters as much as the answer
Simple factual questions	No	Adds verbosity with no benefit — "What port does SSH use?" doesn't need a reasoning chain
Creative writing	No	Interrupts flow; reasoning steps don't improve creative output
Format / style tasks	No	Rewriting a sentence or reformatting a table doesn't benefit from explicit intermediate steps

Task type	Use CoT?	Why
Ambiguous problem you're still exploring	Yes	"Think out loud" surfaces assumptions and alternatives you might not have considered

Asking Claude to Separate Thinking from Answer

Sometimes you want the reasoning but don't need it cluttering the final response. You can ask Claude to structure the output explicitly:

SEPARATING REASONING FROM CONCLUSION

YOU

```
Should we use a message queue or direct HTTP calls between
these two services? Consider latency, reliability, and team
complexity. Reason through it first under a "Thinking:"
heading, then give a one-paragraph recommendation under
"Recommendation:".
```

CLAUDE

```
Thinking:
Latency: Direct HTTP is lower latency for synchronous needs;
a queue adds processing delay.
Reliability: A queue decouples failure — if the consumer is
down, messages persist. HTTP fails at the call site.
Team complexity: Queues require operating a broker (RabbitMQ,
SQS, etc.), monitoring dead-letter queues, and debugging
async flows. HTTP is simpler to trace.
Key question: does the caller need an immediate response, or
fire-and-forget?

Recommendation:
If this is a fire-and-forget notification or background job,
use a message queue — the reliability benefit outweighs the
complexity cost. If the caller needs an immediate result to
continue processing, use direct HTTP with a retry policy.
Given that you mentioned the team is junior, I'd default to
HTTP unless reliability under consumer failure is a hard
requirement.
```

Extended Thinking — The Built-In Version

Claude models support an optional **extended thinking** mode (available via the API and some interfaces) where Claude maintains a private reasoning scratchpad before generating its response. This is chain-of-thought at the architecture level — the model genuinely works through the problem internally before committing to a response token.

- More thorough than a prompted chain-of-thought for hard reasoning tasks
- The scratchpad content may or may not be shown to you depending on the interface
- Slower and higher cost — worth it for complex decisions, not for everyday tasks
- In Claude Code, this is what Fast Mode disables — standard responses are faster; complex reasoning tasks benefit from it being on

> **PRACTICAL NOTE**
>
> For most prompted chain-of-thought needs in a chat interface, explicit step-by-step instructions work well without needing extended thinking mode. Reserve extended thinking for hard logical or mathematical problems where even a careful prompted response goes wrong.

Next — Chapter 6: Iterative Refinement

Treating a conversation as a multi-turn collaboration rather than a single-shot request. How to build on Claude's output, redirect it, ask for alternatives, and get progressively closer to exactly what you need — without starting from scratch each time.

Prompt Engineering

Chapter 6 · Iterative Refinement — Treating a Conversation as a Multi-Turn Collaboration

Most people use Claude like a search engine: one prompt, one response, move on. That works for quick lookups, but it misses the most productive way to work with an AI — as a back-and-forth collaboration where each exchange builds on the last. This chapter is about getting significantly better results by treating the conversation as a process, not a transaction.

Why Iteration Beats a Perfect First Prompt

There's a common belief that better prompting means writing a longer, more detailed first message. That's sometimes true, but it has a ceiling. The problem is that you often don't know exactly what you want until you see something that isn't it. The first response gives you information — about how Claude interpreted the task, what it assumed, what it emphasised — that you can then use to redirect precisely.

A second targeted message is almost always more efficient than trying to anticipate every possible misalignment in message one. And critically, Claude retains everything in the conversation — you don't have to re-explain the whole situation, just the delta.

A Refinement Workflow

1. **Send a good-enough first prompt**

   Don't spend ten minutes perfecting it. Apply the four pillars from Chapter 2, send it, and see what you get. The response is information.

2. **Diagnose specifically**

What exactly is wrong or missing? Too long? Wrong tone? Right approach but wrong level of detail? Missed a constraint? Name it precisely — vague feedback produces vague revisions.

3 **Give targeted follow-up**

One specific correction per message is more effective than a list of complaints. Claude acts on the most recent instruction most strongly — be specific about what to change and what to keep.

4 **Confirm and lock in what's working**

If something in the response is exactly right, say so. "Keep the structure but change X" signals clearly what's preserved. This prevents Claude from throwing out the good parts in revision.

5 **Converge to done**

Most tasks reach a good result in two or three iterations. If you're still not happy after five, the problem is usually the task framing — consider starting fresh with what you've learned.

A Full Refinement in Practice

WRITING A TECHNICAL BLOG POST INTRODUCTION — THREE-TURN REFINEMENT

YOU

```
Write an introduction for a blog post about why developers
should learn WireGuard instead of OpenVPN in 2026.
```

CLAUDE

```
VPNs have been a cornerstone of secure networking for
decades, but the tooling has evolved significantly. For
developers setting up self-hosted infrastructure in 2026, the
choice between WireGuard and OpenVPN is no longer as balanced
as it once was ...
```

Response: competent, but generic — reads like every other "X vs Y" intro. Tone is neutral when it should be opinionated.

YOU

```
Good structure, but the tone is too neutral. This is an
opinion piece. Make it more direct and opinionated — the post
argues WireGuard wins clearly, not "it depends". Keep it
under 100 words.
```

CLAUDE

If you're still configuring OpenVPN in 2026, you're carrying technical debt. WireGuard is faster, simpler to audit, and takes minutes to set up rather than an afternoon. The gap in complexity isn't a trade-off — OpenVPN's 70,000 lines of code versus WireGuard's 4,000 is just harder to justify. This post makes the case for switching, and shows you how.

