Vol. 1 · Deep Dive edition

The Claude Code *Field Manual*

The step-by-step edition: walkthroughs, worked examples and exercises for working with an AI teammate on real code.

MIAN SHAFIQ · SHAFIQ.PK

free to share!



25 years of code, one new teammate

Hi, I'm Shafiq.

I've shipped software since the days of FTP deploys. Coding agents changed my day faster than anything before them.

This isn't a list of magic prompts. It's how to set up your repo so Claude does good, safe, checkable work.

This is the deep dive: steps, examples and an exercise for every chapter. Short on time? Read the 30-slide edition first.

How to read this edition

Same ten chapters as the short version, with the "how" filled in. Four kinds of slide:

STEP BY STEP

Numbered steps

Do them in your own repo, in order.

WORKED EXAMPLE

A real session

What you type, and what Claude does.

WRONG / RIGHT

Side by side

The mistake everyone makes, and the fix.

TRY IT

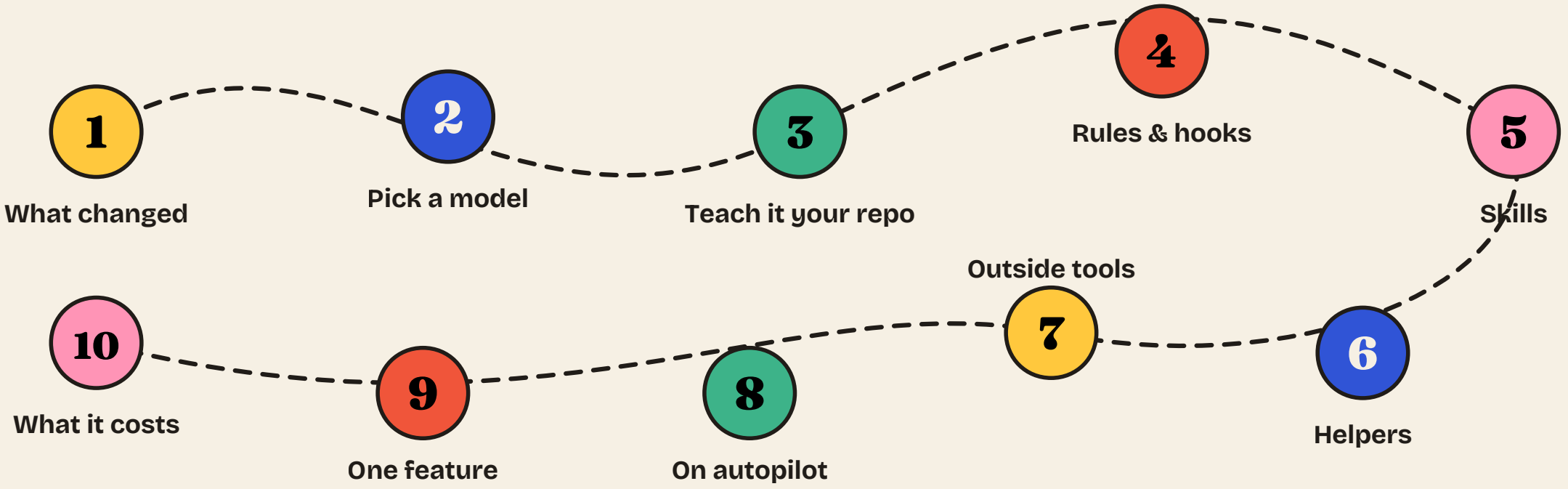
Your homework

One exercise per chapter, with a "done when".

tip: one chapter a day beats the whole deck in one night

The route

Ten stops. The first four are the foundation.



Set up in five minutes

You need a Claude Pro, Max, Team or Enterprise plan, or a Console (API) account. The free plan doesn't include Claude Code.

no terminal? there's a desktop app too

1. **Install it** (macOS, Linux, WSL)

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell: `irm https://claude.ai/install.ps1 | iex` ·
Homebrew: `brew install --cask claude-code`

2. **Open a new terminal and check it**

```
claude --version
```

3. **Go to a real project and start**

```
cd ~/code/my-project && claude
```

The first run opens your browser to log in.

4. **Something odd?** `claude doctor` checks your install and settings.

1

a true story...

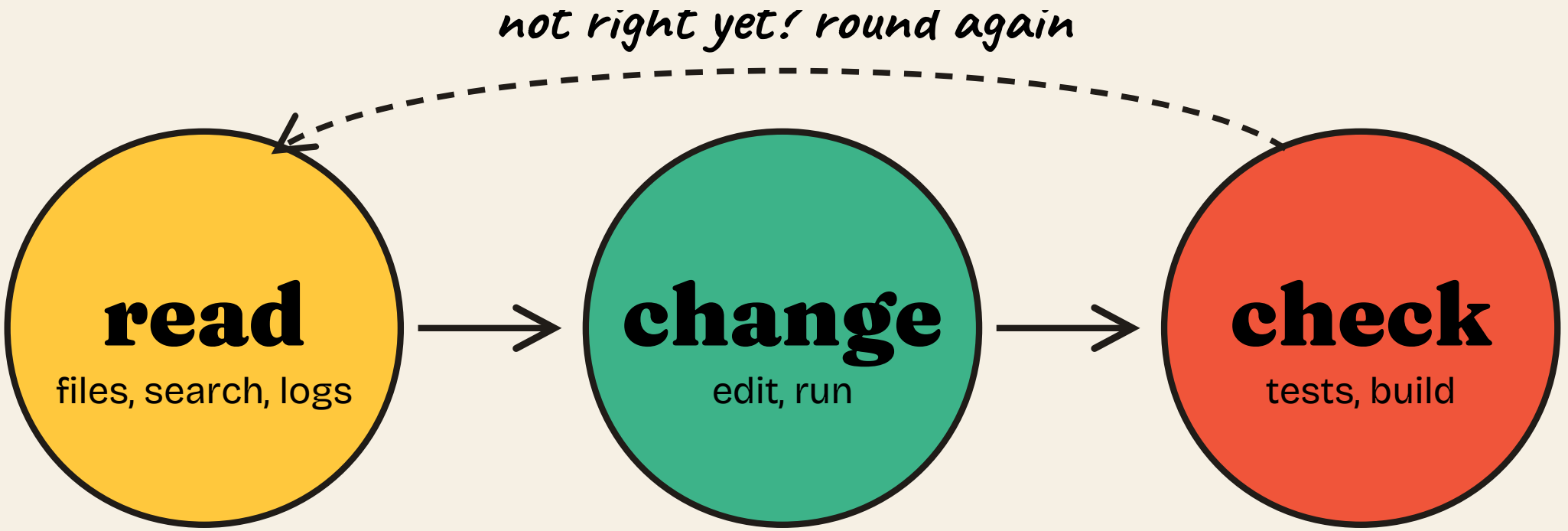
I gave it a whole ticket and a failing test, then went to make tea.

