AI Field Guide

CHAPTER EIGHT

Autopilot

Plan mode, custom commands, hooks, and subagents — taught from zero, using the files that are now sitting in your actual projects.

Compiled 14 July 2026 · every example in this chapter is a real file on your machine

CHAPTER EIGHT

What this chapter is

Chapter Seven told you that you'd never used plan mode, never written a command, never written a hook, and never specialised a subagent. This chapter teaches all four — and it does it differently from a normal tutorial.

THE FILES IN THIS CHAPTER ALREADY EXIST

Every example below is **a real file I wrote into one of your real projects** while writing this. You are not reading about a hypothetical `/ship` command. You are reading about *your* `/ship` command, which works, right now, in Laqta.

A PDF has never once made anyone configure anything. So the configuration came first, and this is the explanation of it.

The goal is the thing Boris Cherny, who runs Claude Code, describes:

> *I don't prompt Claude anymore. I have loops running that prompt Claude... My job is to write loops.*
>
> BORIS CHERNY, HEAD OF CLAUDE CODE

You are currently a very good pilot flying manually. Everything below is the autopilot.

CHAPTER EIGHT

1 · Plan mode

What it is

A mode where Claude **investigates and writes a plan, but is physically prevented from editing anything**. It reads your code, works out what it would do, and shows you — then waits. Nothing is written until you approve.

It is not a prompt or a politeness convention. It's a *permission mode*: the edit tools are switched off.

How to use it

Press `Shift+Tab` to cycle permission modes until it says **plan**. Then type your request as normal. Claude comes back with a plan; you either approve it, or you edit it, or you say "no, do it differently" — *before* a single file has changed.

> WHERE YOU ARE NOW
>
> Your transcripts: `bypassPermissions` on **656** turns, `default` on 15, **plan on zero.** You optimised for never being interrupted — and paid for it by being wrong autonomously. Plan mode is a thirty-second gate in front of an hour-long build.

When to use it — and when not to

USE PLAN MODE	SKIP IT
Anything touching a **schema, auth, or money**	You could describe the diff in one sentence
Anything against a **live client system** (Noor, X-Grid — `jat-erp.com` is production)	Renaming a variable, fixing a typo
When you're not sure the approach is right	You've done this exact thing four times already
Multi-file refactors	Anything you'd happily undo with `git checkout`

Planning has real overhead. Don't gate a one-line CSS fix behind it. **Gate the migration.**

Why it matters more for you than for most people

The most expensive incident in your transcripts — the GitHub request that burned a million tokens because it never said *"don't touch any code"* — **could not have happened in plan mode.** Claude would have come back with "here's my plan: clone the repo, analyse the codebase, create the project, push the code," and you would have said *"no, just the empty repo"* in four seconds, for free.

CHAPTER EIGHT

2 · CLAUDE.md — write it down once

What it is

A markdown file at the root of a project. **Claude reads it automatically at the start of every session in that project.** It is how you stop re-typing the same facts.

> **WHY YOURS NEVER WORKED: YOU RUN CLAUDE FROM YOUR HOME DIRECTORY**
>
> A project CLAUDE.md **only loads when Claude *starts* in that project.** You launch from `~` 74% of the time, so even if you'd written one, it would never have loaded.
>
> **The habit that unlocks everything in this chapter: `cd` into the project before you start Claude.** One session per project. Commands, hooks, agents and CLAUDE.md are all scoped to the project — from `~`, none of them exist.

What I wrote for you

Six of them — Laqta, the three Elite apps, Sandow, and Asas. Here's the real one from Laqta, and why each part earns its place:

```
# Laqta

## Commands
npm run dev      # vite + express API together
npm run build    # vite build → dist/
/ship            # build, deploy, verify the live URL

## Architecture
- Production API is Cloudflare Pages Functions in `functions/api/`.
- `server/index.js` is a LOCAL DEV STAND-IN ONLY. It does not deploy.
  If you add an endpoint, it must exist in `functions/api/` or it does not exist.

## Rules that are not obvious
- `src/lib/pricing.ts` is the single source of truth for money.
  Never hardcode a price, a plan, or a credit cost anywhere else.
- English, LTR only. i18n and RTL were deliberately REMOVED.
  Do not reintroduce a language switcher or `dir="rtl"`.
- framer-motion mount animations get stuck in this project. Keep components static.

## Definition of done
The DEPLOYED url shows it — not localhost, and not "the build passed."
```

The rule for every line

Anthropic's test: "**Would removing this line cause Claude to make a mistake?**" If no, delete it.

