Vibe Coding for Beginners

How to Learn Coding With AI and Build Real Products

A prompt field guide from 8,452 real AI conversations

— Khoa Nguyen · 1devtool.com

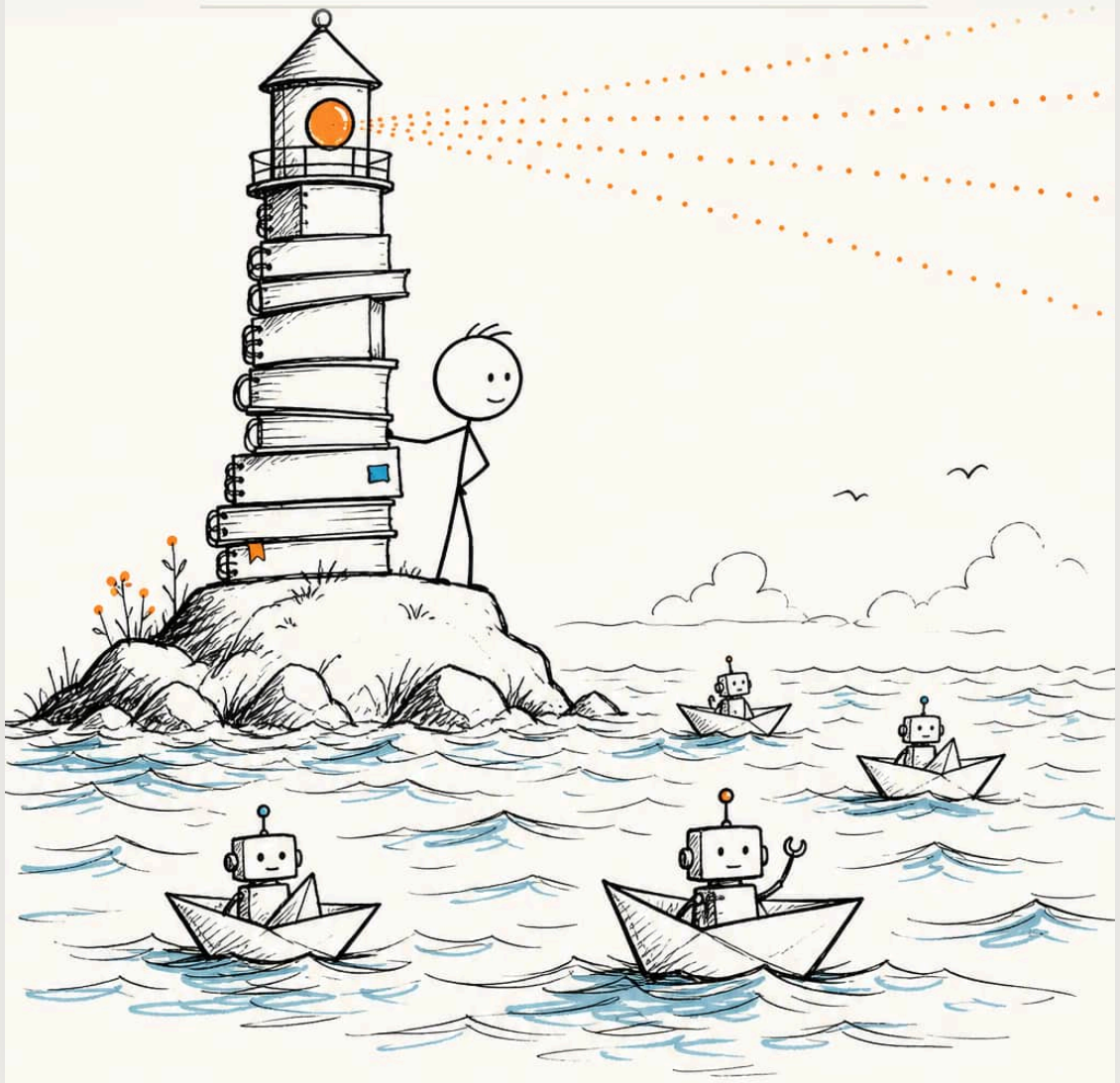
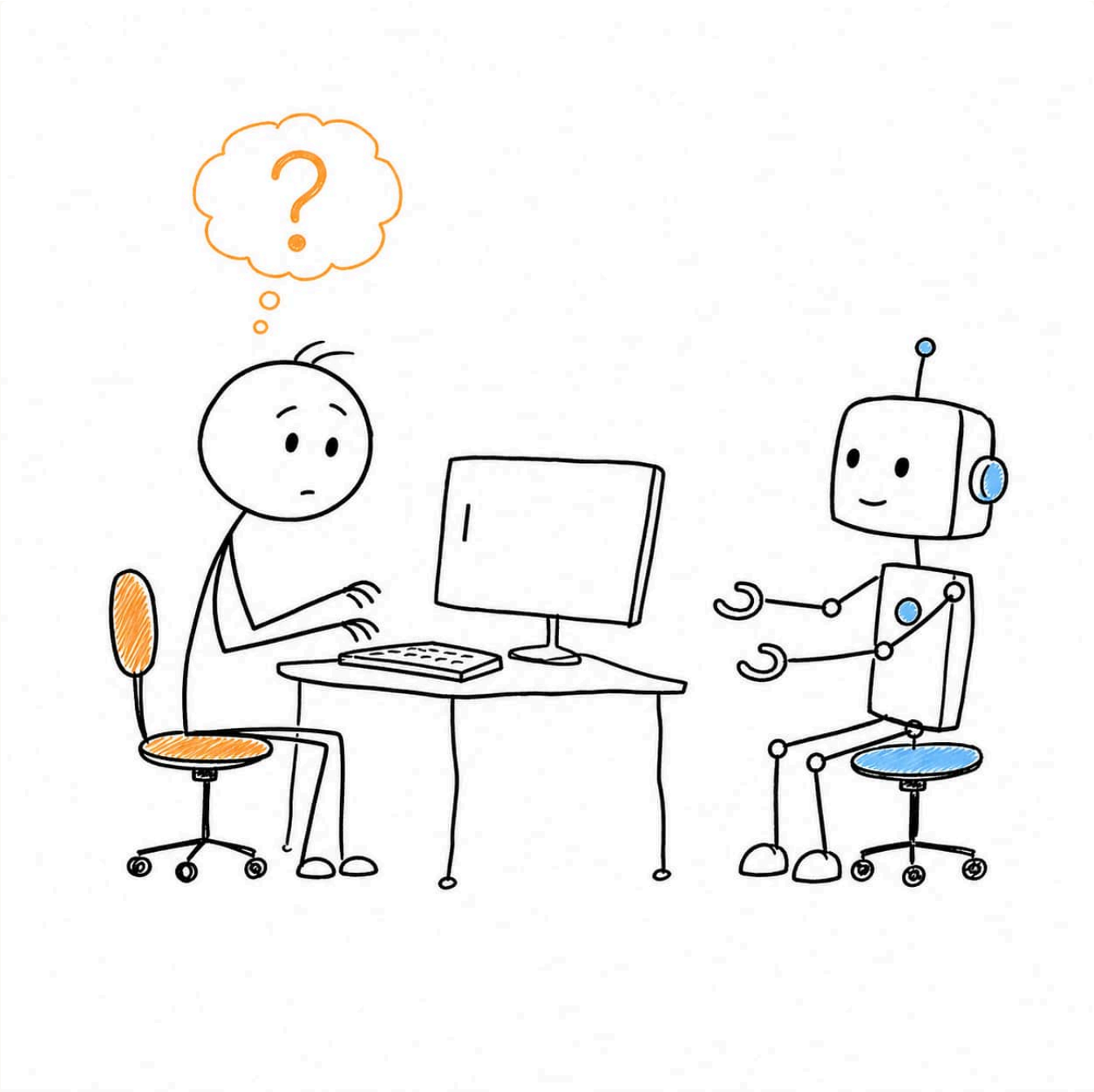


Table of Contents

Part 1 - Foundations

Chapter 1 - What Is Vibe Coding?



The first blank space: a partner is ready to work, but you do not know how to ask.

When I first switched to using AI to code, I sat staring at the terminal for a long time without typing anything. Not because I did not know how to use a computer. Not because I had never coded before. But because, for the first time in my life, I had something sitting next to me that knew more frameworks than I did, typed faster than I did, and was ready to do almost anything I asked.

The problem was: I did not know what to ask.

I once opened a project, pasted a random error into AI, and typed:

```
fix this
```

AI fixed it. The app ran. I felt great.

The next day, something else broke. AI fixed that too. Then the UI shifted. Then deploy failed. Then I opened a 100-turn session, and the more we fixed, the more tangled everything became, until I no longer knew what the original bug was. That feeling is familiar to many people new to AI: powerful and powerless at the same time.

This book was born from that gap. The gap between "AI can do a lot of things" and "I know how to work with AI to build a real product."

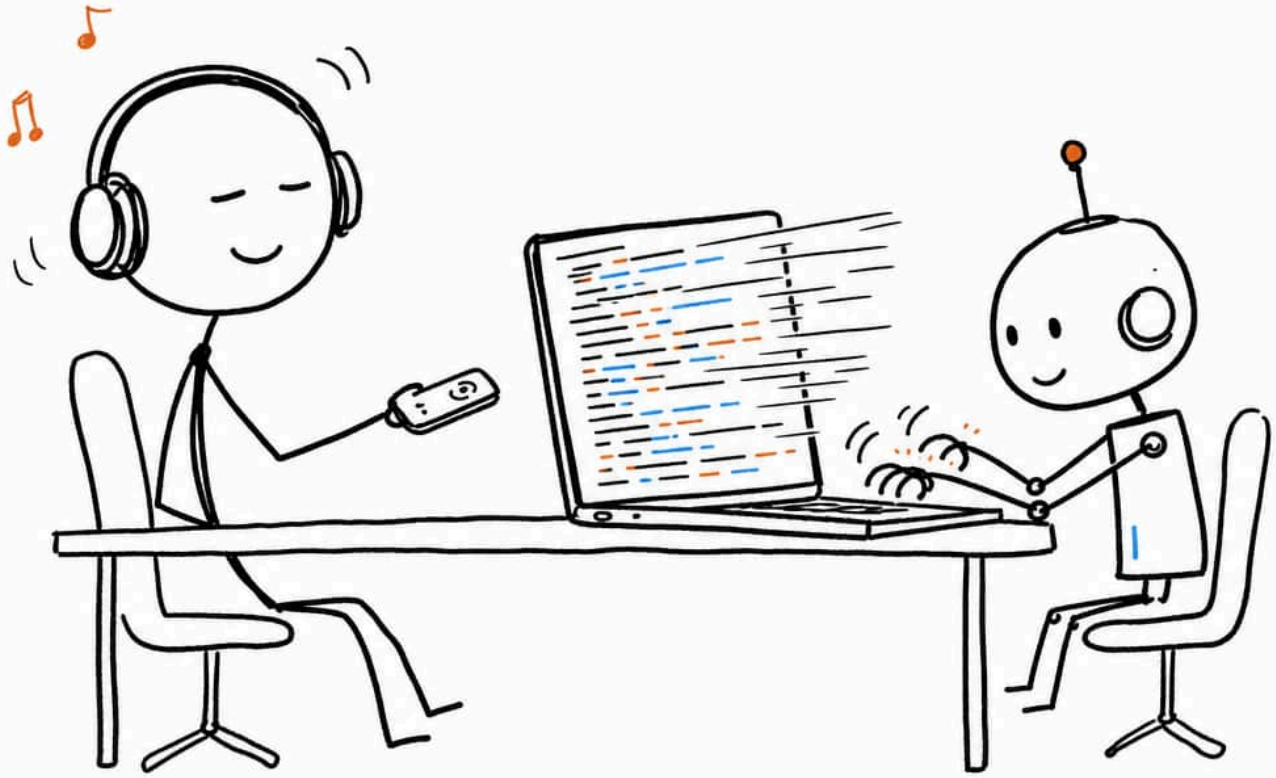
Over the last two years, I saved almost every prompt I typed while coding, debugging, deploying, writing, and analyzing. When I looked back, I had a painfully useful dataset: **8,452 prompts** from **52 real projects**. Some prompts helped me ship products. Some pushed me into a `fix again, fix again` loop. Some are prompts I now reread and cannot even understand what I wanted.

The most painful part was the data:

- Only **8.0%** of my prompts were strong.
- **58.6%** were thin: missing context, missing verification, missing the pieces AI needed.
- Only **15.1%** asked AI to verify the result.
- File/path targets appeared fairly often, but a target alone did not save the outcome.

In short: I did not lack tools. I lacked a way of working.

Vibe coding is not "AI codes everything for you"



Good vibe coding is like pair programming: you keep direction, AI adds speed.

Many people hear "vibe coding" and misunderstand it. They think it means sitting with coffee, typing one vague sentence, and letting AI build a finished product by itself. Some days it really can do a few magic tricks like that. But if you make that your main method, sooner or later you will lose the plot.

Vibe coding, as I use the term in this book, means:

Using AI as a technical collaborator to move from idea to working product, while you still own the decisions about goals, scope, quality, and verification.

You do not need every API in your head. You do not need to remember the syntax of every framework. But you do need to know what you are building, why it is correct, and how you will know it works.

AI can write code quickly. But AI does not automatically know:

- What your business goal is.
- Which parts of the codebase must not be touched.
- Which bug matters more.
- Which trade-off you accept.
- When to stop a session and reset.

That is your job.

If you treat AI like a homework machine, you will become passive. If you treat AI like an extremely fast coworker who guesses too confidently, you will prompt very differently.

You will not type:

```
make this better
```

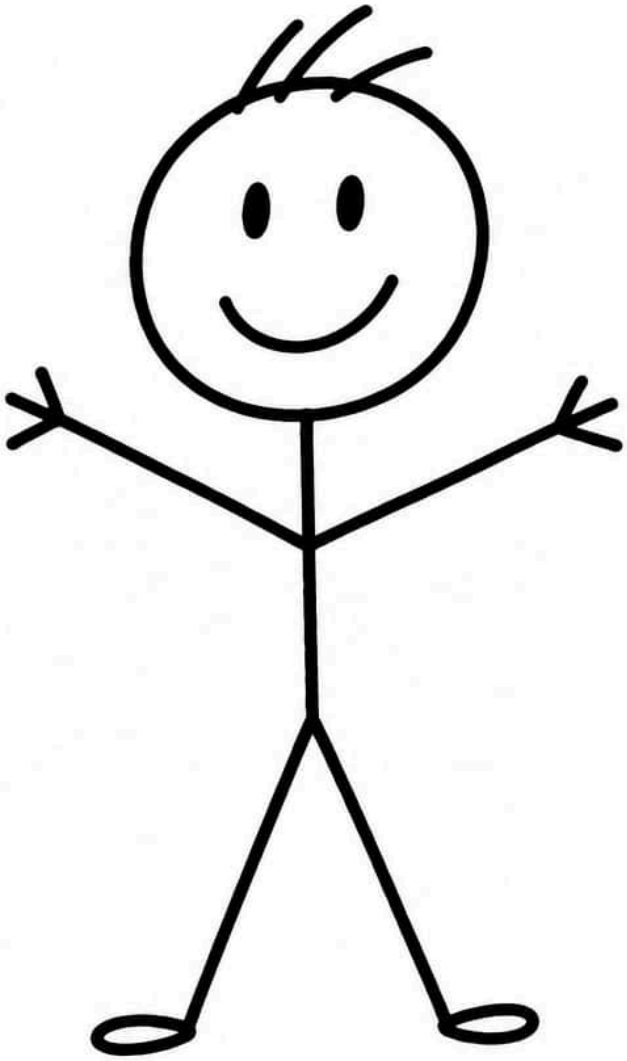
You will type:

```
Review `src/components/CheckoutSummary.tsx`.

Current issue:
- Mobile layout overflows at 375px.
```

```
- Desktop layout is acceptable and should not change.

Goal:
- Fix mobile overflow only.
- Preserve current pricing logic and API calls.

Verify:
- Run the relevant tests if available.
- Tell me exactly what changed and what you did not touch.
```

These two prompts are not different because the second is more "polite." They are different because the second tells AI the field, the rules, and the scoring criteria.

Vibe coding is not a slot machine either



A vague prompt turns AI into a slot machine: sometimes you win, but you are not in control.

There is one very addictive way to use AI: type randomly and wait for a miracle.

The first time it works, you are happy. The second time it works, you believe. By the tenth time, when it breaks an important file, you realize you were never really controlling the process. I call this "slot-machine coding."

Slot-machine coding has a few signs:

- You paste an error but do not provide a reproduction.
- You tell AI to "fix" but do not describe expected behavior.
- You let AI edit many files without setting scope.
- You do not read the diff.
- You do not run tests.
- You keep extending an already-too-long session because you do not want to lose context.

It is like gambling because the early reward is strong. A 20-word prompt can produce 200 lines of working code. But the cost comes late: warped architecture, hidden bugs, strange dependencies, changed config, and eventually you no longer understand your own project.

This book is not anti vibe coding. I still use AI every day. But I am against the kind of AI use that makes me dumber.

Good vibe coding should make you clearer, not more vague. After a good session, you should understand the project better than before: where the bug was, what code changed, which tests ran, and what risk remains. If after a session you only have a pile of changed files and do not know why, that is technical debt in disguise.

What should beginners learn first?

If you do not know how to program yet, I do not think you should immediately ask AI to build a SaaS with auth, billing, dashboard, deploy, database, and background jobs. Could you make a demo? Maybe. Will you understand enough to operate it? Not necessarily.

You do not need to become a senior engineer before using AI. But you should have a few foundations:

- Know what files and folders in a project are for.

- Know how to read basic terminal errors.
- Know the difference between frontend, backend, database, and server.
- Know how to run an app locally.
- Know how to commit code and go back when needed.
- Know a little about testing, even if it is only manual testing with a checklist.

AI helps you move faster, but it does not make ambiguity free. If your foundation is hollow, AI will build very quickly on that hollow foundation.

This is not a threat. It is a time saver. A beginner who knows how to ask clearly, verify, and keep scope can go very far with AI. Someone who only knows how to paste and hope will move very fast for the first three days, then get stuck in their own mess.

How to read this book

I wrote this book as a practical field manual, not an intro programming textbook. You can read it from start to finish, but an even better way is to read it while reopening your own old prompts.

Each chapter tries to answer one question:

- What is vibe coding, and what role should I keep?
- How does AI actually "think"?
- What parts make a strong prompt?
- What does the data from 8,452 real prompts say?
- When AI is wrong, how should I correct it?
- When building UI, deploying, or starting a first project, how should prompts differ?
- When a session gets long and context fills up, how should I reset?

- Which templates are worth copying, and which recipes should I avoid?

You do not need to memorize everything. The goal is not to know 50 templates by heart. The goal is that after reading, before you press Enter, a few questions light up in your head:

- Have I made the goal clear?
- Does AI know which files to read?
- Have I said what must not be touched?
- Have I asked for verification?
- If this session gets long, do I know where to stop?

If those questions appear, your prompt is already different.

The core principle

If I had to compress the whole book into one sentence, I would say:

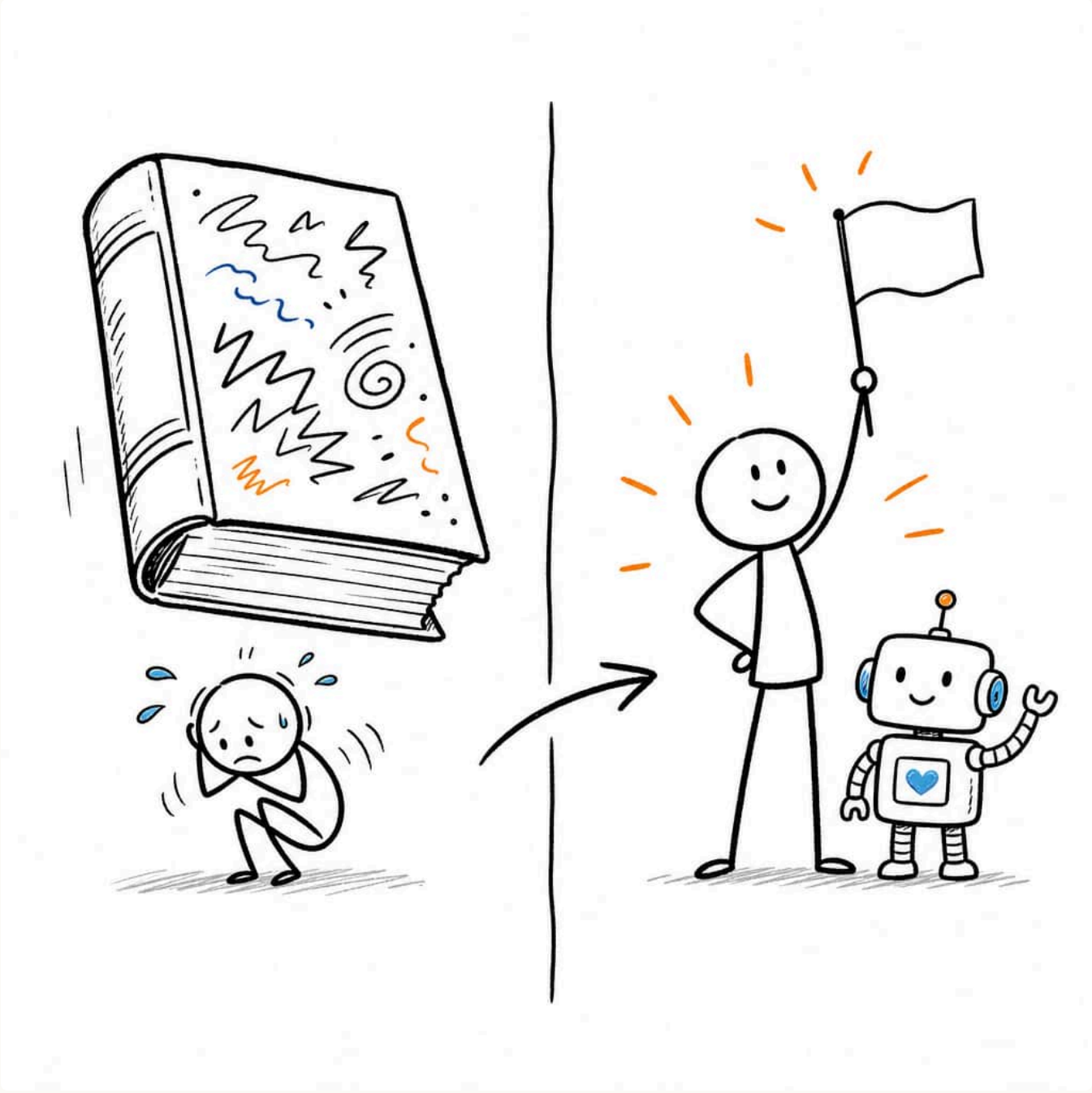
The stronger AI gets, the clearer your prompt needs to be.

