mid 2026 edition

```
$ claude --version
2.1.145 (Claude Code)

$ git log --oneline -3
a3f2c11 fix: race condition in auth handler
9b1e447 feat: add streaming response support
c88d903 refactor: extract validation layer
$ claude "write tests for auth.ts"
```

WORKING WITH

Claude Code

Practical workflows for developers who ship

CHAPTER 01 | CHAPTER 02 | CHAPTER 03 | . . .

```
$ claude "refactor this for readability"
✓ Analyzing 847 lines across 12 files...
✓ Identified 6 improvement opportunities
✓ Applied changes — 3 files modified
$ ▊
```

Nuno Freire
www.booleanstate.com

About this guide

This is not a beginner's guide to Claude Code. You already know it exists. You've probably used it for auto-completion in your IDE, to generate a component, fix a bug, or explain some unfamiliar code. It worked. Mostly.

But if you're honest, you're still using it like a very expensive Stack Overflow. Ask a question, get an answer, paste it in, break something else, repeat. The context window fills up. The outputs start going sideways. The tool feels unpredictable.

The problem isn't Claude Code. The problem is the workflow around it.

This guide is a compilation of real workflows: patterns that experienced developers have settled on after months of daily use. Some come from the developer community. Some come from hard-won trial and error. I'm also including some principles from Boris Cherny, the Staff Engineer at Anthropic who built Claude Code and shared his own setup publicly.

The goal: help you stop using Claude Code as a chatbot and start using it as a structured part of your development process. These are tips and workflows I use daily.

Target reader: mid-to-senior developers, comfortable with the terminal, already using Claude Code, want to get significantly more out of it without the chaos.

A note on cost and model choice

Claude Code is metered per token. Heavy use isn't free, and that matters for the workflows in this guide: plan mode, sub-agents, and Superpowers all involve multiple back-and-forth turns rather than a single shot.

For day-to-day work I use Sonnet 4.6, not Opus 4.6 or 4.7. Sonnet is significantly cheaper per token and good enough for almost everything in this guide: planning, implementation, code review, automating dev admin. I reach for Opus only when a task is genuinely hard reasoning: complex architectural decisions, debugging something subtle, or planning a large refactor where the cost of a wrong call is high. For everything else, Sonnet is the right default.

This is worth being deliberate about. The temptation is to always pick the most powerful model. In practice, picking Sonnet for most work and Opus for the 10% that needs it gives you better economics with no meaningful quality loss.

A note on when NOT to use Claude Code

The workflows in this guide make Claude Code more useful, but more useful isn't always the same as the right tool. Skip it when:
You want to learn by doing. If the goal is to internalise a new framework or pattern, Claude doing the work for you defeats the purpose.

The question is small enough that asking it is slower than answering it. If you already know roughly what you'd write, just write it.

This guide is about getting more out of Claude Code, not using it more. Different things.

(c) 2026 — Working with Claude Code

Contents

1. The Mental Model Shift
2. CLAUDE.md: Your Project's Brain
3. Plan Mode: Stop Letting Claude Guess
4. Context Management: The 60% Rule
5. Sub-Agents: One Session, One Job
6. Superpowers: A Skills Framework for Claude Code
7. Context7: Stop Claude Hallucinating Your Library's API
8. Automating Dev Admin (Jira, PRs, Standups)
9. Custom Commands: Shorthand for the Prompts You Use Daily
10. What NOT to Do

(c) 2026 — Working with Claude Code

SECTION 1

The Mental Model Shift

The most important thing to understand about Claude Code is what it actually is.

It is not a smarter autocomplete. It is not a chatbot that happens to know code. It is an agentic tool, which means it can plan, execute, use tools, read files, run commands, and make sequences of decisions autonomously. The gap between treating it as a chatbot and treating it as an agent is where most of the productivity is hiding.

Here is the clearest way to see the difference:
Chatbot usage:

```
"Explain how Angular signals differ from observables. When would I prefer one over the other?"
```

Claude explains. You read. You decide. You implement.

Agentic usage:

```
"Read the router config in src/app/app.routes.ts and refactor the three heaviest modules to use lazy loading, following the patterns already used in the workspace module."
```

Claude reads your actual files, understands your actual patterns, and does the actual work.

Both are valid uses. The difference is who is doing the work. Most developers stay stuck in chatbot mode because it's familiar. The leverage lives in the second mode, but it only works reliably when Claude has the context, conventions, and constraints to make good decisions on your behalf. That setup is what the rest of this guide is about.

