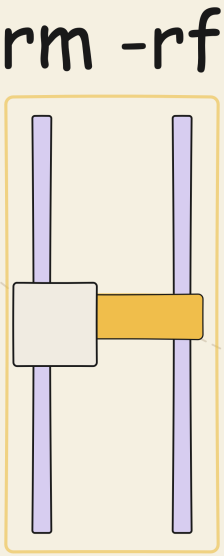
timurista-ai

THE HARNESS SERIES



The Claude Code Safety Kit

Put a human in front of the write, in about ten minutes

Tim Urista

FREE EDITION · NOVEMBER 2026

THE CLAUDE CODE SAFETY KIT



The Claude Code Safety Kit

Put a human in front of the write, in about ten minutes

Tim Urista

TIMURISTA.AI

The Claude Code Safety Kit
by Tim Urista

First edition, revised · November 2026
`version 0.2`

Copyright © 2026 Tim Urista. All rights reserved.

This book is given away free. You may share the file as it is. You may use every configuration and script in it in your own work, commercial or not, without attribution and without warranty of any kind.

The figures are drawn by the same code that draws the author's explainer videos. Product names belong to their owners; this book is not affiliated with or endorsed by any vendor it names.

Corrections and the current version: https://timurista.ai

Contents

Preface 9

1 The loop with nothing in front of it 11

- Reversible and irreversible are not a spectrum 12
- How fast the bad case is 12
- What a gate has to be 14

2 The gate 15

- What the hook sees 15
- Installing it 16
- The script 17
- Why a deny list, when a deny list is always incomplete 19
- Choosing an allow list instead 20
- What belongs behind the gate 21

3 The file the agent actually reads 23

- What reaches the model 23
- Write rules, not descriptions 24
- A skeleton 25
- The section that pays for itself 26
- Keeping it honest 27

4 The ten-minute audit 29

- Decide your posture first 29
- The checklist 30
- The last check is the real one 31
- Where to go from here 31

What comes next 33

FRONT MATTER

Preface

This is a short book about one control. Not a methodology, not a philosophy of AI-assisted development: one control, which you can install this afternoon, and which changes what a coding agent is allowed to do to your repository without asking.

I run a software product alone, with an agent doing a large share of the typing. That arrangement only works because the agent cannot do anything irreversible to the repository without my reading it first. The gate that makes this true is about ten lines of configuration and a short script, and it is the highest-value thing in my setup by a wide margin. Everything else, the project file, the review workflow, the skills, makes the agent better. The gate is what makes it safe to be wrong about all of that.

Who this is for

You have used Claude Code, or Codex, or Cursor's agent mode, for more than a weekend. You have had the experience of reading a diff you did not expect, or watching a command run that you would not have run yourself. You are not looking for an argument about whether agents are good; you are looking for the part of the setup that keeps the bad afternoon from happening.

You do not need to be an expert in the tool. Every file in this book is short enough to read in full, and nothing here depends on a particular editor or operating system.

What is in it

Chapter 1 is the problem, stated precisely enough to be worth solving: the agent's loop has a human at the start and a human at the end, and nothing in the middle. Chapter 2 is the gate itself — the hook, the script, and what to put behind it. Chapter 3 is the project file, which is the other half of the control and the half most people write badly. Chapter 4 is a ten-minute audit you can run on a repository you already have.

The last page says what the paid companion adds. Nothing in this one is a teaser: the gate described here is the gate I run.

A word about versions

Agent tooling changes faster than books do. The shape of the control in this book — intercept the tool call, decide, allow or refuse — has been stable across every version I have used, and I expect it to stay stable, because it is the only place the decision can live. The exact key names in a settings file are a different matter. Where a detail is likely to drift, I say so and point at the vendor's own documentation, which is the only thing that can be current.

> **NOTE** Every script in this book is short enough to read in full before you run it, and you should. Where a detail depends on a vendor's current release rather than on the shape of the idea, the text says so and points at their documentation.

What was tested for this edition: the gate script, run directly with bash 3.2 and jq 1.7.1 on macOS, against every case the book describes; and the hook wired into Claude Code 2.1.312, in a throwaway repository, for the refusal, the renamed-script check and a timeout. Linux and Windows were not tested.

CHAPTER 1

The loop with nothing in front of it

An agent's working loop has exactly two human checkpoints, and they are both in the wrong place.

Draw the loop an agent actually runs and you get four steps. You ask for something. It makes a plan. It writes. You find out what it did.