Notice what is *not* in there. No "use clean code." No list of the directory structure. No explanation of what React is. Every line is a thing Claude would get *wrong* without being told — the Express file that looks like an API but isn't, the pricing file, the RTL that was deliberately removed, the animation bug that made you write "*keep static*" in your memory.

> KEEP IT UNDER 200 LINES. THIS IS NOT A STYLE PREFERENCE.
>
> "Bloated CLAUDE.md files cause Claude to **ignore your actual instructions**." A long file doesn't get followed more carefully — it gets followed *less*, because the important line is drowning.
>
> Your `noor-erp-rebuild.md` is **68KB**. That is not a CLAUDE.md, and it's currently in global memory taxing every unrelated session you open.

The three diagnostic signals

- Claude **ignores a rule you wrote** → the file is too long.
- Claude **asks something the file answers** → the phrasing is ambiguous.
- You make **the same correction twice** → it belongs in CLAUDE.md. Or better, in a hook.

CHAPTER EIGHT

3 · Custom slash commands

What it is

A markdown file in `.claude/commands/`. The filename becomes the command. `ship.md` gives you `/ship`. The body is **a prompt you wrote once** instead of retyping.

That's genuinely all it is. A saved prompt with a name. But the frontmatter is where it becomes powerful.

Your real `/ship`, annotated

```
---
description: Build, deploy Laqta to Pages, and verify the live URL works.
disable-model-invocation: true
allowed-tools: Bash(npm run build), Bash(npx wrangler *), Task
---

Ship Laqta to production.

Current state:
- Branch: !`git branch --show-current`
- Uncommitted changes: !`git status --short | head -20`

## Steps
1. Build. `npm run build`. If it fails, STOP. Never deploy a broken build.
2. Deploy. `npx wrangler pages deploy`
3. Verify — do not skip. Dispatch the **verifier** subagent against
   https://laqta.pages.dev …
4. Report the URL and the verifier's verdict.

If the verifier says BROKEN, say so plainly. Do not tell me it shipped
if it didn't work.
```

LINE	WHAT IT DOES
`disable-model-invocation: true`	Claude can **never** trigger this on its own — only you, by typing `/ship`. Essential for anything with side effects. It also costs **zero context** until you type it.
`allowed-tools:`	These commands run **without a permission prompt**. Narrowly scoped, so it can run `wrangler` but not `rm`.
`!`git status --short``	**The best feature almost nobody knows.** The backtick-bang runs the command *before Claude sees the prompt* and pastes the output in. Claude doesn't call a tool — it just *already knows* the branch and the uncommitted files. Saves a round-trip and a chunk of context.
"If the verifier says BROKEN, say so plainly"	You are writing the standard into the command, once, instead of asking "is it really deployed?" every time.

> **WHAT THIS RETIRES**
>
> In two weeks you hand-typed `wrangler` **567 times**, `curl` **1,182 times**, `npx tsc` **300 times**, and `npm run build` **255 times** — the same build → typecheck → deploy → poke-the-URL loop, across ten projects, by hand.
>
> That loop is now one word.

CHAPTER EIGHT

4 · Hooks — the part that isn't a suggestion

What it is, and why it's different from everything else

This is the key distinction and it's worth reading twice.

> CLAUDE.MD IS CONTEXT. A HOOK IS CODE.
>
> A CLAUDE.md line is a *strong suggestion* that Claude is very likely to follow. **A hook is a shell command the harness runs whether Claude likes it or not.**
>
> If a rule must hold *every single time, with no exceptions* — it is not a memory line. It is a hook. **Every rule you've ever had to repeat is a rule that wanted to be a hook.**

Your real typecheck hook

It lives at `.claude/hooks/typecheck.sh`, and it's wired up in `.claude/settings.json`:

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

Read that as a sentence: "**After Claude uses the Edit or Write tool, run this script.**" That's a hook. An event, a matcher, and a command.