One honest caveat: agentic work compounds error. Even if Claude makes the right call on most individual decisions, a feature with many decision points has many opportunities for small wrong calls to stack up. This is why structure matters more than prompt quality. The developers getting the most out of Claude Code are not writing better prompts, they are building better systems around the tool, so each decision has less room to drift.

SECTION 2

CLAUDE.md: Your Project's Brain

CLAUDE.md is a markdown file you place in your project root. Claude Code reads it at the start of every session. It is the single most impactful thing you can configure.

Think of it as the onboarding document for a senior developer joining your project. It should answer: What is this project? What are the conventions? What should never happen? What should always happen?

Getting started

For new projects, the right sequence is: scaffold the project first, then ask Claude to generate the CLAUDE.md once the structure is in place:

```
"Look at the codebase structure, existing components, and patterns in use, then
 generate a CLAUDE.md that reflects how this project actually works."
```

For existing projects, Claude Code can generate a starting point automatically with /init. This scaffolds a baseline from your project structure. It won't capture all your conventions yet, but it gives you something to build on.

Either way, treat the generated file as a first draft, not a finished product.

What to put in it

Project overview: one paragraph. Framework, key libraries, what the app does.

Tech stack: explicit versions matter. Claude's training data includes older versions of everything. Stating "PrimeNG 21, AgGrid 32, Angular 21" prevents it from suggesting deprecated APIs.

Development commands: the exact commands to start, build, test. Claude will use these.

Architecture overview: folder structure, module patterns, naming conventions. The more specific, the better.

Coding rules: this is where most of the value is. Be explicit about what to always do, what to never do, patterns to follow, and anything Claude gets wrong repeatedly. Add those as rules immediately when they happen.

Safety protocols: for risky operations like bulk migrations, define the exact procedure Claude must follow before touching files.

What not to put in it

Keep it under 200 lines. Every line costs tokens at session start. An 800-line CLAUDE.md means Claude has spent significant context before you've typed anything.

Do not put things that belong in code comments or README files. CLAUDE.md is for Claude, not for humans reading the repo.

Do not put aspirational rules nobody enforces. If Claude breaks a rule and you don't correct it, the rule is noise.

The living document rule

> When Claude does something wrong, add it to CLAUDE.md so it doesn't repeat it. Every mistake is a missing rule. The document compounds over time.

The three principles (Boris Cherny, creator of Claude Code)

Add these to the top of every CLAUDE.md you maintain:

> **SIMPLICITY**
> Make every change as simple as possible. Minimal code. If you can delete lines instead of adding them, do that.

> **ROOT CAUSES**
> Find root causes. No temporary fixes. No band-aids. Hold yourself to senior developer standards.

> **MINIMAL FOOTPRINT**
> Only touch what's necessary. No side effects. No introducing new bugs while fixing old ones.

Adapting Claude to your working style

Beyond project conventions, CLAUDE.md is also where you define how you want Claude to behave with you as a developer. This block is worth adding to every project:

```
## Working Style

- Be direct and concise. No fluff, no unnecessary caveats.
- Always write down assumptions and ask clarifying questions upfront.
- Call out if something seems off: honest pushback is preferred over agreement.
- Do not over-explain things the developer clearly already knows.
```

Adjust the tone to match how you actually work. The point is that CLAUDE.md isn't just about the codebase, it's about the working relationship.

Multiple CLAUDE.md files

Claude Code supports CLAUDE.md files at different levels: project root applies to everything, subdirectory applies only to that folder. For a large codebase with distinct concerns, this lets you have different rules per context without bloating the root file.

SECTION 3

Plan Mode: Stop Letting Claude Guess

This is the highest-ROI habit change in this guide. It costs nothing except discipline.

The problem with jumping straight to implementation: Claude starts making decisions the moment you hit enter. Framework choice, file structure, naming, patterns, edge case handling: all decided implicitly. Most are fine. Some are not. By the time you notice a wrong assumption, Claude has built on top of it across a dozen files.

Plan mode front-loads all decision-making before a single file is touched.

The workflow

Step 1: Ask for a plan, not an implementation:

```
"Give me a plan for implementing [feature]. Do not write any code or touch any files yet."
```

The phrase "do not write any code or touch any files yet" is not optional. Without it, Claude will start implementing mid-plan.

Step 2: Review the plan. Annotate where Claude got something wrong: wrong file paths, wrong patterns, missing edge cases.