When I came back, the test passed, the diff was small, and the explanation was better than mine.

The unit of work changed from a line of code to a task.

What actually changed

Every session is a *loop*



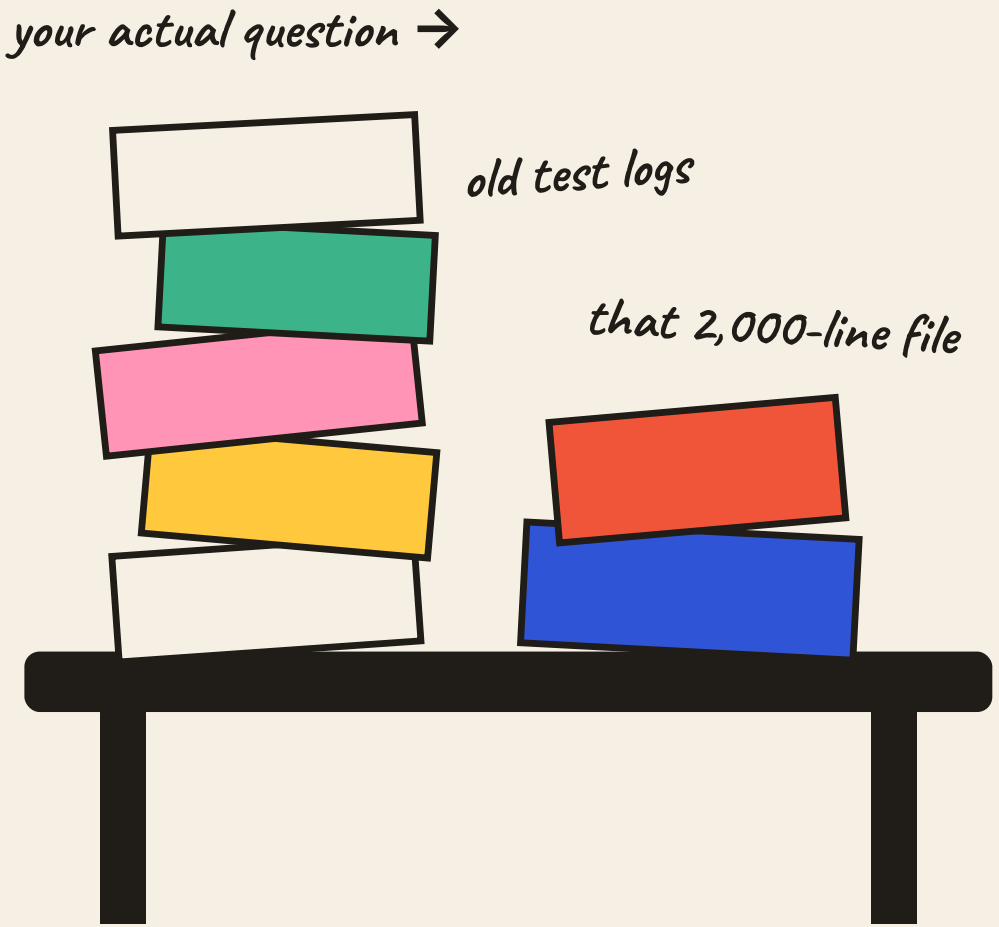
Claude stops when it thinks it's done. **Give it a way to check,** or "looks done" is all it has.

Think of context as a *desk*

Your messages, every file it reads, every bit of test output: it all piles up on one desk.

A messy desk means worse work. Clear it between tasks.

`/clear` fresh start `/context` see the pile



Anatomy of a good request

Three parts: the outcome, where to look, and how to check.

× VAGUE

```
fix the logout bug
```

Claude has to guess what "the bug" is, where it lives, and when it's fixed. Then it stops when it thinks it's done.

✓ CLEAR

```
Users get logged out after ~5 minutes instead of 24 hours. Session code is in src/auth/. Write a failing test that shows it, fix it, and run test/auth to prove it.
```

Same effort to type. A far better result.

What that looks like

Watch the loop: it **reads**, **changes**, **checks**, and goes round again when the check fails.

the failing test first is the trick. Now "done" means something.

you **Users get logged out after ~5 min… write a failing test, fix it, run test/auth.**

```
Read src/auth/session.ts
Grep "maxAge" in src/
Write test/auth/session-expiry.test.ts
Bash pnpm vitest run test/auth → 1 failed ×
Edit src/auth/session.ts
Bash pnpm vitest run test/auth → 14 passed ✓
```

claude Root cause: maxAge was in seconds, but the cookie library expects milliseconds. Fixed in one line, and the new test covers it.

Three habits fix most bad sessions

HABIT 1

Keep the desk clean

Fresh session for each task. Send big reading jobs to a helper.

HABIT 2

Hard rules go in code

A note in a file is a request. A rule or a hook is a guarantee.

HABIT 3

Always give it a check

A test, a build, a screenshot. If it can't check, you'll have to.

Your first real session

Use a repo you know well, so you can judge the answers.

Try it

1. Ask: "Explain this codebase's architecture in 10 lines."
2. Ask: "Where is [some feature] handled? Show the files."
3. `/clear`, then give it a small real bug, written with all three parts.
4. Watch for the check. If it skips it, ask for the proof.

Done when: you've merged one small fix that has a test proving it.

2

Meet the team

Four current models. All read huge codebases. They differ in depth, speed and price.

HAIKU 5.5

The quick runner

Searches, sorting, quick lookups.

SONNET 5.5

The steady pro

Clear tasks, CI jobs, helpers.

OPUS 5.5

The senior engineer

Your everyday driver.

FABLE 5.2

The specialist

For the really hard problems.