That may sound backwards. Many people think that the better the model gets, the more vaguely they can speak. That is partly true. A better model can guess better. But when you build real products, a good guess is still a guess. And one wrong guess in a real codebase can cost hours.

Clear prompts are not for making AI happy. Clear prompts protect you from AI's speed.

Before moving on, open the last 10 prompts you sent to AI. How many of them clearly stated current state, desired state, scope, and verification?

Chapter 2 - The Mindset of Someone Who Codes With AI



The same AI, but a different mindset creates different results.

Mindset matters not because it sounds nice. It matters because it decides what kind of prompt you type when the app is broken at 1 a.m.

Beginners often use AI like an answer machine:

```
Fix this bug.
```

A strong vibe coder uses AI like a collaborator who needs a brief:

```
Debug the login redirect bug.

Observed:
- After successful login, the user remains on `/login`.
- Network request `/api/session` returns 200.

Expected:
- User is redirected to `/dashboard`.

Please inspect auth flow first, explain root cause, then patch the smal
```

Both prompts want the bug fixed. But the second prompt has something the first one does not: an investigative attitude.

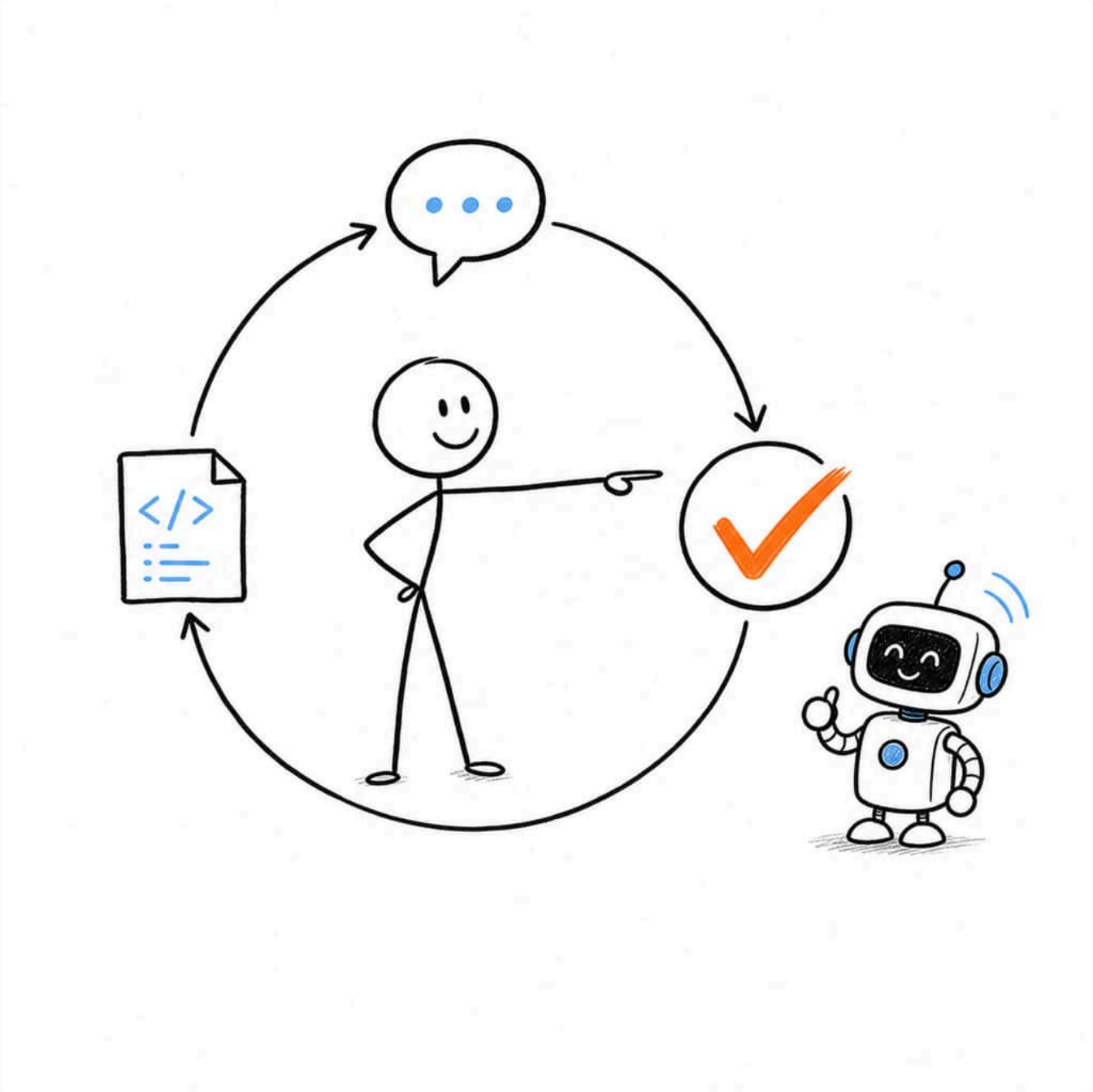
When working with AI, the best mindset is not "AI is smarter than me." It is also not "AI is just a dumb tool." The more useful mindset is:

AI is a very fast coworker who has read widely and writes quickly, but does not live inside your project.

It does not know the history of decisions. It does not know which code was just changed by someone else. It does not know whether you are prioritizing speed or safety. It does not know that this small bug is actually blocking tomorrow's launch. If you do not say it, AI will guess.

And AI guesses with confidence.

You are the keeper of context



A good prompt does not end with the first question. It is a loop: brief, verify, redirect.

The first thing I learned after many bad sessions was this: I cannot outsource context.

AI has a context window. But a context window is not real memory. It is a limited workbench. The more things you put on the bench, the harder it is to see what matters.

A good session usually has this rhythm:

1. You brief the goal and scope.
2. AI reads files and makes a plan or patch.
3. You inspect the diff, run tests, or provide new evidence.
4. AI adjusts.
5. When the session starts to get long and vague, you reset with a handoff.

A bad session often has the opposite rhythm:

1. You type vaguely.
2. AI edits randomly.
3. You see it is wrong and type "not that."
4. AI guesses again.
5. You add a new unrelated requirement.
6. Context becomes soup.

I have fallen into the second pattern many times. The problem was not that AI was bad. The problem was that I gave AI roles it should not own all at once: product owner, architect, QA, and historian.

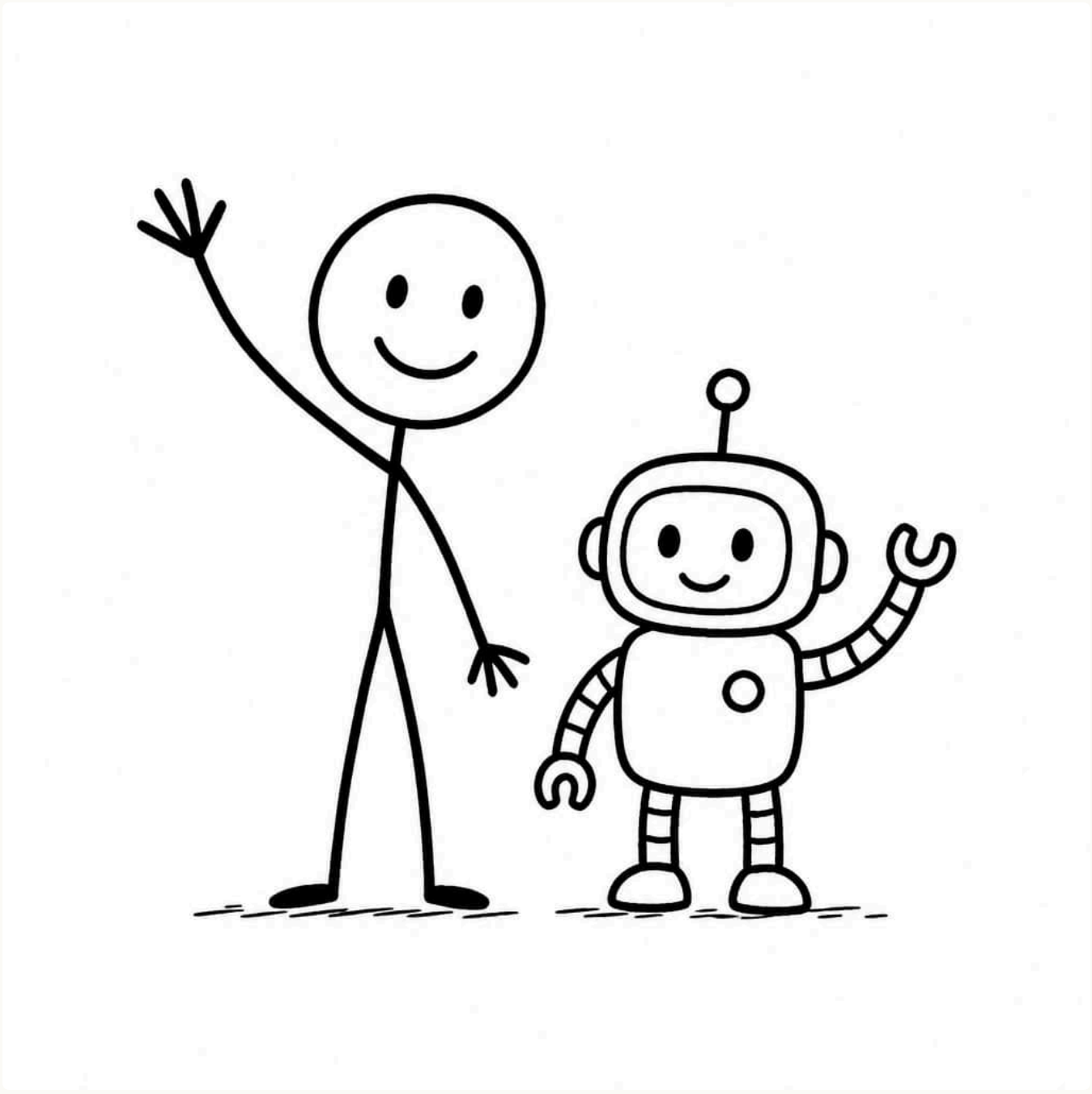
You must keep context. Not by writing prompts as long as essays, but by giving AI the right things it needs to start:

- The goal of the task.
- Relevant files/paths.
- Current behavior.
- Desired behavior.

- Constraints.
- How to verify.

If one of these six is missing, AI may still work. But it works by guessing.

Understand AI's brain



AI does not have project memory the way you think. It works inside a finite context window.

You do not need to study deep learning to use AI. But there are a few practical truths worth remembering.

One: AI does not read the codebase unless you make it read the codebase.

If you ask:

```
why is checkout broken?
```

AI may answer in a way that sounds very reasonable. But if it has not read `checkout.ts`, `payment.ts`, the route handler, env config, and the error log, that answer is only a hypothesis. Hypotheses can be useful. But do not confuse them with evidence.

A good prompt often starts with:

```
First inspect these files before proposing a fix:
- `src/checkout/page.tsx`
- `src/api/checkout/route.ts`
- `lib/stripe.ts`
```

Two: AI tends to complete the request, even when the request is incomplete.

Humans can ask follow-up questions when information is missing. AI can too, but often it chooses the more "helpful" path: filling the gaps itself. This is the source of many bugs.

You say:

```
add export feature
```

It may add CSV export. Or PDF. Or a download button on the wrong page. Or a new package. Or change the entire data model. Not because it is malicious. Because you did not define what "export" means.

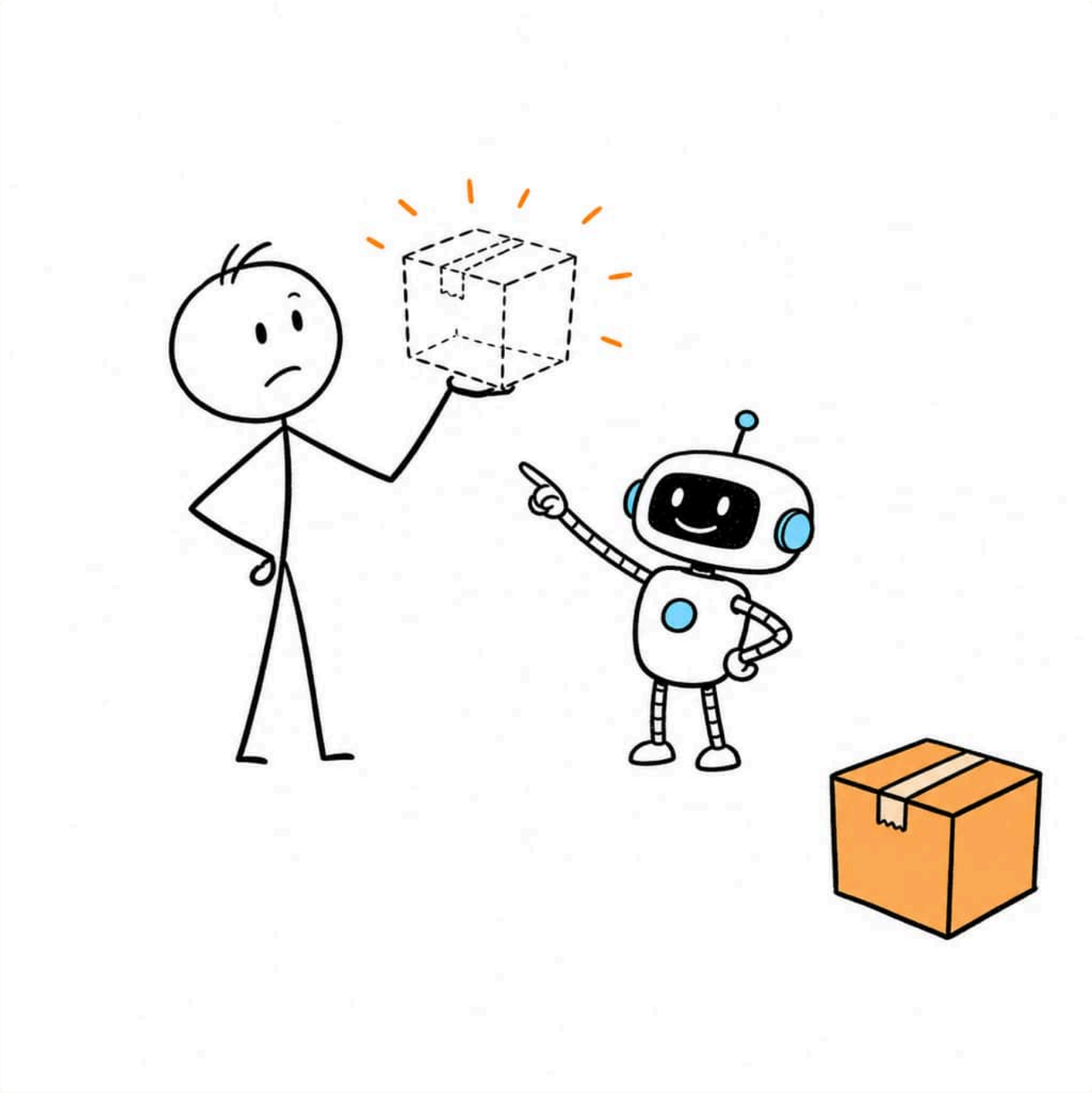
Three: AI learns more from the current prompt than from the intention in your head.

In your head, you may be thinking "do not change the UI." But if the prompt does not say that, AI cannot hear it. In your head, you may know "this file is being edited by someone else." But if the prompt does not say it, AI does not know.

One small section can save a lot:

```
Out of scope:
- Do not change UI layout.
- Do not touch billing logic.
- Do not modify files outside `src/auth`.
```

Do not let AI hallucinate dependencies



AI knows many packages, but it can also invent a package that sounds completely real.

One of the funniest mistakes I ran into was AI recommending a package that sounded extremely plausible but did not exist. The name had the right npm style. The README sounded real. The API looked clean. I only found out when install failed.

Since then I have had one rule:

Anything AI gives you that touches the outside world - packages, APIs, CLI flags, pricing, laws, new docs - must be verified.

In code, verification can be:

- `npm view <package>`
- reading official docs
- running tests
- running the build
- checking the diff
- opening the local app
- reading real logs

Do not turn AI into your only source of truth. AI is a powerful generator of hypotheses and drafts. Truth still lives in the code, tests, terminal, docs, and running product.

Your three roles

When vibe coding, I find that I need to hold three roles.

1. The briefer.

You provide the goal, scope, constraints, and relevant resources. The first prompt is like a brief for a coworker jumping into the task.

Bad brief:

```
fix auth
```

Good brief:

```
Fix auth redirect after login.

Current:
- Login succeeds.
- User stays on `/login`.

Expected:
- Redirect to `/dashboard`.

Read:
- `src/app/login/page.tsx`
- `src/lib/auth.ts`
- `src/middleware.ts`

Out of scope:
- Do not change signup.
- Do not change database schema.

Verify with the existing auth test or a manual reproduction.
```

2. The reviewer.

AI writes quickly, so you need to read a little more slowly. You do not need to read every line like a compiler, but you need to know:

- Which files changed?
- Did anything change outside scope?
- Was a dependency added?
- Was old logic deleted?
- Was there a test or verification?

If you do not read the diff, you are signing your name on code you do not understand.

3. The person who decides when to stop.

AI is very good at continuing. It can keep fixing forever. But many sessions drift farther from the original goal the longer they go. When a task has passed 60-80 turns and still is not clean, the problem is usually no longer "one more prompt is missing." The problem is dirty context.

A good stopping point is:

```
Stop here. Summarize:
- Goal
- Files changed
- Current status
- Remaining bugs
- Exact next prompt for a fresh session
```

Reset is not failure. Reset is context hygiene.

A pragmatic mindset

I do not believe in either extreme: "AI will replace all developers" or "AI is only autocomplete." Both are too lazy.

AI really changes how I work. It helps me build faster, try more ideas, and learn faster. But it also amplifies bad habits: vagueness, laziness about verification, and the desire to fix faster than you understand.

A pragmatic mindset is:

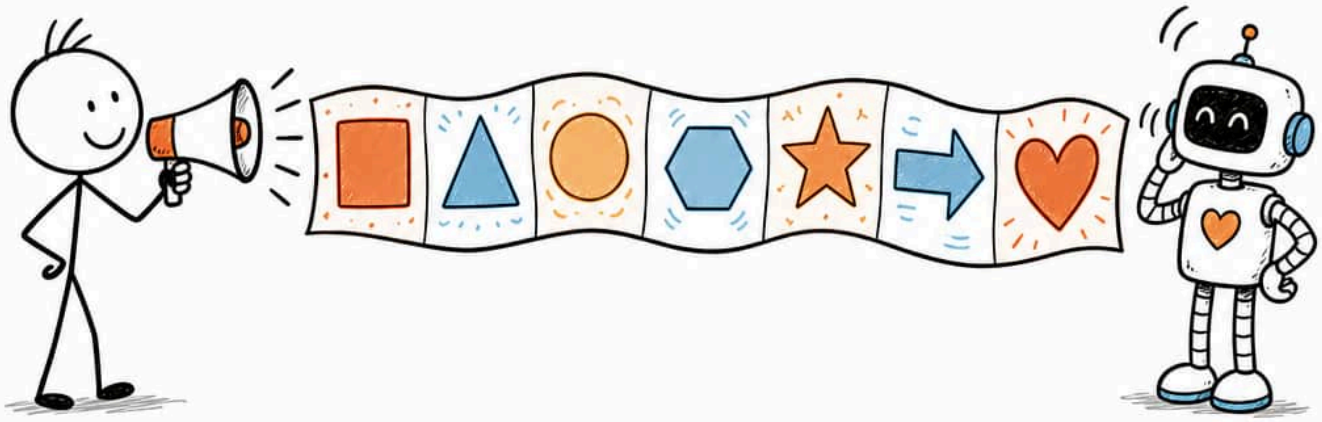
- Use AI to accelerate, not to avoid thinking.
- Make AI read evidence, not only guess.
- Give clear scope, not a pile of confusion.
- Verify with terminal, tests, browser, and diff.
- Reset when context gets too long.