Step 3: Send it back:

```
"Address all my notes. Do not implement yet."
```

Repeat until every decision is resolved.

Step 4: Only then:

```
"The plan looks good. Implement it now."
```

The keyboard shortcut

Shift + Tab switches between Plan Mode and Edit Mode without typing anything. Press it before your request to enter Plan Mode. Press it again when you're ready to implement.

When to use it

Use plan mode for: new components or features touching multiple files, refactors touching more than three files, anything with unclear requirements, anything you haven't done before in this codebase.

Skip it for: single-file changes, things you've done before, trivial queries.

How long should a plan be?

A plan for a two-hour feature should be one to two pages. If it's longer than a developer can read and verify in five minutes, it's too long. Say so:

```
"This is too detailed. Give me a shorter plan: just the key decisions and file changes."
```

SECTION 4

Context Management: The 60% Rule

Claude Code has no memory between sessions. Even within a long session, quality degrades as the context window fills. Most developers don't notice until outputs start going sideways mid-feature.

Think of Claude's context window like RAM. Fixed size. As you fill it with conversation, code, tool outputs, and file reads, older decisions get compressed or dropped. By the time you're deep into a big feature, Claude is working with a degraded version of everything decided earlier.

How much context is too much?

There's no exact number: quality degradation is gradual, not a cliff. But practitioners commonly report a noticeable drop somewhere between 40% and 60% capacity. Auto-compaction fires near 100% and is lossy: it summarises in memory and you don't see what survived.

The rule of thumb most developers settle on: keep context below roughly 60-80% during focused work. Beyond that, the marginal quality loss starts outweighing the convenience of staying in one session.

The fix: PROGRESS.md

The standard fix is a PROGRESS.md file you keep in the project root during multi-session work. The structure and rules for it should live in your CLAUDE.md (see the sample in the appendix), so Claude knows exactly what to write and when, without you re-explaining it every time.

With that in place, the prompt becomes simple:

```
"Update PROGRESS.md."
```

Claude reads the conventions from CLAUDE.md and writes the file accordingly. Then run /clear to reset context. Start the next session with:

```
"Read PROGRESS.md and continue where we left off."
```

This is better than /compact. Compact summarises in memory and you don't see what survived. PROGRESS.md is explicit. You can read it, edit it, and correct it before the next session. You control exactly what survives.

> PROGRESS.md isn't a recovery tool, it's a precaution. Like a git stash, you create it before you need it, not after. Create it at the start of any non-trivial task.

When to start a fresh session

Start fresh when: context approaches 60-80%, Claude starts contradicting earlier decisions, outputs feel increasingly generic, or you're switching to a completely different concern within the same project.

SECTION 5

Sub-Agents: One Session, One Job

Most developers use Claude Code as one big general assistant: build this feature, fix that bug, review the code, write the PR. That's like hiring one person to be your architect, builder, and quality inspector simultaneously. Conflicts happen, context gets polluted, quality drops.

Sub-agents is a discipline, not a different tool. One Claude Code session equals one clearly defined job.

Instead of this

```
"Build the notification widget, make sure it follows our patterns, write tests, and
 give me a PR description when done."
```

Do this

Session 1, the builder:

```
"Your job: implement the notification widget. Follow CLAUDE.md conventions exactly. Do
 not write tests. Do not review. Just implement. Read PROGRESS.md first if it exists."
```

Session 2, the reviewer:

```
"Your job: review the changes in git diff for this feature. Check against CLAUDE.md
 rules only. List violations. Do not fix anything."
```

Session 3, the PR writer:

```
"Your job: write the PR description. Read the git diff. Format: what changed, why, how
 to test."
```

Why it works

Each session starts with clean context focused on one concern. The reviewer isn't biased by having written the code. The PR writer isn't trying to remember what it built two hours ago. Each agent has a tight, unambiguous job.

(c) 2026 — Working with Claude Code

SECTION 6

Superpowers: A Skills Framework for Claude Code

Superpowers is an open-source skills framework that gives Claude Code a structured methodology for the full development workflow: brainstorming, planning, execution, testing, review, and branch management. Think of it as a set of reusable playbooks that Claude follows consistently, rather than ad-hoc prompts you write from scratch each time.

A note on cost: Superpowers consumes more context than ad-hoc prompting because the skills are loaded into context at session start. For small tweaks and quick fixes, it's overhead you don't need. For non-trivial features, where the alternative is improvising your own structure and losing time to rework, the upfront context cost pays for itself many times over.

