How I Use Every Claude Code Feature

A brain dump of all the ways I've been using Claude Code.

SHRIVU SHANKAR

NOV 02, 2025

I use Claude Code. A lot.

As a hobbyist, I run it in a VM several times a week on side projects, ofte with `--dangerously-skip-permissions` to vibe code whatever idea is o my mind. Professionally, part of my team builds the AI-IDE rules and tooli for our engineering team that consumes *several billion tokens per month* just for codegen.

The CLI agent space is getting crowded and between Claude Code, Gemi CLI, Cursor, and Codex CLI, it feels like the real race is between Anthropic and OpenAI. But TBH when I talk to other developers, their choice often comes down to what feels like superficials—a “lucky” feature implementation or a system prompt “vibe” they just prefer. At this point these tools are all pretty good. I also feel like folks often also over index o the output style or UI. Like to me the “you’re absolutely right!” sycophan isn’t a notable bug; it’s a signal that you’re too in-the-loop. Generally my goal is to “shoot and forget”—to delegate, set the context, and let it work Judging the tool by the final PR and not how it gets there.

Having stuck to Claude Code for the last few months, this post is my set reflections on Claude Code's entire ecosystem. We'll cover nearly every feature I use (and, just as importantly, the ones I don't), from the foundational `CLAUDE.md` file and custom slash commands to the powerful world of Subagents, Hooks, and GitHub Actions. **This post ended up a l long and I'd recommend it as more of a reference than somethin to read in entirety.**

CLAUDE.md

The single most important file in your codebase for using Claude Code effectively is the root `CLAUDE.md`. This file is the agent's "constitution," it primary source of truth for how your specific repository works.

. For my hobby projects, I l

UDE.md is strictly maintaine growing to 25KB).

30% (arbitrary) or more of c roduct or library specific

max token count for each selling "ad space" to teams ot ready for the CLAUDE.md

ated philosophy for writing effective `CLAUDE.md`.

1. **Start with Guardrails, Not a Manual.** Your `CLAUDE.md` should start small, documenting based on what Claude is getting wrong.
2. **Don't `@`-File Docs.** If you have extensive documentation elsewhere, it's tempting to `@`-mention those files in your `CLAUDE.md`. This bloats t context window by embedding the entire file on every run. But if you just *mention* the path, Claude will often ignore it. You have to *pitch* th agent on *why* and *when* to read the file. "For complex … usage or if y encounter a `FooBarError`, see `path/to/docs.md` for advanced troubleshooting steps."
3. **Don't Just Say "Never."** Avoid negative-only constraints like "Neve use the `--foo-bar` flag." The agent will get stuck when it thinks it *mu* use that flag. Always provide an alternative.
4. **Use `CLAUDE.md` as a Forcing Function.** If your CLI commands are complex and verbose, don't write paragraphs of documentation to explain them. That's patching a human problem. Instead, write a sim bash wrapper with a clear, intuitive API and document *that*. Keeping your `CLAUDE.md` as short as possible is a fantastic forcing function for simplifying your codebase and internal tooling.

Here's a simplified snapshot:

```
# Monorepo

## Python
- Always ...
- Test with <command>
... 10 more ...

## <Internal CLI Tool>
... 10 bullets, focused on the 80% of use cases ...
- <usage example>
- Always ...
```

```
- Never <x>, prefer <Y>

For <complex usage> or <error> see path/to/<tool>_docs.md

...
```

Finally, we keep this file synced with an `AGENTS.md` file to maintain compatibility with other AI IDEs that our engineers might be using.

If you are looking for more tips for writing markdown for coding agents se "AI Can't Read Your Docs", "AI-powered Software Engineering", and "How Cursor (AI IDE) Works".

The Takeaway: Treat your `CLAUDE.md` as a high-level, curated set of guardrails and pointers. Use it to guide where you need to invest in more (and human) friendly tools, rather than trying to make it a comprehensiv manual.

Compact, Context, & Clear

I recommend running `/context` mid coding session at least once to understand how you are using your 200k token context window (even wit Sonnet-1M, I don't trust that the full context window is actually used effectively). For us a fresh session in our monorepo costs a baseline ~20l tokens (10%) with the remaining 180k for making your change — which c fill up quite fast.

A screenshot of **/context** in one of my recent side projects. You can almost think of this like disk space that fills up as you work on a feature. After a few minutes or hours you'll need to clear the messages (purple) to make space to continue.