If you keep this mindset, vibe coding does not make you weaker. It gives you another layer of leverage.

Before the next chapter, ask yourself: in recent sessions, are you the briefer, the reviewer, and the person deciding when to stop - or are you letting AI hold all three roles by itself?

Part 2 - The Art of Prompting

Chapter 3 - How to Write a Strong Prompt



A strong prompt does not need beautiful writing. It needs enough bones for AI to hold on to.

After rereading thousands of my own prompts, I found that strong prompts are not mysterious. They are not magic spells. Writing a long prompt in English does not automatically make it good. Roleplay does not automatically make it better.

Strong prompts usually share the same skeleton:

1. **Role** - What role is AI taking?

2. **Task** - What specific work needs to be done?
3. **Target** - Which file, screen, module, or artifact?
4. **Context** - What is happening now, and why does it need to change?
5. **Constraints** - What must not be touched?
6. **Output** - What form should the answer take?
7. **Verification** - How will we know it is correct?

Not every prompt needs all 7 parts. Changing one word on a button does not require a long brief. But when a task touches real code, multiple files, or user-facing behavior, the missing part often shows up later as a bug.

Weak prompts are not weak because they are short

A short prompt can be very strong:

```
Fix typo "Singup" -> "Signup" in `src/app/login/page.tsx`. Do not chang
```

This prompt is short but sufficient: task, target, constraint.

A long prompt can still be weak:

```
I am building an app kind of like Notion but simpler, with a dashboard,
editor, settings, and maybe later team workspaces. Can you look at this
It feels kind of weird, and I want it to be more professional and clear
```

It has many words, but it lacks target, current/desired, scope, and verification. AI has to guess what "this part" is and what "professional" means.

Do not ask whether a prompt is long or short. Ask whether it has enough information to act correctly.

Role: set the lens for AI

Role is not decoration. It changes how AI reads the task.

Compare:

```
Improve this API.
```

and:

```
You are reviewing this API as a backend engineer focused on reliability
Find edge cases first, then patch the smallest fix.
```

The second prompt does more than say "make it better." It tells AI to look through the lens of reliability, review first, and patch second. For complex tasks, role helps AI choose criteria.

Role is most useful when you need to:

- Review architecture.
- Debug root cause.
- Audit security.
- Refactor while preserving behavior.
- Design UI in a specific style.

Role does not need to be long. One line is enough.

```
You are a strict code reviewer.
```

or:

```
You are reviewing AND PATCHING this checkout flow.
```

Task and target: do not make AI guess "this"

The worst prompt in my dataset was not grammatically incorrect. It was:

```
fix them
```

"fix" is an action. "them" is nothing.

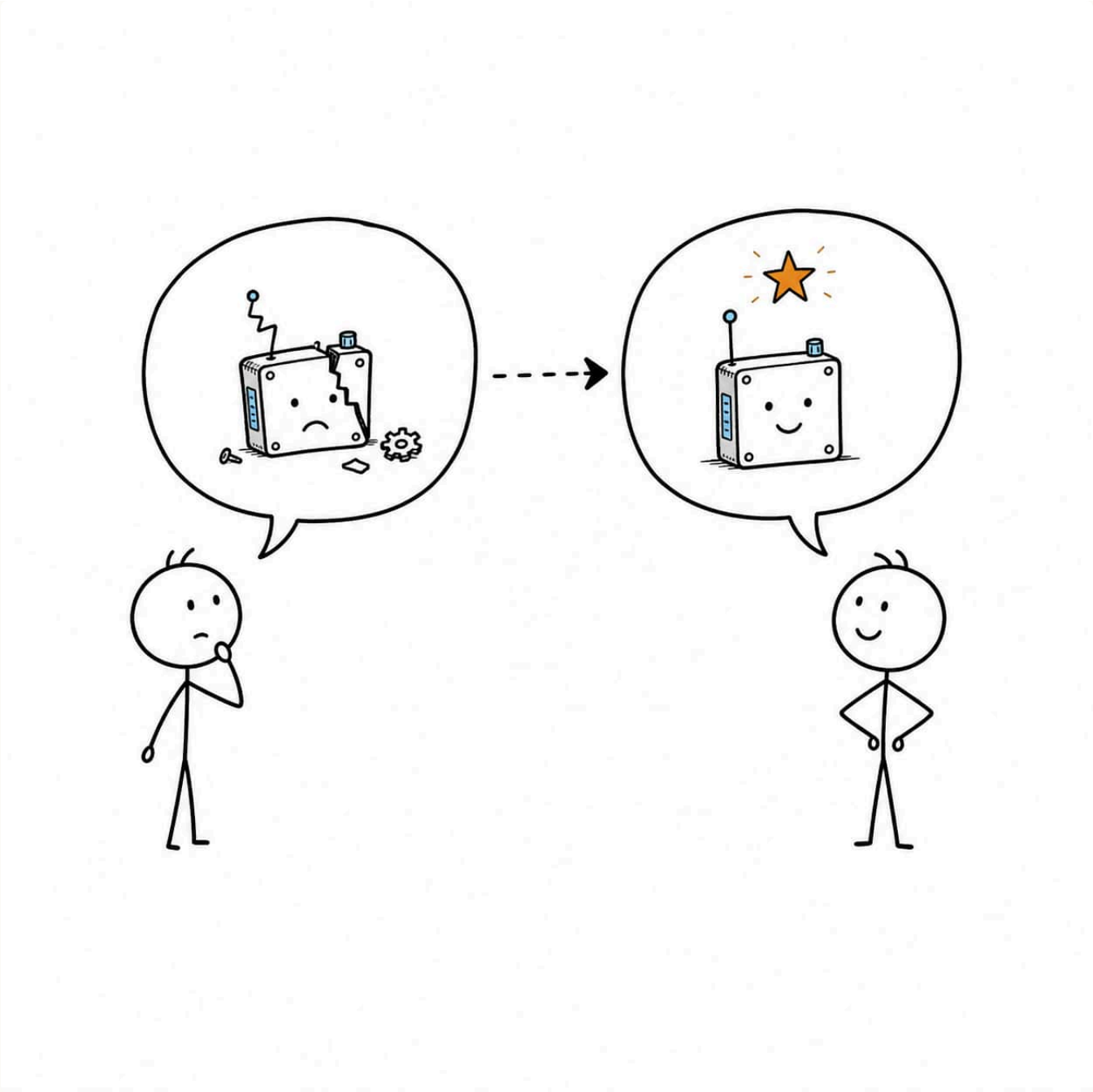
AI does not know whether "them" means tests, bugs, layout, warnings, or user feedback. If there are many things in the previous context, it will choose something that seems plausible. It may be right. It may be wrong.

Good targets are usually:

- File/path: `src/components/Sidebar.tsx`
- Route: `/settings/billing`
- Component: `CheckoutSummary`
- Test: `auth.redirect.test.ts`
- A specific log/error block
- A screenshot with a description

A prompt with a clear target keeps AI from having to grep the whole project in the dark.

Context: current vs desired



Two current/desired lines often save more time than a long description.

One pattern that improves prompts quickly is writing two sections:

```
Current:
- Mobile menu opens, but cannot close after selecting a link.

Desired:
- After selecting a link, menu closes and route changes normally.
```

Current/desired looks simple, but it forces you to clarify the problem. Often, while writing these two lines, you realize you do not fully understand the bug yet.

If you do not know the current state, ask AI to investigate first:

```
First inspect the current behavior and explain what is happening.
Do not patch yet.
```

This is an extremely strong prompt. It fights the habit of "fix first, understand later."

Constraints: the most commonly forgotten part

AI has a bias toward helping. It sees ugly code near the bug and fixes it too. It sees a component that could be refactored and refactors it. It sees an old package and upgrades it. Sometimes those changes are technically correct but wrong for the task.

That is why a good prompt should include `Out of scope`.

```
Out of scope:
- Do not change pricing logic.
- Do not touch database schema.
- Do not refactor unrelated components.
- Do not add new dependencies.
```

This is the brake. Without a brake, AI can move very fast in a direction you do not need.

In my data, good sessions usually had clearer context and constraints. File/path helps AI start, but constraints help it stay inside the guardrails.

Output: say what you want back

Output is not always "code." Sometimes you need a plan first. Sometimes you need findings first. Sometimes you want an immediate patch. Sometimes you only need a checklist.

Example:

```
Return:
1. Root cause in 5–8 lines.
2. Files you will change.
3. Patch.
4. Verification result.
```

For review tasks:

```
Return findings first, ordered by severity, with file/line references.
Only patch after listing findings.
```

Clear output makes review easier. Otherwise, AI chooses a random format: explanation, code, apology, and extra suggestions all mixed together.

Verification: the small but expensive part

Before pressing Enter, score your prompt quickly. It does not need to be formal; it just needs attention.

Only 15.1% of my prompts asked for verification. This was the biggest gap.

No verification means you are trusting "done." But in code, "done" is not evidence. Evidence is a passing test, a passing build, a reproduced fix, a correct browser view, or at least a diff that stayed in scope.

A verification prompt can be simple:

```
After patching, run the relevant test if available.
If no test exists, describe the exact manual verification steps.
```

Or:

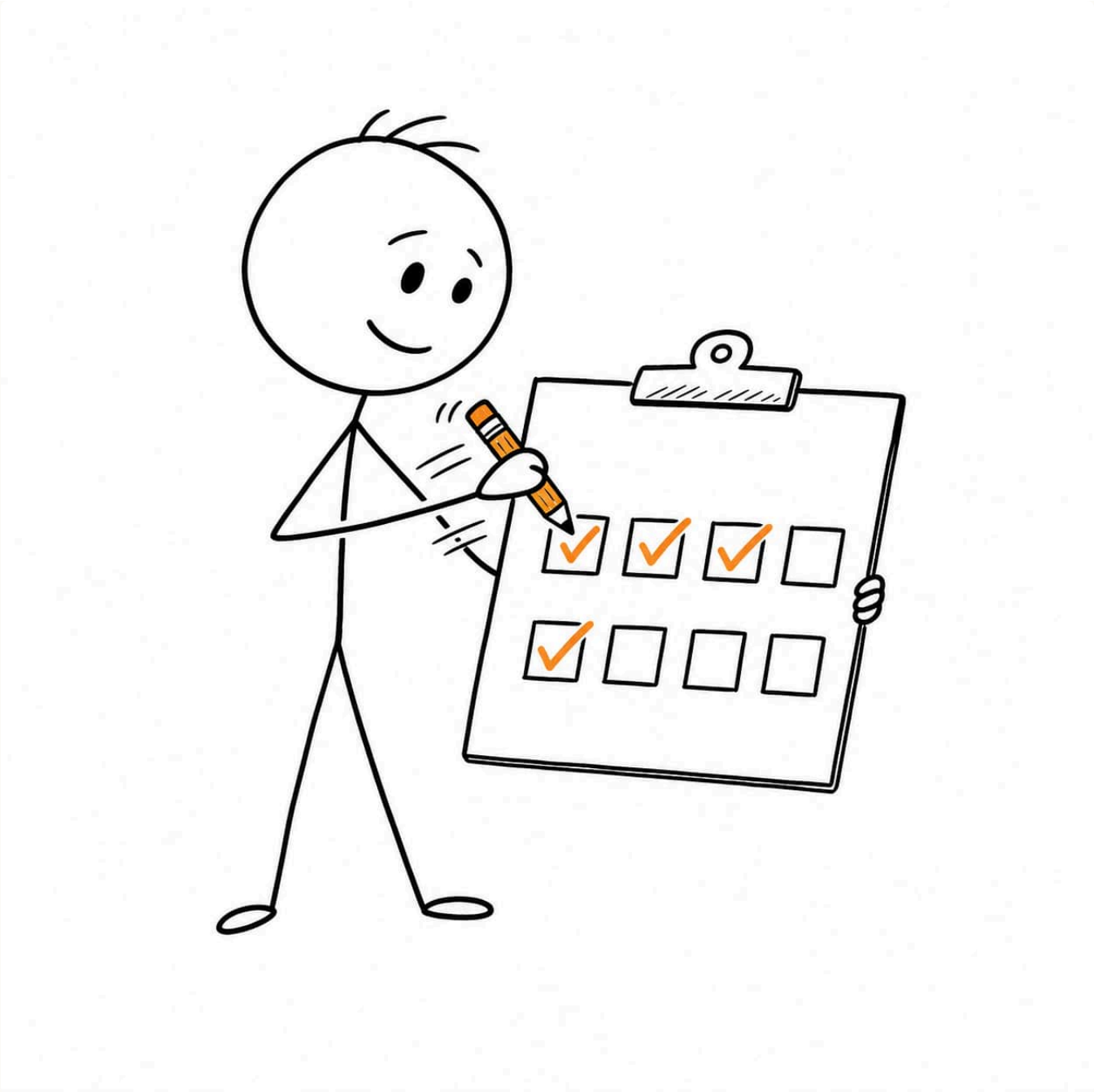
```
Verify on desktop 1280px and mobile 375px.
```

Or:

```
Do not claim fixed unless you rerun the same reproduction.
```

One verification sentence changes the attitude of the whole session. AI is no longer just "making a fix"; it has to think about how to prove the fix.

A 10-point scorecard before sending



The scorecard is not an exam. It is a 20-second brake before you send.

The 7-part framework helps you write prompts. The scorecard helps you grade them quickly before pressing Enter. The two are not a perfect one-to-one map, but they serve the same goal: reducing ambiguity.

Before sending an important prompt, ask:

- **Is the action clear?** Does AI know whether to review, fix, debug, refactor, or design?

- **Is the target clear?** Is there a file/path, route, component, log, or screenshot?
- **Is the context sufficient?** Do you have current and desired?
- **Are there constraints?** Did you say what must not change?
- **Is there evidence?** Do you have a real error, real output, or real reproduction?
- **Is the output format clear?** Did you ask for findings, patch, checklist, or summary?
- **Is verification included?** Is there a way to prove correctness?
- **Is the scope reasonable?** Is one prompt holding three unrelated tasks?

If the prompt is 8/10, send it. If it is 5-7, improve it a little. If it is below 5, do not send it yet. You are not saving time by sending a vague prompt. You are only moving the cost into the cleanup.

Strong prompt template

This is the template I use when a task is serious enough:

```
You are <role> for <project/context>.

Task:
<work to do>

Target:
- `<file_or_path_1>` - <why it matters>
- `<file_or_path_2>` - <why it matters>

Current:
- <current behavior>

Desired:
- <desired behavior>

Out of scope:
- <thing not to touch>
```

```
- <thing not needed for this task>

Return:
1. Brief diagnosis or plan.
2. Patch.
3. Verification result.
```

Do not copy this template as a ritual. Delete what you do not need and keep what you do. The best prompt is not the longest template. The best prompt is just enough for AI not to guess the dangerous parts.

A before/after example

Before:

```
fix sidebar mobile
```

After:

```
Fix mobile sidebar close behavior.

Target:
- `src/components/Sidebar.tsx`
- `src/components/MobileNav.tsx`

Current:
- At 375px, sidebar opens correctly.
- After tapping a nav link, route changes but sidebar stays open.

Desired:
- Sidebar closes after link tap.
- Desktop sidebar behavior unchanged.

Out of scope:
- Do not redesign sidebar.
```

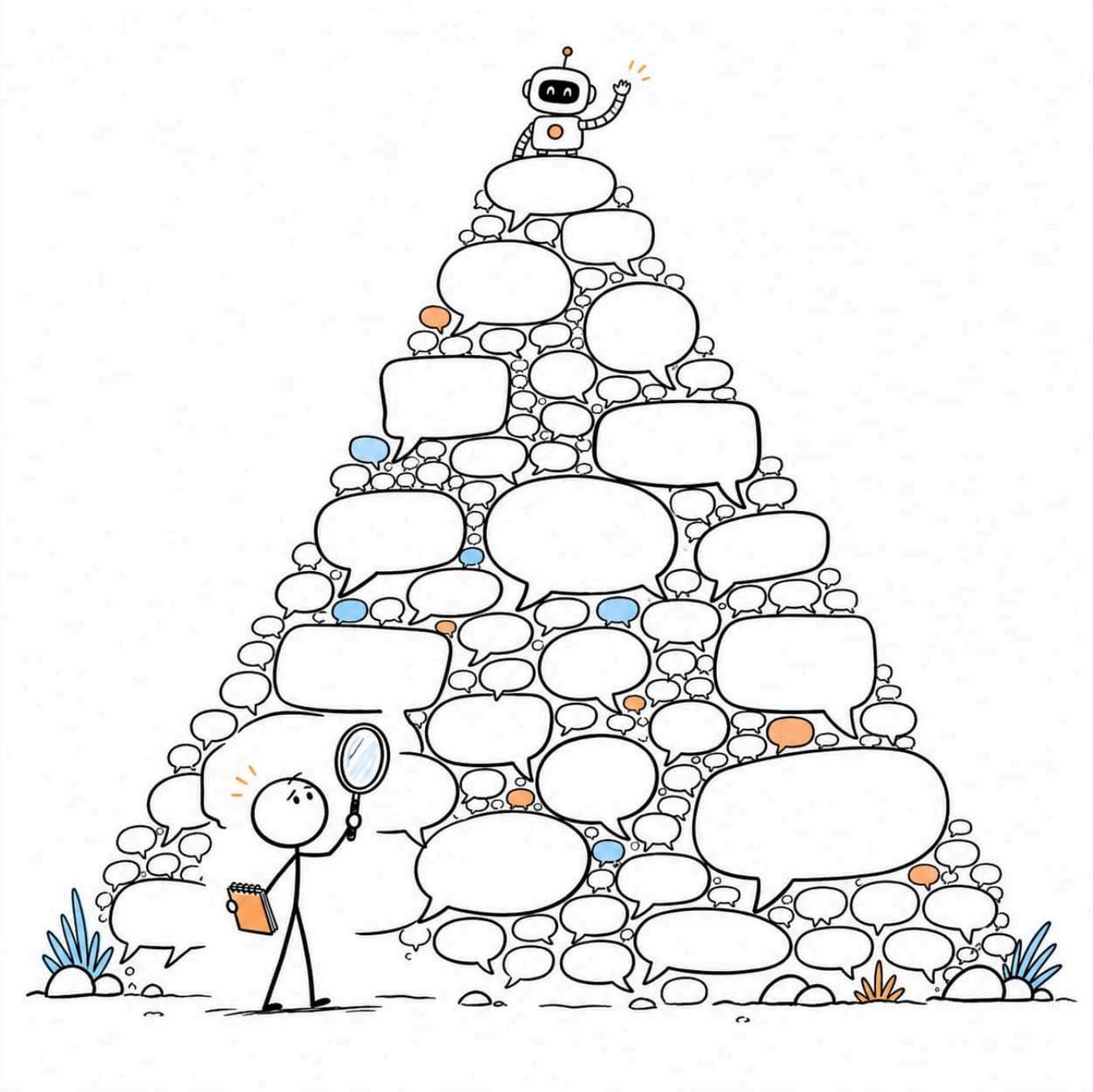
```
- Do not change route structure.
- Do not add dependencies.

Verify:
- Check mobile width 375px.
- Confirm desktop width 1280px unchanged.
```

The second prompt is longer, but not because it rambles. It is longer because it clarifies the places where AI is likely to guess wrong.

This chapter has one job: slow you down for 20 seconds before you press Enter. Those 20 seconds are usually cheaper than 2 hours fixing a patch that went outside scope.

Which part is missing from the next prompt you are about to send: target, context, constraint, or verification?

Chapter 4 - Lessons From 8,452 Real Prompts



8,452 prompts is a mountain. Each prompt is one step.

One day I asked myself: "I use AI this much, but am I actually getting better at prompting?"

It felt like I was. I felt faster, more confident, less stuck. But feelings lie very well. So I exported my prompt history and wrote a script to analyze it.

The result hurt.

Not the kind of hurt that says "I am terrible." More like looking in the mirror in the morning: this is the real face of my work habits, with no filter. Some prompts I thought were normal turned out to be thin. Some patterns I repeated hundreds of times without noticing. Some sessions I thought were productive, but the turn count and correction count showed I was going in circles.

The data in this chapter comes from **8,452 prompts** across **590 sessions**, and the deeper outcome analysis is based on **1,126 sessions** with enough user + assistant traffic to score. The outcome score is heuristic: correction count, apology language, and completion signal in the final message. Recipes with a small n are directional signals, not laws of physics.

In other words: do not read this chapter as a statistics paper. Read it as a work journal inspected with a flashlight.

Story 1: most of my prompts were too thin

I scored prompts by shape: action verb, target, context, constraint, verification. The distribution was clear:

- **Thin (0-4 points): 58.6%**
- Workable (5-7 points): 33.4%
- **Strong (8-10 points): 8.0%**

More than half of the prompts I typed were the kind I should not send if the task mattered.

Weak prompts are not always useless. Sometimes the session is already clear, and "ok next" is fine. But when the first prompt of a new task is too thin, AI starts by guessing. If the guess is right, AI feels magical. If the guess is wrong, you spend 20 turns steering back.

The three most common anti-patterns were:

1. Fragment prompts.

```
fix them
```

or:

```
add this
```

There is an action, but no target. AI does not know what "them" is.

2. Image-only without context.

Paste a screenshot and explain nothing. AI can look at the image, but it does not know whether you want to fix layout, copy, spacing, hierarchy, or flow.