Two versions exist. The original at github.com/obra/superpowers is cross-platform and works across Claude Code, GitHub Codex, Gemini CLI, and others. Because it's cross-platform, it can't use Claude Code-native features. The Claude-specific fork at github.com/pcvelz/superpowers removes that constraint and adds dependency tracking between tasks, real-time progress visibility, and a pre-commit hook that blocks commits when tasks are incomplete. If you use Claude Code exclusively, use the fork.

Installation

```
/plugin marketplace add pcvelz/superpowers
/plugin install superpowers-extended-cc@superpowers-extended-cc-marketplace
```

Verify with /help: you should see brainstorming, writing-plans, and executing-plans in the command list.

The core workflow

1. /brainstorming: Before writing any code. Refines your rough idea through targeted questions, explores alternatives, presents design in sections for your validation, saves a design document. Nothing moves forward until you approve.

2. /writing-plans: Breaks the approved design into tasks of two to five minutes each. Every task has exact file paths and verification steps. Creates native Claude Code tasks with dependency tracking: task two cannot start until task one is complete.

3. /executing-plans or subagent-driven-development: Executes the plan in batches with human checkpoints, or dispatches a fresh subagent per task with two-stage review.

4. /requesting-code-review: Reviews against the plan, reports issues by severity. Critical issues block progress.

5. /finishing-a-development-branch: Verifies tests, presents merge and PR options, cleans up the worktree.

The brainstorming skill

This is the one to start with. Before any non-trivial feature:

```
/superpowers-extended-cc:brainstorming

I want to add [describe feature]. Here is the context: [brief description of codebase and constraints].
```

Claude will ask clarifying questions: architecture choices, edge cases, integration points. You answer. It proposes a design section by section. You validate each section before it proceeds. Every decision is explicit before implementation begins.

Visual design output during brainstorming

One of the most useful things the brainstorming skill does is generate visual output autonomously when it decides visuals will communicate better than prose. It might render an HTML page showing two or three design options for a component, or a diagram of data flow, and ask you to pick an approach before moving on.

You don't trigger this manually. The skill decides when a visual would be clearer than text, generates it, and presents it as part of the design conversation. It's most common when the feature involves layout, hierarchy, or state transitions where prose descriptions get ambiguous quickly.

Recommended configuration

Add to .claude/settings.json to prevent Claude Code's auto-plan mode conflicting with the Superpowers workflow:

```
{ "permissions": { "deny": ["EnterPlanMode"] } }
```

SECTION 7

Context7: Stop Claude Hallucinating Your Library's API

Claude Code's training data has a cutoff. PrimeNG, AgGrid, Angular, Ionic: all release frequently. The version you're running today may differ significantly from the version Claude trained on. Claude doesn't know that. It will write code that looks correct and isn't: suggesting deprecated methods, APIs that moved, or patterns that changed in a major version.

Context7 fixes this by fetching current, version-specific documentation for any library and injecting it into Claude's context before it writes code. Instead of guessing from stale training data, Claude reads the actual current docs.

Installation

```
npx -y @upstash/context7-mcp
```

Or add to your MCP config:

```
"context7": { "command": "npx", "args": ["-y", "@upstash/context7-mcp"] }
```

How to use it

Add "use context7" to any prompt where library-specific APIs matter:

> "use context7: Implement a server-side row model in AgGrid Enterprise 32 with infinite scrolling and custom filters."

> "use context7: What is the correct way to lazy-load a standalone component in Angular 21?"

> "use context7: Show me how to use PrimeNG 21's ConfirmationService with a custom dialog component."

When it matters most

Use Context7 when working with libraries that have frequent breaking changes, features you know changed recently, any API you're not 100 percent certain about, or third-party libraries with smaller communities and less training data coverage.

Add it to your CLAUDE.md

```
## External Documentation
When working with [your libraries], always use context7 to fetch
current API documentation before suggesting implementation approaches.
```

SECTION 8

Automating Dev Admin

The same agentic capabilities that make Claude Code useful for building features make it equally useful for the administrative work that surrounds development: the parts that eat 30 to 60 minutes of a developer's day without producing a line of code.

Jira digest

The cleanest way to integrate Jira is via the Atlassian MCP. Connect it once through your Claude settings and you never touch credentials again: Claude queries your Jira directly from any session.

Once connected, your morning briefing is a single prompt:

```
"What Jira tickets are assigned to me? Highlight anything blocked or high priority, and flag anything updated recently that looks relevant to my work."
```

```
"Give me standup prep from my Jira tickets: what I did, what I'm doing next, any blockers. 60 seconds spoken."
```

No browser, no filter navigation, no context switching.