I have three main workflows:

- `/compact` **(Avoid):** I avoid this as much as possible. The automatic compaction is opaque, error-prone, and not well-optimized.
- `/clear` **+** `/catchup` **(Simple Restart):** My default reboot. I `/clear` t state, then run a custom `/catchup` command to make Claude read al changed files in my git branch.
- "Document & Clear" **(Complex Restart):** For large tasks. I have Claude dump its plan and progress into a `.md`, `/clear` the state, then start a new session by telling it to read the `.md` and continue.

The Takeaway: Don't trust auto-compaction. Use `/clear` for simple reboots and the "Document & Clear" method to create durable, external

“memory” for complex tasks.

Custom Slash Commands

I think of slash commands as simple shortcuts for frequently used promp nothing more. My setup is minimal:

- `/catchup`: The command I mentioned earlier. It just prompts Claude t read all changed files in my current git branch.
- `/pr`: A simple helper to clean up my code, stage it, and prepare a pul request.

IMHO if you have a long list of complex, custom slash commands, you’ve created an anti-pattern. To me the entire point of an agent like Claude is that you can type *almost* whatever you want and get a useful, mergable result. The moment you force an engineer (or non-engineer) to learn a ne documented-somewhere list of essential magic commands just to get wo done, you’ve failed.

The Takeaway: Use slash commands as simple, personal shortcuts, not a replacement for building a more intuitive `CLAUDE.md` and better-tooled agent.

Custom Subagents

On paper, custom subagents are Claude Code’s most powerful feature fo context management. The pitch is simple: a complex task requires X toke of input context (e.g., how to run tests), accumulates Y tokens of working context, and produces a Z token answer. Running N tasks means `(X + Y Z) * N` tokens in your main window.

The subagent solution is to farm out the `(X + Y) * N` work to specialized agents, which only return the final Z token answers, keeping your main context clean.

I find they are a powerful idea that, in practice, *custom* subagents create two new problems:

1. **They Gatekeep Context:** If I make a `PythonTests` subagent, I've no hidden all testing context from my *main* agent. It can no longer reasc holistically about a change. It's now forced to invoke the subagent jus to know how to validate its own code.
2. **They Force Human Workflows:** Worse, they force Claude into a rig human-defined workflow. I'm now dictating *how* it must delegate, wh is the very problem I'm trying to get the agent to solve for me.

My preferred alternative is to use Claude's built-in `Task(...)` feature to spawn clones of the *general* agent.

I put all my key context in the `CLAUDE.md`. Then, I let the *main agent* deci when and how to delegate work to copies of itself. This gives me all the context-saving benefits of subagents without the drawbacks. The agent manages its own orchestration dynamically.

In my "Building Multi-Agent Systems (Part 2)" post, I called this the "Mast Clone" architecture, and I strongly prefer it over the "Lead-Specialist" model that custom subagents encourage.

The Takeaway: Custom subagents are a brittle solution. Give your main agent the context (in `CLAUDE.md`) and let it use its own `Task/Explore(..` feature to manage delegation.

Resume, Continue, & History

On a simple level, I use `claude --resume` and `claude --continue` frequently. They're great for restarting a bugged terminal or quickly rebooting an older session. I'll often `claude --resume` a session from da ago just to ask the agent to summarize how it overcame a specific error, which I then use to improve our `CLAUDE.md` and internal tooling.

More in the weeds, Claude Code stores all session history in `~/.claude/projects/` to tap into the raw historical session data. I have scripts that run meta-analysis on these logs, looking for common exceptions, permission requests, and error patterns to help improve ager facing context.

The Takeaway: Use `claude --resume` and `claude --continue` to resta sessions and uncover buried historical context.

Hooks

Hooks are huge. I don't use them for hobby projects, but they are critical steering Claude in a complex enterprise repo. They are the deterministic "must-do" rules that complement the "should-do" suggestions in `CLAUDE.md`.

We use two types:

1. **Block-at-Submit Hooks:** This is our primary strategy. We have a `PreToolUse` hook that wraps any `Bash(git commit)` command. It checks for a `/tmp/agent-pre-commit-pass` file, which our test script *only* creates if all tests pass. If the file is missing, the hook blocks the commit, forcing Claude into a "test-and-fix" loop until the build is gre
2. **Hint Hooks:** These are simple, non-blocking hooks that provide "fire and-forget" feedback if the agent is doing something suboptimal.