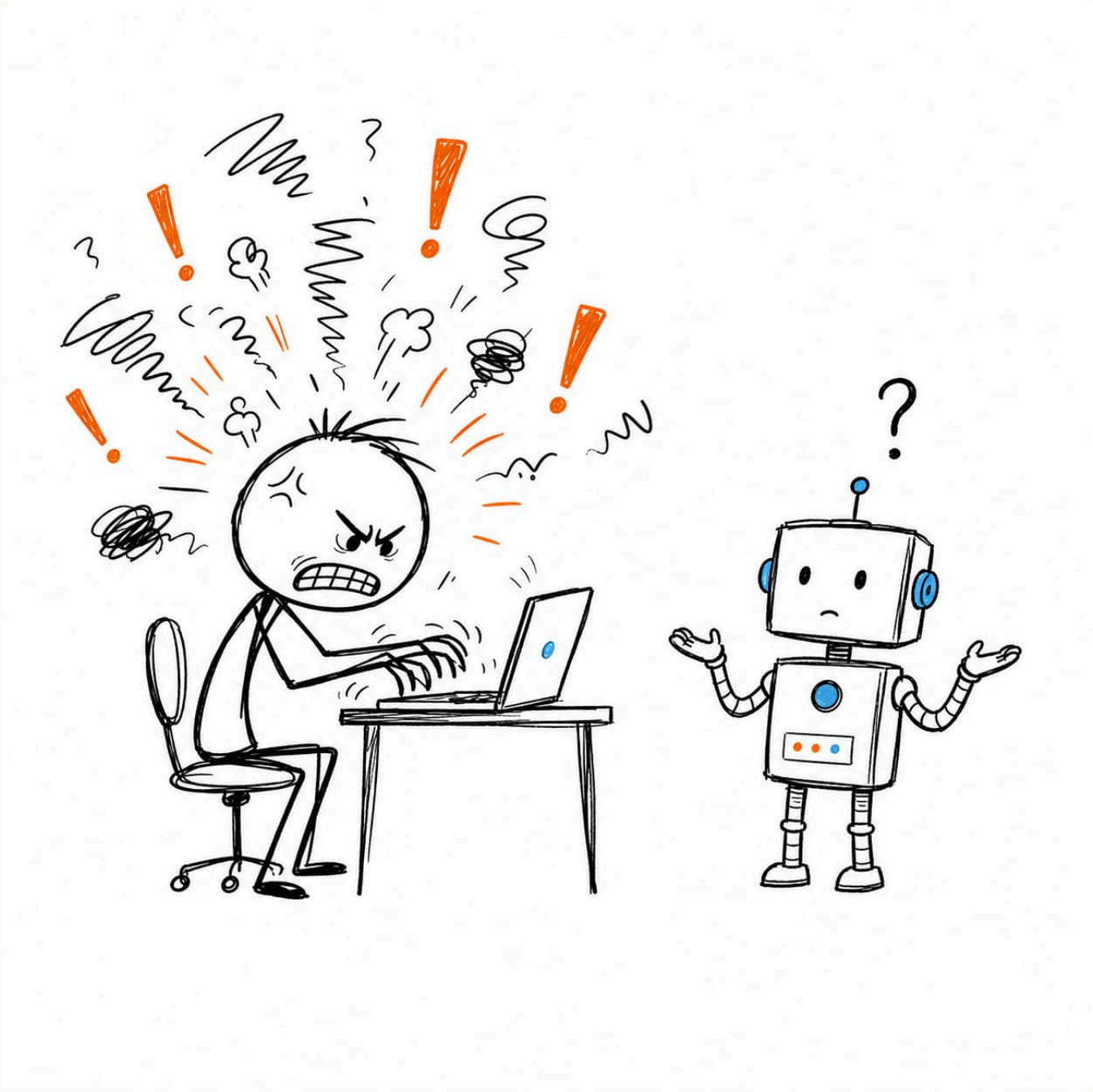
A better prompt only needs one more line:

```
In this screenshot, the issue is the primary CTA is visually weaker tha
Fix hierarchy only. Do not redesign the whole page.
```

3. Multi-intent without order.

```
fix auth bug, add dark mode, update README, deploy staging
```

These are four tasks. AI may do the easy one first, skip the hard one, and let the session drift away from the original goal. One session, one intent. If there are multiple intents, number them and state the order.



Plenty of emotion, no information - AI has nothing to hold on to.

Story 2: targets help AI start, constraints help AI finish

When I connected outcomes to sessions, one result surprised me: bad sessions did not necessarily lack targets. Some bad sessions still had clear files/paths and

evidence. At first I thought my script was wrong. After rereading them, it made sense.

Target and evidence are admission tickets. Without them, AI cannot start well. But having them does not guarantee a clean outcome.

Example:

```
Fix error in `src/api/billing/route.ts`.
Here is the stack trace: ...
```

This prompt tells AI where to start. But it does not say:

- Do not change pricing logic.
- Do not change schema.
- What the expected behavior is.
- Which test should verify it.

AI can fix the error by changing the logic incorrectly. The error disappears. The product breaks.

In good sessions, two things usually appeared more clearly: **context** and **constraints**. Context tells AI where it is. Constraints tell AI where the boundaries are.

A good prompt does not only say:

```
Fix checkout bug in `route.ts`.
```

It says:

```
Current:
- Stripe checkout session is created.
```

```
- User is redirected to success URL even when cart is empty.

Desired:
- Empty cart should return 400 and not create Stripe session.

Out of scope:
- Do not change pricing calculation.
- Do not touch webhook handling.
- Do not change database schema.
```

That is the difference between "knowing the file" and "knowing the job."

Story 3: the first prompt in a session should be longer than a mid-session prompt

I used to be confused by two numbers that seemed to contradict each other.

In the shape analysis, strong prompts often landed around 35-85 words. But in the outcome analysis of the first prompt in a session, the good range was much longer: about 1,600-3,000 characters.

In reality, these numbers describe two different prompt types.

The first prompt in a session has to set the playing field. You need role, task, files, current/desired, scope, and verification. It should be longer.

A mid-session prompt usually only needs to steer the delta. Once context is established, a 50-word prompt can be enough:

```
Keep the fix, but revert the unrelated change in `PricingCard.tsx`.
Then rerun the same checkout reproduction.
```

The problem is that many people do it backward: the first prompt is 20 words, then mid-session prompts become 600 words because everyone is firefighting.

My rule:

- First prompt for a new task: write it carefully.
- Continuation prompt: keep it short and specify the delta.
- If a continuation prompt starts becoming an essay, you may need to reset the session.

Story 4: a few recipes clearly win

Across the 1,126 sessions with enough data, a few recurring patterns stood out.

The **Reviewer/Patcher frame** showed a very strong signal, even with a small n:

```
You are reviewing AND PATCHING <artifact> for <project>.
Read the plan from `<path>`.

Focus on:
1. Architectural gaps
2. Hidden bugs
3. Edge cases

Out of scope:
- <non-goal>

Return findings first, then patch, then verification.
```

This frame wins because it forces the right order: review first, patch second. AI does not rush into editing. It reads, evaluates, and only then changes code.

Out of scope is another block worth adding. It is short, but it has a large effect:

```
Out of scope:
- Do not redesign UI.
```

```
- Do not change billing logic.
- Do not add dependencies.
```

A **role frame** such as `You are a strict code reviewer` or `You are a senior frontend engineer focused on accessibility` also helps AI choose the right criteria.

The common thread: these recipes frame AI before asking for work. Before AI starts, you tell it where it stands in the project, which lens to use, and which line not to cross.

Story 5: 100 turns is a real wall

No session in my data went past 100 turns and still had a clean outcome. Zero.

This does not mean turn 101 is guaranteed to fail. But it says something practical: when a session gets too long, errors begin to compound. Context fills up. The original goal gets diluted. AI remembers part, forgets part, and mixes part.

I use one rule:

When a session reaches 70-80 turns and the task is still not done, prepare a handoff.

Handoff prompt:

```
Stop patching.
Summarize for a fresh session:
- Original goal
- Current status
- Files changed
- What works
- What is still broken
```

```
- Commands/tests run
- Recommended next prompt
```

Then open a new session with that summary. Do not cling to context. Dirty context is not an asset.

The skill curve is not a straight line

One thing that helped me stop blaming myself was realizing prompt skill does not rise linearly. Some months my best-rate went up. Some months it dropped because I was trying a new tool, doing more complex projects, or letting scope outrun skill.

That is normal. When you take on harder tasks, old prompts may no longer be enough. The feeling that "I am getting worse" sometimes just means you are playing a higher level.

The fastest way to recover is not reading 100 more tutorials. The fastest way is rereading your last 50 prompts.

You will see:

- I often forget verification.
- I often do not say what is out of scope.
- I often open a new task inside an old session.
- I often paste a screenshot without explaining it.
- I often let AI patch before understanding root cause.

Personal data is an annoying teacher, but a fair one.

A 30-minute exercise

You do not need to write a script like I did. This exercise is enough:

1. Open the AI tool you use.
2. Copy your last 50 prompts into a text file.
3. Mark which prompts have a clear target.
4. Mark which prompts have current/desired context.
5. Mark which prompts have a constraint.
6. Mark which prompts have verification.

Do not fix anything while reading. Just look.

If you feel a little embarrassed, good. I did too. Mild embarrassment means you just saw your real habits.

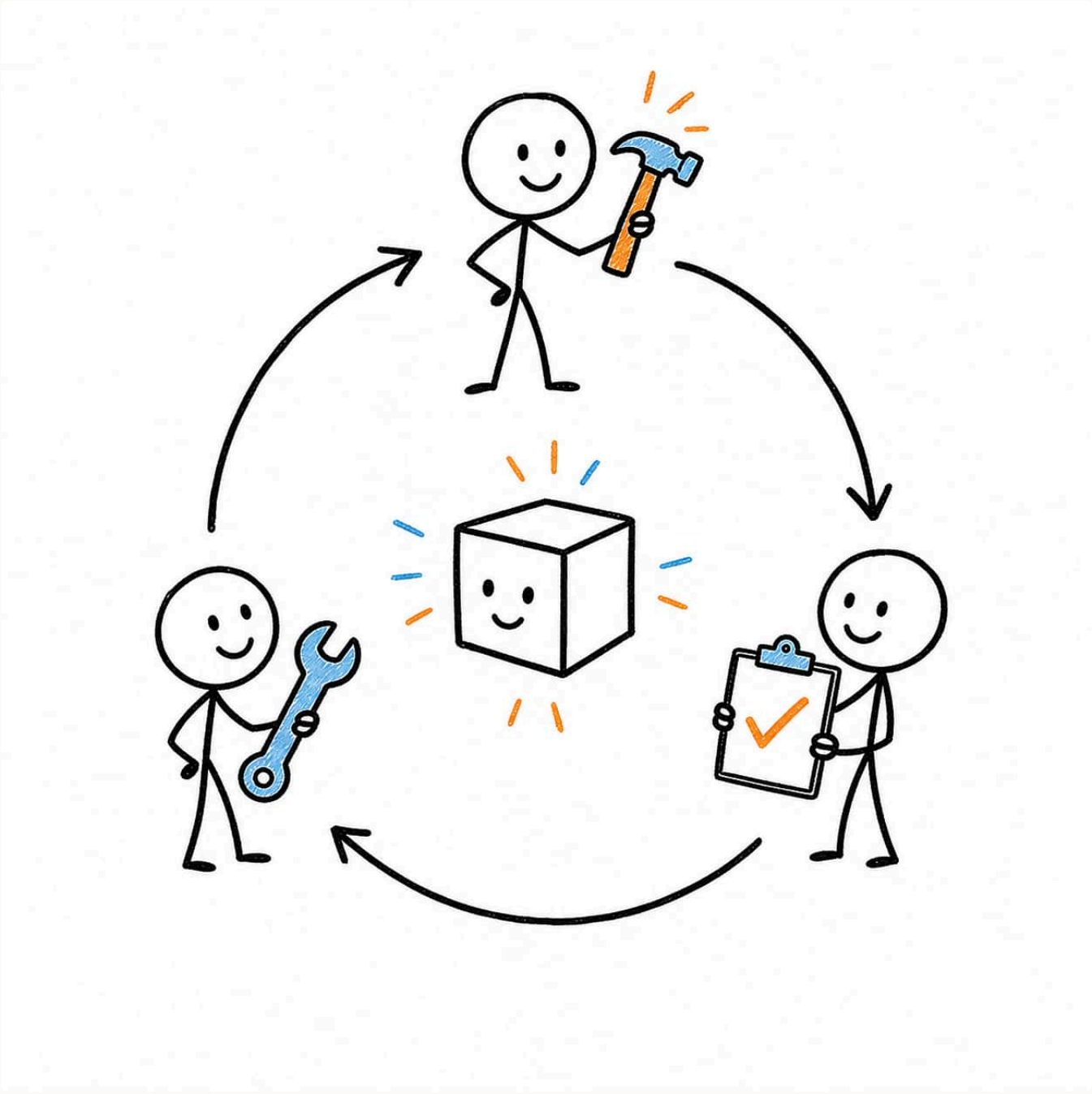
Three numbers to remember after this chapter:

- **58.6%** of my prompts were too thin.
- **15.1%** of my prompts had verification.
- **100+ turns** is almost always a reset signal.

You do not need perfect prompts. You only need prompts that are a little better than your old ones, repeated long enough.

Part 3 - The Build Loop

Chapter 5 - The Build, Test, Fix Loop



Vibe coding is not a one-shot prompt. It is a build, test, fix loop.

One of the biggest misunderstandings beginners have is thinking vibe coding happens in one prompt.

You type. AI codes. The app runs. Done.

Sometimes that is real. But with real products, the flow is usually different:

1. Build a small slice.
2. Test against evidence.
3. Fix based on evidence.
4. Verify again with the same reproduction.
5. Commit or reset context.

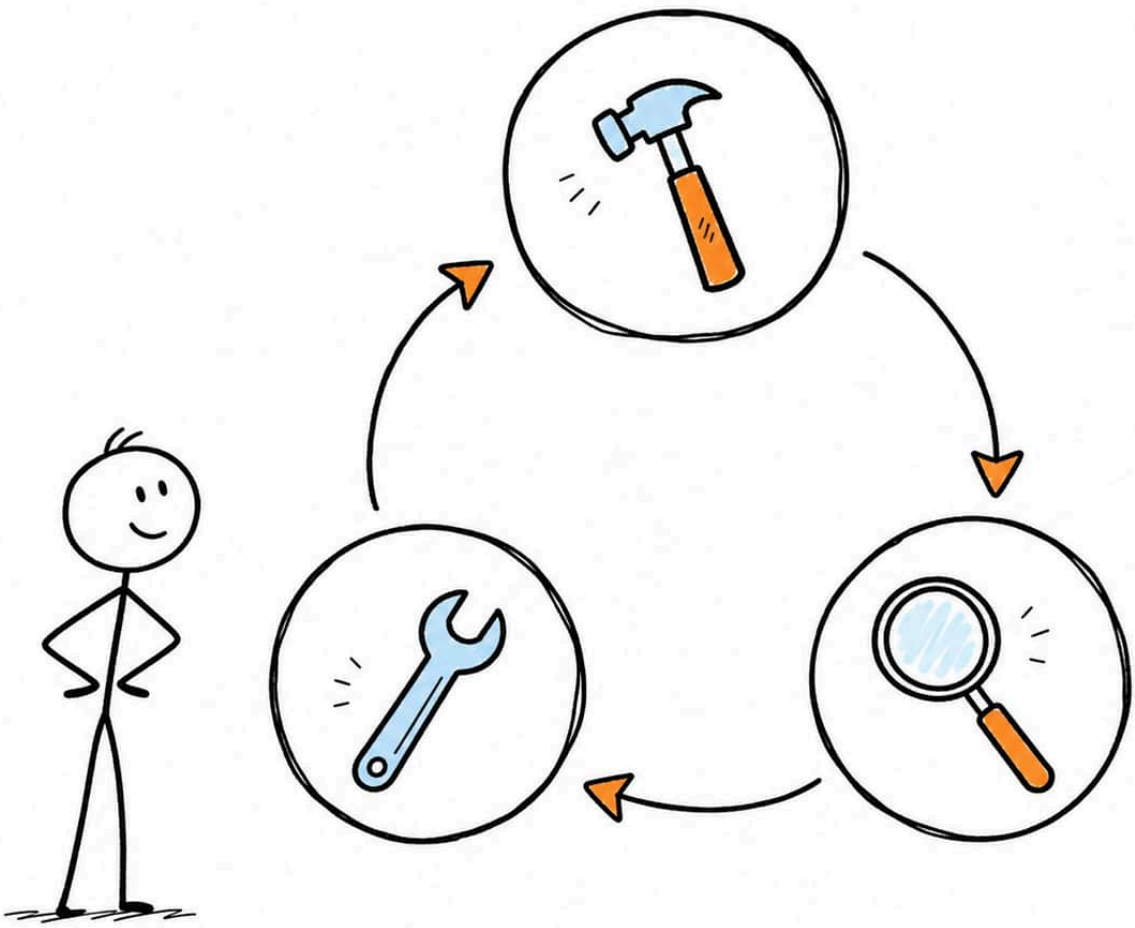
This loop sounds simple, but it is the boundary between using AI like a product builder and using AI like a slot machine.

If you only build and do not test, you get a fragile demo.

If you only test and do not know how to fix, you get stuck.

If you fix based on emotion, you create new bugs.

Build smaller than you think



Do not try to build the whole product in one loop. Each loop should have one clear slice.

AI makes it easy to get greedy. One prompt can generate many files, so you want to pack in too much:

```
Build a full dashboard with auth, analytics, settings, billing, dark mo
```

This prompt looks efficient, but it mixes too many risks. If something breaks, you do not know whether the issue is auth, chart, API, CSS, env, or deploy.

A good build should be smaller:

```
Build the first vertical slice for the dashboard.

Scope:
- `/dashboard` route only.
- Show current user's name from existing session.
- Show 3 placeholder metric cards with static data.

Out of scope:
- No charts yet.
- No billing.
- No settings page.
- No new dependencies.

Verify:
- App builds.
- `/dashboard` renders for logged-in user.
```

A vertical slice is a thin slice that works end to end. It is not complete, but it is real. You have a route, fake or small real data, basic UI, and a way to verify.

Good vibe coding is not "AI builds a lot." Good vibe coding is "I always know what the current loop is proving."

Testing is not only unit tests

Many people hear "test" and think they need a serious test suite. If your project has tests, great. But if you are building your first product, testing can start as a manual checklist.

Example for login:

```
Manual verification:
1. Open `/login`.
2. Enter valid account.
3. Submit.
4. Expect redirect to `/dashboard`.
5. Refresh `/dashboard`.
6. Expect session still valid.
```

Prompt for AI:

```
After patching, verify against this exact checklist.
If you cannot run the browser, tell me which steps I should run manuall
```

The important word is "exact." Do not let AI verify with a generic sentence:

```
Looks good.
```

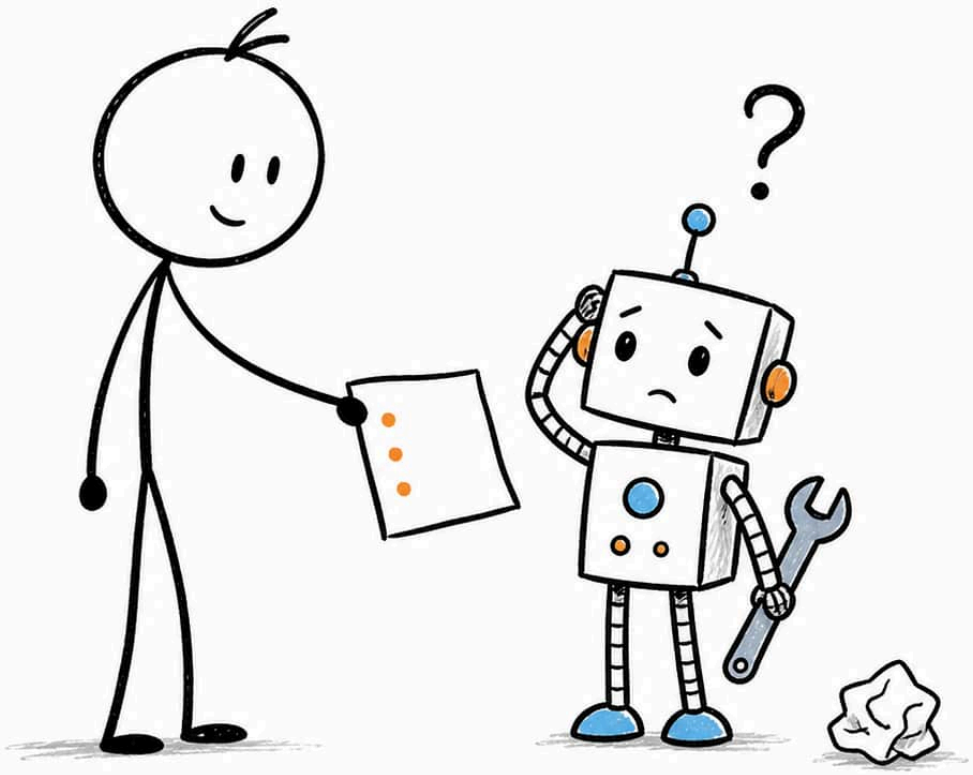
"Looks good" is not evidence.

Evidence is:

- The command that ran.
- The output that was seen.
- Which tests passed or failed.
- Which viewport a screenshot used.
- Which reproduction was retried.

The clearer your evidence, the more accurately AI can fix.