Jira with multiple accounts

If you work across multiple clients with separate Atlassian accounts: a common situation for contractors: the MCP only authenticates with one account at a time. The workaround is a small jira-digest.js script per project, using a .env file with that client's credentials. Each project folder has its own script and its own .env. Claude Code runs the script and reads the JSON output to produce the same briefing.

```
"Run jira-digest.js and give me a briefing on my tickets. Highlight anything blocked or high priority."
```

Add .env to your .gitignore so credentials never reach version control.

PR descriptions

Before pushing any branch:

```
"Read the git diff for this branch and write a PR description. Format: what changed, why, how to test. Keep it under 200 words."
```

Takes five seconds. Saves ten minutes. The description is accurate because Claude read the diff, not your memory of what you changed.

Weekly client summary

```
"Read git log --since='1 week ago' and the Jira tickets I closed this week. Draft a brief status update: what was delivered, what's in progress, any blockers to flag."
```

Useful before a weekly standup. Produces something you'd otherwise spend twenty minutes writing from scratch.

SECTION 9

Custom Commands: Shorthand for the Prompts You Use Daily

By now you've noticed something: the prompts in this guide can be long. Typing these out every day adds friction to the workflows that are supposed to remove friction.

Claude Code's solution is custom slash commands. A small markdown file becomes a slash command you can run with two keystrokes. The prompt template lives in the file. You type the shortcut and Claude does the rest.

How it works

Custom commands live in one of two places. .claude/commands/ is committed to the project and shared with your team. ~/.claude/commands/ holds your personal commands, available across every project.

Each file in these folders becomes a command. The filename without the .md extension is the command name. The content of the file is the prompt that runs when you invoke it.

A minimal example

Create ~/.claude/commands/progress.md with this content:

```
Update PROGRESS.md in the project root following the structure defined
in CLAUDE.md. If PROGRESS.md doesn't exist yet, create it.
```

Now in any Claude Code session, type /progress and the command runs. That's it.

A starter pack worth having

Six commands worth having from day one: /progress, /resume, /pr, /review, /standup, and /plan. Each is a single markdown file with a short prompt.

The full text for all six is in Appendix D at the end of this guide, ready to copy and drop into ~/.claude/commands/.

Passing arguments

Commands accept arguments via $ARGUMENTS. So /plan add a notification widget injects that phrase into the prompt template.

Project-specific vs personal

Project commands (.claude/commands/) suit prompts tied to one codebase. Personal commands (~/.claude/commands/) work across any project. Start there.

> Custom commands codify your workflow. Once /progress and /resume exist, you'll actually use them. Friction is the enemy of habit. Removing it is how good practices become defaults.

SECTION 10

What NOT to Do

No guide is complete without the failure modes. These are the most common ones, in order of how much time they waste.

One massive session for everything

Long sessions with mixed concerns: building, debugging, reviewing, asking questions: degrade context and produce inconsistent output. Keep sessions focused. One job per session.

Skipping plan mode for "simple" features

Most features that seem simple aren't. The ones that don't get a plan are usually the ones that require the most rework. Default to planning unless the change is genuinely trivial.

Ignoring CLAUDE.md drift

Your CLAUDE.md is only useful if it reflects reality. If Claude keeps making the same mistake and you keep correcting it manually instead of adding it as a rule, you're paying the same tax every session.

Using /compact instead of saving state

/compact is convenient but lossy. You don't control what survives. Use PROGRESS.md for anything you'd be upset to lose.

Running migrations without a backup stash

Before any bulk migration, create a recoverable point: a git stash reference or a clean commit you can roll back to. When multiple files need refactoring, Claude Code often writes a bulk script, and in my experience that seldom works perfectly the first time. Always have somewhere safe to return to.

Not installing Context7

If you work with any modern JavaScript framework or UI library, you might be getting wrong answers without it. The install takes two minutes.

Retyping the same prompts every day

If you find yourself writing the same prompt three days in a row, it should be a custom command. Friction kills habits.

Treating Claude Code as a code generator

The developers getting the most value are using Claude Code to make decisions, not just produce syntax. Ask it to reason, propose, review, and critique: not just write.

(c) 2026 — Working with Claude Code

Closing

These workflows are current as of mid-2026. Claude Code is evolving rapidly: new commands, new features, new model versions. Verify specific commands and configuration syntax against the latest documentation at code.claude.com before assuming anything here is still exact.