We intentionally do not use “block-at-write” hooks (e.g., on `Edit` or `Writ` Blocking an agent mid-plan confuses or even “frustrates” it. It’s far more effective to let it finish its work and then check the final, completed resul at the commit stage.

The Takeaway: Use hooks to enforce state validation at commit time (`block-at-submit`). Avoid blocking at write time—let the agent finish its plan, then check the final result.

Planning Mode

Planning is essential for any “large” feature change with an AI IDE.

For my hobby projects, I exclusively use the built-in planning mode. It’s a way to align with Claude before it starts, defining both *how* to build something and the “inspection checkpoints” where it needs to stop and show me its work. Using this regularly builds a strong intuition for what minimal context is needed to get a good plan without Claude botching th implementation.

In our work monorepo, we’ve started rolling out a custom planning tool b on the Claude Code SDK. Its similar to native plan mode but heavily prompted to align its outputs with our existing technical design format. It also enforces our internal best practices—from code structure to data privacy and security—out of the box. This lets our engineers “vibe plan” a new feature as if they were a senior architect (or at least that’s the pitch)

The Takeaway: Always use the built-in planning mode for complex changes to align on a plan before the agent starts working.

Skills

I agree with Simon Willison's: **Skills are (maybe) a bigger deal than MCP.**

If you've been following my posts, you'll know I've drifted away from MCP for most dev workflows, preferring to build simple CLIs instead (as I argue in "AI Can't Read Your Docs"). My mental model for agent autonomy has evolved into three stages:

1. **Single Prompt:** Giving the agent all context in one massive prompt. (Brittle, doesn't scale).
2. **Tool Calling:** The "classic" agent model. We hand-craft tools and abstract away reality for the agent. (Better, but creates new abstractions and context bottlenecks).
3. **Scripting:** We give the agent access to the raw environment—binaries, scripts, and docs—and it writes code *on the fly* to interact with them.

With this model in mind, **Agent Skills** are the obvious next feature. They are the formal productization of the "Scripting" layer.

If, like me, you've already been favoring CLIs over MCP, you've been implicitly getting the benefit of Skills all along. The `SKILL.md` file is just a more organized, shareable, and discoverable way to document these CLIs and scripts and expose them to the agent.

The Takeaway: Skills are the right abstraction. They formalize the "scripting"-based agent model, which is more robust and flexible than the rigid, API-like model that MCP represents.

MCP (Model Context Protocol)

Skills don't mean MCP is dead (see also "Everything Wrong with MCP"). Previously, many built awful, context-heavy MCPs with dozens of tools th

just mirrored a REST API (`read_thing_a()`, `read_thing_b()`, `update_thing_c()`).

The “Scripting” model (now formalized by Skills) is better, but it needs a secure way to access the environment. This to me is the new, more focus role for MCP.

Instead of a bloated API, an MCP should be a simple, secure gateway that provides a few powerful, high-level tools:

- `download_raw_data(filters…)`
- `take_sensitive_gated_action(args…)`
- `execute_code_in_environment_with_state(code…)`

In this model, MCP’s job isn’t to abstract reality for the agent; its job is to manage the auth, networking, and security boundaries and then get out the way. It provides the *entry point* for the agent, which then uses its scripting and `markdown` context to do the actual work.

The only MCP I still use is for Playwright, which makes sense—it’s a complex, stateful environment. All my stateless tools (like Jira, AWS, GitHub) have been migrated to simple CLIs.

The Takeaway: Use MCPs that act as data gateways. Give the agent one or two high-level tools (like a raw data dump API) that it can then script against.

Claude Code SDK

Claude Code isn’t just an interactive CLI; it’s also a powerful SDK for building entirely new agents—for both coding and non-coding tasks. I’ve

started using it as my default agent framework over tools like LangChain/CrewAI for most new hobby projects.