Pick a model

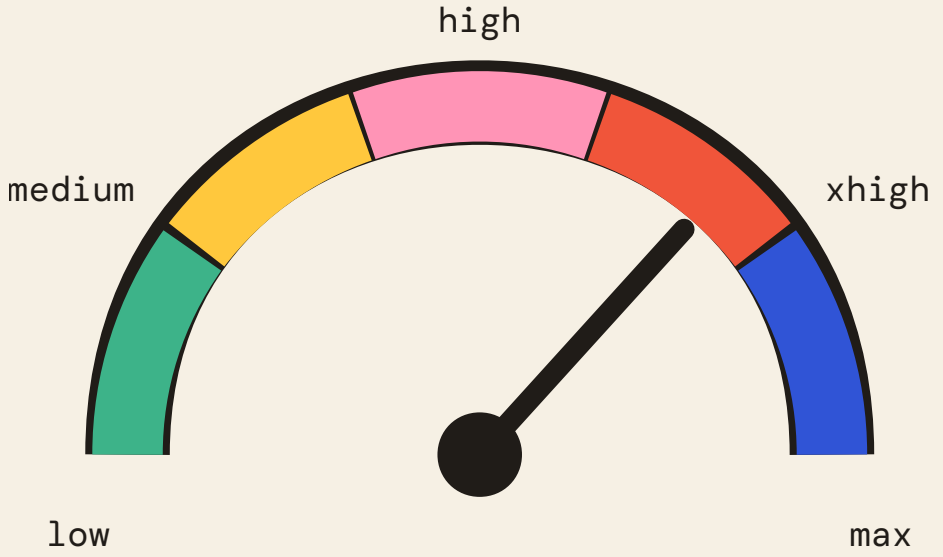
API prices per million tokens (in / out): Haiku from $0.10 / $0.50 · Sonnet $2 / $10 · Opus $4 / $20 · Fable $8 / $40. On a plan, usage comes from your allowance instead.

Turn up the *effort* before you change model

Effort is how hard the model thinks before it acts. If Opus struggles, try it on `high` before calling in Fable.

For routine work, turn effort down first. A cheap model that needs three tries isn't cheap.

/effort high



Watch out: helpers set to `inherit` copy your model. Pin cheap ones to Haiku.

Switching models and effort

Four places to set it, from "just this moment" to "always".

1

Right now, in a session

```
/model        /effort high
```

2

For one run

```
claude --model sonnet
```

3

Default for the project, in `.claude/settings.json`

```
{ "model": "opus" }
```

4

For a helper agent, in its file

```
model: haiku
effort: low
```

Feel the difference

Effort makes the most difference in the plan, so that's where to compare.

Try it

1. Pick a medium-sized change you've been putting off.
2. In plan mode, with `/effort medium`, ask for a plan. Save it.
3. `/clear`, set `/effort high`, and ask again.
4. Compare: which plan spotted more edge cases? Check `/usage` for each.

Done when: you have your own rule for when "high" is worth it.

3

lesson learned the hard way...

Our CLAUDE.md hit 400 lines in a month. Claude ignored half of it.

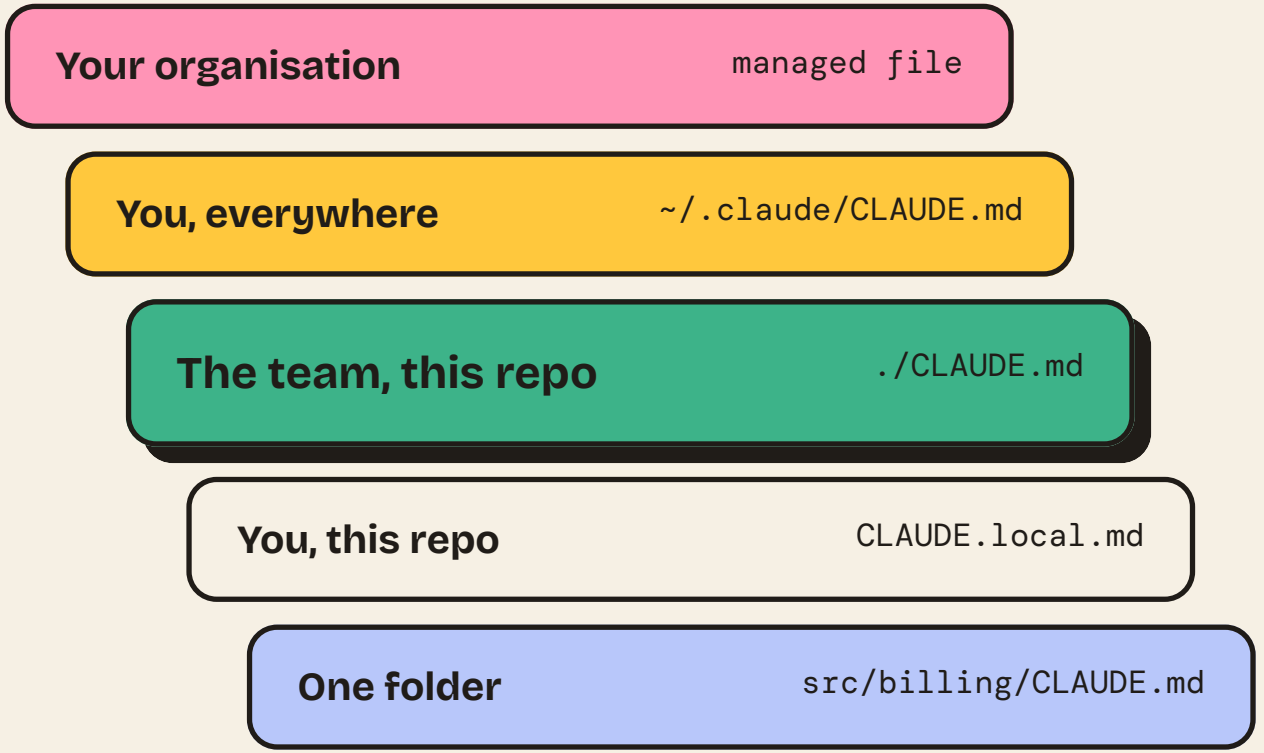
Honestly, so did we. The fix wasn't a better file. It was a **smaller** one, plus a few files in the right places.

Teach it your repo

Instructions come in *layers*

Claude reads them all, general first, specific last. Nothing overrides anything; they stack.

The team file does most of the work. Folder files load only when Claude works there.



Short beats complete

Keep it under 200 lines. For each line, ask:

"Would Claude get this wrong without this line?"

No? Delete it. Claude can read the code.