When AI is wrong



A good correction is not yelling at AI. A good correction adds new evidence.

AI being wrong is normal. The issue is not that it was wrong. The issue is how you correct it.

Weak correction:

```
Not right.
```

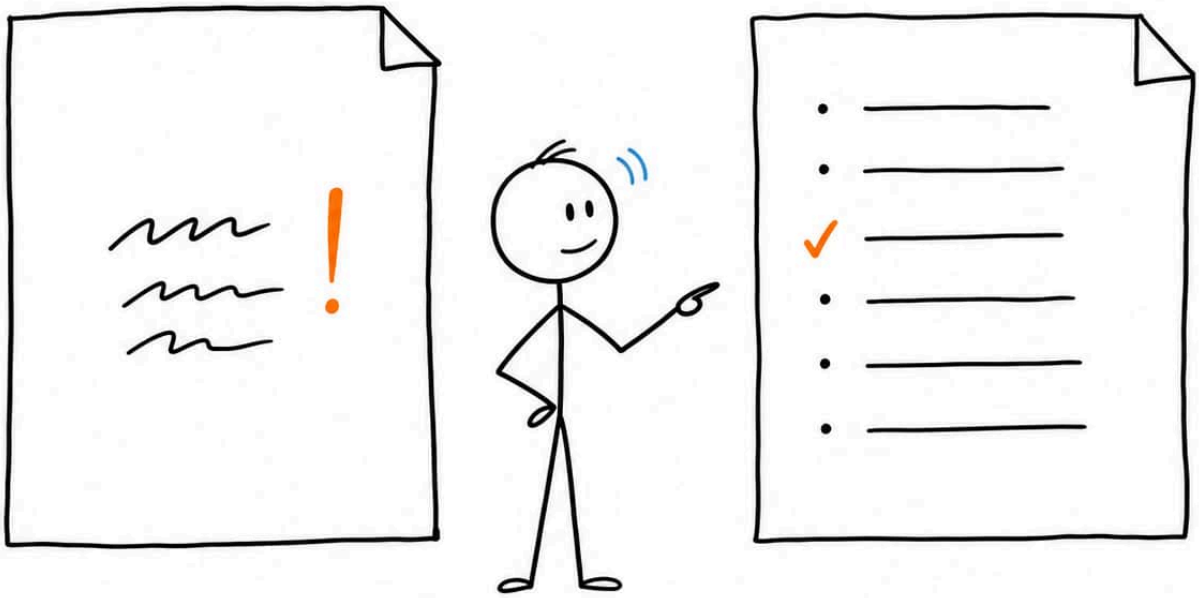
AI will ask again or guess again. Often it apologizes and edits in another wrong direction.

Good correction:

```
This is not fixed.

Observed after your patch:
- At 375px, sidebar still stays open after tapping "Settings".
- Console has no error.
- Route changes to `/settings`.

Expected:
- Sidebar should close after route change.

Please inspect why `onNavigate` is not called for mobile links.
Do not redesign sidebar.
```

You do not need to be overly polite, but you do need to be specific. Correction is not where you vent. It is where you add evidence.

Correction template



The more evidence a correction has, the less AI has to guess again.

Template I use:

```
The previous fix did not solve the issue.

Observed:
- <actual behavior after patch>

Expected:
```

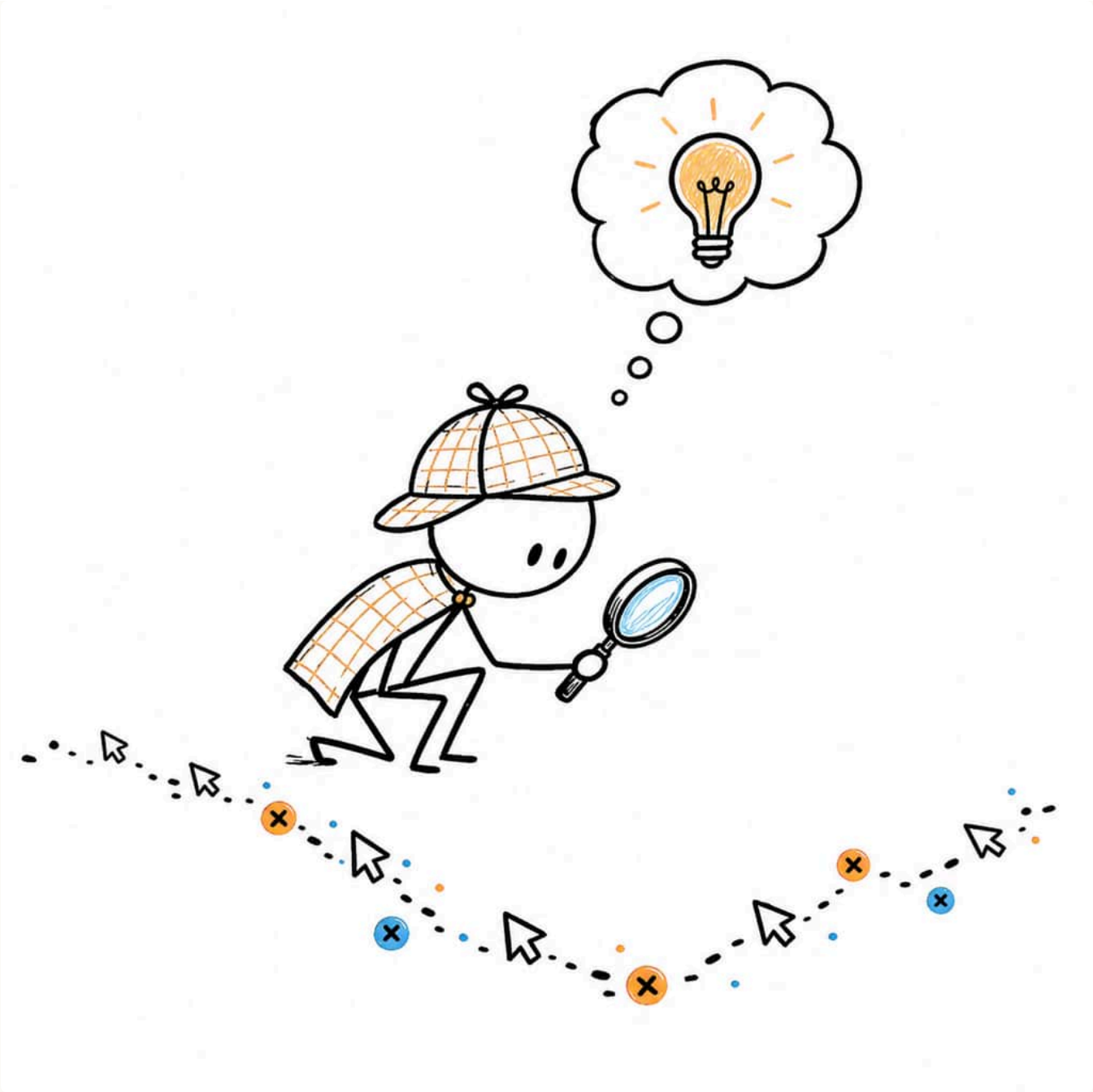
```
- <correct behavior>

Evidence:
- <log, screenshot, command output, reproduction>

What changed:
- <if you know what the previous patch changed>

Please:
1. Explain why the previous fix missed it.
2. Patch the smallest fix.
3. Rerun the same reproduction.

Out of scope:
- <thing not to touch>
```

The two strongest lines here are:

```
Explain why the previous fix missed it.
Rerun the same reproduction.
```

The first line forces AI to learn from the mistake inside the session. The second line prevents it from changing the test halfway through.

Debug with evidence



A bug is not the enemy. A bug is information.

When a bug appears, beginners ask:

```
why doesn't it run?
```

A strong vibe coder asks:

```
why does it behave that way?
```

These two questions differ in attitude. The first is emotion. The second is investigation.

Emotional debugging produces prompts like:

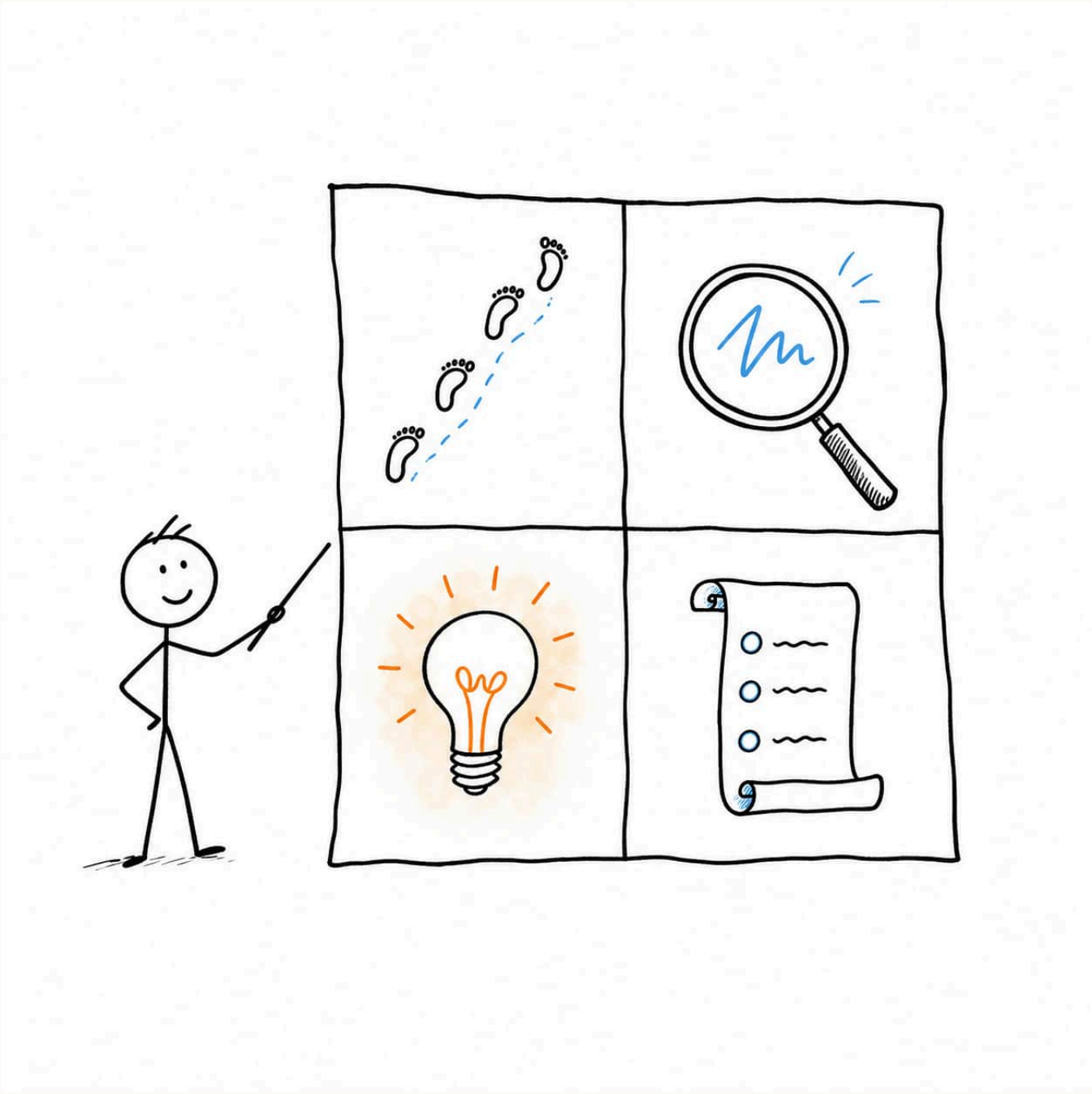
```
Fix this error fast.
```

Evidence-based debugging produces prompts like:

```
Debug checkout session creation failure.

Reproduction:
1. Add one item to cart.
2. Click Checkout.
3. Observe 500 response from `/api/checkout`.

Observed:
<paste exact server log>

Expected:
- API returns checkout URL.

Please:
1. Inspect route handler and Stripe helper before patching.
2. Explain root cause in 5–10 lines.
3. Patch the smallest fix.
4. Rerun the same reproduction.
```

A bug is not an insult. It is output from the system. The more you respect real output, the more chance AI has to fix the right thing.

The four boxes of a debug prompt



Reproduction, observed, expected, constraints - if one box is missing, AI has to guess.

A strong debug prompt has four boxes:

1. Reproduction

The steps that create the bug. Without a reproduction, AI does not know how to check the fix.

```
Reproduction:
1. Open `/settings/billing`.
2. Click "Update card".
3. Submit empty form.
```

2. Observed

What actually happened. Logs, stack traces, screenshots, network responses. Do not paraphrase too much.

```
Observed:
- Modal closes.
- No validation message.
- Console: `Cannot read properties of undefined (reading 'id')`
```

3. Expected

What should happen.

```
Expected:
- Modal stays open.
- Field error appears under card number.
- No request is sent.
```

4. Constraints

The boundaries of the fix.

```
Constraints:
- Do not change Stripe helper.
```

```
- Do not redesign modal.
- Keep existing form library.
```

If you only paste a stack trace, AI may fix the symptom. If you add expected behavior and constraints, it has a chance to fix the root cause in the right scope.

Do not change the test halfway through

One mistake I used to make often: each time AI fixed something, I tested in a different way. The original bug was the mobile menu. After the patch, I clicked the desktop dropdown instead. I found another bug and asked AI to fix it in the same thread. A few turns later, the session no longer knew what the original task was.

Rule:

Each fix loop must rerun the same reproduction first.

If the old reproduction passes, you can open a new bug. But do not mix the new bug into the old loop.

Prompt:

```
Before doing anything else, rerun the original reproduction:
1. ...
2. ...
3. ...

Tell me pass/fail.
Do not investigate new issues yet.
```

This is how you keep the session clean.

When should AI plan first?

Not every task needs a plan. But if the task touches many files, data migration, auth, billing, deploy, or a large refactor, make AI plan first:

```
Do not patch yet.
First inspect the relevant files and return:
1. Root cause hypothesis.
2. Files that need changes.
3. Risks.
4. Verification plan.
```

Planning first helps you catch errors before code changes. If the plan is wrong, the patch will almost certainly be wrong.

For small tasks, patching directly is fine:

```
Fix typo in `SignupButton.tsx`. No plan needed.
```

The important part is that you choose the mode instead of letting AI choose it for you.

Commit is a breathing point

A clean build-test-fix loop should end at a breathing point:

- Diff is within scope.
- Test or manual verification passes.
- There are no strange files.
- You understand the change.
- It can be committed.

If you are not there yet, do not open a new task. Starting a new task on top of a dirty diff is the fastest way to lose the session.

Closing prompt:

```
Summarize this change for commit:
- Goal
- Files changed
- Behavior changed
- Verification run
- Known remaining risk
```

This summary helps you write a better commit message, and it is also a handoff if you need a new session.

A sample loop

A real work loop might look like this:

```
Build first slice of profile settings.

Scope:
- Route `/settings/profile`.
- Fields: display name, avatar URL.
- Save to existing user profile API.

Out of scope:
- Password change.
- Email verification.
- Billing settings.

Before patching:
- Inspect existing settings routes and profile API.
- Return a short plan.
```

```
After patching:
- Run build/test if available.
- Provide manual verification checklist.
```

After AI patches:

```
The build passes, but manual test failed.

Observed:
- Saving display name returns 200.
- UI still shows old name until refresh.

Expected:
- UI updates immediately after save.

Please patch the smallest fix and rerun the same verification.
```

After it passes:

```
Stop here. Summarize for commit with files changed and verification.
```

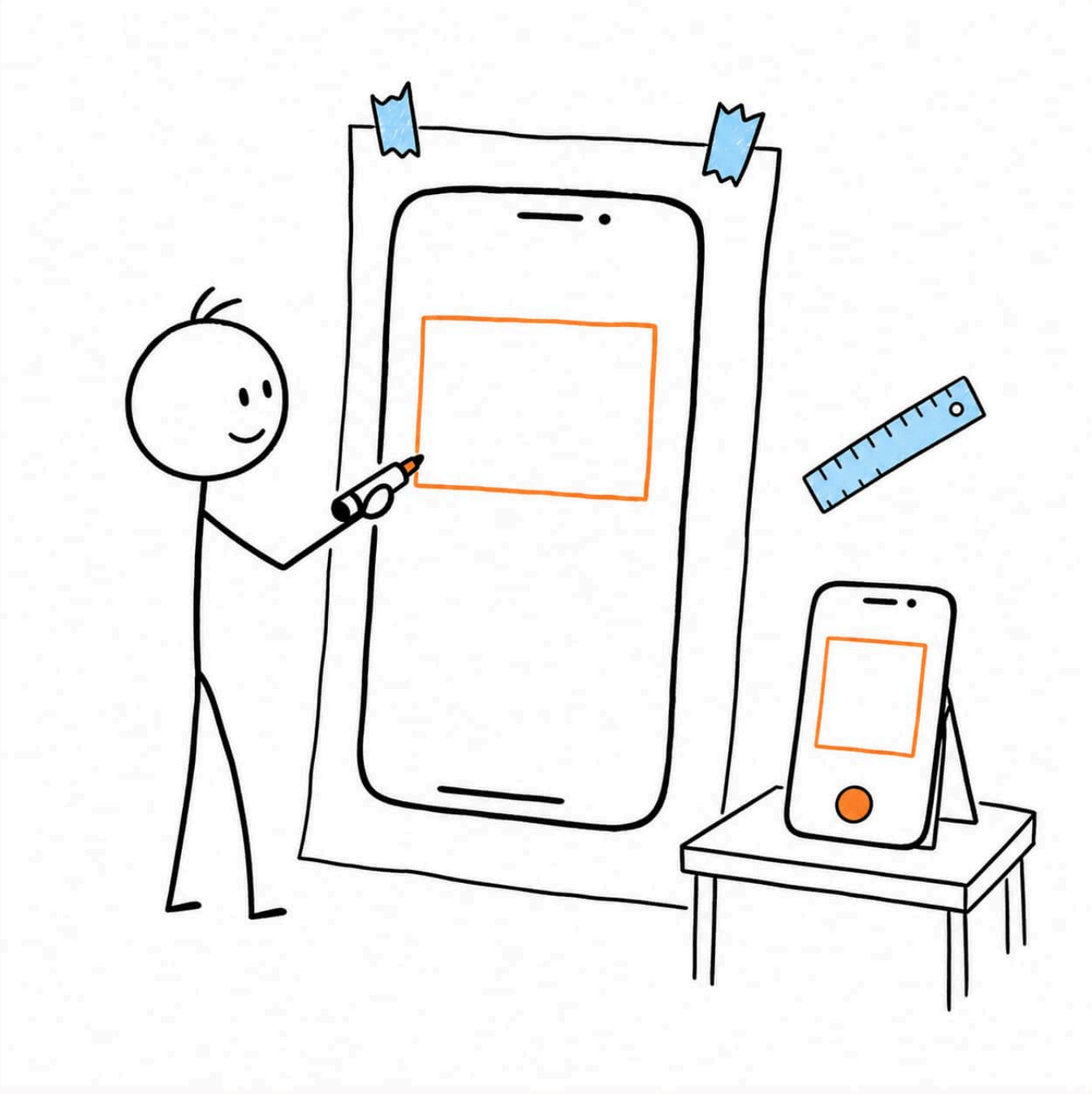
That is real vibe coding: not shiny, not magical, but very effective.

This chapter can be compressed into one sentence: **do not let AI only build; make it prove with you.**

What is the reproduction for your most recent task? If you cannot answer, you may not be ready to ask AI to fix it.

Part 4 - Shipping Real Products

Chapter 6 - Prompts for Real Products



When you move from demo to product, prompts must say clearly what users will see and where the risk lies.

A real app is not only running code. It has UI that users see, data users trust, servers that can go down, and small decisions that accumulate into product feel.

This is where the downside of vibe coding becomes visible. AI can build UI quickly, add deploy config quickly, and scaffold features quickly. But if the prompt is vague, it can also create a product that "looks done" while being hard to use, hard to operate, and hard to debug.

In this chapter, I group the three most practical prompt areas:

- UI and design.
- Ops, deploy, and server.
- Capstone: how your first project should be built.

These groups are different, but they share one thing: the closer you get to a real product, the more clearly your prompt must state **success criteria** and **the areas that must not break**.

UI prompts: do not only say "make it prettier"

Good UI has layers: hierarchy, layout, interaction, responsive behavior. "Prettier" is not enough.

The worst UI prompt is:

```
make it beautiful
```

What will AI do? It may add a gradient. It may increase border radius. It may add shadows. It may change the font. It may make your admin dashboard look like a landing page.

A good UI prompt must say:

- What the user is trying to do.
- What kind of screen this is: dashboard, form, editor, checkout, settings.
- Whether the current issue is hierarchy, spacing, responsive behavior, copy, interaction, or state.
- What needs to be preserved.
- Which viewport to verify.

Example:

```
Review and patch `src/components/OnboardingPanel.tsx`.

User goal:
- New user needs to understand the next 3 setup steps quickly.

Current issue:
- Primary CTA is visually weaker than secondary action.
- Body copy is too dense.
- Mobile layout at 375px feels cramped.

Design direction:
- Quiet SaaS onboarding.
- Dense but readable.
- No marketing hero treatment.

Out of scope:
- Do not change onboarding logic.
- Do not add animation.
- Do not introduce a component library.

Verify:
```

```
- Desktop 1280px.
- Mobile 375px.
```

Notice that this prompt does not say "beautiful." It says what the user needs to do and where the UI is failing.

