Member-only story

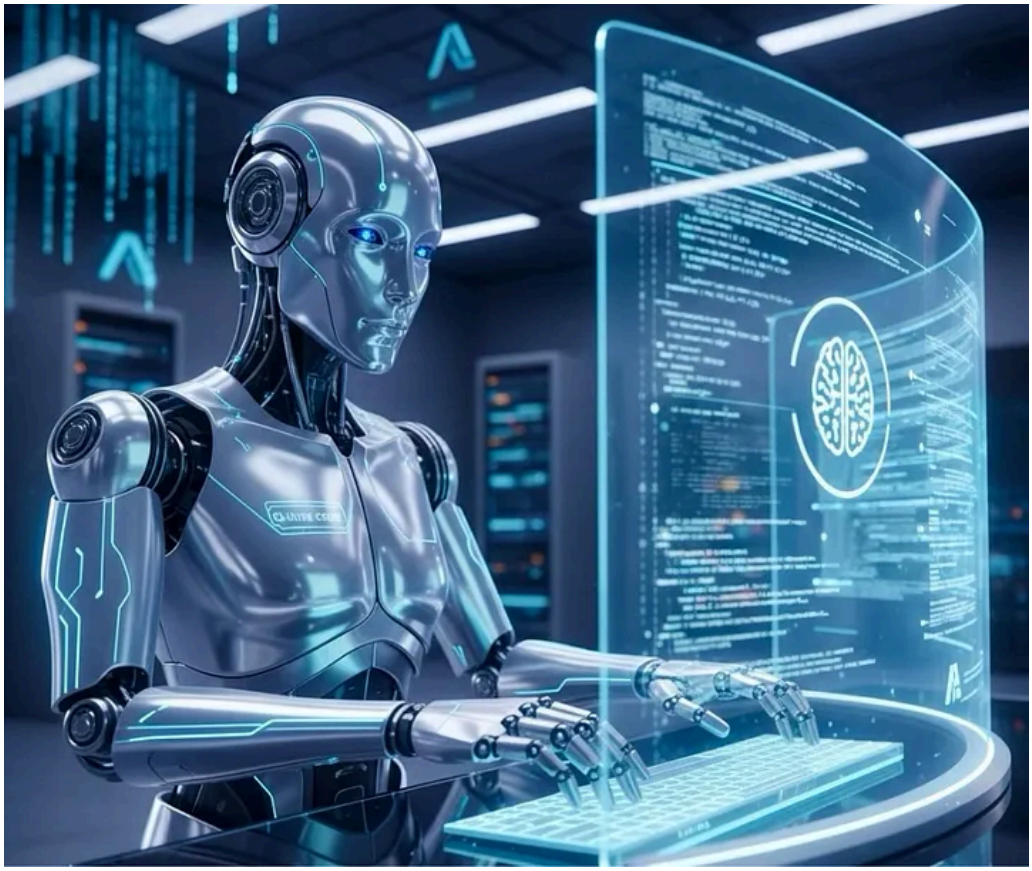
10 Game-Changing CLAUDE.md Entries That Turned My Claude Code Sessions into a Coding Superpower

Reza Rezvani Follow 22 min read · 1 day ago

TL;DR: Tired of wrestling with AI that writes spaghetti code? Meet CLAUDE.md – your project's persistent brain that teaches Claude how to code like a pro. Below are 10 killer CLAUDE.md prompts (with examples) that transformed my agentic AI coding workflow from all-nighters to autopilot.

A Nightmare in Node (and the CLAUDE.md Epiphany)

It's 2:13 AM on a Tuesday. I'm hunched over my keyboard, eyes bloodshot, debugging a Node.js API that's throwing cryptic 500s.



Claude Code at work . Shot by Google Nano Banana

I've got console.log statements littered everywhere, Postman open on one screen, Stack Overflow on another – living every developer's worst insomnia-fueled groundhog day.

The more I poke, the more the bug hydra grows new heads. By dawn, I'm questioning my career choices *(and downing my 5th coffee)*. Sound familiar? This was me before I discovered CLAUDE.md.

Fast-forward a week: I spin up a new side project (a tiny React + Express SaaS). This time, I start by crafting a CLAUDE.md file – Anthropic's secret weapon for agentic AI coding. The difference? Night and day.