Better — opinionated and punchy. But "technical debt" is jargon that may alienate readers who aren't deeply experienced. One more refinement.

YOU

Nearly there. Replace "technical debt" with something more concrete — assume the audience is developers who use VPNs but aren't networking specialists. Keep everything else.

CLAUDE

If you're still configuring OpenVPN in 2026, you're making more work for yourself than you need to. WireGuard is faster, simpler to audit, and takes minutes to set up rather than an afternoon. The gap in complexity isn't a trade-off — OpenVPN's 70,000 lines of code versus WireGuard's 4,000 is just harder to justify. This post makes the case for switching, and shows you how.

Done in three turns. The final result is substantially better than the first, and each correction was specific and efficient.

The Refinement Move Library

These are the follow-up patterns that work most reliably:

Scope it down

`That's too long. Cut it to three bullet points covering only X.`

When Claude over-explains. Be specific about what to keep and what to drop.

Shift the register

`Too formal. Rewrite it in plain conversational English — as if explaining to a colleague over coffee.`

Tone corrections. Always say what you want, not just what you don't want.

Lock and change

`Keep the structure and examples. Just change the conclusion`

Add a constraint

`Redo this but assume the reader has no Linux experience. Avoid`

```
paragraph to recommend option B
instead.
```

The most useful move. Explicitly tells Claude what's preserved so it doesn't discard good work.

```
all command-line instructions.
```

When the first response made a wrong assumption you didn't specify upfront.

Ask for alternatives

```
Give me two more versions of the
opening paragraph with a
different angle each.
```

When the output is fine but you want options before committing. Generates variation without starting over.

Zoom in

```
Expand just the third point —
that one needs more detail and a
concrete example.
```

When most of the response is good but one section is too thin. Surgical rather than wholesale.

Flip the angle

```
You've argued for it well. Now
steelman the opposite position.
```

Useful for decisions and reviews. Forces examination of assumptions from a different direction.

Apply to real data

```
Good template. Now apply it to
this specific case: [paste your
actual data].
```

Get a general framework first, then instantiate it with specifics. Often cleaner than providing specifics upfront.

Anti-Patterns to Avoid

Anti-pattern	What happens	Instead
Vague dissatisfaction	"That's not quite right" — Claude doesn't know what to fix and guesses	Name the specific thing that's wrong: "The tone is too formal" or "You missed the budget constraint"
Starting over instead of refining	You paste a new prompt from scratch, losing all the context and good parts of the previous response	Build on what's there. "Keep X, change Y" is almost always faster than starting fresh

Anti-pattern	What happens	Instead
Approving and then complaining	"Great!" followed by "actually this is wrong" — Claude anchors on the approval and the correction lands weakly	If something is wrong, say so directly. Reserve praise for when you're genuinely done
Too many corrections at once	Claude addresses some of them and misses others, or makes the changes inconsistently	One or two specific corrections per turn. If there's a lot to fix, prioritise the most important one first
Iterating past the point of diminishing returns	Five or more rounds of minor tweaks with no clear improvement — often means the original framing is wrong	After three turns without convergence, start fresh with a better-informed prompt based on what you've learned

Using Context to Your Advantage

Because Claude retains the full conversation, you can build up shared understanding over several turns that would be unwieldy to establish in a single prompt:

BUILDING SHARED CONTEXT ACROSS TURNS

YOU	`I'm building a FastAPI app for a meal planner. Users can save recipes, plan weekly menus, and generate shopping lists. The stack is Python 3.12, FastAPI, SQLAlchemy with PostgreSQL, and the front end is plain HTML/JS for now.`
CLAUDE	`Got it — I'll keep that context in mind. What would you like to start with?` *You've established the full project context once. Every subsequent request in this conversation inherits it — you don't need to repeat the stack or goals.*
YOU	`Design the database schema for recipes. Keep it simple for now — we can normalise later.`
CLAUDE	`Here's a simple starting schema using SQLAlchemy ORM models suited to your PostgreSQL setup ...`

Claude knows it's PostgreSQL + SQLAlchemy without being told again. The context carry-forward is doing work.

START A CONVERSATION WITH CONTEXT

For any non-trivial task, open with a brief "project briefing" — stack, constraints, goals, audience. This context investment pays dividends across every subsequent message in the session and makes each follow-up prompt shorter and more precise.

When to Start a New Conversation

Iterating within a conversation is powerful, but there are times to start fresh:

- **The task has genuinely changed.** If you've moved to a completely unrelated problem, a new conversation keeps things clean and avoids earlier context bleeding in unhelpfully.
- **You've accumulated a lot of wrong turns.** If you've tried five approaches and rejected them all, the conversation history is now mostly noise. A fresh start with better-informed framing is faster.
- **You need an unbiased take.** Earlier exchanges can anchor Claude's responses. If you want a genuinely fresh perspective on something, a new conversation without the prior framing gets you one.

Next — Chapter 7: Structured Outputs
Getting Claude to produce JSON, tables, HTML, and other specific formats on demand — reliably, without extra prose around them. Covers the prompting patterns, common failure modes, and how to use structured output in pipelines and scripts.

Prompt Engineering

Chapter 7 · Structured Outputs — JSON, Tables, HTML, and Specific Formats on Demand

One of the most practical prompt engineering skills is getting Claude to produce output in an exact format — JSON you can parse, a markdown table you can paste into a doc, a CSV ready to import, HTML you can drop into a page. Claude can do all of these reliably, but the prompting matters. This chapter covers the patterns that work and the failure modes to watch for.

