How I work with Claude Code

2026-06-10

My practical workflow for using Claude Code: project-level rules, persistent memory across sessions, plan mode, and what needs scaffolding to work well.

Since 2025, I have been using AI tools as part of my daily work. Not as a replacement for thinking — as a productivity aid. An AI coding agent has become a regular part of how I work on infrastructure, blog posts, and software projects. Most of what follows likely applies to other agents as well — Cursor, Copilot, or whatever comes next. I happen to use Claude.

This post is not a tutorial and not a pitch. It is a reflection on what actually works — for me, on my projects — across months of real use, across 40+ projects, and across months of agent use.

The VS Code Setup

I use Claude Code through the official VS Code extension, which embeds the full CLI directly into the editor. Alongside it, Continue covers the inline layer — quick completions and context-aware chat while editing. The two complement each other: Claude Code for multi-step agentic work, Continue for the moment-to-moment editing flow.

`CLAUDE.md` — Rules the Model Must Follow

The single most impactful thing I do is maintain a `CLAUDE.md` file at the root of every project. This file defines hard rules that Claude must follow throughout a session.

Without it, every session starts from scratch. Claude makes decisions inconsistently, repeats mistakes, and occasionally suggests patterns that contradict earlier choices. With it, the model has a clear contract to follow from the first message.

The content varies completely by project — a Rust networking library looks nothing like a Kubernetes-based SaaS backend. But the *categories* are consistent across all of them:

Session onboarding. A mandatory sequence at the start of every session — read memory files, check git log, map the directory structure, confirm the current goal before touching anything.

Verification steps. What must pass after changes: build commands, type checks, smoke tests. Without this, Claude considers a task done when the file is written. With it, the task is done when the build is green.

Language and library rules. Decisions already made that should not be revisited. Which UUID library, which HTTP client, which date format. Prevents Claude from suggesting an alternative on every session.

Infrastructure and cloud gotchas. Quirks that took time to discover: provider-specific API constraints, timeouts that need raising, configuration patterns that look correct but are not. The kind of thing that only appears in a post-mortem or a support ticket.

Security constraints. What must never appear in commits. Encrypted files that must stay encrypted, identifiers that must not leak into public repositories.

A concrete example of the onboarding block, which I use in every project:

```
## Session onboarding
1. Read memory files from .claude/projects/*/memory/
2. Run `git log --oneline -20` and `git branch -a`
3. Map the directory structure
4. State your understanding of the current task before making any changes
```

The `CLAUDE.md` does not need to be long. Two hundred lines covering the project's key decisions, conventions, and gotchas is enough to make the difference between a session that builds on prior work and one that reinvents it.

The full list of files Claude Code reads — including `CLAUDE.local.md`, path-scoped `rules/`, and memory files — is documented at code.claude.com/docs/en/memory.

Memory — Context Across Sessions

Claude Code supports a file-based memory system at `~/.claude/projects/<project>/memory/`. Each memory file has a YAML frontmatter with a type — `user`, `feedback`, `project`, or `reference` — and a short description. A `MEMORY.md` index file is loaded automatically at session start.

The types I use most:

- **feedback** — things I do not want to explain again. Examples: "do not use ~ for approximation in Markdown tables — it renders as strikethrough; use ≈ instead" or "never say *empfohlen* (recommended) in blog posts — circumstances change."
- **project** — current decisions that evolve. Version numbers, known broken dependencies, ongoing work, architectural choices still in flux.
- **reference** — where to find things. Which repo holds the example code, which dashboard is the one oncall watches, which Linear project tracks the bugs.

The memory is not magic. It is a structured set of markdown files that Claude reads on startup. The value comes from maintaining it: updating stale entries, adding new learnings after a session, keeping the index honest.

My largest project has accumulated a 102 KB open-items log across 70+ sessions. Claude reads it at the start of each session. Progress is visible. Decisions are traceable. That file has saved more time than any individual prompt optimization.