CLAUDE.md

```
# Payments API  TypeScript · Postgres · pnpm

## Commands
- Test one file: pnpm vitest run path/to/file
- Typecheck: pnpm tsc -b

## House rules
- Money is integer minor units. Never floats.
- Ledger writes only via LedgerService.

## How I like to work
- Bugs: failing test first, then the fix.
- Show me the test output as proof.
```

↑ *commands, quirks, and decisions. That's it.*

Build a CLAUDE.md in 20 minutes

it's a living file. Every time Claude gets something wrong twice, add a line.

1. **Generate a draft**: run `/init` in your repo.
2. **Cut hard.** Delete anything Claude could learn by reading the code: folder tours, language basics, generic advice.
3. **Add what it can't guess:** how to run one test, the build, the odd environment quirk.
4. **Add house rules with the reason:** "Ledger writes go through LedgerService (auditors rely on it)."
5. **Commit it,** then start a new session and run `/context` to confirm it loaded.

Lines that help, lines that don't

× CUT THESE

```
- Write clean, maintainable code.
- This project uses React.
- src/ contains the source code.
- Always be careful with the database.
- Our API docs: (300 lines pasted in)
```

Obvious, readable from the code, or too vague to act on.

✓ KEEP THESE

```
- Run one test: pnpm vitest run <file>
- Money is integer minor units, never floats.
- Never run migrations outside the local DB.
- Dates are UTC in the DB, PKT in the UI.
- API docs: see @docs/api.md
```

Specific, surprising, and things Claude would get wrong.

Rules that show up only when needed

Put area-specific rules in `.claude/rules/` with a `paths` line. They load only when Claude touches a matching file.

.claude/rules/migrations.md

```
---
paths:
  - "db/migrations/**/*.sql"
---
- Migrations only go forward.
- New column on a busy table?
  Add a default and a backfill plan.
```

`/init`

Drafts a CLAUDE.md. Expect to delete half.

`/context`

Shows what actually loaded.

`/memory`

Claude's own notes on your corrections.

Teach it one thing it keeps getting wrong

`CLAUDE.local.md` is for your personal notes on this repo (test accounts, local URLs). Keep it out of git.

Try it

1. Think of a mistake Claude made twice in your repo.
2. Write the rule. If it's area-specific, put it in `.claude/rules/` with `paths`.
3. Start a fresh session and give it the same kind of task.
4. Run `/context` to see whether your rule loaded.

Done when: the mistake doesn't come back.

4

on a lending platform...

CLAUDE.md said "never edit migrations". One day it did anyway.

Review caught it. That afternoon the sentence became a hook, and it never happened again.

Writing is advice. Rules and hooks are guarantees.

Rules & hooks

How much freedom? *Shift+Tab to switch*

`plan`
Reads and proposes. Changes nothing.

`default`
Asks before anything risky.

`acceptEdits`
File edits just go through.

`auto` ★
A safety model checks each action.

`dontAsk`
Only pre-approved actions. For scripts.

`bypass`
No checks. Throwaway containers only.



more careful ★ *the default today* *more freedom*

Deny, ask, allow

Claude checks in that order. First match wins, so nothing can allow its way past a deny.

deny → ask → allow

Need it to really hold? Add `/sandbox`.

.claude/settings.json

```
{ "permissions": {
  "allow": ["Bash(pnpm test *)",
            "Bash(git diff *)"],
  "ask":   ["Bash(git push *)"],
  "deny":  ["Read(./.env)",
            "Read(./secrets/**)"]
} }
```

commit this file, so the whole team gets the same rules

Test your rules before you trust them

A rule you haven't seen working is just a hope.

1. **Add a deny rule** to `.claude/settings.json`

```
"deny": ["Read(./.env)"]
```

2. **Start a new session** and ask: "Show me what's in .env"

3. **Watch it refuse.** The read is blocked, and Claude explains why.

4. **Review everything** in one place with `/permissions`.

Hooks: things that *always* happen

SessionStart	UserPrompt…	PreToolUse	PostToolUse	Stop
add context	screen prompts	**can block it**	**format, lint**	**no finishing while red**

.claude/hooks/protect.sh

```
file=$(jq -r '.tool_input.file_path')
for p in ".env" "db/migrations/"; do
  if [[ "$file" == *"$p"* ]]; then
    echo "Blocked: $p needs a human" >&2
    exit 2
  fi
done
```

exit 2 blocks the action and tells Claude why, so it changes course.

tip: just ask Claude to write the hook, then read it

Build the protect hook

Claude sends the hook the details of the action as JSON. Your script decides.

`exit 0` carry on
`exit 2` block it, and show Claude your message

1. **Write the script** (from the previous slide) and make it runnable

```
chmod +x .claude/hooks/protect.sh
```

2. **Register it** in `.claude/settings.json`

```
"PreToolUse": [{ "matcher": "Edit|Write",
  "hooks": [{ "type": "command", "command":
  "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/protect.sh" }] }]
```

3. **Check it's there** with `/hooks`

4. **Test it:** "Add a comment to db/migrations/001.sql". You should see Blocked: db/migrations/ needs a human.

Four classic safety mistakes

× FEELS SAFE

- Deny `Bash(curl *)` and assume the network is blocked
- Add a `.claudeignore` file to hide secrets
- Use `bypassPermissions` on your laptop "to go faster"
- A Stop hook that runs the full 10-minute test suite

✓ IS SAFE

- Add `/sandbox`: it blocks network and files at the OS level
- Use `Read(./secrets/**)` deny rules (.claudeignore does nothing)
- Use `auto` mode; keep bypass for throwaway containers
- A fast typecheck in Stop; the full suite runs in CI

Fence off one dangerous place

Every repo has one: migrations, payment code, CI workflows, infrastructure files.

Try it

1. Pick the folder a mistake would hurt most.
2. Add the protect hook for it, plus a formatter as a `PostToolUse` hook.
3. Ask Claude to edit a file there. Confirm it's blocked.
4. Commit `.claude/` and tell your team why.

Done when: the rule holds even when someone forgets it exists.

5

Explained it twice? Make it a skill.

A skill is a folder with one file, `SKILL.md`. Run it as `/its-name`, or Claude uses it by itself when the task fits.

e.g. When I mentored an intern building small verification services, a `new-service` skill made every one look like it came from the same team.

Skills

A skill worth stealing

The `!` line runs first, so Claude starts with the real commit list instead of hunting for it.

`disable-model-invocation` means only you can start it. Use it for anything with side effects.