UI needs state, not only screenshots

A real screen has many states:

- Empty state.
- Loading state.
- Error state.
- Success state.
- Disabled state.
- Long text.
- Mobile.
- Desktop.

AI often builds the happy path. If you only paste the prettiest screenshot, it will optimize for that screenshot.

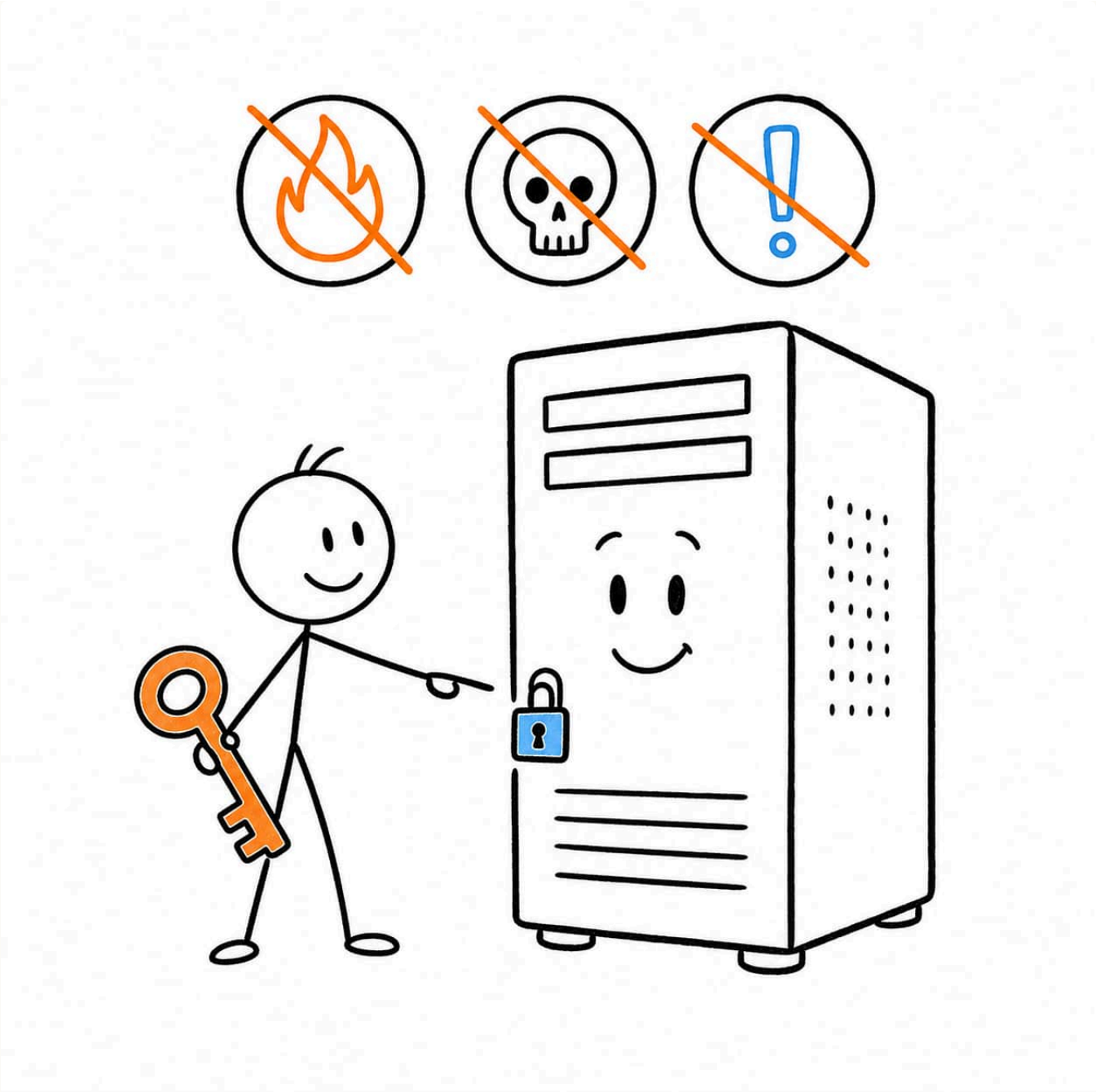
A better prompt:

```
Patch the billing settings UI.

States to handle:
- Loading current plan.
- No payment method.
- Payment method exists.
- API error when saving.
- Long plan name should not break layout.
```

```
Preserve:
- Existing API calls.
- Existing color tokens.

Verify:
- 375px mobile.
- 768px tablet.
- 1280px desktop.
```

If your app only looks good in the normal state, it is not done. Users often meet products in bad states: slow network, missing data, long text, API errors.

Ops prompts: safety before speed



For ops and deploy, a good prompt starts with safety boundaries.

Ops is where I am strictest with AI.

A UI bug annoys users. A deploy bug can take the app down. A wrong command can delete data. An env leak can cost you a whole day of cleanup.

So ops prompts must not be vague:

```
help deploy this
```

Not enough.

Ops prompts should include safety rails:

```
You are helping me debug a staging deploy.

Environment:
- Staging only.
- Do not run destructive commands.
- Do not modify production config.
- Do not print secrets.

Current:
- Build passes locally.
- Deploy fails at migration step.

Evidence:
<paste sanitized log>

Please:
1. Identify likely root cause.
2. Suggest read-only inspection commands first.
3. Ask before any command that changes remote state.
```

The important phrase is: **read-only first**.

Before fixing a server, look. Before running a migration, inspect. Before changing config, back up or understand rollback.

Do not paste secrets

It sounds obvious, but when a deploy is failing, people paste too much. `.env`, API keys, database URLs, tokens, private keys - all can slip into a prompt if you are not

careful.

My rule:

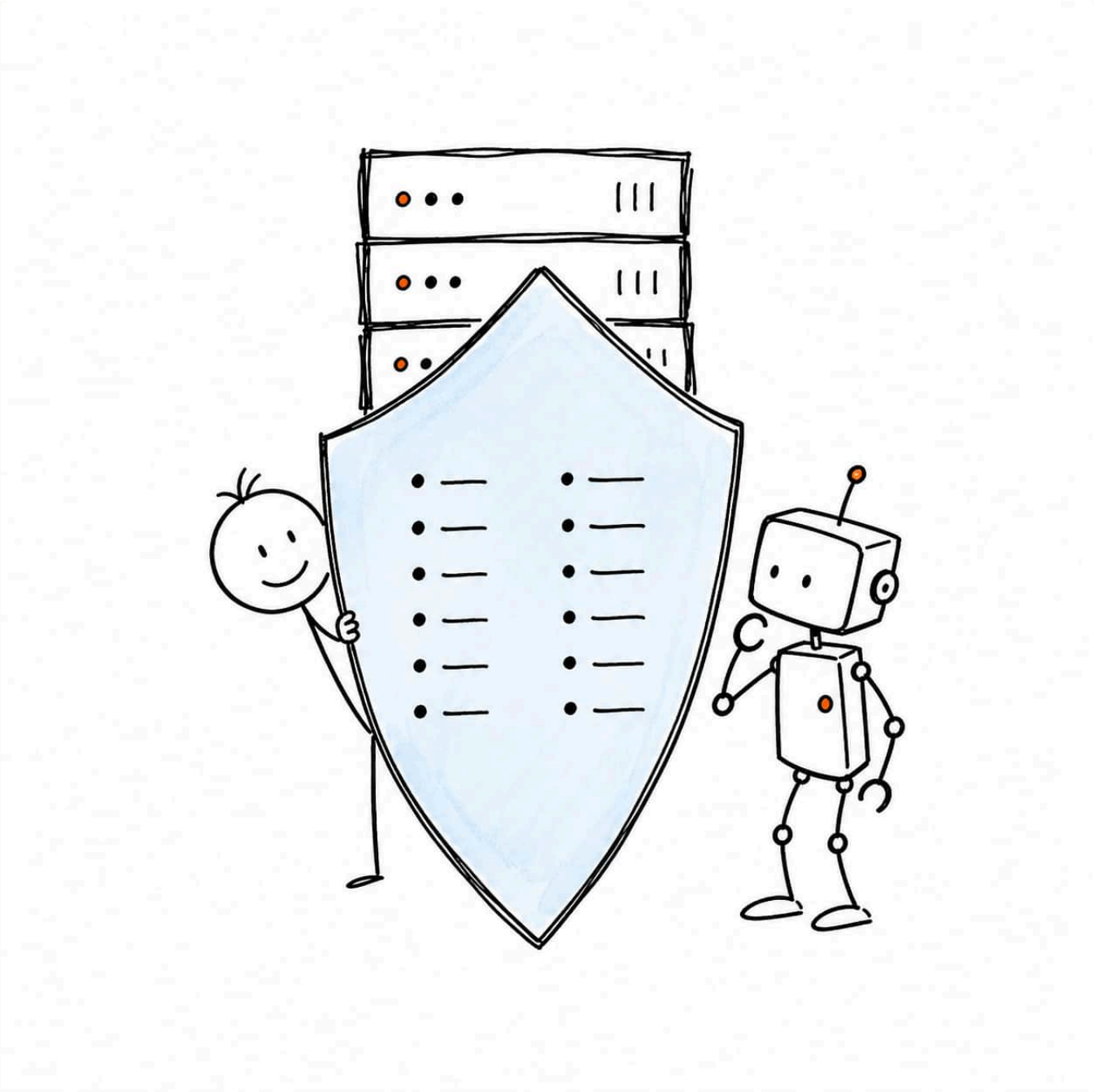
- Do not paste real secrets.
- Sanitize logs before sending.
- If you need to show format, use placeholders.
- If AI asks for a secret, refuse and provide another way to verify.

Prompt:

```
I will not paste secrets.
Assume env vars exist with these names:
- `DATABASE_URL`
- `STRIPE_SECRET_KEY`
- `NEXT_PUBLIC_APP_URL`

Debug based on names and sanitized logs only.
```

AI does not need the real key to debug 90% of issues. It needs to know which variables exist, what error happened, and which command ran.

CLAUDE.md or project rules are a shield



A good rules file helps AI remember safety rules in the middle of a session.

If you use coding agents often, keep a project rules file such as `CLAUDE.md`, `AGENTS.md`, or the equivalent. It does not replace prompts, but it is a default shield.

Keep the content short:

```
# Project rules

- Never commit secrets.
```

```
- Never run destructive database commands without explicit approval.
- For production changes, propose a plan first.
- Prefer existing patterns and helpers.
- Do not add dependencies without explaining why.
- Run tests/build when relevant.
```

This file is especially useful when sessions are long or multiple agents are working together. You do not want to repeat "do not touch production" in every prompt.

But do not be naive. A rules file does not replace attention. It reduces risk; it does not remove risk.

Capstone: your first project should not be a portfolio

A first project does not need to be large. It needs to prove you can complete the loop.

Many beginners ask what they should build to learn vibe coding. The common answer is a portfolio. I do not like that answer much.

A portfolio is easy to make pretty, but it does not force you to learn the full product loop. It may have no auth, no data, no error state, no complex deploy, and no real user flow.

Your first project should be a **proof of process**, not a showpiece.

It should be small enough to finish in 1-2 weeks, but real enough to have:

- One main user flow.
- Data stored somewhere.
- UI with at least 2-3 states.
- A bug you can debug.
- A public deploy.
- A README that says how to run it.

Good examples:

- Personal habit tracker.
- Mini CRM for freelancers.
- Family expense tracker.
- Bookmark manager.
- Writing tracker.
- Simple waitlist + admin view.

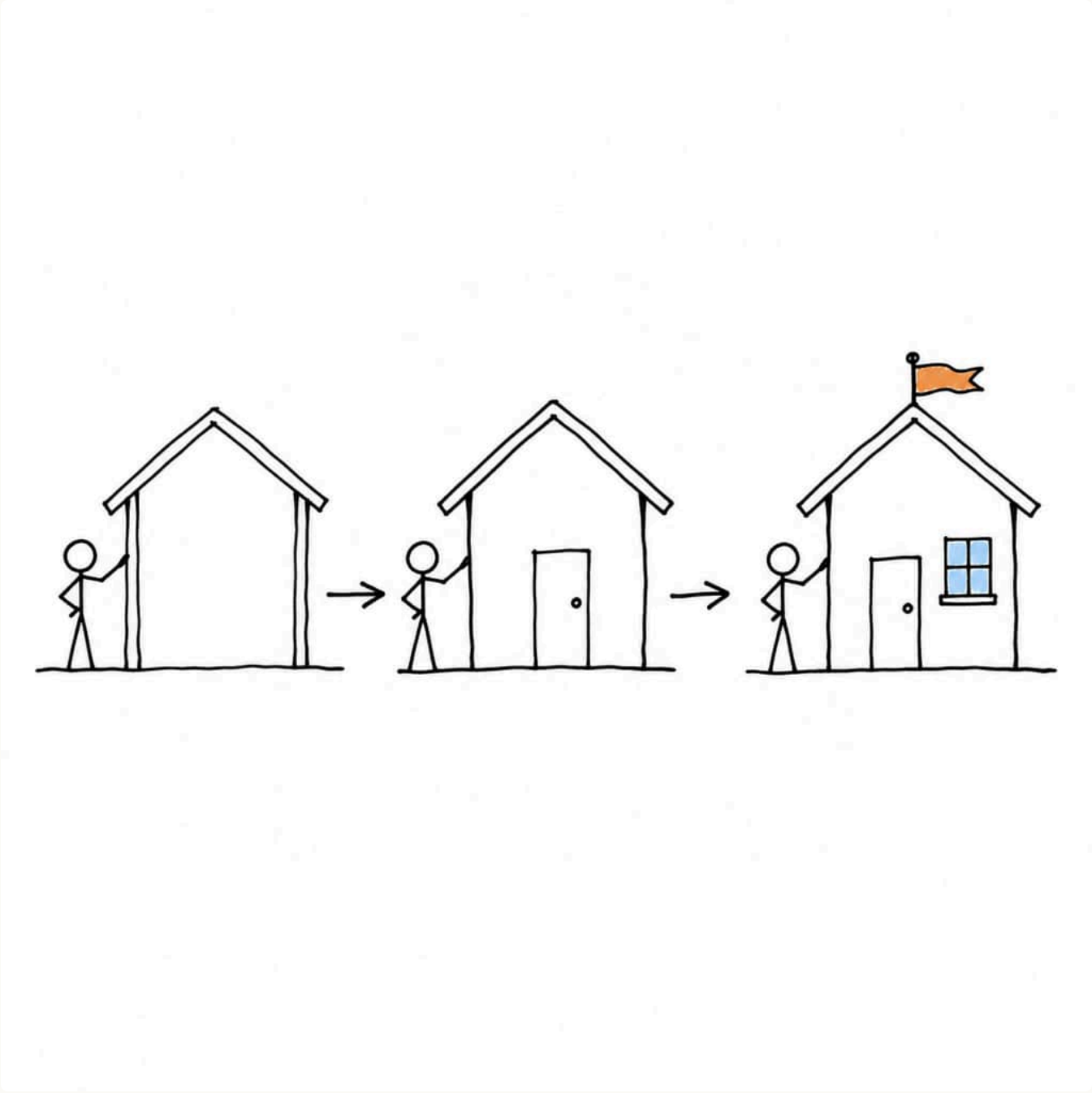
Too easy:

- Static portfolio.
- Text-only landing page.
- Todo app with no persistence.

Too ambitious:

- Marketplace.
- Social network.
- Full LMS.
- Multi-tenant SaaS billing from the beginning.

Skeleton, feature, polish



Do not polish before the skeleton runs. Do not add features before the core flow is clean.

A good capstone should move through three layers.

1. Skeleton

Goal: the app works end to end, even if it is ugly.

Prompt:

```
Build skeleton for a habit tracker.

Scope:
- Home route.
- Habit list route.
- Add habit form.
- Store data locally or in existing DB helper.

Out of scope:
- Auth.
- Charts.
- Streak logic.
- Notifications.

Verify:
- User can add a habit and see it in list.
```

2. Feature

Goal: one useful main flow.

```
Add daily check-in for habits.

Current:
- User can create habits.

Desired:
- User can mark a habit done for today.
- List shows today's completion state.

Out of scope:
- Weekly charts.
- Reminder notifications.
```

3. Polish

Goal: make the product usable, not only runnable.

```
Polish habit list UX.

Focus:
- Empty state.
- Loading state.
- Long habit names.
- Mobile 375px.

Out of scope:
- No new features.
- No data model changes.
```

The order matters. If you polish before the skeleton, you are painting walls before the house has a foundation. If you add features before the core flow is clean, you multiply bugs.

Definition of done for a first project

Your first project is done when:

- Someone else can open the URL and understand what the app does.
- The main flow works without you standing next to them explaining it.
- The README explains how to run it locally.
- There is a public deploy.
- There are no secrets in the repo.
- You know which files do what.
- You can explain the 3 biggest bugs you hit and how you fixed them.

Do not wait for the app to be perfect. Perfect is a very polite trap. It makes you polish forever and never ship.

Ship a small version, learn from it, then build the next version.

Full prompt for a capstone

You can start a project with this prompt:

```
You are helping me build my first vibe-coding capstone.

Project:
<describe the app in 3-5 lines>

Goal for this session:
- Build the first vertical slice only.

Must include:
- One user flow that works end-to-end.
- Basic UI states.
- Simple persistence.
- README update.

Out of scope:
- Advanced auth.
- Billing.
- Complex charts.
- Notifications.
- Any feature not needed for the first slice.

Before patching:
1. Inspect current project structure.
2. Propose a 3-step plan.

After patching:
1. Run build/test if available.
```

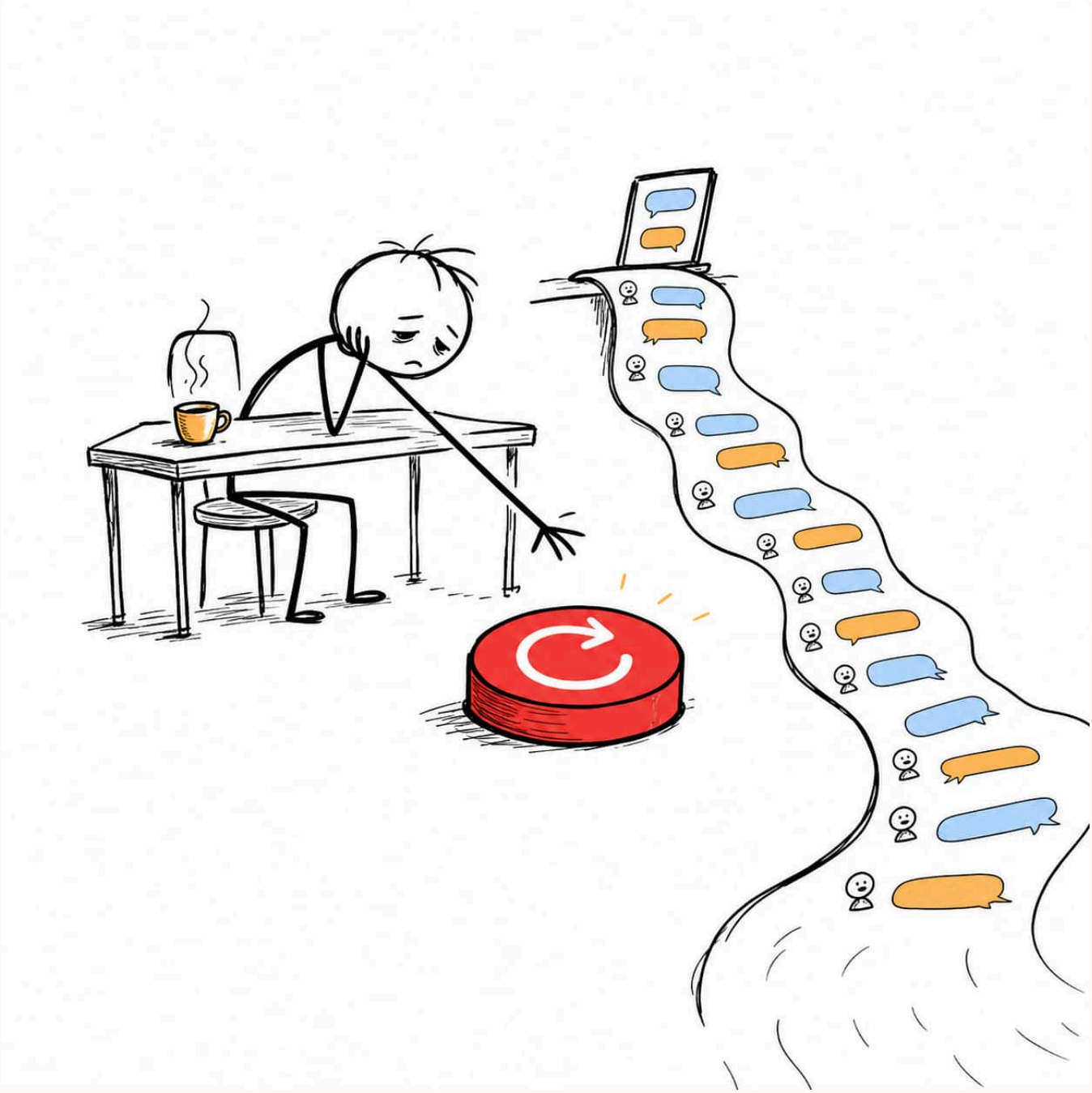
```
2. Give manual verification checklist.
3. Summarize what remains.
```

If AI proposes too much, cut it back. If it wants to add a new package, ask why. If it changes scope, pull it back to the vertical slice.