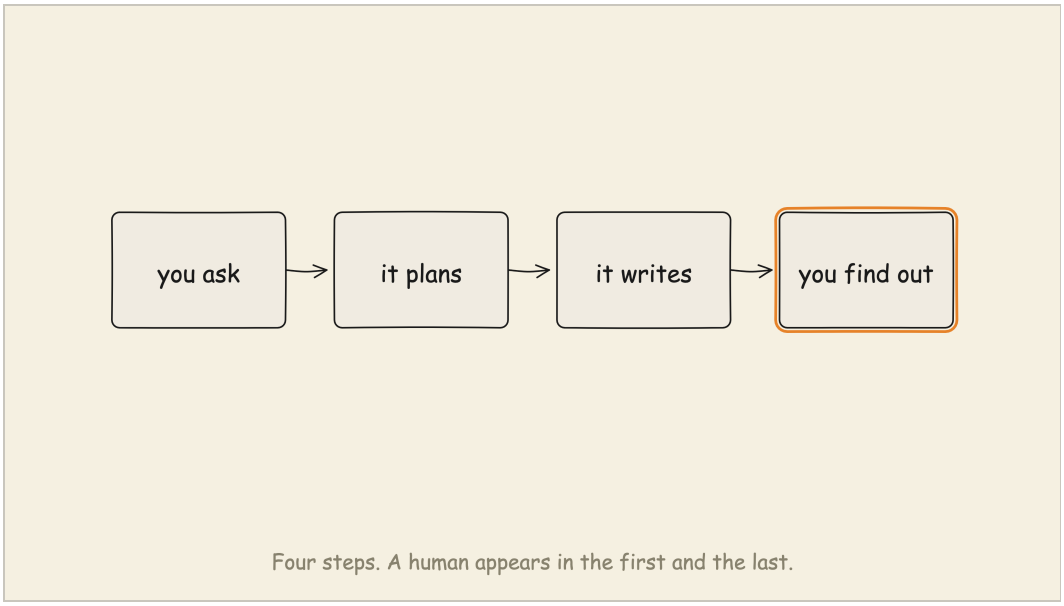


Figure 1-1. The loop as shipped. A human appears in the first step and the last one, and the two steps in between – the ones that touch your files – have nobody in them.

That picture is not a criticism of the tooling. It is the design, and for most of what an agent does it is the right design: if you had to approve every file read, every search, and every line of reasoning, you would be typing the code yourself and more slowly. The loop is fast because most of it does not need you.

The trouble is that the two unsupervised steps are not the same kind of step. Planning is cheap and reversible; nothing on disk changes, and a bad plan costs you

the tokens it took to write. Writing is neither. A write is the moment the agent stops being a conversation and starts being a process with your credentials.

Reversible and irreversible are not a spectrum

It is tempting to think of agent actions as more or less risky, and to imagine a dial you could turn. In practice there are two piles and almost nothing between them. One pile is everything you can undo with a keystroke: a file edit in a tracked repository, a branch, a commit, a local build. The other pile is everything that reaches past your working copy — a force push, a migration, a deploy, a delete against anything that is not a working copy, a request to a service that charges money or sends mail.

The first pile does not need a gate. Version control is already the gate, and it is a better one than any prompt, because it does not depend on you reading carefully at four in the afternoon. The second pile needs a gate, because nothing else is going to catch it.

NOTE This is why "approve every edit" is a worse policy than it sounds. It puts a prompt in front of the pile that is already protected, which trains you to click through prompts, which is exactly the habit you cannot afford when the dangerous one finally appears.

How fast the bad case is

The thing that makes the unsupervised middle dangerous is not that agents are reckless. It is that the whole chain runs at machine speed, and a human watching the output is not a control at that speed — they are an audience.