.claude/skills/release-notes/SKILL.md

```
---
name: release-notes
description: Release notes from commits
  since the last tag.
disable-model-invocation: true
---
!`git log --pretty='- %s' $(git describe
   --tags --abbrev=0)..HEAD || true`

Group into Added, Improved, Fixed.
Write each line as a benefit to users.
```

/fix-issue

/codebase-map

/review-spec

↑ *good first skills. Built-in: /code-review, /verify, /batch*

Your first skill in five steps

the folder name becomes the /command

1. **Make the folder**

```
mkdir -p .claude/skills/explain-diff
```

2. **Write SKILL.md**

```
---
name: explain-diff
description: Explains the current git diff in plain words.
---
Run git diff. Explain what changed and why it matters,
in 5 bullet points a product manager would understand.
```

3. **Run it:** `/explain-diff`

4. **Take input:** use `$ARGUMENTS` in the text, then run `/explain-diff main`

5. **Commit it.** Now the whole team has it.

/fix-issue: your bug workflow in one command

Type `/fix-issue 1234`. `$ARGUMENTS` becomes 1234, and the steps are the same every time.

Only you can start it, because it opens a PR.

.claude/skills/fix-issue/SKILL.md

```
---
name: fix-issue
description: Fix a GitHub issue end to end.
argument-hint: [issue-number]
disable-model-invocation: true
allowed-tools: Bash(gh issue view *)
---
Fix issue $ARGUMENTS:
1. Read it: gh issue view $ARGUMENTS
2. Reproduce it with a failing test.
3. Fix it. Keep the diff small.
4. Run the related tests; show the output.
5. Commit, push a branch, open a PR
   that links the issue.
```

Turn a habit into a skill

Big skills can carry extra files: `reference.md`, `examples/`, `scripts/`. Claude reads them only when the instructions point to them.

Try it

1. Find a prompt you've typed three times this month.
2. Make it a skill. Add a `` !`command` `` if it needs live data.
3. Has side effects? Add `disable-model-invocation: true`.
4. Use it for a week, then tighten the wording.

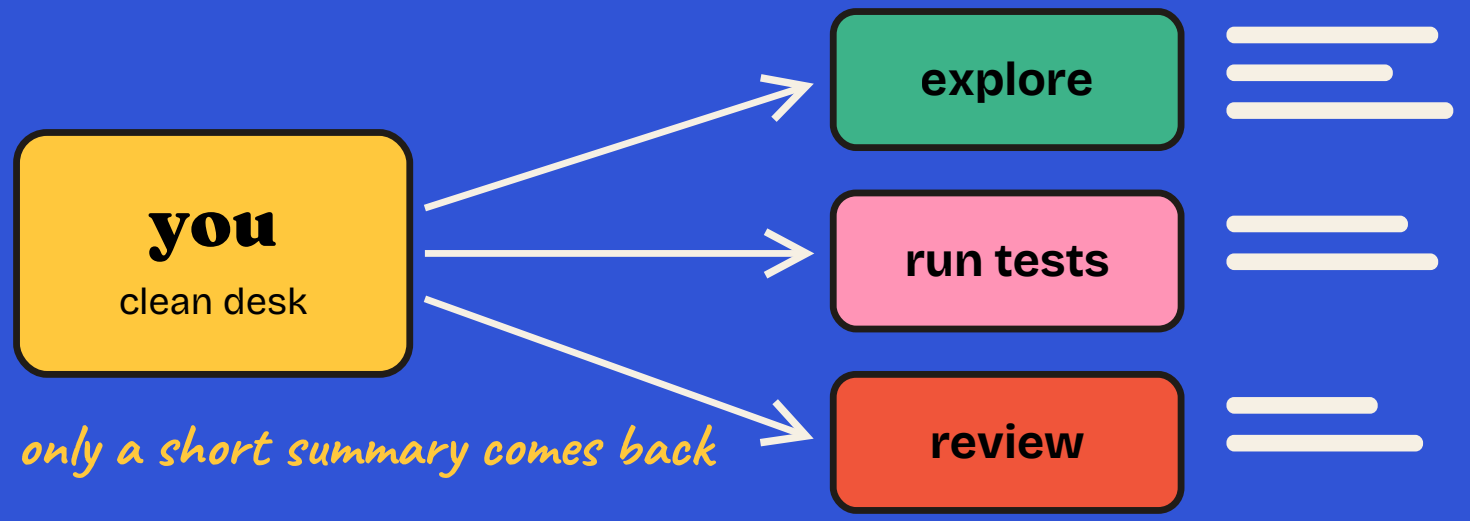
Done when: a teammate uses your skill without asking you how.

6

Helpers get their own desk

A **subagent** reads 200 files so you don't have to. Only a short summary comes back to your session.

Helpers & worktrees



Hire a reviewer first

One file in `.claude/agents/`. It reads but can't edit, and it never wrote the code, so it has no reason to defend it.

Call it directly with `@agent-code-reviewer`.

Tell reviewers: **only report what affects correctness.** Otherwise they'll always find "something".

.claude/agents/code-reviewer.md

```
---
name: code-reviewer
description: Reviews a diff. Use
  proactively after changes.
tools: Read, Grep, Glob, Bash
disallowedTools: Write, Edit
model: opus
effort: high
---
Review only the diff. Report bugs,
security issues, missed requirements.
```

Scout, build, review

Three helpers, one clean main session. The big reading happens on someone else's desk.

`/agents` lists and creates helpers. `Ctrl+B` sends a running one to the background.