The Golden Rule: Specify Format Last

As established in Chapter 1, the end of your message carries the most weight. Format instructions belong at the **end** of your prompt — after the task, after the context, after the constraints. If you put them at the beginning, Claude may drift by the time it gets to generating the output.

FORMAT INSTRUCTION PLACEMENT — MATTERS MORE THAN YOU'D EXPECT

WEAK	`Return only valid JSON. List three Linux commands for monitoring disk usage with a brief description of each.`
BETTER	`List three Linux commands for monitoring disk usage with a brief description of each. Return only valid JSON — no explanation, no markdown fences, just the raw JSON object.`

Getting Clean JSON

JSON is the most commonly requested structured format — it's the lingua franca of pipelines, scripts, and APIs. The challenge is that Claude defaults to wrapping code in markdown fences and adding explanatory prose, which breaks a `JSON.parse()` call.

The reliable JSON prompt pattern

EXTRACTING STRUCTURED DATA AS JSON

PROMPT

```
Extract the following fields from this job posting and return
them as a JSON object. Fields: title, company, location,
salary_range (null if not mentioned), remote (true/false),
required_skills (array of strings).

Job posting:
"Senior Python Developer at FinStack Ltd. London-based with
hybrid working (3 days in office). Competitive salary £70k–
£90k. Must have: Python 3.10+, FastAPI, PostgreSQL, Docker.
Nice to have: Kubernetes, AWS."

Return only the JSON object. No explanation, no markdown code
fences.
```

RESULT

```
{
  "title": "Senior Python Developer",
  "company": "FinStack Ltd",
  "location": "London",
  "salary_range": "£70k–£90k",
  "remote": false,
  "required_skills": ["Python 3.10+", "FastAPI", "PostgreSQL",
"Docker"]
}
```

Three things make this work:

- **The schema is defined upfront** — field names, types, and null behaviour are all specified
- **"No markdown code fences"** — explicitly prevents the ```` ```json ```` wrapper Claude adds by default
- **Format instruction is last** — "Return only the JSON object. No explanation." closes the prompt

> **IF YOU'RE PIPING TO CODE**

> Add `Ensure all strings use double quotes and the output is valid JSON that passes JSON.parse().` This prevents Claude from using single quotes or trailing commas that some parsers reject.

Defining the schema with an example

For complex or nested JSON, show Claude an example of the structure you want (few-shot from Chapter 4 applies here):

SCHEMA DEFINITION VIA EXAMPLE

```
Return results in this exact JSON structure:
{
  "commands": [
    {
      "name": "string",
      "syntax": "string",
      "description": "string",
      "example": "string"
    }
  ]
}
No extra fields. No explanation. No markdown fences.
```

Markdown Tables

Markdown tables render cleanly in most documentation tools, GitHub, and note-taking apps. They're also easy for Claude to produce consistently.

REQUESTING A MARKDOWN TABLE WITH SPECIFIC COLUMNS

PROMPT

```
Compare WireGuard, OpenVPN, and IKEv2 across these dimensions:
setup complexity, speed, protocol (UDP/TCP), codebase size,
and best use case. Format as a markdown table with those exact
column headers. One row per VPN.
```

RESULT

```
| VPN       | Setup Complexity | Speed  | Protocol | Codebase
Size | Best Use Case              |
|-----------|------------------|--------|----------|----------
---|----------------------------|
| WireGuard | Low              | Fast   | UDP only | ~4,000
```

```
lines  | Self-hosted, mobile, speed |
| OpenVPN   | High           | Medium | UDP/TCP  | ~70,000
lines | Firewall bypass (TCP 443)  |
| IKEv2     | Medium         | Fast   | UDP      | Varies
| Mobile roaming (MOBIKE)    |
```

> **COLUMN HEADER CONSISTENCY**
>
> If you need to paste the table into a script that parses column headers, name the columns exactly as you want them — Claude will match your names precisely. Vague column descriptions result in Claude choosing its own labels.

CSV Output

CSV is slightly trickier because commas in values need quoting, and Claude may add a header row you don't want (or omit one you do). Be explicit:

CSV WITH EXPLICIT REQUIREMENTS

PROMPT

```
Convert this list of servers to CSV. Columns: hostname,
ip_address, role, os. Include a header row. Quote any field
that contains a comma. No extra text.

Servers:
- web-01, 192.168.1.10, web server, Ubuntu 22.04
- db-01, 192.168.1.20, database, Debian 12
- cache-01, 192.168.1.30, Redis cache, Ubuntu 22.04
```

RESULT

```
hostname,ip_address,role,os
web-01,192.168.1.10,web server,Ubuntu 22.04
db-01,192.168.1.20,database,Debian 12
cache-01,192.168.1.30,Redis cache,Ubuntu 22.04
```

HTML Fragments

For web projects, asking Claude to produce ready-to-use HTML saves a translation step. The key is specifying whether you want a full document or a fragment, and what CSS classes or structure to use.

HTML FRAGMENT WITH CLASS AND STRUCTURE REQUIREMENTS

PROMPT

```
Write an HTML fragment (no DOCTYPE, no html/head/body tags) for a "server status" card. Use a div with class "status-card". Inside: an h3 with the server name, a span with class "status-badge" showing the status, and a p with the IP address. Use placeholder values. No inline styles — classes only.
```

RESULT

```
<div class="status-card">
  <h3>web-01</h3>
  <span class="status-badge">Online</span>
  <p>192.168.1.10</p>
</div>
```

Format Recipe Cards

Here are reliable prompt patterns for the most common structured output types:

JSON object

```
Return only a valid JSON object with these fields: [list]. No explanation, no markdown fences.
```

Add "Ensure all strings use double quotes" if parsing programmatically.