Your first product does not need to impress people. It needs to teach you one thing: I can use AI to complete the loop from idea, code, test, fix, deploy, to a real URL.

Are you trying to build your first project to show off, or to prove that your process is solid enough?

Chapter 7 - Working With AI Over the Long Haul



The problem with long sessions is not that they are long. The problem is having no reset point.

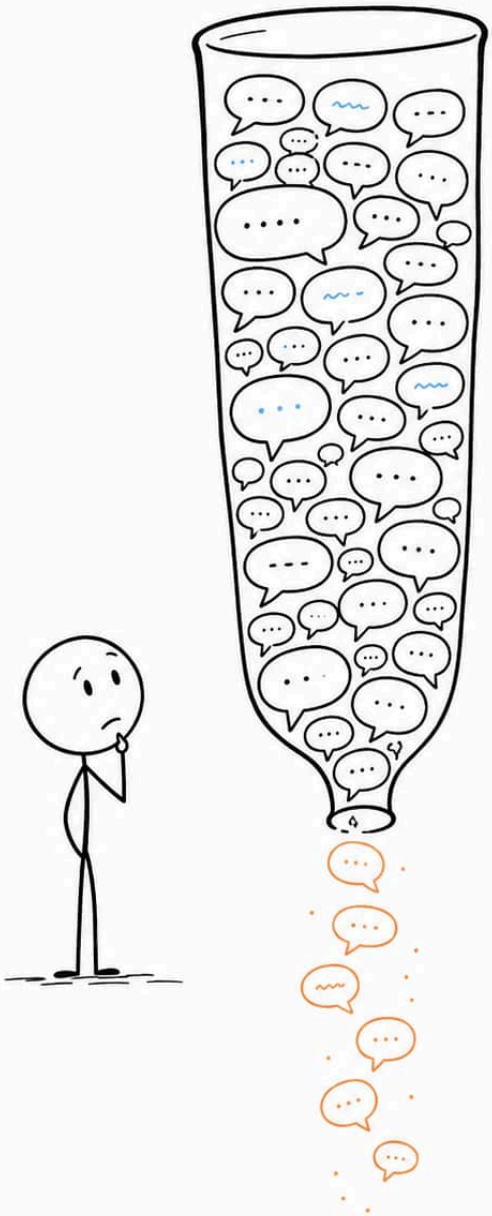
A good prompt helps you start. A good workflow helps you go far.

When I first started using AI, I hated losing context. If a session had a lot of history, I wanted to keep it. I thought AI already "knew" the project, and opening a new session would mean explaining everything again. So I kept going.

Then a 30-turn session became 60 turns. 60 became 100. By then, AI started mixing old bugs with new requests, mentioning files that had changed but were no longer accurate, or fixing something I had already reverted. I was tired too, so my prompts became shorter and more irritated.

That was when I understood: **long context is not automatically good context.**

The context window fills up



Context is like a workbench. Put too much on it, and the important thing gets covered.

AI works inside a context window. You can imagine it as a table. At the start of a session, the table is clean. You place the goal, files, logs, and plan on it. AI can see fairly clearly.

After 50 turns, the table has:

- The original bug.

- Patches already tried.
- Patches that were reverted.
- A side request.
- An old screenshot.
- A log that is no longer true.
- A file that changed later.
- An "actually, also..." from 20 turns ago.

AI still answers fluently, so you think it still has everything. But in reality, the important signal has become noisy.

A long session is not automatically bad. A long session can be very good if it has clear milestones. But when a long session drifts away from its goal, that is context debt.

Three reset signals

I usually reset when I see one of three signals.

1. AI starts editing outside scope.

You ask it to fix the mobile sidebar, and it changes the entire theme provider. Maybe the prompt was unclear. Maybe context has drifted.

2. You have to repeat the same thing too many times.

```
Don't change billing.
```

If you have to say it a third time, open a new session with that constraint clear from the beginning.

3. The session passes 70-80 turns and the main task is still not done.

This is not a mystical number. It is practical experience. In my data, 100+ turn sessions did not have clean outcomes. The closer you get to that zone, the cheaper a reset becomes.

Reset is not giving up

A good reset is a structured handoff:

```
Stop here. Do not patch.

Create a handoff for a fresh session:
1. Original goal.
2. Current status.
3. Files changed.
4. What works now.
5. What is still broken.
6. Commands/tests already run.
7. Constraints that must be preserved.
8. Recommended next prompt.
```

Then open a new session:

```
Continue from this handoff.

<paste handoff>

Before patching:
- Re-read the listed files from disk.
- Confirm current state.
- Then propose the smallest next fix.
```

The line "re-read from disk" matters. Do not let the new session trust only the summary. The summary is the map; disk is the land.

Milestones for long sessions

If the task is large, do not wait until things are tangled before resetting. Split milestones at the start.

Example for a large module refactor:

```
We will do this refactor in milestones.

Milestone 1:
- Inspect current behavior.
- Write characterization tests if feasible.
- No behavior change.

Milestone 2:
- Extract helper functions.
- Preserve public API.

Milestone 3:
- Remove duplicate code.
- Run full verification.

Only work on Milestone 1 now.
```

AI often jumps to Milestone 2 when it sees an easy opportunity. You have to hold it back.

Mid-session prompt:

```
Stay on Milestone 1.
Do not refactor yet.
```

```
Only add tests that capture current behavior.
```

Milestones do two things: reduce scope for AI and reduce your anxiety. You no longer feel like you have to solve the whole mountain in one prompt.

Handoff between human and AI

A good handoff is not only for AI. It is for you tomorrow.

After a long session, ask for:

```
Write a human-readable session summary:
- What I asked for.
- What you changed.
- Why those changes were made.
- How to verify.
- What remains risky.
```

Do not accept a generic summary. If AI writes:

```
Updated the UI and fixed bugs.
```

Make it rewrite:

```
Be specific. Include file paths and exact behavior changed.
```

A good summary is an asset. It helps you commit, write a changelog, open an issue, or reset the session.

Build a personal prompt library

Good prompts are assets. Do not let them disappear into chat history.

After a while, I realized I was retyping the same prompts again and again:

- Debug a bug with reproduction.
- Review a plan before implementation.

- Patch responsive UI.
- Deploy staging safely.
- Reset a long session.
- Handoff to a new session.
- Review diff before commit.

If a prompt has saved you twice, save it.

A personal prompt library does not need a complex tool. A folder is enough:

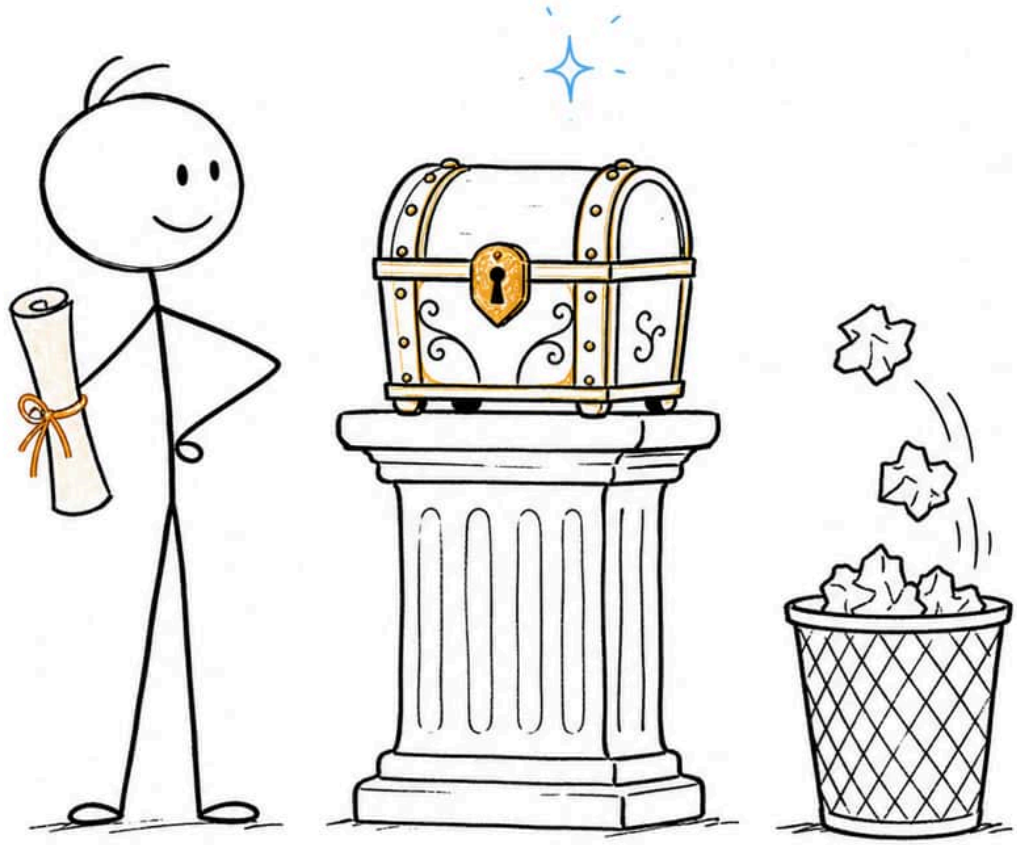
```
prompts/
  debug.md
  ui-review.md
  ops-safe-deploy.md
  reviewer-patcher.md
  session-handoff.md
  commit-summary.md
```

Each template should include:

- When to use it.
- Prompt template.
- One real example.
- Required fields.
- Common mistakes.

Do not save 100 templates. You will not use them. Start with the 5 templates you use most.

Templates are living assets



A template is not a magic spell. It is a living asset: use it, then adjust it.

A good template today may become weak when the project changes. When you move from a small frontend to a full-stack app, your old UI template may not be enough. When you start doing real deploys, your ops template needs safety rails. When you use a new tool, your reset prompt may need to change.

So each template should have a light version marker:

```
## debug.md

Last updated: 2026-05-08

Use when:
- Bug has reproducible behavior.
- Need root cause before patch.

Template:
...

Notes:
- Add viewport for UI bugs.
- Add sanitized logs for server bugs.
```

After each good session, ask:

```
What part of this prompt made the biggest difference?
```

After each bad session, ask:

```
What was missing from my first prompt?
```

That is how templates evolve from real experience, not theory.

Three templates to have immediately

If you do not have a prompt library yet, start with these three.

1. Debug with evidence

```
Debug <bug>.
```

```
Reproduction:
1. ...
2. ...
3. ...

Observed:
<actual output/log/screenshot>

Expected:
<correct behavior>

Constraints:
- ...

Please inspect first, explain root cause, patch smallest fix, rerun rep
```

2. Reviewer/Patcher

```
You are reviewing AND PATCHING <artifact> for <project>.

Relevant files:
- `<path>` - <purpose>

Focus on:
1. ...
2. ...
3. ...

Out of scope:
- ...

Return findings first, then patch, then verification.
```

3. Session handoff

```
Stop patching.
Create a handoff:
- Goal
- Files changed
```

```
- Current status
- What works
- What is broken
- Verification run
- Next prompt
```

These three templates cover most real work: fixing bugs, reviewing/patching, and resetting.

Do not turn your library into a graveyard

Another mistake: saving many prompts and never using them. The prompt library becomes a template graveyard.

How to avoid that:

- Put templates where you see them while working.
- Keep filenames clear.
- Delete templates you do not use.
- After each use, edit the template if needed.
- Do not save prompts that are too specific to reuse.

A good template should help you start faster, not make you spend 10 minutes choosing a template.

Long-haul work is energy management

Vibe coding is not only managing AI. It is also managing your own energy.

When you are tired, prompts get worse. You omit context. You do not read diffs. You trust "done." You add scope because you want to finish everything now.

A few personal rules:

- If I am too tired to read the diff, I stop.
- If I am irritated, I do not correct with emotion. I write observed/expected.
- If the session is too long, I hand off.
- If the task passes, I commit before opening a new task.
- If the first prompt is vague, I rewrite it before sending.

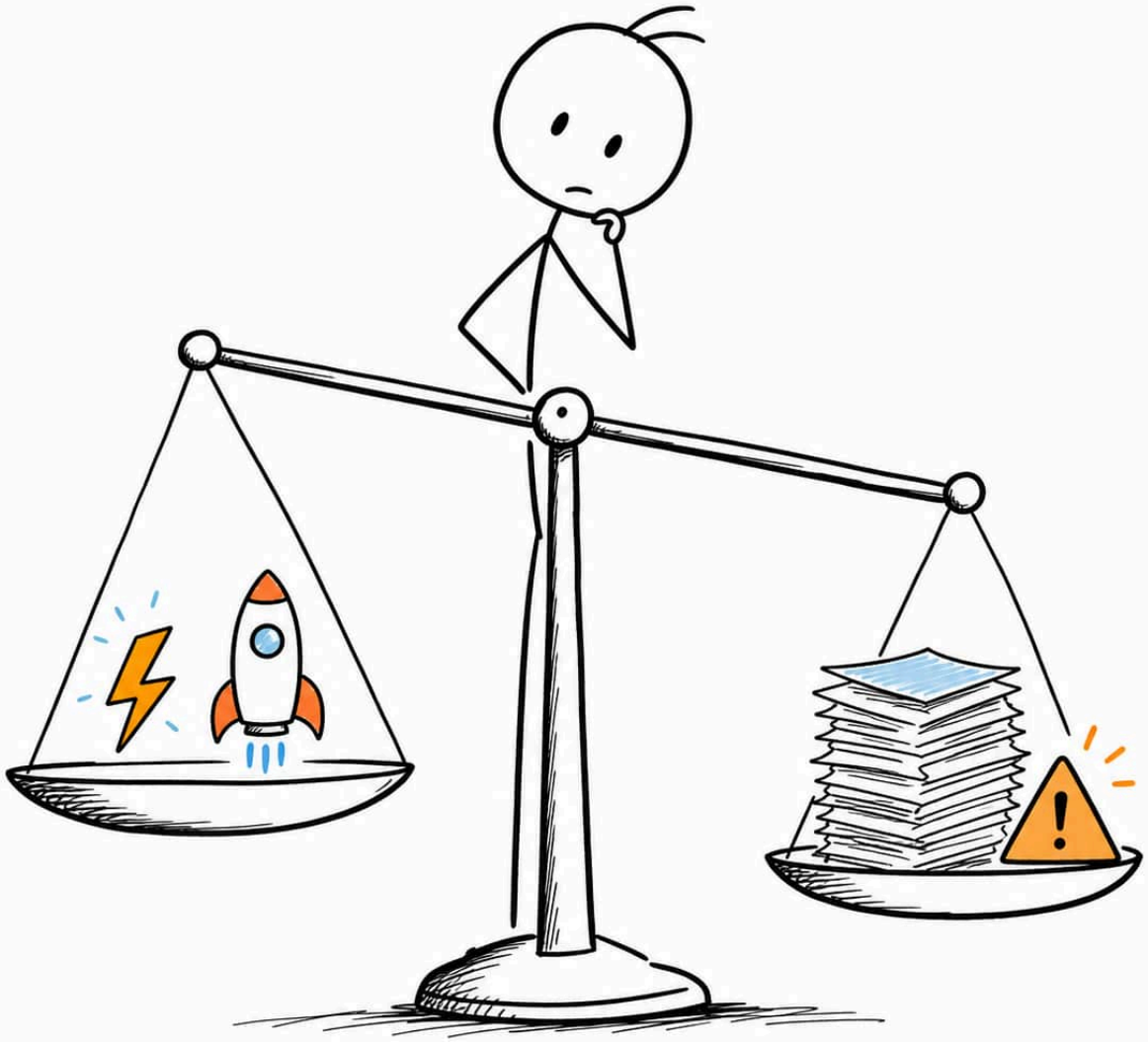
This sounds simple, but it is hygiene. Good AI users are not the people with the fanciest prompts. They are the people who keep the workflow clean long enough.

This chapter wants you to remember one thing: **AI can run very fast, but you have to set checkpoints.**

Did your most recent session have a checkpoint, or was it just one chat line that kept stretching?

Part 5 - Toolbox

Chapter 8 - Toolbox and Closing Thoughts



Vibe coding has a bright side and a dark side. Whether you can go far depends on how you keep the balance.

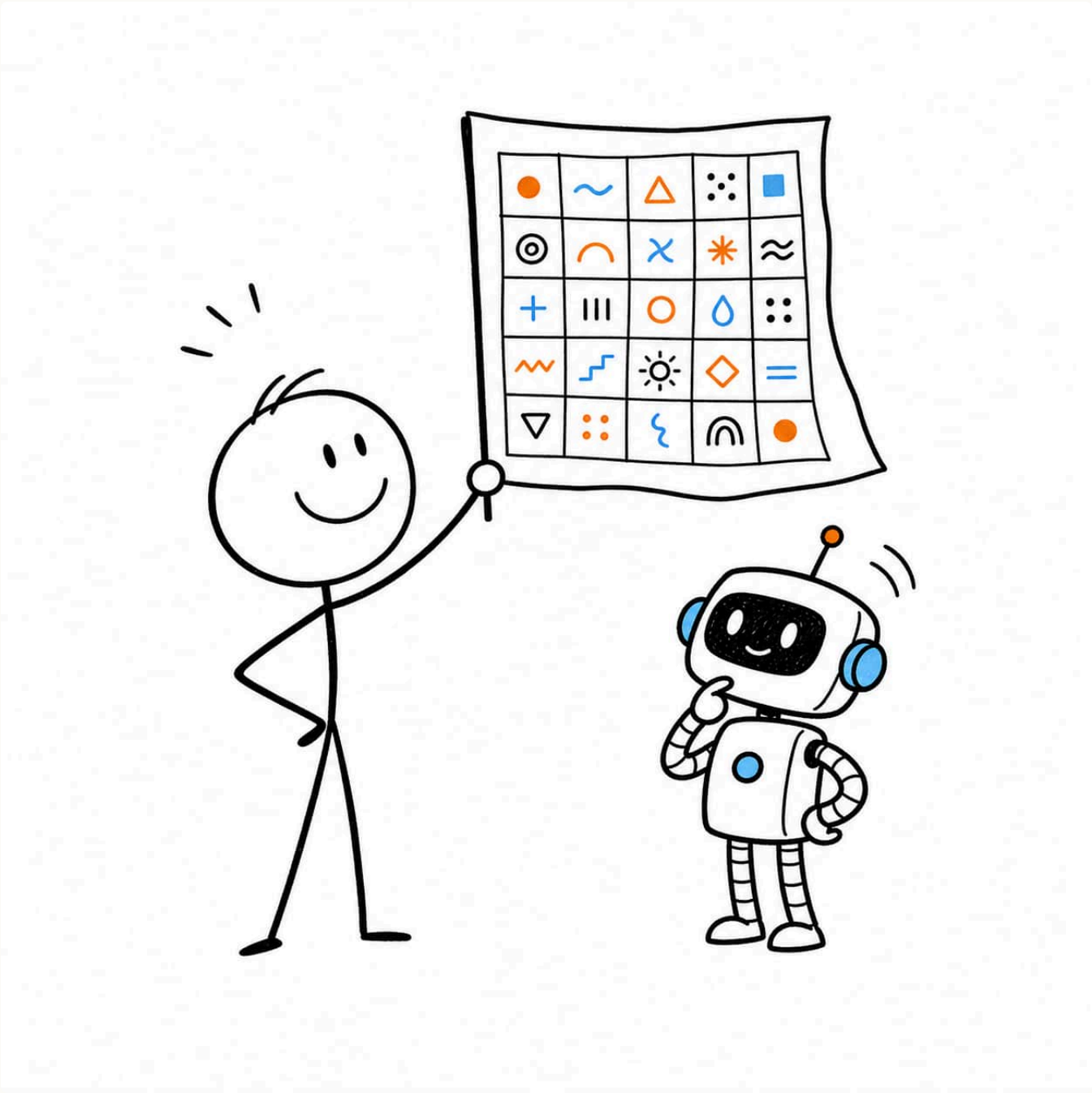
If the previous seven chapters were about how to think and how to work, this chapter is the toolbox.

I collected here the things you can copy and use immediately: cheat sheet, templates, the Reviewer/Patcher frame, winning recipes, and anti-recipes to avoid. At the end

is the part I want to say plainly about the two sides of vibe coding, because only describing the bright side would not be fair.

Do not try to use every template at once. Choose 2-3 that fit your current workflow, use them for real, edit them for real, and only then add more.

Cheat sheet before sending a prompt



One page is enough to pull a prompt from vague to clear.

Before pressing Enter on an important prompt, ask quickly:

1. **Is the action clear?**
   Does AI need to review, fix, debug, refactor, design, deploy, or summarize?

2. **Is the target clear?**
   Is there a specific file/path, route, component, log, screenshot, or artifact?

3. **Do you have current/desired?**
   What is happening now, and what do you want instead?

4. **Do you have evidence?**
   Is there a real error, real output, or real reproduction?

5. **Do you have out of scope?**
   What must not be touched?

6. **Do you have verification?**
   How will you know it is correct?

If your prompt is missing 2-3 items in this checklist, it is not ready for a large task.

Six small questions before sending are often cheaper than six correction loops afterward.