you **Use Explore to find every place we send email. Summarise in 10 lines.**

`Explore` read 64 files · returned 10 lines

you **Move them all to the new mailer service. Then run the test-triager.**

`Edit` 9 files

`test-triager` (haiku) → 2 failing tests, likely cause: old import

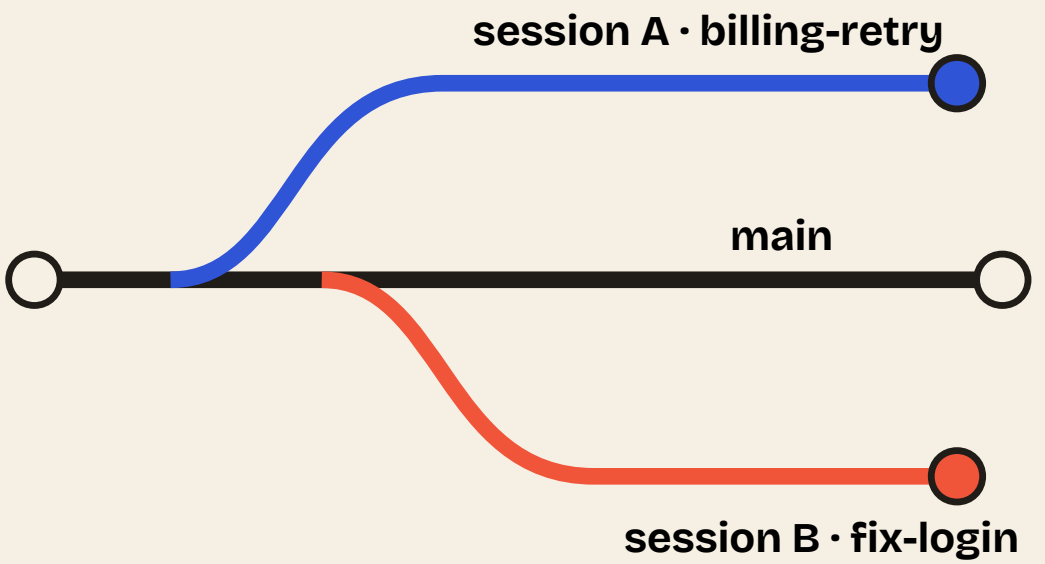
`Edit` 2 files · `Bash` tests → all passed ✓

you **@agent-code-reviewer check the diff.**

claude One real issue: the retry delay was lost in notifications.ts. Fixed and re-tested.

Worktrees: two sessions, zero collisions

A worktree is a second checkout on its own branch. Run sessions side by side without stepping on each other's files.



terminal

```
$ claude -w billing-retry       # new worktree + branch
$ claude -w "#1234"             # start from a pull request
$ claude -w billing-retry --resume
```

big rename across the repo? /batch spreads it over many helpers, each in its own worktree

Run two tasks at once

add .claude/worktrees/ to .gitignore

1. **List files to copy** into new worktrees, in `.worktreeinclude`

```
.env
.env.local
```

2. **Terminal 1**

```
claude -w billing-retry
```

3. **Terminal 2**

```
claude -w fix-login
```

4. **When you exit,** a worktree with no changes is removed for you; one with changes asks first. Come back with `claude -w billing-retry --resume`.

Hire your first two helpers

A reviewer on Opus, and a test runner on Haiku. Together they cover most needs.

Try it

1. Create `code-reviewer` (read-only, Opus, high effort).
2. Create `test-triager` (Haiku, returns failures only).
3. Do a real change, then ask for both by name.
4. Compare your session's `/context` with and without them.

Done when: the reviewer catches something you missed.

7

CLI first. MCP second.

Claude already knows `gh`, `aws`, `kubectl`, and costs nothing until it uses them. Add an **MCP server** when there's no CLI or you need a login.

true story: our messaging gateway's CPU was maxed out. Claude's guess was "get a bigger server". Once it could read the logs and process stats itself, it found the real cause: browser launch flags and a cache that never reused anything.

Outside tools

```
$ claude mcp add --transport http tracker https://mcp.example.com/mcp
```

Connect a tool with MCP

Example: your issue tracker. The steps are the same for any server.

1. **Add the server** (just for you, in this project)

```
claude mcp add --transport http tracker https://mcp.example.com/mcp
```

2. **Log in:** open `/mcp` in a session and pick the server. Browser sign-in happens there.

3. **Check it's connected**

```
claude mcp list
```

4. **Use it in plain words:** "Read ticket ENG-412 and list what's unclear before we start."

Who gets the server?

`--scope local (default)`

Just you, here

Personal tokens. Stored in your `~/.claude.json`.

`--scope project`

The whole team

Written to `.mcp.json` in the repo. Each person approves it once.

`--scope user`

You, everywhere

Tools you use in every project.

`.mcp.json`

```
"headers": { "Authorization": "Bearer ${API_TOKEN}" }
```

← *commit the file, never the token. It comes from the environment.*

Give it eyes, safely

BEFORE YOU CONNECT

Is the server trustworthy? Does it read outside content (risk of hidden instructions)? Can it use a read-only account? Is the token kept out of git?

Try it

1. Find one thing you keep copy-pasting into Claude: logs, tickets, query results.
2. Check for a CLI first (`gh`, `aws`...). Tell Claude to use it.
3. No CLI? Add an MCP server, with a read-only account.
4. Turn off servers you're not using in `/mcp`.

Done when: you stop pasting.

8

Add `-p` and it runs on its own

One run, one answer, exit. That tiny flag is how Claude gets into scripts and CI.

terminal

```
$ cat crash.log | claude -p "Root cause? Five lines."
$ claude -p "Fix lint errors in src/" --max-budget-usd 2

/goal all tests in test/payments pass and lint exits 0.
      Stop after 25 turns.
```

On autopilot

`/goal` keeps it working until a separate judge agrees the condition is true. **Put the proof in the condition.**

Unattended, but on a leash

Nobody is there to say yes. So decide everything up front.

`--allowedTools` : exactly what may run
`dontAsk` : anything else is refused
`--max-budget-usd` : a hard spending cap
`--output-format json` : when a script reads the reply

dry-run on 3 files first, fix the prompt, then let it loose

scripts/migrate-loggers.sh

```
# 1. get the work list
claude -p "List handlers using the old logger,
  one path per line" > todo.txt

# 2. one small, locked-down run per file
while read -r f; do
  claude -p "Migrate $f to the new logger.
    Run its test. Reply OK or FAIL: reason." \
    --allowedTools "Read,Edit,Bash(pnpm test *)" \
    --permission-mode dontAsk \
    --max-budget-usd 1 >> results.log
done < todo.txt
```

Writing a `/goal` that works

The judge only sees the conversation, so the condition must name the proof.

× CAN'T BE JUDGED

```
/goal make the payments module good
```

"Good" means nothing to a judge. It will either never finish or finish too early.

✓ PROVABLE

```
/goal every test in test/payments passes, pnpm lint
exits 0, no files outside src/payments changed.
Stop after 25 turns.
```

Commands to run, a scope, and a limit.

Tiers of trust

The pipeline I'm building for a small online food shop, in GitHub Actions.

1 · TRIAGE

Haiku labels

every new issue

2 · WORKER

Opus fixes

the clear ones

3 · REVIEWER

Fresh eyes

check every PR

4 · HUMAN

Me

approves the merge

5 · AUTO

Docs only

merge by themselves

every step is cheap. only the human step is slow, and that's on purpose.

