Claude Code's Source Code: What It Reveals and How to Use It Better

This guide breaks down the key insights from the Claude Code source code that became public, and translates them into practical habits that will help you use Claude Code like a top 1% user.

By: Nate Herk

How the Source Code Became Public

Anthropic published an npm package for Claude Code that included a `.map` file pointing to readable TypeScript source hosted on their servers. A security researcher found it, followed the trail, and the full source was mirrored publicly on GitHub within hours.

This is not a reverse-engineered approximation. The codebase includes roughly 1,900 files and over 512,000 lines of production code, covering the tool system, command system, memory system, MCP client/server support, IDE bridge, task system, coordinator mode, feature flags, and environment configuration.

Important note: Anthropic still owns the copyright on this code and has issued DMCA takedown notices. Everything covered here focuses on practical takeaways for how you can use Claude Code better, not the source code itself.

Insight #1: It's Not What You Think It Is

Most people think Claude Code is basically Claude but in your terminal, like a chatbot with access to your local files. That is completely wrong.

What the source code reveals is that Claude Code is a **full agent runtime**. It is a proper application built with Bun, TypeScript, and React. It has a tool system, a command system, a memory system, a permission engine, a task manager, a multi-agent coordinator, and an MCP client and server, all wired together under one execution pipeline.

The flow works like this: your input hits a CLI parser, goes to something called the Query Engine, calls the LLM API, runs a tool execution loop, and then renders results back in your terminal.

Why this matters: If you are using Claude Code like a chatbot (just typing questions and hoping for good answers), you are using maybe 10% of what it can actually do. The rest of the value is in the systems built around the model, and that is what the remaining insights are about.

Insight #2: The Command Surface You're Ignoring

The source code reveals roughly **85 slash commands** in Claude Code. Most users know maybe five of them. Power users treat these commands like shortcuts, and they are one of the biggest levers for getting more value out of the tool.

Here are the ones that matter most:

`/init` sets up your project context. It generates a `CLAUDE.md` file that acts as Claude Code's operating manual for your repository. More on why this file is critical in the next section.

`/plan` **and** **`/ultraplan`** put Claude Code into planning mode. Instead of immediately executing, it maps out the full approach first and asks you before it starts touching files. This is a big deal when you are working on something complex and you do not want it editing things everywhere. You will also save tokens in the long run.

`/compact` compresses your conversation history so you can keep the important context but drop the noise. If you are burning through tokens, this is one of the fastest fixes. You can also run `/compact` with a specific prompt, like telling it to keep all the important information about a particular integration while compressing everything else.

`/review` **and** **`/security-review`** are built-in code review workflows. Instead of asking Claude Code to "look at my code," you use these commands and it runs a structured review. The fact that these exist as dedicated commands tells you that review is a first-class workflow in this product, not an afterthought.

`/context` manages what files Claude Code is actually paying attention to. This matters because every file in context costs you tokens.

`/cost` shows you what you have actually spent. Most people have no idea how much a session is costing them until they look at the bill.

`/resume` **and** **`/summary`** let you pick up where you left off between sessions without having to re-explain everything.

The takeaway: Better prompts are one lever, but knowing these commands is a completely separate lever that most people are not even touching.

Insight #3: The Memory System Is Way More Important Than You Think

The source code reveals that Claude Code has a full memory system, and the center of it is a file called **`CLAUDE.md`**.

Most people either ignore this file or dump a bunch of random notes into it. That is a huge mistake because `CLAUDE.md` is not documentation. It is **operating context**.

Think of it this way: if Claude Code is an employee, then `CLAUDE.md` is their onboarding document. It tells them how you do things, what matters, what you never do, and how the project is structured.

The best users keep `CLAUDE.md` short, opinionated, and operational. Things like:

- "We use TypeScript strict mode. Always."
- "Tests go in test folders next to the source file."
- "Never modify the database schema without running migrations."
- "Use PNPM, not npm."

Decision rules, constraints, and conventions. Not a novel about your project's history.

The source also reveals multiple layers to this system. There is user-level memory, extracted memories, and even team memory synchronization. This means Claude Code has persistent memory mechanisms across project, user, and session contexts, and `CLAUDE.md` is one of the highest-leverage inputs shaping how it behaves in future sessions.

The takeaway: If you get nothing else from this guide, go update your `CLAUDE.md` file today, whether on the global level or the project level. Remember, these get injected every single session before every single chat.

Insight #4: Permissions Are Why Claude Code Feels Slow

If you have used Claude Code, you have hit this. You ask it to do something and it keeps asking: "Allow me to do this? Can I run this? Can I edit this?" Over and over.

Most people think the fix is better prompting. It is not. The fix is **permissions**.

The source code shows a deep permission system with multiple modes: default mode (where it asks you about everything), plan mode, and then bypass/auto modes where it can just execute.