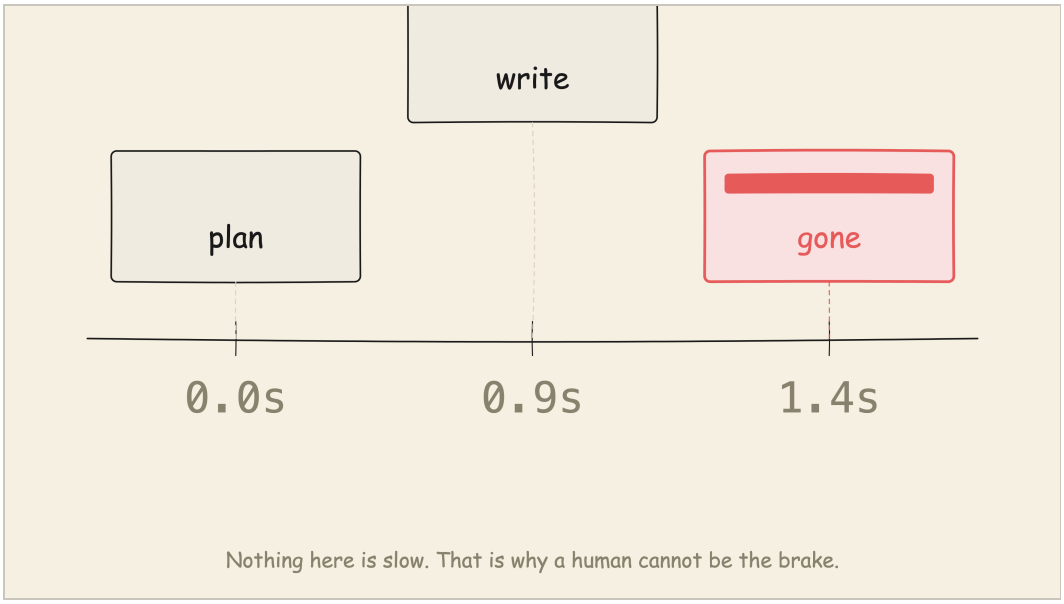


Figure 1-2. A chain with no gate in it. The elapsed times are the point: there is no moment in here long enough for a person to read what is happening and intervene.

The cheap version of this story runs in a tracked repository: a cleanup instruction, a plausible reading of it, a recursive delete that takes a directory nobody thought to mention, and a report that it is done. Recovery costs twenty minutes and nothing of value, because everything irreplaceable was under version control somewhere else.

The expensive version reaches a production database, a cloud account, or a directory that was never in a repository at all. Those stories circulate, usually told by the person they happened to, and they are worth reading sceptically, but the mechanism they describe is always this one. Something irreversible happened, and there was no step between deciding and doing.

READING INCIDENT STORIES HONESTLY

When one of these goes around, the headline usually names the model. That is the least interesting fact in it. The useful questions are: what credential did the process hold, what safeguard was supposed to stop it, and why did that safeguard not apply?

Almost every time, the answer is that the safeguard was a human reading output in real time, and there was no real time to read it in. Name the failed control, not the model.

What a gate has to be

So the control we want is narrow. It should sit in front of the irreversible pile only. It should stop the process, not warn about it — a message that arrives after the command has run is a log entry, not a gate. It should fail closed, so that a broken gate blocks rather than waves things through. And it should be small enough that you can read it in one sitting and believe it.

All four of those are achievable with a feature that is already in the tool. That is the next chapter.

CHAPTER 2

The gate

A pre-tool-use hook, a short script that fails closed, and the question of what belongs behind it.

Claude Code can run a program of yours immediately before it runs a tool, and let that program decide whether the tool call proceeds. That feature is called a pre-tool-use hook, and it is the whole mechanism in this book. Everything else is a matter of what you put behind it.

What the hook sees

When the agent is about to use a tool, the hook runs with a JSON object on standard input describing the call: which tool, and what arguments. Your program inspects it and exits, and the exit code is the decision.

The exit codes matter more than they look

Three outcomes, and only one of them stops anything.

- **Exit 2** blocks the call. Whatever the program wrote to standard error is handed back to the agent as the reason.
- **Exit 0** does not approve the call. It hands it on to the normal permission flow, which may still ask you.
- **Any other code**, exit 1 included, is a non-blocking error: the hook has not blocked anything. Normal permissions still apply, so the call may or may not run — which is exactly the uncertainty a gate is supposed to remove.

WARNING A gate that exits 1 is not a gate. It logs its objection, lets the command run, and tells you nothing. Use 2.

TIP A refusal is a chance to redirect. "Blocked: this deletes an untracked directory. Move it to a branch and try again" produces a better next attempt than "Blocked."

