The Claude Decision Guide

7 forks in the road that separate Claude power-users from tourists

By **Jackson Lai** — author of 300+ Claude Code skills

For anyone who's used Claude (or ChatGPT) for at least a week and feels like they're not getting the leverage other people seem to be getting.

Why this is a *decision* guide, not a feature guide

Feature guides tell you what buttons exist. They look like: "Claude has chats, projects, Code, skills, hooks, subagents, MCP, prompt caching..." A list. Useful for orientation. Not useful when you're staring at a real task at 9pm on a Tuesday.

What stops most people from going from tourist to power-user isn't ignorance of features. It's not knowing which fork to take when a decision shows up in the moment.

This guide is **7 decisions**. Each one is a fork in the road you'll hit this week if you use Claude seriously. Each fork has:

- **The fork** — what you're choosing between
- **Symptom you picked wrong** — how to recognize it in your own work
- **The right call (and why)** — what I'd do, and the reasoning
- **The 30-second test** — a tiny experiment to feel the difference

I've shipped 300+ Claude Code skills since 2025. Most of what's below, I learned by getting one of these forks wrong, badly, in front of real consequences.

Fork 1 — Chat, or terminal?

The fork: when you have a Claude task in front of you — do you go to claude.ai (the chat tab) or to Claude Code (your terminal)?

Symptom you picked wrong: you find yourself copy-pasting Claude's output from a browser tab into your IDE, file by file, line by line. If "copy from browser → paste into editor" is showing up anywhere in your loop, you've picked chat for a terminal job.

The right call — pick by the verb in your task.

- Verb is **ask, explain, draft, brainstorm, summarize** → chat is right
- Verb is **edit, create, run, deploy, fix-across-files, refactor** → terminal is right

The reasoning: chat-Claude can read what you paste, but it can't *act* on your computer. Terminal-Claude can read your files, edit them in place, run your tests, and report the actual output — without you in the middle as a copy-paste relay.

The 30-second test: next time you're about to paste 200+ lines of code into a chat window, stop. That's the signal you should be in Claude Code instead. Ask yourself: am I asking, or am I editing? If it's editing, switch tools.

Fork 2 — Continue the conversation, or restart?

The fork: this conversation has been going for a while and Claude is starting to feel "off." Do you keep going (give more context, re-explain things), or do you start a fresh one?

Symptom you picked wrong:

- Claude repeats a mistake you've already corrected
- Claude mixes context from earlier in the conversation with something new ("but earlier you said...")
- *You* find yourself scrolling up to remind it of something it should remember

The right call — apply the "scroll-up tax." The moment you scroll up to remind Claude of something earlier — or feel the urge to — start a new conversation. Cleanly. Don't fight the drift.

The reasoning: long conversations don't just confuse Claude; they confuse you. The sunk-cost feeling of "but I've already taught it so much" is exactly the wrong gut to trust. The cost of a fresh conversation with one clean paragraph of context is almost always less than the cost of dragging a polluted 50-turn thread further. The 50-turn thread *feels* informed; in practice it's noise wrapped around the 3 things that mattered.

The 30-second test: if you've already thought "let me remind it of..." or "it seems to have forgotten..." even once in this conversation — that's the signal. Open a new one. Take 30 seconds to write what matters in one paragraph. Move on.

Fork 3 — Ask a question, or write a brief?

The fork: you have a task you want Claude to do. Do you ask a question, or write a brief?

Symptom you picked wrong: the first response is the wrong shape, wrong direction, or wrong scope. You then "iterate" — really, you correct — for 3, 5, 8 turns before getting close. That's not iteration. That's clean-up from an under-specified ask.

The right call — for anything with more than one moving part, brief; don't ask. A brief has four sections, four lines is enough:

```
Goal:        what done looks like, in one sentence
Context:     who this is for, what's already true
Constraints: what NOT to do, what's off the table
Done when:   how I'll verify it worked
```

The reasoning: when you brief Claude like you'd brief a smart colleague, you get the first response 80% right. When you ask a question, you get the first response 30% right and spend the rest of the conversation correcting. The math is asymmetric — **90 seconds of briefing saves 10 minutes of correction**.

The 30-second test: the next medium-complexity task you give Claude, write a 4-line brief instead of a question. Notice how the first response differs.

Fork 4 — Do it yourself, or send a helper?

The fork: you need Claude to look something up, search through a codebase, scan a long document, or otherwise do work that produces a small answer after a lot of intermediate sifting. Do you ask Claude to do it directly in the main conversation, or send a subagent?

Symptom you picked wrong: the conversation you're in fills up with intermediate search results, file contents, log dumps. Your "main thread" is now 80% noise from the search. Future turns get slower, the model loses the plot, and you find yourself starting a new conversation (Fork 2) sooner than you should have to.

The right call — if the task involves *searching for* the answer (not just answering), delegate. Send a subagent. The subagent does the searching, comes back with just the answer. The intermediate 27 minutes of noise stays in their context, not yours.

The reasoning: context is finite. Everything that lands in the conversation is paid for, attended to, and competes for the model's focus. Search debris is the worst kind of context inflation because it's not even useful — it's the *byproduct* of getting to something useful.

The 30-second test: next time you'd type "find all files that use X" or "look through the docs and tell me where Y is mentioned" — say **"send a subagent to find…"** instead. Notice how clean your main conversation stays.