My next project didn't spiral out of control; it deployed flawlessly in under 4 hours. I felt like I'd unlocked god mode for coding – as if chaotic, caffeinated

intern-Claude had morphed into a grizzled software architect overnight.

The secret sauce was feeding Claude a persistent set of rules and context before any code was generated. That's exactly what CLAUDE.md does: it's the *"persistent brain"* of your project, a Markdown file that auto-injects crucial knowledge and coding rules into every Claude prompt .

In other words, CLAUDE.md serves as Claude's long-term memory – loaded at the start of each session – covering everything from your tech stack and style guide to those quirky project-specific gotchas.

Why does this matter? *Because agentic AI thrives on context.* Give **Claude Code** the right context and it transforms into an indefatigable pair programmer. In fact, Anthropic's own devs found that teams with well-tuned CLAUDE.md files achieved 2 – 3× faster execution on tasks .

No more re-explaining your project's basics every session; CLAUDE.md keeps Claude perpetually in-the-know. It's like hiring a new dev who reads the entire wiki and codebase before writing a line of code. *(No wonder 80% of Anthropic's engineers use Claude Code daily now – they essentially gave it a brain transplant!)* And the best part?

Claude will even draft the initial CLAUDE.md for you with a single command – run */init* and watch it compile a quick project overview that you can refine .

So what exactly should go into this magic file? According to Anthropic's guidance *(and hard-won personal experience)*, CLAUDE.md is ideal for documenting *"common commands, core patterns, style guides, testing protocols, repo conventions, and any critical knowledge"* your AI buddy should remember .

The goal is to bake in those *"one weird trick"* insights that turn a mediocre code assist into your ultimate sidekick. Below, I've compiled the 10 game-changing CLAUDE.md entries that leveled up my Claude Code sessions.

Each entry comes with a real-world scenario, a copy-paste-ready Markdown snippet, and a pro tip for tweaking it to your needs. Steal these snippets *(I won't tell)*, drop them into your CLAUDE.md, and watch Claude go from good to un-freaking-stoppable at agentic AI coding. Let's dive in!

1. Architecture Blueprint

Give Claude the 10,000-foot view of your app's structure and tech stack upfront. When I first ran */init* in Claude Code, it auto-generated an Architecture section that read like an architect's cheatsheet.

This entry became the bedrock of my CLAUDE.md. By outlining the high-level design – frameworks, layers, and how everything connects – I ensured Claude always codes with the grand plan in mind. No more randomly mixing MVC patterns or misplacing files; the Architecture Blueprint makes Claude aware of the "big picture" from the get-go.

Real-world example: I was building a personal finance tracker. Initially, Claude kept intermixing frontend logic with backend code (yikes).

Once I defined the architecture in CLAUDE.md – e.g. React SPA frontend, Express API backend, SQLite database – Claude strictly followed that separation. It even knew where to place new modules without me telling it. Here's a condensed snippet inspired by that project's CLAUDE.md:

The Core Architecture

This project is a full-stack Personal Finance Tracker with a React frontend and Node.js/Express backend:

- **Frontend:** React 19 single-page app (Vite) with a component-based UI.
- **Backend:** Express.js REST API with a SQLite database.
- **Database:** SQLite3 with tables for `transactions`, `categories`, `savings_goals`.
- **Communication:** Frontend calls backend at `http://localhost:3001/api/`.
- Design Pattern: Backend follows MVC – routes for API endpoints, models for DB operations.

This *"blueprint"* entry tells Claude exactly how our system is laid out and enforces key patterns (like MVC).

Immediately, my AI pair-programmer stopped proposing code that didn't fit the mold. Claude now sees the forest, not just the trees.