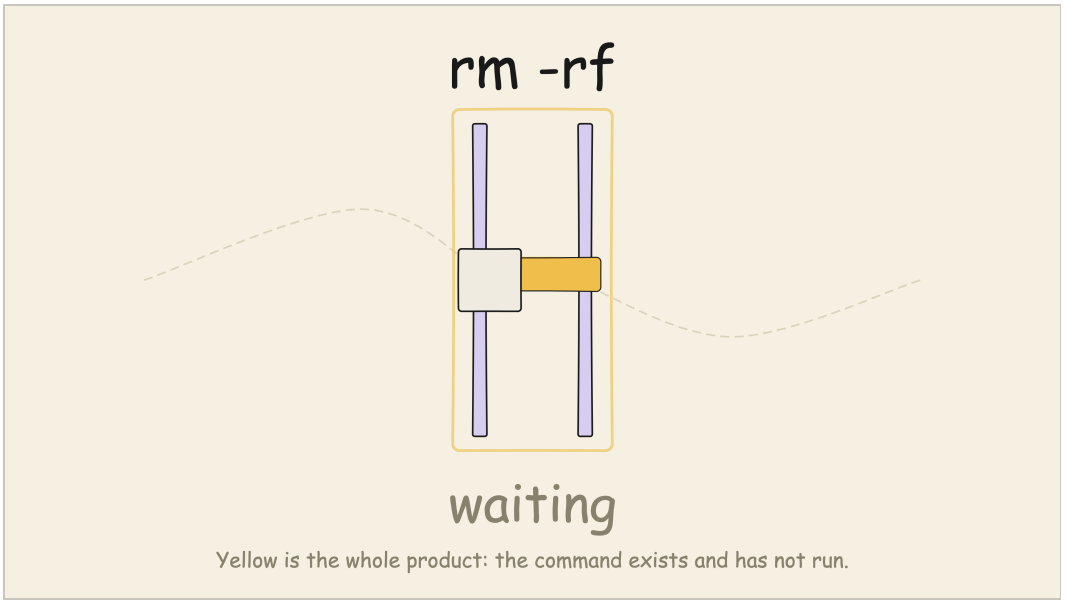


Figure 2-1. The gate holding a command. The command exists, has been fully specified by the agent, and has not run. This state is the entire product.

Installing it

Hooks are configured per project, in the project's settings file. The matcher selects which tools the hook applies to; everything else is left alone, which matters, because a gate in front of file reads would be a gate you learn to ignore.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "$CLAUDE_PROJECT_DIR/.claude/gate.sh",
            "timeout": 10,
            "onFailure": "block"
          }
        ]
      }
    ]
  }
}
```

Example 2-1. The hook, in .claude/settings.json. This wires up a script; the script is the gate.

WARNING `"onFailure": "block"` is what makes a broken gate refuse. Without it, a script that is missing, not executable, crashes or runs past its `timeout` is a non-blocking error, and the command goes on to the normal permission flow. It needs Claude Code 2.1.295 or later. On an older version, check the vendor's hooks reference for what your version offers before you rely on the gate failing closed.

NOTE Key names here have changed between releases and may change again. Check your installed version against the vendor's hooks reference before you debug the script; the script is usually not the problem.

WARNING `"matcher": "Bash"` covers shell calls and nothing else. File writes and edits made through other tools do not pass through this hook at all. Widening the matcher widens the coverage and the friction together; decide which tools you actually want gated.

The script

Here is a gate small enough to audit. Every path through it either exits 2 or exits 0, and every way it can go wrong exits 2. That last clause is the one worth testing, and the one an earlier draft of this chapter got wrong.

```
#!/usr/bin/env bash
# No set -e: a failure here must refuse, not exit with some other
# status that the agent is free to treat as a non-blocking error.
set -uo pipefail
trap 'echo "Gate: internal error. Refusing." >&2; exit 2' ERR

deny() { echo "Gate: $1" >&2; exit 2; }

# A missing dependency is not a reason to wave the command through.
command -v jq >/dev/null 2>&1 || deny "jq is not installed."

payload="$(cat)" || deny "could not read the payload."
cmd="$(printf '%s' "$payload" | jq -er '.tool_input.command // empty')"   || deny "could r
[ -n "$cmd" ] || deny "no command in the payload."

# One call, one command. Anything that chains, substitutes or
# redirects can hide a second command from every check below.
# A newline and a single & separate commands too.
case "$cmd" in
  *";"*|*"&"*|*"|"*|*'`'*|*'$('*|*">"*|*"<"*|*$'\n'*)
    deny "shell composition. Run it yourself." ;;
esac

# Match flags, not one spelling: rm -rf, -fr, -Rf and -r -f are
# the same delete; -f and +branch are the same force push.
if [[ $cmd =~ (^|[[:space:]/])rm([[:space:]].*)?[[:space:]](-[[:alpha:]]*[rR]|--recursive)
  deny "recursive delete."
fi
if [[ $cmd =~ (^|[[:space:]])git[[:space:]].*push.*[[:space:]](--force|-[[:alpha:]]*f|[+])
  deny "force push. Push a branch instead."
fi