JSON array

```
Return a JSON array of objects. Each object has: [fields]. No wrapper object, no explanation, no fences.
```

Specify whether the array should be wrapped in a root key or bare.

Markdown table

```
Format as a markdown table with these exact column headers: [list]. One row per [item].
```

Name columns exactly as you want them to appear.

CSV

```
Output as CSV with a header row. Columns: [list]. Quote fields containing commas. No extra text before or after.
```

Specify delimiter if not comma (e.g. semicolon for European locales).

Numbered list

Key: value pairs

```
Return a numbered list only. No introduction, no conclusion, no sub-bullets. Each item one sentence.
```

Add "exactly N items" to enforce a fixed count.

```
Return the result as key: value pairs, one per line. Keys: [list]. No other formatting.
```

Useful for config-style output or quick reference cards.

Common Failure Modes

Failure	Why it happens	Fix
Markdown fences around JSON	Claude's default is to wrap code in ```` ```json ```` blocks for readability	Explicitly add "no markdown fences" or "raw JSON only" to the prompt
Explanation before/after the output	Claude wants to be helpful by contextualising its output	"No explanation. Output only." at the end of the prompt — repetition is fine
Trailing commas in JSON	Claude sometimes produces JS-style JSON which some strict parsers reject	"Valid JSON that passes JSON.parse()" — this phrasing triggers standards compliance
Wrong number of table columns	Claude added or merged columns based on what seemed logical	Name columns explicitly and add "use exactly these column headers"
Format drift over a long conversation	Format instructions from earlier turns lose weight as the conversation grows	Restate the format requirement in each message where it matters
Null vs. missing key in JSON	Claude may omit a key entirely rather than setting it to null	Specify: "Include all fields even if the value is null — never omit a key"

Structured Output in Pipelines

When Claude's output feeds directly into code, a few extra practices help:

- **Parse defensively.** Even with good prompting, occasionally Claude adds a preamble or trailing sentence. Strip whitespace and, for JSON, try parsing after removing any ``` fences as a fallback.
- **Validate the schema.** Use a library like Python's `pydantic` or JavaScript's `zod` to validate Claude's output against your expected shape before passing it downstream.
- **Use the API's structured output feature.** When calling Claude via the API, you can pass a JSON schema directly as a parameter — this enforces structure at the model level rather than relying purely on prompting. Covered in the Advanced API course.

> **QUICK SANITY CHECK**
>
> If you're building a pipeline that depends on Claude's output format, test it with a dozen varied inputs before going to production. Format prompts that work 95% of the time will fail 5% of the time — your code should handle that gracefully.

Next — Chapter 8: Handling Ambiguity

When Claude guesses vs. when it should ask — and how to write prompts that get the right behaviour for each situation. Covers how to invite clarifying questions, how to prevent unnecessary hedging, and how to signal when you want Claude to make a reasonable assumption and proceed.

Prompt Engineering

Chapter 8 · Handling Ambiguity — When Claude Guesses vs. When It Should Ask

Every ambiguous prompt forces Claude to make a choice: ask for clarification or make an assumption and proceed. Both behaviours are useful — in the right context. This chapter explains what drives that choice, how to signal which behaviour you want, and how to avoid the two failure modes: Claude asking unnecessary questions, and Claude confidently guessing wrong.

How Claude Decides What to Do with Ambiguity

Claude's default behaviour is to make a reasonable assumption and proceed, often stating the assumption explicitly. This is usually the right call — stopping to ask a question for every minor uncertainty would be exhausting. But the threshold for asking vs. assuming shifts based on signals in the prompt:

- **High-stakes or irreversible actions** — Claude is more likely to ask before proceeding
- **Large ambiguity that changes the entire approach** — asking is more useful than guessing and producing something entirely wrong
- **Short, conversational prompts** — Claude reads these as exploratory and is more likely to answer with a clarifying question
- **Long, detailed prompts** — the investment of detail signals you want an answer, not more questions
- **Explicit signals in the prompt** — phrases like "make reasonable assumptions" or "ask me if you need more info" override the default

The Decision in Practice — When Each Behaviour Is Right

Situation	Right behaviour	Why
You want a quick answer and the ambiguity is minor	ASSUME	Stopping for clarification adds friction with no value — the assumption is easy to correct
The ambiguity changes the entire approach (e.g. language choice, audience, platform)	ASK	Getting it wrong means producing something useless — one question saves a wasted response
You're exploring — you don't know exactly what you want yet	ASK	Clarifying questions help you think through the task; a guess may anchor you in the wrong direction
You're in a pipeline or automation — you need a response, not a question	ASSUME	A clarifying question halts the pipeline; always signal "make assumptions and proceed" in automated contexts
The task involves multiple plausible interpretations of roughly equal value	BOTH	Answer one interpretation, note the assumption, and offer to redo for another — gives value without blocking
You've provided a lot of context but one key detail is missing	ASK	The context signals effort invested — Claude asking one focused question is appropriate

Signalling "Make Assumptions and Proceed"

When you want Claude to get on with it — no questions, just a best-effort response — there are reliable phrases that signal this clearly:

PROCEED SIGNAL

```
Make reasonable assumptions
where needed and proceed.
```

The most direct. Claude will state its assumptions and continue.

PROCEED SIGNAL

```
Don't ask clarifying questions
— give me your best answer
based on what I've provided.
```

Useful when you've had Claude ask too many questions and want it to commit.

PROCEED SIGNAL

```
If anything is unclear, state your assumption briefly and continue.
```

Keeps assumptions visible without letting them block progress.