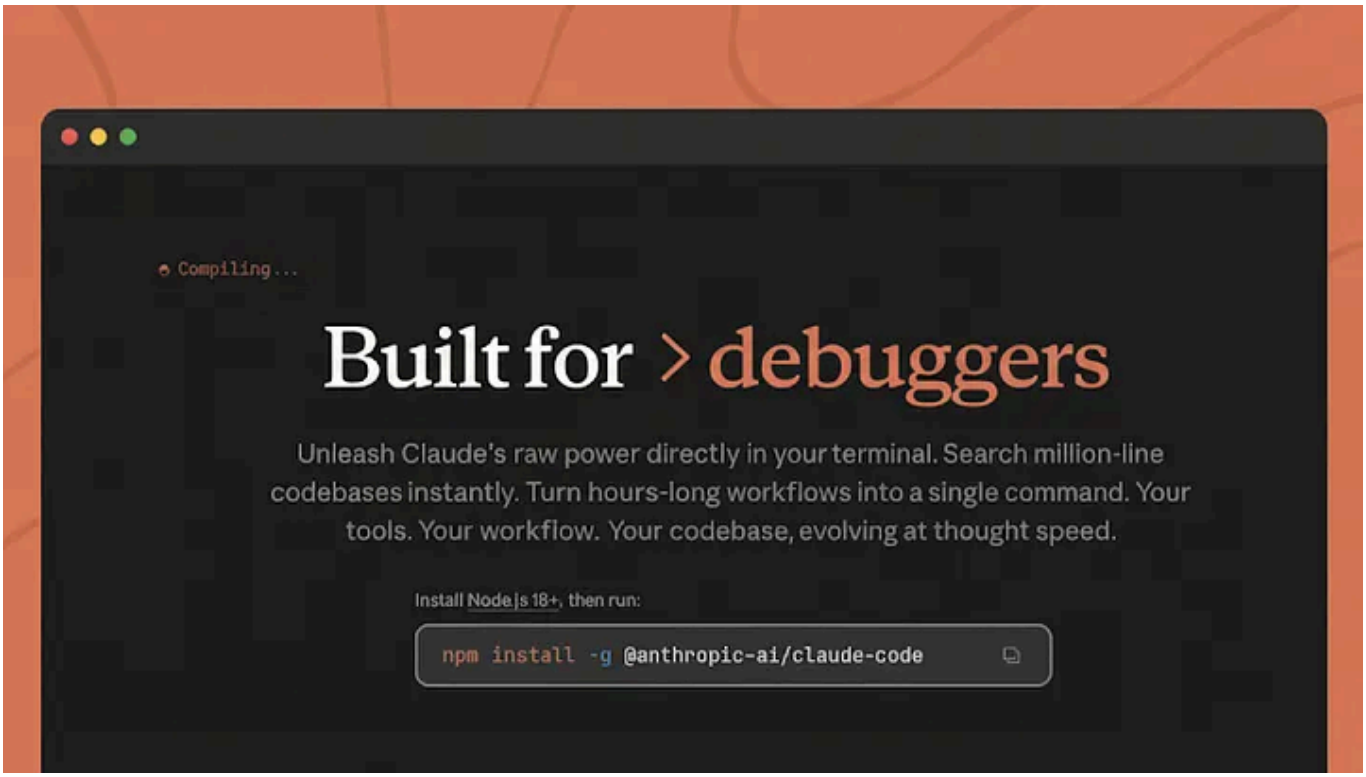
Figure: Hierarchical CLAUDE.md files allow you to organize project knowledge at multiple levels – from global and team-wide rules down to project-specific and even folder-specific guidelines .

Claude automatically pulls in all applicable CLAUDE.md contexts, prioritizing the most specific guidance for each task.

Pro Tip: If your project has distinct subsystems *(e.g. frontend vs backend)*, consider using nested CLAUDE.md files in those directories for more targeted guidance.

Claude will pull in the CLAUDE.md from the root and the one in frontend/ when working on UI code, for example. This hierarchy keeps context relevant and tight.

Also, whenever you finish a major refactor, have Claude update the architecture section *(you can literally tell it "update the Architecture section in CLAUDE.md to reflect our new module structure")*. Keeping this blueprint current pays dividends in code quality!



Anthropic Claude Code Agentic Coding Tool

2. Command Center

Never let Claude forget how to build, run, or test your app. One of the first pain points I fixed was Claude not knowing my project's dev scripts. *(It would sometimes ask "How do I run the app?")*.

The Command Center entry is a simple list of common commands and what they do. This spares Claude from scanning your package.json or Makefiles repeatedly and speeds up its iterations .

Think of it as a cheat sheet for all the CLI tasks in your project.

Real-world example: In my Node API, I added a *"Bash Commands"* section listing npm run dev, npm run test, npm run lint with descriptions. Suddenly, Claude stopped guessing and started executing the right commands automatically. When I asked it to run tests, it already knew to use npm run test *(and even what testing library was in use)*.