lower="$(printf '%s' "$cmd" | tr '[:upper:]' '[:lower:]')"
case "$lower" in
  *"git reset --hard"*)        deny "hard reset. Commit or stash first." ;;
  *"git clean"*)               deny "git clean removes untracked files." ;;
  *"drop table"*|*"truncate"*) deny "destructive SQL. Run it by hand." ;;
  *" -delete"*)                deny "find -delete removes files in bulk." ;;
  *"mkfs"*)                    deny "writes to a device." ;;
esac

exit 0
```

Example 2-2. `.claude/gate.sh`. Every failure path refuses, including its own.

Four things in that script are doing more work than they look. `set -e` is deliberately absent: with it, a failing command exits with that command's status, and a status that is not 2 is a non-blocking error, so a broken gate would let the call through. The `jq` check is there for the same reason. The composition check comes first, because every pattern after it inspects one string: without it, `npm test; rm -rf .` is a single command that no later rule matches, and a newline or a single `&` separates commands just as well as a semicolon. And the delete and force-push rules match flags rather than one spelling, because `rm -Rf` and `git push -f` are how people actually type them.

Make it executable, restart the session so the settings are re-read, and try it. Then break it on purpose: rename the script and confirm you are refused rather than allowed. With `onFailure` set, the refusal comes from Claude Code itself, because the hook could not run. Without it, the command goes ahead to the normal permission flow, and that is the gap the setting closes.

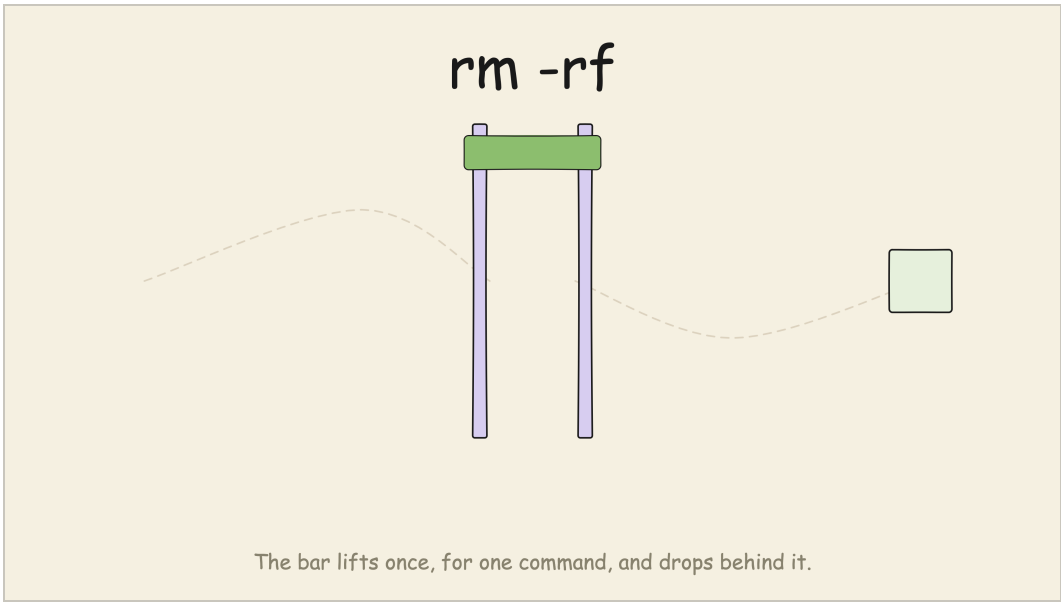


Figure 2-2. The same gate letting a call through. Exit 0 does not approve anything: it hands the call on to the normal permission flow, which may still ask you.

Why a deny list, when a deny list is always incomplete

A reviewer will point out, correctly, that the list above does not catch everything. It does not catch a destructive command inside a script the agent wrote a minute ago, or

one run from inside `node` or `python`. It does not catch `bash -c` wrapping a command it would otherwise refuse. A sufficiently inventive sequence gets past it.

That objection is right about the mechanism and wrong about the threat. The failure mode this gate exists for is not an adversary trying to get past you. It is a plausible misreading of an ordinary instruction, executed confidently and quickly. Those misreadings concentrate in a small number of shapes, because they come from the same place: the agent reaching for the standard way to do a thing. The standard way to clean a directory is `rm -rf`. The standard way to make a branch match is `--force`. A list of twenty lines covers the overwhelming majority of the cases that actually occur.

WARNING A deny list is not a security boundary, and this chapter is not telling you it is one. If your threat model includes a deliberate attempt to get past the gate — a prompt injected through a file the agent reads, for instance — you need an allow list and a sandbox, not this. The allow list is next; for the sandbox, see the vendor's sandboxing documentation.

Choosing an allow list instead

If you want the stronger version, invert it: refuse everything except a named set of commands. This is genuinely safer and genuinely more annoying, and whether it is worth it depends on how varied your work is. On a repository where the agent mostly runs a test command, a formatter and a build, an allow list is four lines and no friction at all.