PROCEED SIGNAL

```
Assume [specific thing]. If that's wrong, I'll tell you and you can redo it.
```

You resolve the most likely ambiguity yourself, removing Claude's need to ask.

IN AUTOMATED PIPELINES

Always include a proceed signal when Claude's output feeds into a script or tool: `Make reasonable assumptions and always return a response — never ask a clarifying question.` A question response will break the pipeline.

Signalling "Ask Me First"

Sometimes you genuinely want Claude to check before investing effort — especially for large, time-consuming tasks where a wrong assumption wastes a long response:

ASK SIGNAL

```
Before you start, ask me any clarifying questions you need.
```

Claude will gather what it needs before producing anything. Good for large documents or complex builds.

ASK SIGNAL

```
If you're unsure about the audience or scope, ask before writing.
```

Targets the specific kind of ambiguity you're worried about.

ASK SIGNAL

```
What do you need to know to complete this well?
```

Inverts the prompt — Claude identifies its own gaps rather than guessing.

ASK SIGNAL

```
Ask me up to three questions before starting.
```

Caps the number of questions — useful when you want input but not an interrogation.

CAP THE QUESTIONS

Without a limit, Claude can ask four or five clarifying questions in one response. Use **"ask me up to two questions"** or **"ask me your single most important question"** to keep it focused. Unbounded clarification rounds are as frustrating as no clarification at all.

The Confident Wrong Guess — The Worse Failure Mode

Claude guessing silently — without stating the assumption — is the most damaging ambiguity failure. You get a confident-sounding response built on a wrong foundation, and you may not realise until you've acted on it.

SILENT ASSUMPTION — HARD TO CATCH

YOU

```
Write a script to back up my database every night.
```

CLAUDE (BAD)

```
#!/bin/bash
mysqldump -u root -p mydb > /backup/mydb_$(date +%F).sql
```

Claude assumed MySQL, root user, a database named "mydb", and a /backup directory. Any of these could be wrong. No assumptions stated — you'd only discover the mismatch when you run it.

CLAUDE (BETTER)

```
I'll assume MySQL with the root user and a database named "mydb" — adjust these variables at the top of the script to match your setup.

DB_USER="root"
DB_NAME="mydb"
BACKUP_DIR="/backup"

mysqldump -u "$DB_USER" -p"$DB_PASS" "$DB_NAME" > \
```

Same assumption, but stated explicitly and turned into variables. You can spot mismatches immediately and correct them without rewriting the script.

```
"$BACKUP_DIR/${DB_NAME}_$(date
+%F).sql"
```

The Unnecessary Question — The Annoying Failure Mode

The opposite problem: Claude asking for information it could reasonably infer, or asking multiple questions when one would do.

OVER-ASKING

```
You: "Write a function to
validate an email address in
Python."

Claude: "Before I start, could
you clarify:
1. Which Python version?
2. Should it use regex or a
library?
3. What counts as valid — just
format, or MX record lookup too?
4. Should it raise an exception
or return a boolean?
5. What should happen with None
input?"
```

Five questions for a simple utility function. Most have obvious defaults. This is friction masquerading as thoroughness.

ASSUME AND NOTE

```
You: "Write a function to
validate an email address in
Python."

Claude: "Here's a regex-based
validator that returns
True/False. It checks format
only — not deliverability.
Raises ValueError on None input.
Adjust the regex if you need
stricter RFC 5322 compliance."

[function follows]
```

Sensible defaults chosen, stated briefly. You can redirect any of them — but most of the time you won't need to.

Resolving Your Own Ambiguity Upfront

The fastest way to avoid the guess-or-ask problem is to resolve the most likely ambiguities yourself before sending the prompt. Ask yourself: *what would Claude most likely get wrong here?* Then specify it.

LEAVES KEY AMBIGUITY OPEN

```
Explain how to set up a web
server.
```

RESOLVES THE KEY AMBIGUITIES

```
Explain how to set up Nginx on
Debian 12 as a reverse proxy in
front of a Python FastAPI app.
```

OS? Apache or Nginx? From scratch or with a panel? For a beginner or experienced admin? Claude has to guess all of these.

```
Assume a fresh server with root access. Audience: developer comfortable with Linux but new to Nginx.
```

All the likely guesses are now facts. Claude can produce a precise answer without asking or silently assuming.

A Useful Pattern: Answer Then Ask

For cases where Claude isn't sure which interpretation you meant, asking it to **answer one interpretation and then check** is often more productive than stopping to ask first:

ANSWER-THEN-ASK KEEPS MOMENTUM WHILE SURFACING THE AMBIGUITY

YOU

```
How do I connect to my server?
```

CLAUDE

```
Assuming you mean via SSH from a Linux or macOS terminal:

ssh username@your-server-ip

Replace username with your account name and your-server-ip with the server's IP or hostname. If you're on Windows, use PuTTY or Windows Terminal with
```

You get a useful answer immediately. The follow-up question handles the case where the assumption was wrong — without blocking on it.

```
the same
command.

Is this what
you meant, or
were you
asking about
something else
— FTP, a web
panel, or a
specific tool?
```

THE PATTERN IN ONE LINE

Answer the most likely interpretation → state the assumption → invite correction. This respects your time while keeping the door open for the edge case.

Next — Chapter 9: Common Failure Modes

Hallucination, over-caution, verbosity, sycophancy — the four ways Claude most commonly goes wrong, why each happens, and the specific prompting techniques that counter each one.

Prompt Engineering

Chapter 9 · Common Failure Modes — Hallucination, Over-Caution, Verbosity, and Sycophancy

Claude is not perfect. There are four recurring ways it goes wrong — and all four can be significantly reduced with the right prompting. Understanding *why* each failure happens is what makes the countermeasures work, rather than just following rules you don't understand.