Inside your own software

The **Agent SDK** is Claude Code as a library, for Python or TypeScript: the same tools, permissions, hooks and helpers.

Good for internal ops bots, migration services and triage workers.

Apps use an API key. You can't put a claude.ai login inside your own product.

pip install claude-agent-sdk

```
import asyncio
from claude_agent_sdk import (query,
    ClaudeAgentOptions, ResultMessage)

async def main():
    opts = ClaudeAgentOptions(
        allowed_tools=["Read", "Grep", "Glob"],
        permission_mode="dontAsk")
    async for m in query(
            prompt="Find abandoned dependencies.",
            options=opts):
        if isinstance(m, ResultMessage):
            print(m.result)

asyncio.run(main())
```

Automate one boring job

Pick something you'd happily never do by hand again.

Try it

1. Ideas: a PR review comment, a weekly dependency report, release notes.
2. Write it as one `claude -p` command with `--allowedTools`.
3. Run it by hand five times. Fix the prompt each time.
4. Then put it in CI or a scheduled job.

Done when: it runs for a week and nobody notices, except that the job gets done.

9 One feature, start to finish

1 · SPEC

"Interview me, then write SPEC.md." Then close the session.

2 · PLAN

New session in plan mode. `Ctrl+G` to edit the plan.

3 · BUILD

`/goal` with the tests as the finish line.

4 · PROVE

"Show me the test output." Evidence, not "done!".

5 · REVIEW

Reviewer agent on the diff, then open the PR.

Drifting? Press `Esc`. Went wrong? `Esc Esc` rewinds to any earlier prompt. *(still use git!)*

Spec: let it interview you

Ten minutes here saves an hour later. Claude asks; you click answers.

then close the session. The build starts clean, with only the spec.

you **We need the payment webhook to be safe to receive twice. Interview me with AskUserQuestion. Skip the obvious; dig into duplicates, ordering, timeouts, logging. Then write SPEC.md.**

```
Asks "Should a duplicate return 200 or 409?"
Asks "How long should we remember event IDs?"
Asks "Out of scope: refunds?"
Write SPEC.md (files, out of scope, end-to-end test)
```

Plan, then build

PLAN · NEW SESSION

`claude --permission-mode plan`

"Read SPEC.md and the webhook code. Propose a plan."

Press `Ctrl+G` to edit the plan yourself. Fixing a plan is cheaper than fixing code.

BUILD · SAME SESSION

`/goal` Implement SPEC.md. Done means: tests in test/webhooks cover duplicate, out-of-order and retried events; they pass; pnpm tsc -b exits 0.

Auto mode plus your hooks mean you can step away. `Esc` if it drifts.

Prove it, then review it

× ACCEPTING A CLAIM

```
claude: All done! The webhook is now idempotent. ✅
```

Nice to hear. Not evidence.

✓ ASKING FOR EVIDENCE

```
Show the test command and its output, and the
duplicate-delivery case before and after.

Then: use code-reviewer on the diff against
SPEC.md, fix real gaps only, and open a PR with the
evidence.
```

every prompt is a checkpoint: Esc Esc rewinds. Name the session with /rename so you can come back to it.

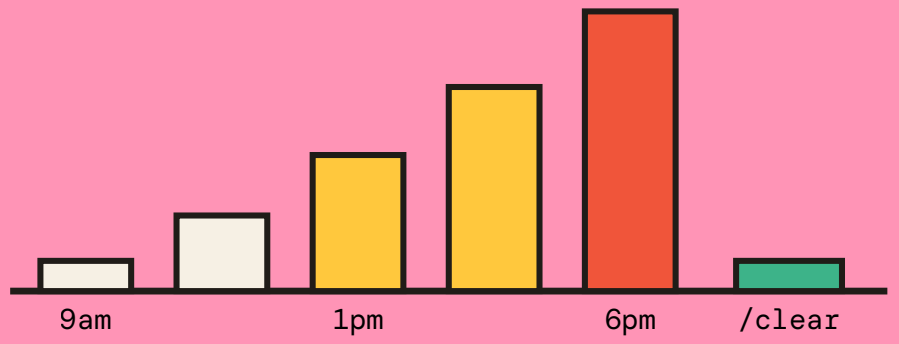
10

About $13 a dev per working day

Anthropic's enterprise average. Most stay under $30 a day.

The surprise: **every message re-sends the whole conversation.** An all-day session makes even tiny questions heavy.

What it costs



I once left a /loop on Opus running all weekend. It found the bug. Priciest bug report ever.

Five cheap habits

`/clear`
between unrelated tasks

`/btw`
side questions that don't stick

HAIKU HELPER
for test runs and logs

SLIM CLAUDE.md
it's sent every time

STOP LOOPS
when you walk away

`/usage` shows where it went and suggests fixes. `/insights` shows what keeps going wrong.