```
case "$cmd" in
  "npm test"|"npm run build"|"git status"|"git diff") exit 0 ;;
  *) deny "not on the allow list: $cmd" ;;
esac
```

Example 2-3. An allow list is the same script with the case inverted. Match whole commands.

WARNING Match the whole command, not a prefix. `"npm test"*` looks harmless and also accepts `npm test; rm -rf .`, because the trailing wildcard matches the rest of the line. If you need arguments, enumerate the forms you actually run rather than opening the end of the pattern.

A reasonable middle: an allow list for the commands you run constantly, falling through to the deny list for everything else. That gives you silence on the common path and a stop on the dangerous one.

What belongs behind the gate

The list is short, and it should stay short. If a rule fires weekly you will start working around it, and a gate you work around is worse than no gate, because you now believe you have one.

- Anything recursive and deleting, against any path.
- Anything that rewrites shared history: force push, hard reset, branch deletion on a remote.
- Anything that touches a database other than a local, disposable one.
- Anything that deploys, publishes, or sends.
- Anything that spends money.

Notice what is not on it: writing files, editing files, creating branches, committing, installing dependencies, running tests. All of those are either reversible or caught by review, and putting them behind a prompt is how you teach yourself to stop reading prompts.

CHAPTER 3

The file the agent actually reads

A project file is the other half of the control, and most of them are written as if nobody will read them.

The gate stops the worst thing. The project file — `CLAUDE.md` at the root of the repository — prevents a much larger number of smaller ones, by telling the agent what this codebase is and how work is done in it.

It is also the file people write worst, for an understandable reason: it looks like documentation, so it gets written like documentation. Documentation describes. A project file instructs. The test for every line is whether it would change what the agent does, and most lines in most of these files would not.

What reaches the model

The file competes for space with everything else in the call. It is worth having a picture of that competition before deciding what to put in it.

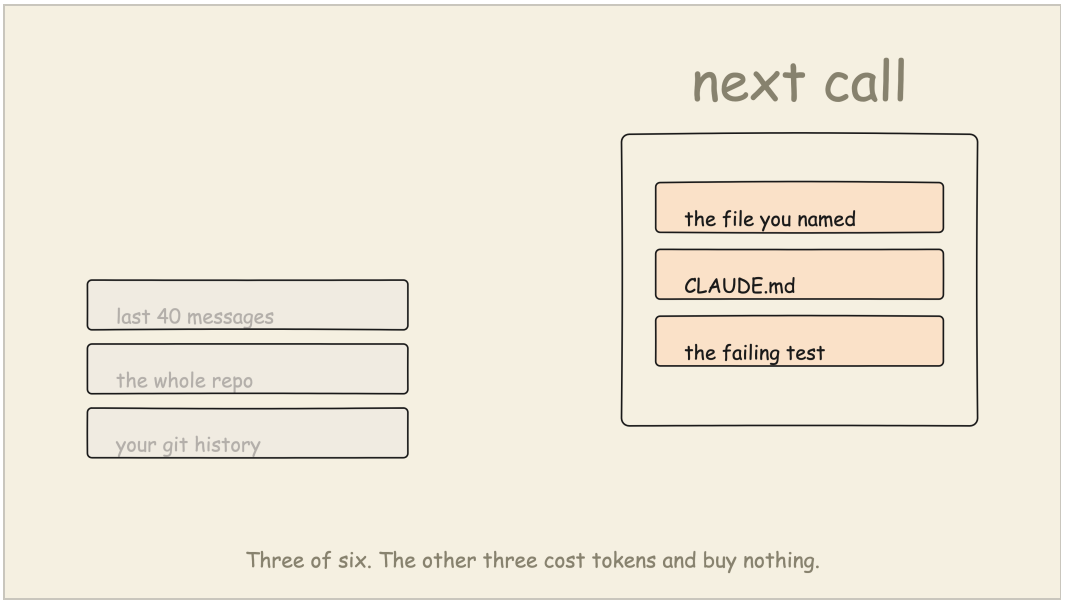


Figure 3-1. What goes into a call. Three of these earn their place; the others are habit.

Everything in that box costs tokens on every single request, for the whole session. A project file that runs to four pages is a tax you pay hundreds of times a day, and the parts of it that do not change behaviour are pure loss. One page that is all instruction beats four pages that are mostly description.

Write rules, not descriptions

Here is the difference, in the two forms I see most often.