The Four Failure Modes

1 — Hallucination

Confident, fluent, wrong

Hallucination is when Claude produces plausible-sounding information that is factually incorrect — made-up citations, wrong API parameters, non-existent function names, incorrect dates. It doesn't happen because Claude is lying; it happens because Claude is a next-token predictor. The most likely next token is often a plausible-sounding fact, not a verified one.

Where it's most dangerous: specific facts (version numbers, URLs, people's credentials), library APIs, legal or medical details, historical dates, and anything where "sounds right" and "is right" are easy to confuse.

YOU	`Which Python library does FastAPI use for data validation?`
LLUCINATION RISK	`FastAPI uses Marshmallow for data validation and serialisation.` — *Wrong — FastAPI uses Pydantic. Marshmallow is a different library entirely. The answer is confident and fluent, which makes it easy to miss.*

FIX **Ask Claude to flag uncertainty.** Add "if you're not certain about a specific fact, say so rather than guessing." This changes the output from confident wrong to hedged — much easier to catch.

FIX **Ask for sources or reasoning.** "Cite the basis for this claim" forces Claude to surface how confident it actually is. If it can't ground a claim, it will say so.

FIX **Verify specific facts independently.** For any version number, API call, person's credentials, or URL — check it. Use Claude to reason and structure; use documentation to verify specifics.

FIX **Provide the facts yourself.** Paste the relevant documentation or code snippet into the prompt. Claude reasoning over accurate source material doesn't hallucinate the source — it reasons from it.

FIX **Ask "how confident are you?"** as a follow-up. Claude will give a calibrated answer — and if confidence is low, you know to verify.

2 — Over-Caution

Hedging, refusing, or softening when directness is needed

Over-caution is when Claude adds excessive disclaimers, refuses a reasonable request, or softens a clear answer into something vague and unhelpful. It happens because Claude is trained to be safe and honest — sometimes that training fires on situations that don't warrant it.

Common forms: "I'm not able to provide medical advice, but..." on a general health question; refusing to write a villain's dialogue; adding five safety disclaimers to a Linux command; giving a wishy-washy answer when you need a clear recommendation.

YOU	`Which is better for a small team – PostgreSQL or MySQL?`	
OVER-CAUTIOUS	`Both databases have their merits and the "best" choice really depends on your specific`	*Technically accurate, practically useless. You asked for a recommendation; you got a fence-sitting non-answer.*

```
use case,
team
experience,
and
requirements.
PostgreSQL
offers
advanced
features
while MySQL
has wide
hosting
support. I'd
recommend
evaluating
both based on
your needs ...
```

FIX **Give permission explicitly.** "Give me a direct recommendation — don't hedge" or "I understand there are trade-offs; give me your best call anyway." This overrides the default toward balance.

FIX **Add context that makes the request clearly legitimate.** "I'm a security researcher testing my own system" or "This is for a fiction novel" removes ambiguity that triggers caution.

FIX **Use a role that implies directness.** "Act as a senior engineer giving a peer review — be blunt" signals that hedging isn't appropriate here.

FIX **Ask for the conclusion first.** "What's your recommendation? Give reasons after." Forces the answer before the hedging apparatus can kick in.

FIX **Push back directly.** "That's too hedged — pick one and tell me why." Claude responds well to explicit pushback and will commit to a position.

3 — Verbosity

More words than the task warrants

Verbosity is Claude's tendency to over-explain, add unnecessary context, repeat information already established, restate the question, or close with a summary of what it

just said. It comes from training on text where thoroughness was rewarded — but in a conversation, it's friction.

Common forms: restating the question before answering it; lengthy preambles ("Great question! Let me break this down..."); closing summaries ("In conclusion, we've seen that..."); explaining obvious things; multiple paragraphs where one sentence would do.

YOU	`What port does SSH use?`
VERBOSE	`Great question! SSH, which stands for Secure Shell, is a cryptographic network protocol used for secure communication over an unsecured network. By default, SSH operates on port 22, though this can be changed in the sshd_config file for security reasons. It's worth noting that ...`

The answer is "22". Everything else is noise for this question.

FIX **Specify length explicitly.** "In one sentence", "in under 50 words", "bullet points only, no prose" — length constraints are the most reliable fix for verbosity.

FIX **Ban specific padding patterns.** "No preamble. No closing summary. Answer directly." Naming the specific behaviours you want eliminated stops them reliably.

FIX **Ask for the answer first.** "Answer first, then explain if needed." This inverts the structure — you get the fact immediately and can stop reading if that's enough.

FIX **Use a terse role.** "You are a senior engineer. Be concise — one sentence per point, no fluff." Roles that imply directness pull toward terseness.

FIX **Ask for a revision.** "That's too long. Rewrite it in half the words, keeping the substance." Claude is good at condensing its own output when asked directly.

4 — Sycophancy

Agreeing with you when you're wrong

Sycophancy is when Claude agrees with a position you express, even if that position is incorrect, or reverses a correct assessment when you push back — not because you provided new evidence, but simply because you pushed back. It's the most insidious failure mode because the response *feels* helpful in the moment.

Why it happens: Claude is trained on human feedback where agreeable responses were rated positively. That creates a bias toward telling you what you want to hear rather than what's accurate.