Here is the real gem: you can set **wildcard permissions**. Things like "allow all git commands" or "allow all file edits in my src folder." The source confirms wildcard patterns like `Bash(git *)` and `FileEdit(/src/*)`.

Instead of Claude Code asking you 15 times whether it can run `git status` or edit a file or run a test, you set the rule once and it just works.

In your `settings.local.json` or `settings.json`, you can set global, user-level, or project-level permissions for things you always want to allow, always want to deny, or always want it to ask about.

The takeaway: For recurring workflows and stuff you do every day, this is one of the highest-ROI changes you can make. It lets you go from babysitting every action to actually letting Claude Code operate like an agent while you step away.

Insight #5: It's Built for Multi-Agent Work

The source code reveals a full coordinator subsystem, agent tools, team tools, and a task system designed for background and parallel work. The task system includes shell tasks, local agent tasks, remote agent tasks, teammate tasks, and even background task concepts.

In plain English, this means the architecture is clearly built to support **decomposition**, splitting work across multiple agents that can run in parallel. Think one agent exploring your codebase, another implementing changes, and another validating tests.

How much of this is fully user-facing today versus still rolling out is harder to say from the source alone. But the architecture makes the intent clear: Claude Code is designed to handle complex multi-step work by breaking it apart, not by cramming it all into one thread.

The takeaway: Structure your requests with decomposition in mind. Instead of one massive prompt like "refactor this entire module, update the tests, and fix the documentation," break it up. Think about it the same way you would think about workflows versus agents in tools like n8n. The best results come from breaking complex work into clear sequential or parallel steps rather than hoping one giant instruction gets everything right.

Insight #6: MCP, Plugins, and Skills Are the Real Extension Layer

If you have been following along with Model Context Protocol (MCP), here is what the source code confirms: MCP is not just supported by Claude Code, it is baked into the architecture. Claude Code is both an **MCP client** and an **MCP server**. It can connect to external tools through MCP, and other systems can connect to it.

But it goes beyond MCP. The source also reveals a skills and plugin layer. Power users can build repeatable workflows, custom capabilities, and domain-specific extensions that compound over time.

This is where Claude Code stops being just a coding tool and starts being an **integration layer**. You can connect it to databases, APIs, internal tools, documentation systems, or anything with an MCP server, and then layer skills and plugins on top for the stuff you do repeatedly.

The takeaway: The more systems you connect to Claude Code, the more useful it becomes. The power is not the tool itself. It is what you connect to it and the workflows you build around it.

Insight #7: There Are Features We Don't Have Access to Yet

The codebase includes checks for something called `USER_TYPE`, and one of the values is `ant` (meaning Anthropic). This tells us that certain capabilities are gated behind internal feature flags.

The source references things like voice mode, a system called Kairos, a daemon mode, and a coordinator mode, all behind flags that suggest they are either internal, experimental, or being rolled out gradually.

We cannot say for sure how different the internal experience actually is. Some of these could be early prototypes. Some might be close to shipping publicly. But what the source does confirm is that Claude Code is **heavily feature-flagged**, meaning different users may already be getting meaningfully different experiences depending on their environment, build, or rollout group.

The takeaway: Pay attention to Claude Code updates. Capabilities that are flagged or limited today are likely on the roadmap. When they land, the people who already understand the architecture will be ready to use them immediately.

Insight #8: It's More Than a Terminal App

The source contains a substantial bridge subsystem for IDE integration, with session management, permission routing, and JWT-authenticated communication. It also contains remote execution and device-handoff flows.

The takeaway: Claude Code's architecture assumes it can operate across terminal, IDE, remote, and multi-device environments. If your workflow spans editor plus terminal, that is aligned with the product's direction.

How to Actually Use All of This: The Top 1% Habits

The single biggest insight from this entire source code is this: top users do not just write better prompts. They **design a better operating environment** for Claude Code.

Here is what that looks like in practice:

1. Treat `CLAUDE.md` like a force multiplier. Keep it short. Keep it opinionated. Update it regularly. Route to other files when needed. This single file shapes every interaction you have with Claude Code.

2. Learn the command surface. You do not need all 85 commands, but `/plan`, `/compact`, `/context`, `/review`, `/cost`, and `/resume` will change your daily workflow immediately.

3. Configure permissions for your recurring workflows. Stop babysitting every action. Set wildcard rules for the stuff you do every day.

4. Think in terms of decomposition, not monolithic prompts. Break complex work into a search phase, a plan phase, an execute phase, and a verify phase. Let Claude Code use its architecture the way it was designed.

5. Manage your context like it's money, because it literally is. Use `/compact` when conversations get long. Use `/context` to control what is loaded. Use `/summary` and `/resume` to carry work across sessions. Every unnecessary file in context is tokens you are paying for. Context management is not a nice-to-have. It is a discipline.