```
## Testing
This project uses Vitest for unit tests. Tests live in the
test/ directory and are organised by module. We care a lot
about test coverage and try to keep it high.
```

Example 3-1. Description. True, and changes nothing about what the agent does.

```
## Testing
- Run `npm test` before you say a change works. Paste the output.
- A new branch of logic needs a test in the same commit.
- Never change a test to make a failing change pass. Say the
  test disagrees and stop.
```

Example 3-2. Instruction. Same subject, written so that a reader can act on it.

The second one is shorter and it does something. The third line in particular is worth its space on its own: changing a test to accommodate a broken change is the most common way an agent reports success on work that does not work, and it stops happening the day you forbid it in writing.

> TIP A good heuristic: if you removed the line, would any behaviour change? If not, delete it. The agent can read the repository; it cannot read your mind about what is forbidden.

A skeleton

Four headings, each one earning its place. Begin here and let the repository add to it.

```
# [Project]

[One sentence: what this is and who uses it.]

## Commands
- Test: `[command]`     - Build: `[command]`
- Lint: `[command]`     - Run:   `[command]`

## Rules
- Run the tests before claiming a change works. Paste output.
- Never weaken a test to make a change pass. Stop and say so.
- Ask before adding a dependency.
- Match the file you are editing: its naming, its comments,
  its error handling.

## Do not touch
- `[generated paths]` - regenerate with `[command]`.
- `[vendored paths]`  - upstream code.
- Anything under `[secrets path]`.