⚠ Agents forget — and need to be reminded

> Memory files reduce the problem, but do not eliminate it. Claude and other AI agents will occasionally ignore documented rules, revert to default behavior, or lose track of a decision made earlier in the same session. This is not a configuration problem — it is an inherent property of how these models work.
>
> The practical response: when something goes wrong in a repeated way, add it to memory or `CLAUDE.md`. And when an agent produces something that contradicts a known rule, correct it explicitly rather than assuming it will not happen again.

Plan Mode — Think Before You Build

For any non-trivial change, I use plan mode before any files are touched. Claude explores the codebase, drafts an approach, and I review it before implementation starts.

What this buys:

- Forces articulating the approach before writing code
- Creates a natural review checkpoint for multi-file changes
- Prevents the "just do it quickly" trap that tends to miss edge cases

A concrete example: splitting a 490-line blog post into two separate posts required updating series navigation in four other files, correcting version numbers in one place, and deleting the original file. The plan laid all of this out explicitly. Without it, at least two of those cross-links would have been missed.

The discipline is: if the change touches more than two or three files, or involves a decision with real consequences, write the plan first.

Plan mode also creates space to push back. Claude's proposals are not always correct, complete, or the right approach for the specific context. Reading the plan before approving it is not a formality — it is the point. Questions worth asking before saying yes: Does this approach match how the rest of the codebase works? Is there a simpler path? Does the plan account for the edge case I already know about?

The plan is a starting point for a conversation, not a deliverable to accept or reject in full. Adjusting the approach at plan stage costs nothing. Adjusting it after implementation costs considerably more. Just like in real life. :smile:

What I Use It For

Area	How Claude helps
Blog series	Structure, SEO descriptions, consistency across 6+ posts
Terraform / Ansible	Recognizing patterns, keeping conventions consistent
Refactoring	Enforcing file-size limits, finding duplicated code
Learning new things	Rust project setup, exploring unfamiliar APIs
Taxonomy cleanup	Making 15+ posts consistent in one session
Incident analysis	Following evidence across logs, configs, and code simultaneously

Cost and Token Awareness

Claude Code can run either on a Claude.ai subscription or directly via the Anthropic API with per-token billing. The subscription plans include a usage allowance; the API bills by input and output tokens, and the numbers add up quickly in intensive sessions.

A few things that drive token consumption fast:

- **Large context windows.** Reading a 400-line file, a 100 KB memory file, and several conversation turns can push a single session into the millions of tokens. Claude Code compresses older context automatically when the window fills — but that compression itself costs tokens.
- **High effort level.** The `effortLevel` setting in `settings.json` controls how much exploratory work Claude does per request. I run at `high`, which produces better results but uses noticeably more tokens than `low` or `medium`.
- **Model selection.** Opus costs roughly five times more per token than Sonnet. I use Sonnet as my default. For tasks where output quality is the bottleneck, I switch to Opus explicitly. For straightforward edits, Haiku is fast and cheap.
- **Multi-agent workflows.** The `Workflow` tool fans out to parallel subagents, each with their own context. A workflow with 20 agents can consume as many tokens as 20 separate sessions. That power is real, but the cost is proportional.

Tracking usage. The `/cost` command in Claude Code shows the token cost of the current session. The `/usage` command shows aggregate usage over time. I check `/cost` at the end of long sessions — especially after large refactoring passes or multi-file blog edits.

What helps keep costs reasonable:

- Plan mode reduces wasted work. A 10-minute planning pass that catches a wrong approach is cheaper than implementing the wrong thing and correcting it.
- Memory reduces re-explaining. Facts stored in memory files do not need to be re-stated every session — Claude loads them automatically, which is far cheaper than repeating context in the prompt.
- Context hygiene matters. `/compact` at natural session boundaries keeps the context window from filling with stale tool output. Starting a fresh session for a new topic is often more efficient than continuing an old one.

The cost model rewards exactly the same behaviors that make the tool work well: clear upfront context, structured memory, and not redoing things from scratch every session.