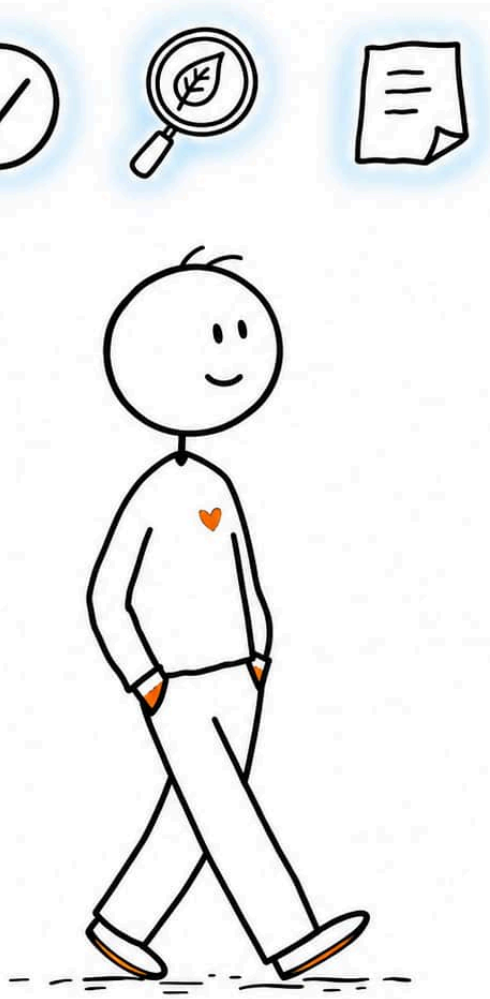
A good minimum prompt:

```
Fix <issue>.

Target:
- `<file/path>`
```

```
Current:
- <current behavior>

Desired:
- <correct behavior>

Out of scope:
- <thing not to change>

Verify:
- <test or manual check>
```

You can add role, output format, plan, or evidence depending on the task. But if target, context, constraint, and verification are missing, be careful.

Three permanent habits



Good prompting is a habit, not a one-time trick.

If I could keep only three habits after reading this book, I would choose:

1. Current vs desired.

Do not only say "fix." Say how the current behavior is wrong and what the correct behavior should be.

2. Out of scope.

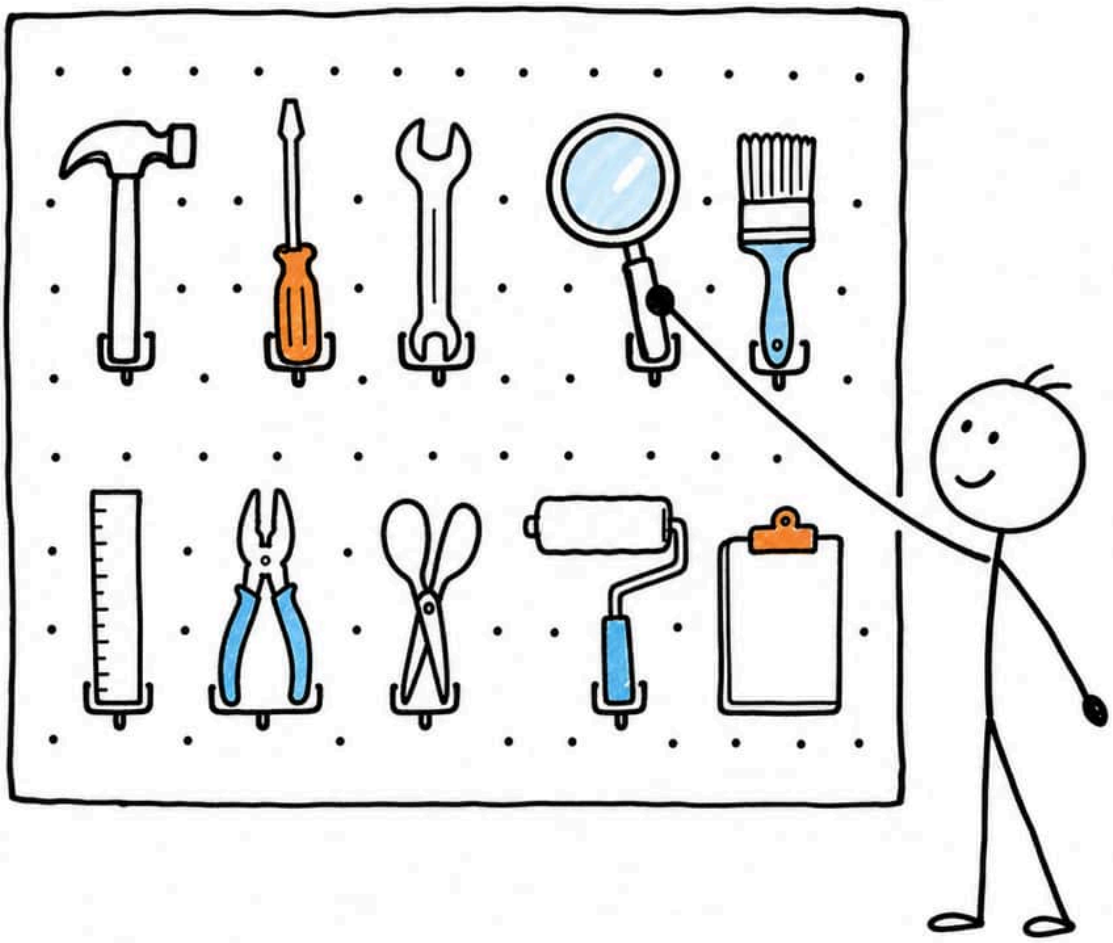
Do not only say what needs to be done. Say what must not be touched.

3. Verify.

Do not accept "done" as a result. Make AI prove it with test, build, reproduction, browser, or checklist.

These three habits do not make you a flashy prompt engineer. They make you a clearer collaborator.

Copy-paste template set



Good templates do not replace thinking. They help you start from the right structure.

1. Small code change

```
Patch <small change>.

Target:
```

```
- `<file>`

Requirements:
- <requirement 1>
- <requirement 2>

Out of scope:
- <non-goal>

Verify:
- <test/build/manual check>
```

2. Debug bug

```
Debug <bug>.

Reproduction:
1. ...
2. ...
3. ...

Observed:
<actual log/output>

Expected:
<correct behavior>

Please inspect first, explain root cause, patch smallest fix, rerun rep
```

3. UI review

```
Review and patch UI for `<component/path>`.

User goal:
- <what user is trying to do>
```

```
Current issue:
- <hierarchy/spacing/responsive/copy/state issue>

Design direction:
- <product style>

Out of scope:
- No redesign outside this component.
- No new component library.

Verify:
- 375px mobile.
- 1280px desktop.
```

4. Safe ops

```
Help debug <deploy/server issue>.

Environment:
- <local/staging/production>

Safety:
- Read-only inspection first.
- Do not run destructive commands.
- Do not print secrets.
- Ask before changing remote state.

Evidence:
<sanitized log>

Return:
1. Likely root cause.
2. Read-only checks.
3. Proposed fix.
```

5. Session handoff

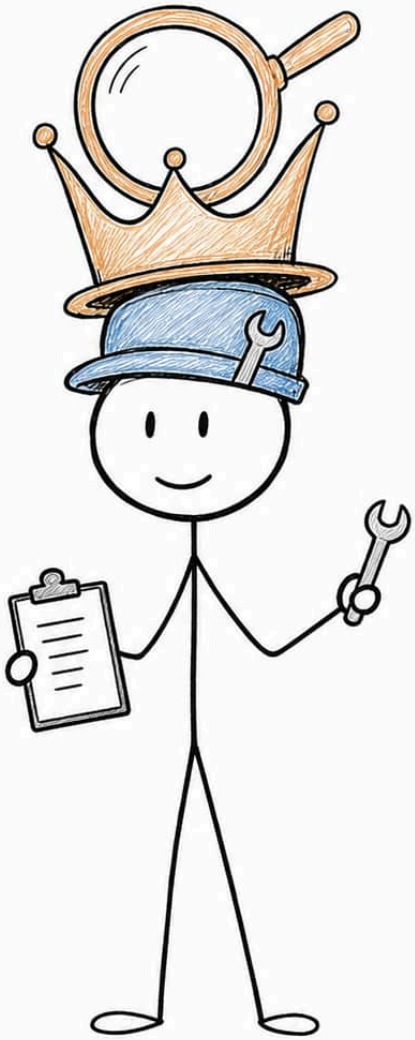
```
Stop patching.

Create a handoff:
- Original goal
- Current status
- Files changed
- What works
- What is still broken
- Verification run
- Constraints to preserve
- Next prompt for fresh session
```

6. Review before commit

```
Review current diff before commit.

Focus:
1. Bugs or regressions.
2. Scope creep.
3. Missing verification.
4. Risky changes.

Return findings first, with file/path references.
Do not patch unless I ask.
```

You do not need to use the exact words. But keep the structure: target, context, scope, verify.

Reviewer/Patcher frame



Review first, patch second: two roles in one head.

If I had to choose the strongest template for complex tasks, I would choose Reviewer/Patcher.

```
You are reviewing AND PATCHING <artifact_type> for <project>.
The plan/document is at `<path>` (read it from disk).

Relevant code locations:
- `<file_1>` - <purpose>
```

```
- `<file_2>` - <purpose>
- `<file_3>` - <purpose>

Focus on:
1. <architectural criterion>
2. <technical gaps / hidden bugs>
3. <edge cases / tests>
4. <user-facing consequences>

Out of scope:
- <non-goal 1>
- <non-goal 2>

You are not alone in the codebase. Do not revert edits made by others.
Your write scope is only <allowed path>.

Return findings first, then patches, then verification.
```

Why does this frame work?

- "reviewing" makes AI read and evaluate first.
- "PATCHING" says the final result must include concrete changes.
- Relevant files reduce blind grepping.
- Numbered focus keeps priorities in order.
- Out of scope prevents scope creep.
- "You are not alone" reminds AI not to revert other people's work.
- Findings first helps you catch errors before the patch goes too far.

The more files a task touches, the more useful Reviewer/Patcher becomes.

When to use it:

- Review a plan/RFC before implementation.
- Audit a security module.
- Refactor a large file.
- Fix a bug whose root cause spans multiple files.
- Review code before merge.

When not to use it:

- Change one line of copy.
- Fix a typo.
- Brainstorm ideas.
- Very short debug chat.

A strong template does not mean use it every time. It is strong when used at the right time.

Winning recipes

A pattern catalog is not scripture. It is a list to try first.

From my data, a few recipes are worth prioritizing:

1. Out of scope section

Simple, short, effective.

```
Out of scope:
- Do not change billing logic.
```

```
- Do not redesign UI.
- Do not add dependencies.
```

If you do not know what else to add to a prompt, add `Out of scope`.

2. Role frame

```
You are a strict code reviewer focused on regressions.
```

Role helps AI choose a lens. It does not need to be long.

3. Numbered focus

```
Focus on:
1. Correctness.
2. Edge cases.
3. Verification.
```

Numbering helps AI miss less, and helps you refer back more easily.

4. Read from disk

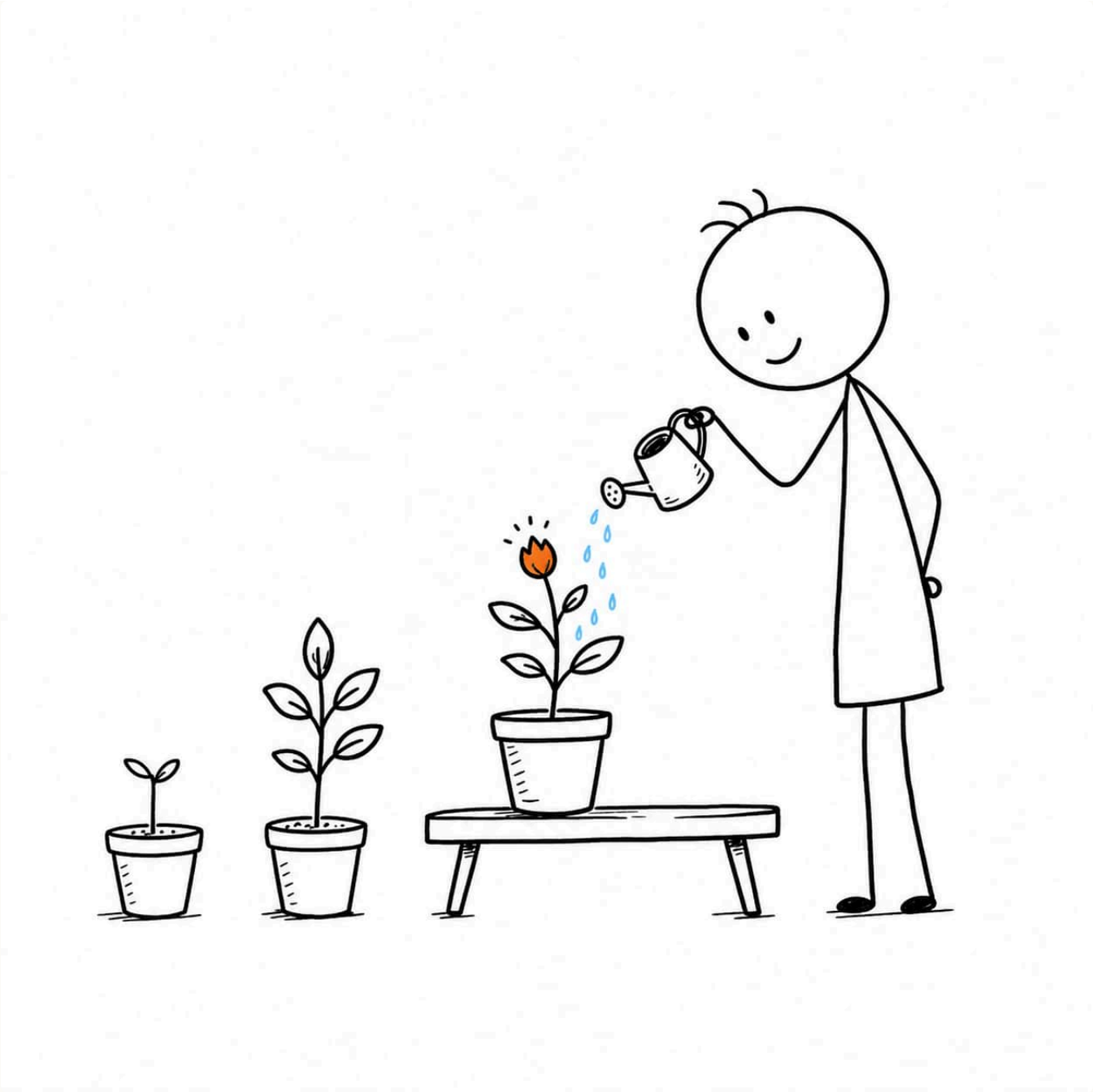
```
Read `docs/billing-plan.md` from disk before patching.
```

Do not let AI rely on a pasted copy if the real file may have changed.

5. Verify with same reproduction

```
Do not claim fixed until you rerun the same reproduction.
```

This prevents the habit of fixing something and then changing the test.



A recipe only has value when you use it, measure again, and adjust it to your project.

Anti-recipes to avoid

A few patterns cost me a lot of time:

1. "Implement X in Y" alone

```
Implement dark mode in settings.
```

It sounds structured, but it lacks context, constraints, and verification. It easily creates a false feeling of safety.

2. Vague short

```
fix
```

```
next
```

If the session is very clear, maybe this is fine. For a new task, do not.

3. Image-only

A screenshot does not speak for you about what the problem is.

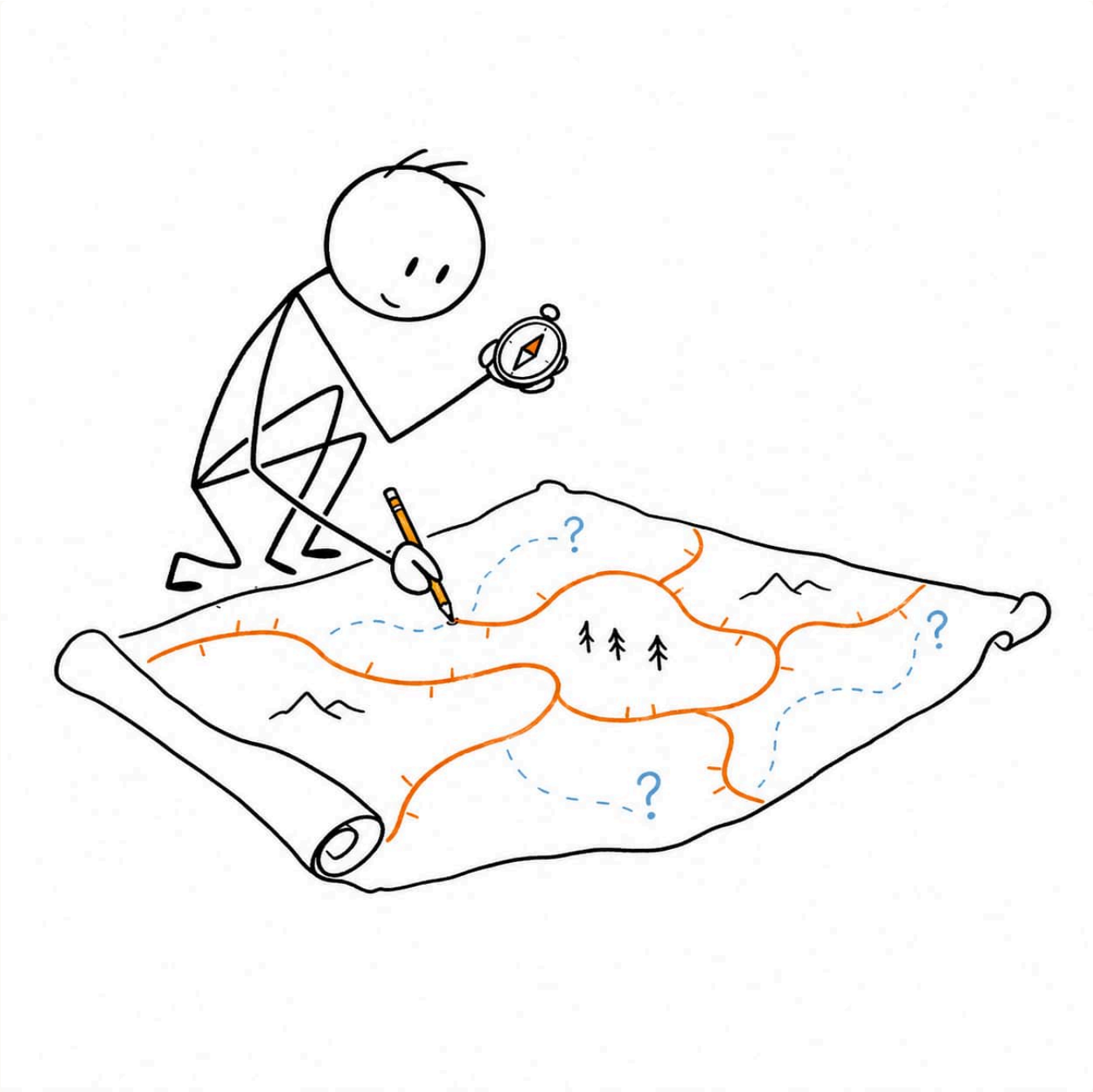
4. Scope creep mid-session

```
also write me a video script
```

inside a session that is debugging auth. This is one of the fastest ways to make context dirty.

5. Trusting "done"

AI saying done does not mean the app runs. Only verification means that.



Data gives direction. You still have to verify inside your real project.

The bright side

Vibe coding has a very real bright side.

It helps beginners touch products faster. In the past, to build an app with auth, database, UI, and deploy, you had to pass through many gates. Now AI can pull you

through the boring syntax and boilerplate so you see the product flow earlier.

It helps experienced people try ideas faster. A prototype no longer has to take a whole week. AI can handle the mechanical part of a refactor. A bug can be analyzed through more hypotheses.

It also helps me learn faster. When AI explains a patch, I can ask follow-up questions: why choose this approach, what is the trade-off, what edge cases exist, what test should be added. Used well, AI does not only write code. It becomes a surface for thinking.

The dark side

But vibe coding also has a dark side.

It can make you lazy about reading. Because AI answers quickly, you can skip the diff.

It can make you lazy about foundations. Because the app runs, you think you understand it.

It can create technical debt very quickly. A slow coder creates debt slowly. AI can create debt at 200 lines per minute.

It can make you addicted to the feeling of fake progress. Every prompt produces code, the terminal moves, files change, and everything feels productive. But if the app is not more stable at the end of the day, that is not progress.

It can also make you lose your own voice. If every decision is copied from AI, your product will feel generic. AI is good at synthesizing what is common. You still have to keep taste, context, and goals.

Balance

AI is a powerful companion, but the steering wheel is still yours.

I do not want you to read this book and become afraid of AI. I still use AI every day. But I also do not want you to worship it.

My current balance is:

- Use AI to draft, but I review.
- Use AI to debug, but evidence must come from the real system.
- Use AI to refactor, but scope must be clear.
- Use AI to learn, but do not skip foundations.
- Use AI to ship faster, but do not skip verification.

You do not need to become a professional prompt engineer. You only need to become someone who delegates clearly, verifies seriously, and knows when to stop.

If you are just starting, do three things:

1. Save your best prompt from this week.
2. Add `Out of scope` to your next code-change prompt.
3. Do not accept an important patch if it has not been verified.

That small. But if you do it long enough, it changes how you work.

Vibe coding is not a shortcut around learning. It is a new way to learn while building. If you keep that in mind, AI does not take away your craft. It gives your craft more leverage.

Final question: will your next prompt be a throw into the slot machine, or a clear brief for a very fast collaborator?