The script itself does three things:

1. Reads the edited file's path from the JSON the harness pipes to it. **If it isn't a `.ts` or `.tsx` file, exit immediately** — no point typechecking a README.
2. Runs `tsc --noEmit`. Takes about three seconds.
3. **If there are errors, print them to stderr and `exit 2`.**

> `EXIT 2` IS THE WHOLE TRICK
>
> Exit code 2 doesn't just fail. It **feeds the error message straight back to Claude**, which then fixes it and carries on — *before* handing control back to you.
>
> The practical effect: **Claude can no longer leave a type error behind.** Not "shouldn't." *Can't.* You will never see the round-trip; you'll just stop finding broken types.

I tested it on Laqta before shipping it to you — injected a real type error, confirmed the hook caught it and exited 2, then restored the file. It's silent when clean, so you'll never notice it working. That's the point.

The other events worth knowing

EVENT	FIRES WHEN	GOOD FOR
`PostToolUse`	After a tool runs	Typecheck, lint, format after edits ← yours
`PreToolUse`	Before a tool runs — **can block it**	Refuse writes to `migrations/` or `.env`
`Stop`	When Claude tries to finish its turn — **can refuse**	"You may not end the turn until the build passes"

And the easiest way to write one: **just ask.** "Write me a hook that blocks any write to the migrations folder." Claude writes hooks.

CHAPTER EIGHT

5 · Subagents — your habit, made automatic and cheap

What it is

A subagent is **a second Claude, with its own fresh context window, that does one job and reports back a summary.** The work it does — all the file-reading, all the false starts — never enters *your* conversation. Only the answer does.

Two reasons that matters:

- **Context.** Exploration is expensive. Delegate it and you get the conclusion without the 40 files.
- **Independence.** A reviewer with a fresh context *sees only the diff* — not the reasoning that produced it. Anthropic: *"a fresh context improves code review since Claude won't be biased toward code it just wrote."*

Your real `verifier` agent

It lives at `~/.claude/agents/verifier.md` — global, so it works in every project. The frontmatter is the interesting part:

```
---
name: verifier
description: Independently verifies that a change actually works by driving
  the real app — loads the deployed URL, checks the console, screenshots the
  result, and reports what it OBSERVED. Read-only: it never edits code.
tools: Bash, Read, Grep, Glob, mcp__Claude_Browser__*
model: haiku
---
```

LINE	WHY
`model: haiku`	**This is the money line.** Of your 249 subagent dispatches, **183 inherited Opus** — you were paying flagship rates for agents doing `ls` and reading screenshots. Verification is not hard reasoning. Haiku does it at a fraction of the cost.
`tools:` *(no Edit, no Write)*	It **cannot** change your code. It can only look. A verifier that can "fix" what it finds is no longer an independent check.
The system prompt	Encodes the standard *you already hold*: never say "verified" without naming the evidence; report what you observed, not what you assume; if you couldn't test it, say so.

> THIS IS NOT A NEW HABIT. IT'S YOUR EXISTING ONE, AUTOMATED.
>
> You made **2,001 preview-eval and screenshot calls** in two weeks, by hand, on Opus. The verifier does exactly what you were already doing — it's just called automatically by `/ship`, and it costs almost nothing.
>
> You can also call it directly: *"use the verifier subagent to check the live site."*

CHAPTER EIGHT

6 · A correction, and an honest note on your skills

Chapter Seven said your `erp-builder` skill had zero invocations and speculated that its trigger needed fixing. **I checked. That speculation was wrong, and I'm not going to pretend otherwise.**

`erp-builder` exists to **stand up a brand-new ERP for a brand-new client.** You haven't onboarded one since you wrote it — you've been working on *Noor* and *X-Grid*, which are existing ERPs it doesn't cover. Zero invocations isn't a broken trigger. It's a skill patiently waiting for the situation it was built for. I've left it exactly as it is.