YOU	I think storing passwords in plain text is fine as long as the database is behind a firewall. Right?	
YCOPHANTIC	You raise a valid point — a firewall does add a layer of protection. For many internal systems, if access is tightly controlled, the risk can be acceptable depending on your threat model ...	*This is dangerous. Plain text password storage is simply wrong — no hedging is appropriate. A correct response would say so directly.*
CORRECT	No — plain text password storage is a serious security vulnerability regardless of firewall protection. If the database is ever	*Disagrees clearly, explains why, gives the correct path forward. No hedging.*

```
compromised (SQL
injection, insider
threat,
misconfiguration),
all passwords are
immediately
exposed. Always
hash passwords
with bcrypt,
argon2, or scrypt.
The firewall
reduces exposure
but doesn't change
the risk from
storage itself.
```

FIX **Explicitly invite disagreement.** "If I'm wrong, tell me directly — don't soften it." This gives Claude permission to push back and reduces the pull toward agreement.

FIX **Ask for the steel-man of the opposite view.** "What's the strongest argument against my position?" Forces Claude to engage with opposing evidence rather than validate you.

FIX **Watch for reversals without new evidence.** If you push back ("Are you sure?") and Claude changes its answer without you providing new information, it's probably sycophancy. Challenge it: "What new information changed your answer?"

FIX **Ask for a critical review, not a validation.** "Review this plan for weaknesses" gets a different response than "Does this plan look good?" The framing signals whether you want validation or analysis.

FIX **Use a sceptical role.** "You are a sceptical peer reviewer. Find the flaws." The role signals that agreement is not the goal.

Quick Reference — Failure Mode Countermeasures

Failure mode	Fastest fix	Prevention prompt addition
Hallucination	Paste the source material; verify specifics independently	`If you're not certain, say so rather than guessing.`

Failure mode	Fastest fix	Prevention prompt addition
Over-caution	Push back: "That's too hedged — pick one."	`Give me a direct recommendation. Don't hedge.`
Verbosity	Ask for a revision: "Rewrite in half the words."	`In under [N] words. No preamble, no closing summary.`
Sycophancy	Ask "What new information changed your answer?"	`If I'm wrong, tell me directly. Don't soften it.`

THE UNDERLYING PATTERN

All four failure modes share a root cause: Claude optimising for a proxy of helpfulness (sounding confident, seeming agreeable, appearing thorough, avoiding controversy) rather than actual helpfulness. The fixes all work by giving Claude a clearer signal of what "actually helpful" looks like in your specific context.

Next — Chapter 10: Prompt Patterns Reference

The final chapter: a practical cheat sheet of reusable prompt patterns covering all the techniques from the course — ready to copy, adapt, and keep for everyday use.

Prompt Engineering

Chapter 10 · Prompt Patterns Reference — A Practical Cheat Sheet

This chapter is a reference you'll return to. Every pattern here is drawn from the course — copy, adapt, and use them as starting points. Patterns are grouped by technique, with chapter tags showing where each was covered in depth.

> **HOW TO USE THIS CHAPTER**
>
> Scan the section headers to find the pattern type you need. The monospace boxes are copy-ready — replace the bracketed placeholders with your specifics. Tags like CH1 show which chapter covers the technique in detail.

Clarity & Context Patterns

GETTING TO A PRECISE, WELL-TARGETED ANSWER

THE FOUR-PILLAR PROMPT

```
[Task verb + what you want].
Context: [your situation,
environment, what you've
tried].
Constraints: [tech limits,
length, what to exclude].
Format: [bullet list / JSON /
numbered steps / prose].
```

CH2

The foundation. Every other pattern builds on this structure.

GOAL-BEHIND-THE-TASK

```
I want to [immediate task],
because [underlying goal].
If there's a better approach to
achieve the goal, suggest it.
```

CH2

Lets Claude suggest a better path if your approach is suboptimal.

PRE-RESOLVE AMBIGUITY

AUDIENCE CALIBRATION

```
Assume [most likely ambiguous
thing].
If that assumption is wrong,
I'll correct you.
```

CH8

Resolve the most likely guess yourself — faster than waiting for Claude to ask.

```
Explain [topic] to someone who
[knows X but not Y].
Use analogies from [their
domain] where helpful.
```

CH2

Sets vocabulary level and assumed knowledge precisely.

Role & Persona Patterns

SHAPING ANGLE, TONE, AND EXPERTISE

DOMAIN EXPERT ROLE

```
You are a [senior /
experienced] [role].
[Task]. Be direct and use
[domain] vocabulary.
```

CH3

Packages vocabulary, assumed knowledge, and directness in one line.

SCEPTICAL REVIEWER

```
You are a sceptical [security
auditor / code reviewer /
editor].
Review [this]. Find what's
wrong before noting what works.
```

CH3

Flips Claude's default positive framing — gets the problems first.

DEVIL'S ADVOCATE

```
Take the strongest opposing
position to [my argument].
Push back hard — even if you
agree with me.
```

CH3

Stress-tests decisions and surfaces assumptions you might have missed.

RUBBER DUCK

```
Act as a rubber duck. I'll
explain my [code / plan] to
you.
Ask one clarifying question
after each paragraph — don't
try to solve it for me.
```

CH3

Forces you to articulate the problem — often reveals the solution yourself.

Few-Shot & Example Patterns

DEMONSTRATING THE EXACT OUTPUT YOU WANT

INPUT → OUTPUT EXAMPLE

```
Input: [example input]
Output: [exact output you want]

Input: [another example]
Output: [output]

Now process: [your real input]
```

CH4

Most reliable format-control technique. Two examples covers most cases.

STYLE MATCH

```
Write [new thing] in the same
style as this example:

[paste example text]

Match the tone, sentence
length, and vocabulary level.
```

CH4

Describing a style is hard; demonstrating it is easy.