A rough sense of scale: a focused Sonnet session for a targeted task typically runs well under a dollar. A long session with plan mode, several exploration passes, and multi-file edits can reach a few dollars. A heavy multi-agent workflow with 20+ parallel agents across a large codebase can run into the tens of dollars. The `/cost` command makes this visible in real time.

What Requires Care

A few things that need deliberate attention:

JavaScript-heavy pages are not readable by the agent. WebFetch fetches the raw HTML — if a page renders its content client-side, the agent sees almost nothing useful. The workaround: print

the page as a PDF or take a screenshot — Claude can read both file types and extract the relevant information from either.

The diff still needs to be read. Claude describes what it *intended* to do, not necessarily what it did. The tool summary and the actual change are not always identical. Reading the diff before moving on is not optional.

Memory is a point-in-time snapshot. A memory file that names a specific function, flag, or file path was accurate when it was written. Before acting on something recalled from memory, verify it still exists in the current code.

> ⚠ **You are responsible for the work — not the agent**
>
> The agent proposes, executes, and explains. You decide, review, and own the result. If something goes wrong in production, the agent is not accountable — you are. Treat every output as a draft that requires your judgment before it becomes reality.

When I Do Not Use Claude Code

Not every problem benefits from an agent. A few situations where I reach for something else instead:

- **One-line fixes and obvious typos.** Just type them.
- **Repetitive edits I know by heart.** A shell one-liner or find-replace is faster and less error-prone than explaining the pattern to an agent.
- **Production incidents where speed matters.** When something is down and I know where to look, I need to act, not explore. An agent that reads twelve files before suggesting anything is the wrong tool for that moment.
- **Infrastructure changes with no rollback path.** Anything irreversible — dropping a database, removing a network policy, modifying production IAM — requires full understanding before touching it. Delegation is not appropriate there.
- **Tasks where the problem is already fully understood.** If I know exactly what needs to change and why, adding plan mode and exploration around a five-minute job creates overhead, not value.

The pattern: Claude Code is useful when the problem involves *exploration* — finding where something is, understanding how pieces fit together, or catching what I would miss working linearly. When the path is already clear, it is faster to walk it directly.

Take Time for Setup — the Benefits Follow

Claude Code works well — but not without setup. The investment in `CLAUDE.md` files, a maintained memory structure, and plan-mode discipline is not overhead. It is the actual value.

These three elements together make it possible to work consistently across many sessions and many projects without losing context. The model brings capability; the scaffolding brings continuity.

That combination is what makes the tool genuinely useful for serious work.

After months of agent use, I no longer think of Claude Code as a chatbot. I think of it as another tool in the toolbox — useful when given structure, unreliable when left without it.

My Learnings

Things I wish I had known earlier:

- **Save early, save often.** Treat session context like unsaved work in an editor. If something important was discussed or decided, write it to memory or a note before the session grows long.
- **Compact manually, do not rely on auto-compaction.** Auto-compaction can trigger at an inconvenient moment and silently drop intermediate state. Running `/compact` deliberately at a natural pause point keeps you in control of what gets summarized.
- **Create a `/save` skill.** A custom skill that saves session state to memory on demand is worth the five minutes it takes to set up. I call it at the end of every significant session.
- **Ask Claude to track progress in todo tasks.** TodoWrite keeps intermediate steps visible across a long session. If compaction happens mid-task, the todo list survives and provides a recovery point.
- **One session, one topic.** Mixing unrelated tasks in a single session inflates context fast and increases the chance that earlier decisions get lost. Start a new session when switching to something unrelated.
- **After compaction, verify before continuing.** Compaction summarizes context — it does not preserve it verbatim. A quick check of the key facts before continuing is faster than recovering from a wrong assumption later.
- **Stale memory is worse than no memory.** A memory file that confidently names a function or path that no longer exists misleads more than silence would. Review memory files periodically and remove or update entries that have drifted from reality.
- **Short, specific prompts over long vague ones.** “Fix the null check in function X on line 42” gets a better result than “improve the error handling.” The narrower the scope, the less room for the model to go in an unexpected direction.