I use it in three main ways:

1. **Massive Parallel Scripting:** For large-scale refactors, bug fixes, or migrations, I don't use the interactive chat. I write simple bash scripts that call `claude -p "in /pathA change all refs from foo to ba` in parallel. This is far more scalable and controllable than trying to ge the main agent to manage dozens of subagent tasks.
2. **Building Internal Chat Tools:** The SDK is perfect for wrapping complex processes in a simple chat interface for non-technical users. Like an installer that, on error, falls back to the Claude Code SDK to ju *fix* the problem for the user. Or an in-house "v0-at-home" tool that le our design team vibe-code mock frontends in our in-house UI framework, ensuring their ideas are high-fidelity and the code is more directly usable in frontend production code.
3. **Rapid Agent Prototyping:** This is my most common use. It's not ju for coding. If I have an idea for any agentic task (e.g., a "threat investigation agent" that uses custom CLIs or MCPs), I use the Claude Code SDK to quickly build and test the prototype before committing t full, deployed scaffolding.

The Takeaway: The Claude Code SDK is a powerful, general-purpose ag framework. Use it for batch-processing code, building internal tools, and rapidly prototyping new agents *before* you reach for more complex frameworks.

Claude Code GHA

The Claude Code GitHub Action (GHA) is probably one of my favorite and most slept on features. It's a simple concept: just run Claude Code in a GHA. But this simplicity is what makes it so powerful.

It's similar to Cursor's background agents or the Codex managed web UI but is far more customizable. You control the entire container and environment, giving you more access to data and, crucially, much strong sandboxing and audit controls than any other product provides. Plus, it supports all the advanced features like Hooks and MCP.

We've used it to build custom "PR-from-anywhere" tooling. Users can trigger a PR from Slack, Jira, or even a CloudWatch alert, and the GHA wil fix the bug or add the feature and return a fully tested PR [1].

Since the GHA logs are the full agent logs, we have an ops process to regularly review these logs at a company level for common mistakes, bas errors, or unaligned engineering practices. This creates a data-driven flywheel: Bugs -> Improved CLAUDE.md / CLIs -> Better Agent.

```
$ query-claude-gha-logs --since 5d | claude -p "see what the other
claudes were getting stuck on and fix it, then put up a PR"
```

The Takeaway: The GHA is the ultimate way to operationalize Claude Code. It turns it from a personal tool into a core, auditable, and self-improving part of your engineering system.

settings.json

Finally, I have a few specific `settings.json` configurations that I've foun essential for both hobby and professional work.

- `HTTPS_PROXY/HTTP_PROXY`: This is great for debugging. I'll use it to inspect the raw traffic to see exactly what prompts Claude is sending For background agents, it's also a powerful tool for fine-grained netw sandboxing.
- `MCP_TOOL_TIMEOUT/BASH_MAX_TIMEOUT_MS`: I bump these. I like runni long, complex commands, and the default timeouts are often too conservative. I'm honestly not sure if this is still needed now that bas background tasks are a thing, but I keep it just in case.
- `ANTHROPIC_API_KEY`: At work, we use our enterprise API keys (via apiKeyHelper). It shifts us from a "per-seat" license to "usage-based" pricing, which is a much better model for how we work.
  - It accounts for the *massive* variance in developer usage (We've se 1:100x differences between engineers).
  - It lets engineers to tinker with non-Claude-Code LLM scripts, all under our single enterprise account.
- "`permissions`": I'll occasionally self-audit the list of commands I've allowed Claude to auto-run.

The Takeaway: Your `settings.json` is a powerful place for advanced customization.

Conclusion

That was a lot, but hopefully, you find it useful. If you're not already usin CLI-based agent like Claude Code or Codex CLI, you probably should be. There are rarely good guides for these advanced features, so the only wa to learn is to dive in.

and support my work.

Type your email... Subscribe

1 To me, a fairly interesting philosophical question is how many reviewers shoul PR get that was generated directly from a customer request (no internal huma prompter)? We've settled on 2 human approvals for any AI-initiated PR for no but it is kind of a weird paradigm shift (for me at least) when it's no longer a human making something for another human to review.

Discussion about this post

Comments **Restacks**

Write a comment...

Josh Devon 20h

Liked by Shrivu Shankar

Great guide, just be careful with Skills! Here's how we hijacked a skill with an invisib prompt inject: https://open.substack.com/pub/securetrajectories/p/claude-skill-hijack invisible-sentence

LIKE (4) REPLY

Jacob Bumgarner 17h

Liked by Shrivu Shankar

Wonderful write up. thank you.

Can you expand on this part a bit?

> I write simple bash scripts that call claude -p "in /pathA change all refs from foo to in parallel.

How do you prevent the agents from overwriting the code each is writing? Switching

LIKE (1) REPLY

1 reply by Shrivu Shankar

3 more comments...