The principles will outlast the syntax. The discipline of structured planning, focused sessions, controlled context, and a living CLAUDE.md isn't going anywhere. The exact way you invoke them might.

If something in this guide stops working, the underlying problem it solves still exists. Adapt the workflow, keep the principle.

> Do you have workflows of your own that belong here? Commands you've built, patterns that changed how you work, things that saved you hours? I'd genuinely like to hear them. Find me at www.booleanstate.com or drop a message — the best ones may end up in a future edition.

Appendix: Copy-paste templates

A. Full sample CLAUDE.md

```
# CLAUDE.md

## Core Principles

- SIMPLICITY: Make every change as simple as possible.
Minimal code. If you can delete lines instead of adding, do that.

- ROOT CAUSES: Find root causes. No temporary fixes. No band-aids.
Hold yourself to senior developer standards.

- MINIMAL FOOTPRINT: Only touch what's necessary.
No side effects. No introducing new bugs while fixing old ones.

## Working Style

- Be direct and concise. No fluff, no unnecessary caveats.
- Always write down assumptions and ask clarifying questions upfront.
- Call out if something seems off: honest pushback preferred.
- Do not over-explain things the developer clearly already knows.

## Project Overview

[One paragraph: what the app does, who it serves, key business context]

## Tech Stack

- Framework: [e.g. Angular 21.x]
- UI: [e.g. PrimeNG 21.x]
- Grid: [e.g. AgGrid Enterprise 31.x]
- State: [e.g. RxJS 7.x, service-based, observable-driven]
- Build: [e.g. Angular CLI]

## Development Commands

npm start        # dev server
npm run build    # production build
npm test         # unit tests

## Architecture

[Folder structure overview, key modules, naming conventions]

## Coding Rules

[Project-specific rules: what to always do, never do, patterns to follow]
```

A. Full sample CLAUDE.md (continued)

```
## External Documentation

When working with [your libraries], always use context7 to fetch
current API documentation before suggesting implementation approaches.

## Session Continuity (PROGRESS.md)

When asked to update PROGRESS.md, write it to the project root
with this structure:

### Status
[Current state of the task: what's done, what's in progress]

### Decisions made
[Decisions taken this session, with brief reasoning]

### Files touched
[Path: brief note on what changed there]

### Next step
[Exact next action, specific enough for a fresh session to continue]

Rules:
- Create PROGRESS.md at the start of any non-trivial task
- Update at natural break points or when context approaches 60-80%
- At session start, always check for and read PROGRESS.md first
- Delete it when the task is done and merged

## Safety Protocols

- Before any bulk migration, create a recoverable point
- Verify with git diff after multi-file changes
- Never assume a migration worked: always verify
```

B. Plan mode prompts

```
# Start
"Give me a plan for [feature]. Do not write any code or touch any files yet."

# Iterate
"Address all my notes. Do not implement yet."

# Execute
"The plan looks good. Implement it now."
```

C. Jira and dev admin prompts

```
# Morning digest
"What's on my Jira today? Assigned tickets, anything high priority,
anything recently updated worth watching."

# Standup prep
"Give me standup prep from my Jira: what I did, what I'm doing next,
any blockers. 60 seconds spoken."

# PR description
"Read the git diff for this branch and write a PR description.
What changed, why, how to test. Under 200 words."

# Weekly summary
"Read git log --since='1 week ago' and the Jira tickets I closed this week.
Draft a brief status update: what was delivered, what's in progress,
any blockers to flag."
```

D. Custom commands starter pack

Drop these files in ~/.claude/commands/

```
# progress.md
Update PROGRESS.md in the project root following the structure defined in CLAUDE.md.
If PROGRESS.md doesn't exist yet, create it.

# resume.md
Read PROGRESS.md in the project root and continue where we left off.
Confirm what you understood the next step to be before doing anything.

# pr.md
Read the git diff for the current branch and write a PR description.
Format: what changed, why, how to test. Keep it under 200 words.

# review.md
Review the changes in git diff against the rules in CLAUDE.md.
List any violations only: do not fix anything. Be concise.

# standup.md
Read my recent git activity (last 7 days) and any Jira tickets I closed
or worked on. Produce standup prep in three bullets: what I did,
what I'm doing next, any blockers. 60 seconds spoken.

# plan.md
Give me a plan for: $ARGUMENTS
Do not write any code or touch any files yet.
Ask clarifying questions if anything is ambiguous.
```