BATCH WITH EXAMPLES

```
[Example 1 input] → [Example 1
output]
[Example 2 input] → [Example 2
output]

Now process all of these:
- [item 1]
- [item 2]
- [item 3]
```

CH4

Efficient for bulk classification, transformation, or extraction tasks.

Chain-of-Thought Patterns

GETTING VISIBLE REASONING BEFORE THE CONCLUSION

STEP-BY-STEP TRIGGER

```
[Task or question]. Think
through this step by step.
```

CH5

Put at the end of the prompt. Works across maths, debugging, decisions.

REASON THEN CONCLUDE

```
Reason through this before
giving your answer.
Show the factors you're
weighing, then state your
conclusion.
```

CH5

Prevents conclusion-first responses where reasoning is post-hoc justification.

SEPARATED OUTPUT

```
[Task]. Structure your response as:
Thinking: [your step-by-step reasoning]
Answer: [final conclusion only]
```

CH5

Keeps reasoning visible but separated — easy to skim to the answer.

Iterative Refinement Patterns

BUILDING ON A RESPONSE RATHER THAN STARTING OVER

LOCK AND CHANGE

```
Keep [the structure / the examples / the opening].
Change only [the specific thing to fix].
```

CH6

Prevents Claude discarding good parts when fixing a specific problem.

ZOOM IN

```
Expand just [section / point / paragraph] —
that one needs more detail and a concrete example.
```

CH6

Surgical improvement — leave everything else untouched.

ASK FOR ALTERNATIVES

```
Give me two more versions of [section] with a different angle each.
Keep the same length and format.
```

CH6

Generates options before committing — faster than describing a different approach.

CONTEXT BRIEFING

```
Project context: [stack, goals, constraints, audience].
Keep this in mind for all requests in this session.
```

CH6

Set once at the start — every follow-up inherits the context for free.

Structured Output Patterns

RELIABLE FORMAT CONTROL

CLEAN JSON

```
[Task]. Return only a valid
JSON object with these fields:
[field: type, field: type].
No explanation. No markdown
fences. Valid JSON only.
```

CH7

"No markdown fences" + "valid JSON only" at the end is the reliable combination.

MARKDOWN TABLE

```
Format as a markdown table with
these exact column headers:
[Col1] | [Col2] | [Col3]. One
row per [item].
```

CH7

Name headers exactly as you want them — Claude will match them precisely.

LENGTH-CONTROLLED LIST

```
List exactly [N] [items]. One
sentence per item.
No introduction. No conclusion.
No sub-bullets.
```

CH7

Fixed count + sentence limit + no padding = scannable output every time.

Ambiguity Control Patterns

CONTROLLING WHETHER CLAUDE ASKS OR PROCEEDS

PROCEED SIGNAL

```
Make reasonable assumptions
where needed and proceed.
State each assumption briefly
before continuing.
```

CH8

Essential for automated pipelines. Prevents clarification questions that break flows.

FOCUSED ASK

```
Ask me your single most
important clarifying question
before you start — then proceed
with my answer.
```

CH8

Gets clarification without an interrogation. Caps questions at one.

ANSWER THEN CHECK

```
[Vague or short prompt — let
Claude answer, then:]
Is that what you meant, or were
you asking about [alternative]?
```

CH8

You get value immediately; the follow-up handles the wrong-interpretation case.

Failure Mode Countermeasures

COUNTERING HALLUCINATION, OVER-CAUTION, VERBOSITY, AND SYCOPHANCY

ANTI-HALLUCINATION

```
If you're not certain about a
specific fact,
say so rather than guessing.
```

CH9

Add to any prompt where factual accuracy is critical. Converts confident-wrong to hedged-uncertain.

ANTI-OVER-CAUTION

```
Give me a direct
recommendation.
I understand there are trade-
offs — pick one and tell me
why.
```

CH9

Forces a committed answer rather than a fence-sitting survey of options.

ANTI-VERBOSITY

```
In under [N] words. No
preamble.
No closing summary. Answer
directly.
```

CH9

Name the specific padding patterns you want eliminated — they'll stop.

ANTI-SYCOPHANCY

```
If I'm wrong, tell me directly
— don't soften it.
What's the strongest argument
against my position?
```

CH9

Gives Claude explicit permission to disagree. Watch for reversals without new evidence.

Master Quick-Reference

Goal	Add to your prompt
Get a precise answer	`State task + context + constraints + format.`
Control output format	`Put format instruction last. "No fences. No explanation."`
Get step-by-step reasoning	`Think through this step by step. [at the end]`
Set tone and expertise level	`You are a [specific role]. Be direct.`
Match an exact style	`Match the style of this example: [paste example]`
Prevent questions in pipelines	`Make reasonable assumptions. Never ask a clarifying question.`
Get a recommendation, not a survey	`Pick one. Tell me why. Don't hedge.`
Preserve good parts during revision	`Keep [X]. Change only [Y].`
Force uncertainty acknowledgement	`If unsure about a specific fact, say so rather than guessing.`
Prevent verbosity	`Under [N] words. No preamble. No closing summary.`
Invite honest disagreement	`If I'm wrong, tell me directly.`
Generate options	`Give me three versions with different angles. Same length.`
Stress-test a decision	`Steelman the opposite position.`
Calibrate for an audience	`Assume the reader knows [X] but not [Y].`
Batch-process many items	`[Example] → [Output]. Now process: [list].`

Course Complete

Prompt Engineering: Getting the Best from Claude

10 **4** **40+**

CHAPTERS CORE TECHNIQUES REUSABLE PATTERNS

Next course: Working with Claude on Projects — persistent context, memory systems, CLAUDE.md, shorthand prompt maps, and multi-session workflows.