## When you are unsure
Say what you are unsure about and stop. A question costs a
minute. A confident wrong answer costs an afternoon.
```

Example 3-3. `CLAUDE.md`, as a starting point. Replace every bracket.

The section that pays for itself

Of those four, "Do not touch" is the one that will save you first. Generated files are the classic case: an agent sees a file that looks wrong, fixes it, and your next build overwrites the fix — or worse, does not, and you now have a generated file that no longer matches its source.

The same section is where vendored code and anything secret-adjacent belongs. An agent will happily read a `.env` file to understand your configuration, and once it has, that content is in a conversation that may be logged.

WARNING A line in a project file is a strong instruction, not an enforcement mechanism. It belongs in the file, and if the path genuinely must never be touched, it also belongs behind the gate from Chapter 2.

Keeping it honest

The failure mode of a project file is drift. The commands change, the file does not, and the agent spends a week running a test command that no longer exists. Two habits prevent it: when a rule stops the agent doing something stupid, write that rule down the same day; when you catch yourself explaining the same thing twice in a conversation, that explanation is a missing line in the file.

CHAPTER 4

The ten-minute audit

Run this against a repository you already have. It is a checklist, not a chapter.

Everything so far is a page of configuration and twenty lines of shell. This chapter is how you find out whether you have it, on a repository where an agent is already working.

Decide your posture first

Before any of the checks, decide which of these you are running, because the checklist means different things in each.

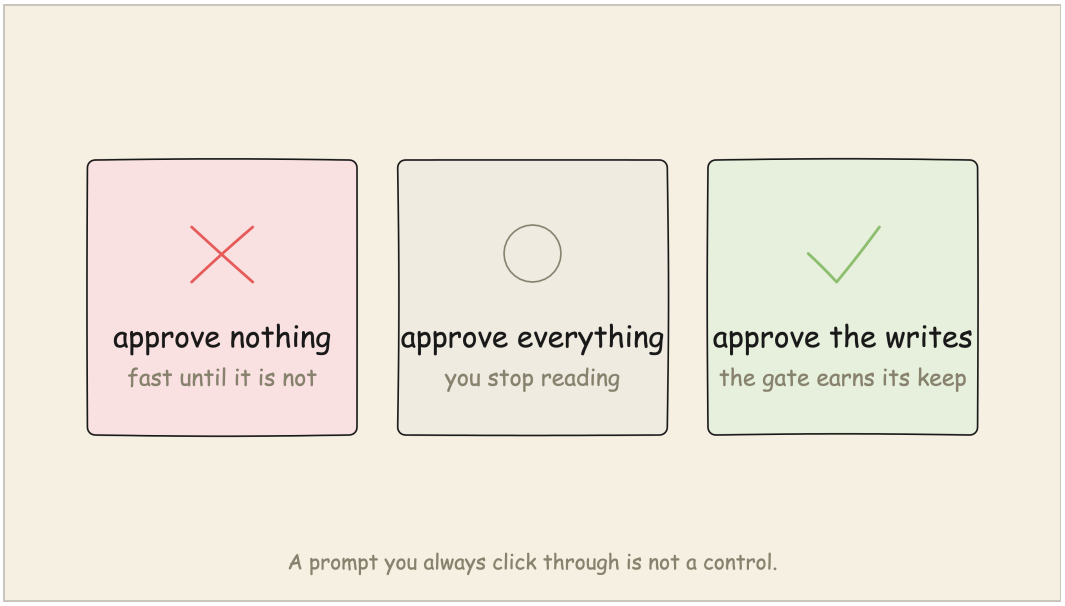


Figure 4-1. Three postures. The middle one is the one people drift into without choosing it.

The middle card is worth naming. Nobody decides to approve everything; they decide to approve a lot, and then the prompts arrive faster than they can be read and the

clicking becomes automatic. If you have a prompt you always say yes to, you do not have a control there — you have a delay. Remove it and put a real gate somewhere narrower.

The checklist

Ten minutes, in this order. Anything you cannot answer in under a minute is a finding.

- ☐ **There is a gate.** A pre-tool-use hook is configured, you can name the file it runs, and you have read that file end to end in the last month.
- ☐ **It fails closed.** Check the hook entry has `"onFailure": "block"`, then break it on purpose: rename the script and confirm the agent is refused rather than allowed. Without that setting a broken gate waves things through to the normal prompts, and a gate that does that is not a gate.
- ☐ **It refuses with exit 2.** Grep the script. Exit 1 is a non-blocking error: the command runs anyway, and nothing tells you.
- ☐ **It actually fires.** Ask the agent to run a recursive delete in a scratch directory. You should get your own refusal message back.
- ☐ **It refuses when it cannot inspect.** Take `jq` off the path and run it again. If a safe command is still allowed, the gate is guessing.
- ☐ **It refuses shell composition.** Try `npm test; rm -rf .` as one command, and the same two commands on two lines. Every pattern check in the script reads one string, so anything that chains or substitutes has to be refused before those checks run.
- ☐ **Any allow list matches whole commands.** A trailing wildcard on a prefix accepts whatever follows it.
- ☐ **Nothing on it fires weekly.** A rule you hit constantly is a rule you will route around. Either the rule is wrong or your workflow is.
- ☐ **There is a project file.** At the repository root, under version control, and under about a page.
- ☐ **Every line of it instructs.** Read it and delete anything that describes rather than directs.
- ☐ **The commands in it are current.** Run each one. This check finds something roughly half the time.

- [ ] **"Do not touch" names generated and vendored paths.** And anything near a secret.
- [ ] **Credentials are scoped.** The shell the agent runs in cannot reach production. If it can, no gate on your laptop is the real control.
- [ ] **You can read the diff.** Open the last change the agent made and explain it out loud. If you cannot, that is the finding, and it is the most important one here.

The last check is the real one

The first thirteen are mechanical and you will pass them within an hour of deciding to. The last one is the one that decides whether any of this was worth doing.

The industry has a name for the speed-up and not yet a good name for the debt. Both arrive together, and only one of them is visible on the day.

THE ONLY RULE I WOULD KEEP IF I COULD KEEP ONE

Before you accept a change, say what it does in one sentence, out loud, without looking at the explanation the agent gave you.

If the sentence does not come, you have found the work: read it properly, or ask for a smaller change. This costs about thirty seconds per change and it is the only habit in this book that compounds.

Where to go from here

If you ran the audit and everything passed, you are in better shape than most repositories I have seen, and the next thing worth your time is not more configuration — it is a review step, so that one agent's work is read by something other than the agent that wrote it.

If things failed, fix them in the order they are listed. The gate first, because it is the only one that prevents an afternoon you cannot get back.

What comes next

This book covers one control. The paid companion I am preparing covers the routine around it, as files you install rather than a document you copy from. It is not finished, and this book stays free either way.

- An installer that shows its plan first, backs up your settings, merges without touching anything else, and uninstalls exactly what it added.
- Plan, checks and handoff templates, so a change is agreed before it is written, checked after every step, and resumable after a break.
- A project file with every line explained, including what it costs.
- A gate that asks before anything that chains, wraps or leaves your machine, as well as refusing the short list in Chapter 2.
- A small practice repository and a short walkthrough.

When it is ready it will be at **timurista.ai**, along with the videos these figures come from. It does not include a sandbox or a container; for those, start with the vendor's documentation.

Corrections

If something here is wrong, or has gone stale against a newer release, I want to know and the next version will say so. The errata list lives on the same page as the book: timurista.ai/books/claude-code-safety-kit.