Fork 5 — Request, or enforce?

The fork: you want Claude to behave a certain way — never push to main, never read `.env` files, always run tests after edits. Do you put it in your prompt (request), or in your settings/hooks (enforce)?

Symptom you picked wrong: Claude follows the rule most of the time but breaks it under pressure — when the task is complex, when the model is reasoning hard, when something else is on its mind. The rule was in the prompt, and prompts are *requests*, not contracts.

The right call — anything that *must* happen, every time, no exceptions, belongs in settings or hooks, not in prompts.

Concretely:

What you want	Where it belongs
"Never touch `.env`, `*.pem`, `~/.ssh/`"	`~/.claude/settings.json` → `permissions.deny`
"Never `git push --force` to main"	Same: `permissions.deny`
"Always run the formatter after editing a file"	A `PostToolUse` hook
"Always check if this command is dangerous before running it"	A `PreToolUse` hook

The reasoning: **prompts are aspirational. Hooks are structural.** Claude in a prompt is a model that agreed in principle to your rules, while focused on solving your actual task. Under pressure, the meta-rule slips. Hooks aren't aspirational — they're code. They can't slip.

The 30-second test: look at your `CLAUDE.md` or your usual prompt template. Find a rule that starts with "always" or "never." If breaking that rule would actually hurt you — move it out of the prompt and into settings. Right now.

Fork 6 — Strong model, or economical model?

The fork: you're about to run a task. Do you use the smartest model available, or the fastest and cheapest?

Symptom you picked wrong: your API bill is 5-10x what it should be, and you can't articulate which specific tasks actually needed the smarter model and which didn't. You just defaulted to "the best one" because it was the safe choice.

The right call — Haiku-first, escalate to Opus when reasoning is the bottleneck.

For most subtasks — listing files, running greps, formatting output, summarizing search results, simple file edits, format conversions — the smaller/faster model is more than enough. You escalate to the bigger model when the work has reasoning at its core: planning, debugging, designing, anything where the *answer* requires synthesis across many things.

In Claude Code, you can route per-subagent. Use it.

The reasoning: cost on AI tools doesn't scale linearly with quality. The 10x cheaper model is often **0.9x quality** on a specific mechanical subtask. That's a phenomenal trade. Most people never make it because the bigger model is the safe default and they never sit down with the routing config.

The 30-second test: next time you spawn a subagent for a mechanical task (search, list, summarize, reformat), explicitly route it to the smaller model. Compare output quality. If you can't tell the difference — and you usually can't — keep the routing.

Fork 7 — Done, or verified?

The fork: Claude tells you the task is complete. The work looks reasonable. Do you trust it ("done"), or do you verify it ("done — *after I check*")?

Symptom you picked wrong:

- The bug you "fixed" comes back the next day
- The tests "all pass" but one was silently skipping
- The deploy "succeeded" but the new version isn't live
- The PDF "exported successfully" but the last 3 pages are missing

The right call — Claude saying "done" is not "done." The last 30 seconds of every task is a verification step, run by *you*, not by the model.

Verification looks like:

Task type	Verification
Code change	Actually run the tests. Look at actual output, not "all pass."
Deploy	Hit the deployed URL. See the version string.
Data task	Pull a sample row. Eyeball it.
Document export	Open the file. Look at first page AND last page.
Schedule / automation	Wait for the first scheduled run. Read the log.

The reasoning: models are systematically biased toward declaring their own work successful. This isn't malice — it's a known artifact of how they're trained. **Self-evaluation has a positivity bias across every LLM that's ever been measured.** The check has to be external to be honest.

The 30-second test: next time Claude says "all done," add 30 seconds to verify the actual artifact (not the model's summary of the artifact). You'll catch issues in roughly 1 in 5 tasks. The 4 you don't catch don't matter; the 1 you do catch is the bug that would have shipped.

If you only do one of these this week

Pick **Fork 5 — Request vs. Enforce.**

Of the seven, Fork 5 has the highest *unrealized* leverage for most people. Fork 1 (chat vs. terminal) is something most users figure out within a month. Fork 7 (verify) is something everyone learns eventually after getting burned. But Fork 5 — moving load-bearing rules from prompts to settings and hooks — most users never figure out, because nothing ever *visibly* fails. Their Claude just behaves slightly worse than it could, forever.

The fix takes ten minutes:

1. Open `~/.claude/settings.json`
2. Add a `permissions.deny` block listing the commands and files Claude should never touch (your `.env`, your SSH keys, `rm -rf`, `git push --force`, your cloud credentials)
3. Save and restart Claude

That's it. From now on, the rules aren't aspirational. They're structural. You'll never have to remind Claude not to touch them, because it can't.

About the author

Jackson Lai — author of 300+ Claude Code skills at github.com/JacksonBuildsAI/ai-skills (MIT licensed). Based in Salt Lake City.

If you found this useful, the highest-leverage thing you can do is send it to one specific person who's where you were a month ago. That's worth more than any like, star, or share.

- **LinkedIn:** linkedin.com/in/slcjackson
- **X:** @JacksonAIBuilds
- **Site:** jacksonlai.com
- **Skills repo:** github.com/JacksonBuildsAI/ai-skills

I take a small number of 1:1 advisory sessions a month for people building seriously with Claude. DMs are open on LinkedIn or X.