What *is* true: **you can always invoke a skill explicitly by name.** Typing `/core-2.0-dashboard` is free, deterministic, and doesn't depend on Claude inferring that you wanted it. You mentioned "Core 2.0" in **171 prompts** and typed the command that loads the entire design system **six times**. That's the gap — not the description.

> COMMANDS AND SKILLS ARE THE SAME THING NOW
>
> `.claude/commands/ship.md` and `.claude/skills/ship/SKILL.md` both give you `/ship`. A skill is just the bigger form — it can bundle reference files and scripts that load *only when needed*. Start with a command. Promote it to a skill when it outgrows one file.

CHAPTER EIGHT

7 · The two you'll want next

Worktrees — for the six-projects-at-once problem

`claude --worktree feature-x` (or `-w`) gives an agent **its own isolated git worktree** — a separate checkout — so two agents can work at once without stepping on each other's files.

> THE ONE LINE THAT MAKES WORKTREES USABLE ON CLOUDFLARE
>
> A fresh worktree doesn't have your `.env` or `.dev.vars` — they're gitignored. So the app won't run, and worktrees feel useless.
>
> **Fix:** a `.worktreeinclude` file (gitignore syntax) listing `.env`, `.dev.vars`, `.env.local`. Those get copied into every new worktree automatically. This is the difference between worktrees being a nice idea and being something you actually use.

`/goal` — the actual autopilot

This is the closest thing to Boris's loop, and it's one line.

```
/goal all tests in test/ pass and npx tsc --noEmit is clean, or stop after 20 turns
```

A **separate evaluator model** re-checks that condition after *every single turn*, and **restarts Claude until it holds.** You are no longer in the loop. You set the condition and walk away.

Two rules: the condition must be **provable from the transcript** (the evaluator can't run commands itself — so "tests pass" works only if the tests were actually run and printed), and **always bound it** with "or stop after N turns" so it can't spin.

Beyond that sit **Routines** — scheduled agents that run on Anthropic's infrastructure and survive your laptop being closed. That is where the AI Brief belongs, rather than a launchd timer firing

into a wall.

CHAPTER EIGHT

Your first week

Everything above already exists. Here's how to actually pick it up.

1. **Today — change one habit.** `cd ~/Desktop/Laqta` *before* you start Claude. Nothing else in this chapter works from your home directory.
2. **Today — type `/ship` once.** Make a trivial change to Laqta and ship it. Watch it build, deploy, dispatch the verifier, and report back with evidence. That single run demonstrates commands, subagents, and the verification standard at once.
3. **This week — use plan mode once, deliberately.** `Shift+Tab` to plan, on something real. Feel what it's like to see the plan before the diff.
4. **This week — let the hook catch you.** You don't have to do anything. Edit some TypeScript, introduce an error, and notice that Claude fixes it before handing back to you.
5. **Then — copy the pattern to the ERPs.** Once you trust it on Laqta and the Elite apps, Noor and X-Grid are the same four files. That's where the real risk is (656 bypass-permission turns against a *production client ERP*), and it's where plan mode earns its keep.
6. **When you're ready — write your first `/goal`.** That's the day you stop prompting and start writing loops.

> THE HONEST FRAMING
>
> None of this makes you a better engineer. You already ship — nine out of nine live. What it does is stop you **personally re-driving a loop that a machine should be driving**: the same build, the same typecheck, the same deploy, the same screenshot, the same "are you sure it works," 2,300 times in a fortnight.
>
> You were never the bottleneck because you lacked skill. You were the bottleneck because you were *in the loop*.

The files this chapter describes. `~/.claude/agents/verifier.md` · `Laqta/CLAUDE.md` + `.claude/commands/ship.md` + `.claude/hooks/typecheck.sh` + `.claude/settings.json` · the same four in `elite-mobile`, `elite-trainer`, `elite-admin`, `Sandow` · `Asas/CLAUDE.md` + `ship.md` (no typecheck hook — Asas has no TypeScript). Sourced from Claude Code docs: memory, skills, sub-agents, hooks, worktrees, /goal, permission modes.