6. Connect things to it. Whether that is MCP servers, CLIs, plugins, or skills, the more tools Claude Code can access, the more valuable it actually becomes.

7. Treat it like infrastructure, not an app. The source reveals a massive configuration surface: model routing, sub-agent model overrides, shell behavior, privacy controls, and the ability to route through different cloud backends. Most people never touch these settings, but if you are

using Claude Code seriously, there is real leverage in tuning the environment to fit your workflow.

Bonus Insights: What the Source Code Documentation Reveals Beyond the Video

The public repository includes detailed architecture docs, tool documentation, and configuration files that go deeper than what was covered in the video. Here are the most useful extras.

Prompt Like an Operator, Not a Chatbot

The source reveals that Claude Code is fundamentally **tool-first**. Each tool in the system has its own input schema, permission checks, execution logic, UI rendering, and concurrency safety. Tools are registered centrally and invoked inside the Query Engine's tool loop. The documented tool surface includes roughly **40 agent tools** covering file read/write/edit, grep/glob, shell execution, notebooks, REPLs, web tools, MCP, LSP, tasks, agents, plans, worktrees, and more.

What this means practically: your requests work best when they imply concrete actions like reading files, searching code, editing specific areas, running checks, or spawning sub-agents. The more action-oriented your prompt, the better Claude Code can leverage its tool system.

Additional Slash Commands Worth Knowing

Beyond the commands covered in the video, the source documents several more:

- **`/memory`** is a first-class command for managing Claude Code's persistent memory directly, not just what is in `CLAUDE.md` but the broader memory system across sessions.
- **`/permissions`** lets you view and configure your permission rules without leaving the session.
- **`/mcp`** manages your MCP server connections, letting you add, remove, or inspect connected tools.
- **`/tasks`** **and** **`/agents`** give you access to Claude Code's multi-agent and task management capabilities from within a session.

The source also reveals that commands are implemented as distinct types: prompt-driven commands (that involve the LLM), local text commands, and local JSX UI commands. This is not just a list of aliases. It is a full command architecture.

Plan Mode Is a Dedicated Product Feature

The source confirms there are dedicated tools for entering and exiting plan mode, meaning it is a deliberate product feature, not just a prompting trick. Use `/plan` or `/ultraplan` when you want Claude Code to map the work before touching files or running commands. Use normal execution mode when you already trust the workflow and want momentum.

MCP Goes Deeper Than You'd Expect

Under the hood, the MCP implementation is thorough. The tooling includes MCP tool invocation, resource listing and reading, auth handling, and deferred tool discovery (meaning Claude Code can find and load new tools from MCP servers on the fly). The repo even ships a dedicated MCP exploration server for studying its own source tree, which gives you a sense of how central this protocol is to the product.

The Specific Feature Flags in the Source

The specific flag names documented in the source include `PROACTIVE`, `KAIROS`, `VOICE_MODE`, `BRIDGE_MODE`, `DAEMON`, and `COORDINATOR_MODE`. The build system uses `bun:bundle` feature gating and dead-code elimination, and the service layer includes GrowthBook-backed analytics and experimentation support. This means Anthropic is actively running experiments and rolling out features to different user segments.

The Configuration Surface Most Users Never Touch

The `.env.example` file exposes settings for auth, model selection, cloud backend routing, shell behavior, output limits, idle thresholds, debugging, remote mode, privacy, and telemetry. Claude Code can route through direct Anthropic access, AWS Bedrock, Google Vertex AI, or Azure Foundry.

You can also configure **sub-agent model overrides**, meaning you can choose which model your sub-agents use independently from your primary model. This is a big deal for managing cost and performance across different types of tasks.

Context Hygiene as a Discipline

The best users are intentional about what files are in context, what memory is being injected, when to compact, and when to resume or summarize. Think of it as a named practice: **context hygiene**. It is not just about saving tokens. It is about making sure Claude Code has the right information to do its best work, and nothing extra cluttering the signal.

What Top 1% Users Likely Do That Most People Don't

Based on the full source documentation, the gap between average and advanced users comes down to these habits:

- They treat prompting as only one layer. They also deliberately use commands, permissions, memory, tasks, agents, and MCP integrations.
- They give `CLAUDE.md` specific conventions, constraints, testing expectations, architecture facts, and non-obvious team norms, not just a project summary.
- They use `/review` and `/security-review` as standard practice, not just when something breaks.
- They expand the system with MCP servers, plugins, and skills when they need new capabilities or repeatable workflows.
- They use the product more like a configurable environment than a chat window, taking advantage of model settings, sub-agent model routing, cloud backends, privacy controls, and shell/runtime options.

The Bottom Line

Claude Code works best when you treat it like a configurable execution environment for an AI agent. Its advantages come from structured tools, explicit commands, permission controls, persistent memory, MCP integrations, and multi-agent orchestration.

Users who learn those layers will get far more leverage than users who only write better prompts.