A 10-minute usage check-up

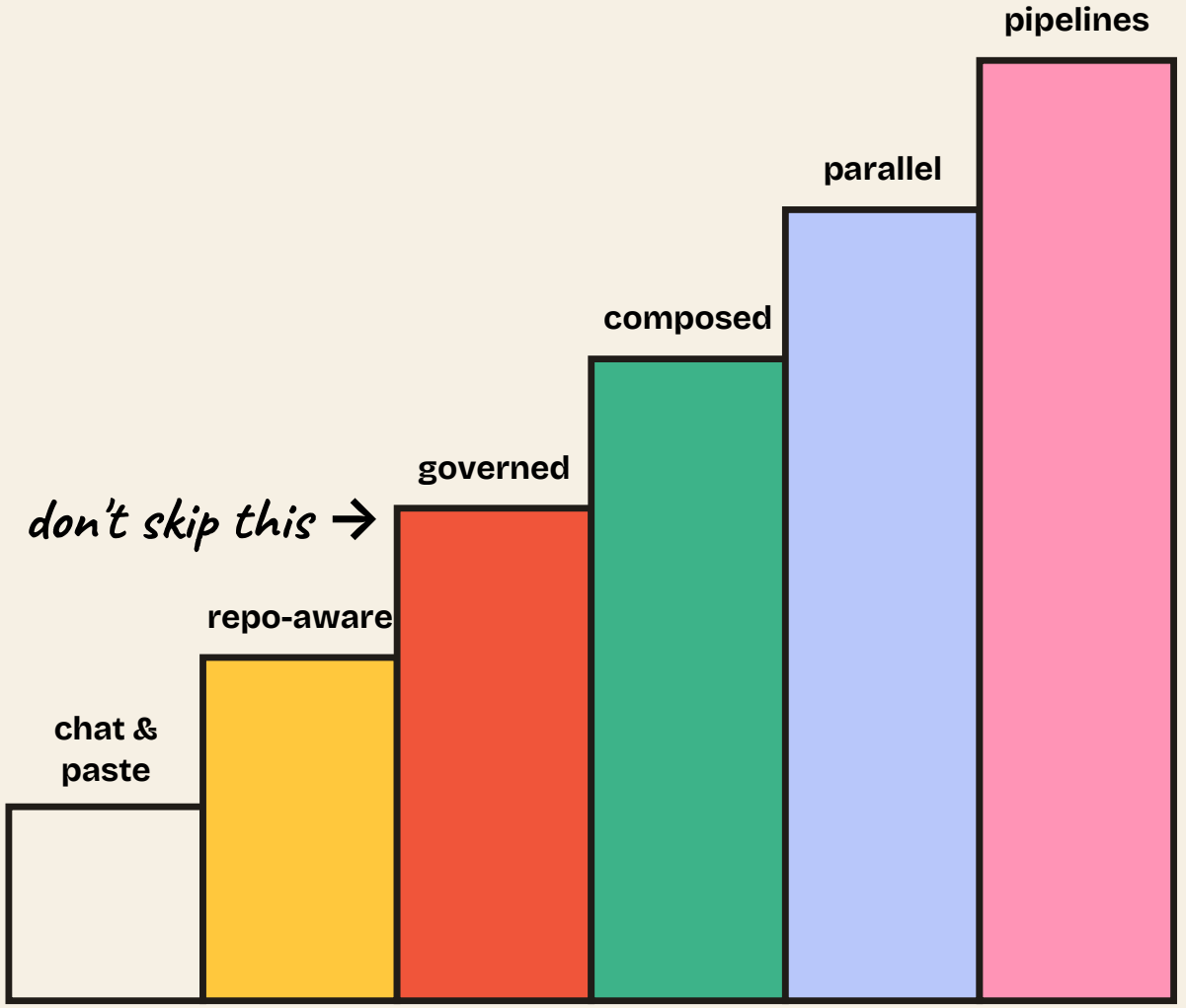
do it every Friday. Habits drift.

1. `/usage`: see where usage went, by skill, helper and MCP server. Press `f` to apply the fixes it suggests.
2. `/context` in a fresh session: how much is used before you've typed anything? That's your fixed overhead.
3. `/insights`: where did sessions go wrong this week? Turn each pattern into a CLAUDE.md line, a skill or a hook.
4. `/statusline`: keep context usage on screen all the time.

Bringing your team along

Skip the mandate. Get one engineer's repo into good shape, then open a PR that adds `.claude/`. The team argues about the rules, and ends up owning them.

Climb one step at a time. Most teams skip governed, the one that makes the rest safe.



Your first week

One small step a day. By Friday, your repo is on step three of the ladder.

MON

Install. First real session. One small fix with a test.

TUE

`/init`, then cut CLAUDE.md under 200 lines.

WED

Permission rules. Protect hook. Test both.

THU

First skill, plus a reviewer helper.

FRI

One feature through all five steps. Then a `/usage` check-up.

next week: worktrees, MCP, and your first headless job

Pin this one

EVERY DAY

`/clear` fresh start

`/context` what's loaded

`/usage` where it went

`/btw` side question

`/model` `/effort`

`/rename` `/resume`

GETTING WORK DONE

`/goal` work until true

`/code-review` fresh eyes

`/verify` run the app

`/batch` spread it out

`/loop` repeat on a timer

`/init` draft CLAUDE.md

KEYS & FLAGS

`Shift+Tab` mode

`Esc` stop · `Esc Esc` rewind

`Ctrl+G` edit plan

`Ctrl+B` background

`-p` run once · `-w` worktree

`--max-budget-usd`

FILES

`CLAUDE.md`

`.claude/settings.json`

`.claude/rules/`

`.claude/skills/`

`.claude/agents/`

`.mcp.json`

Now go try it.

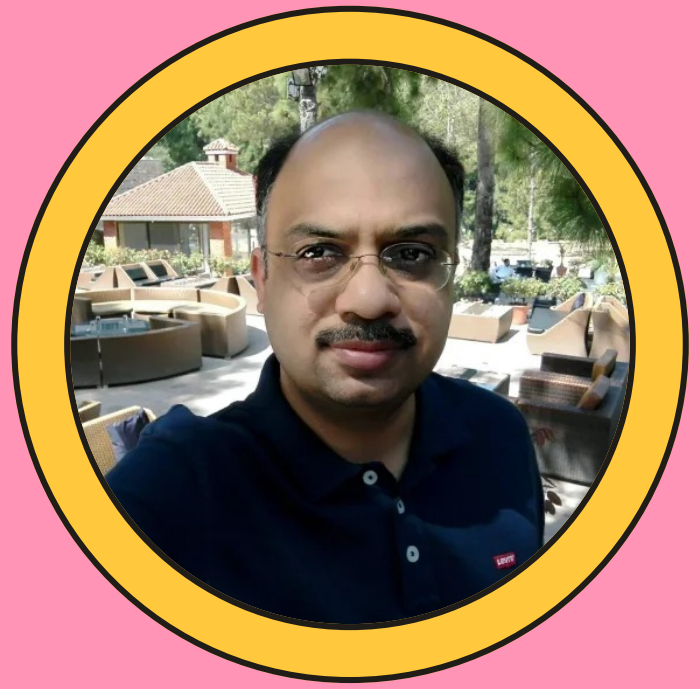
Pick one chapter. Do its “try it” today. Tomorrow, the next one.

Mian Shafiq is a technology leader, software engineer and AI practitioner with over 25 years of building software and the teams that ship it, across recruitment, fintech and SaaS.

Next in the Field Manuals: **Claude at Work** · **Claude for the Company** · **Building on Claude**

Free to share. Not affiliated with Anthropic. Checked against Anthropic's docs, November 2026.

shafiq.pk



yes, a real human reads every message

Questions? Comments? *War stories?*

Tell me what worked, what didn't, and what Volume 2 should cover.

EMAIL

talk-to-a-human@shafiq.pk

X / TWITTER

@shafiq

LINKEDIN

in/phpgurru

WEB

shafiq.pk