Here's how a Command Center snippet might look:

Bash Commands

- `npm run dev` – Start the development server (auto-reloads on code changes).
- - `npm run build` – Build the project for production (outputs to `dist/` folder).
- - `npm run test` – Run the test suite (uses Jest with coverage).
- - `npm run lint` – Lint all files using ESLint (enforces coding standards).

By documenting these, you ensure Claude always uses the correct incantations. No more *"it works on my machine"* moments because the AI ran the wrong command.

As Anthropic's engineers advise: document your frequently used tools and scripts in CLAUDE.md so Claude can use them without hesitation .

Pro Tip: Include tool versions or env quirks if relevant. For example, specify *"Use Node 18"* or *"Requires Python 3.11 venv"* in this section if your setup is finicky. Claude will then proactively handle environment setup as it writes code.

Also, if you find Claude asking permission or confirmation for routine tasks *(like running tests)*, you can preempt that by whitelisting safe commands. *(I personally run Claude Code with a – dangerously-skip-permissions flag in dev, but only do this if you trust your CLAUDE.md and tests!)*.

The key is streamlining Claude's access to your workflow – give it the keys to your Command Center and watch it go to work.

3. Style Guide Sheriff

Keep your code style consistent, no matter who (or what) is writing the code. Nothing's worse than an AI assistant that flips coding styles mid-project – one minute it's using snake_case, the next it's camelCase; or it's mixing tabs and spaces (gasp!).

To prevent such chaos, I deputized a Style Guide Sheriff in my CLAUDE.md. This entry lays down the law on code style and conventions: *from syntax preferences to naming patterns.*

Real-world example: My React project had strict ESLint rules *(single quotes, 2-space indent, semicolons, etc.)*. Initially, Claude would produce code that violated some lint rules, which meant extra fix-up work.

I added bullets like *"Use ES6+ syntax (no CommonJS require calls)"* and *"Follow our ESLint rules (2-space indent, single quotes, no semicolons)*" to CLAUDE.md.

Magically, Claude's outputs started conforming to these standards without reminders.

It's like it internalized our prettier config. Here's a sample snippet:

Code Style Guidelines

- **Syntax:** Use ES Modules (`import`/`export`) rather than CommonJS. Use modern ES6+ features (arrow functions, etc.) where appropriate.
- **Formatting:** 2 spaces for indentation. Use single quotes for strings. ***No*** trailing semicolons *(we run Prettier)* – except where necessary in TypeScript *(enums, interfaces)*.
- **Naming:** Use `camelCase` for variables/functions, `PascalCase` for React components and classes. Constants in `UPPER_SNAKE_CASE`.
- **Patterns:** Prefer functional components with hooks over class components in React. Avoid using any deprecated APIs.

This gives Claude a stylistic compass. After adding a style section, I noticed far fewer nitpicks during code review – Claude's pull requests started to pass linters and look uniform.

The official Claude Code best practices explicitly recommend putting your code style rules in CLAUDE.md for exactly this reason. Consistency is queen in maintainable code, and now Claude polices it for me.

Pro Tip: Enforce tone in comments and commit messages too. If you have a standard for code comments or PR descriptions *(e.g. JSDoc format, or "must include Jira ticket ID in commit message"),* note that down.

Claude will then include well-formed comments and even format its git commits accordingly. One of my entries says *"Prefix all commit messages with the ticket ID and a short scope (e.g. '[PAY-123] feat: …')".* Lo and behold, Claude started crafting commit messages like a seasoned dev. The Style Guide Sheriff isn't just about semicolons and brackets – you can extend it to any convention you care about.

4. Test Bench Coach

Make testing a first-class citizen by prompting Claude to think test-first. If you're like me, writing tests can slip through the cracks when you're in the zone.

But Claude Code loves to generate tests – especially if you explicitly ask or remind it to. I added a Testing Instructions section in CLAUDE.md to ensure that for every feature or bugfix, Claude either writes or updates tests as part of its workflow.

Real-world scenario: While pair-programming on a Redux state management module, I wanted to enforce that new reducers have unit tests. I put guidance in CLAUDE.md like *"Always include tests for new features"* and specifics like *"Use Jest and React Testing Library for UI tests."*

The effect: after implementing a React component, Claude would immediately suggest creating a corresponding test file, often without me even asking. It's essentially a gentle nudge baked into every prompt.

Here's how a Test Bench Coach entry might look:

Testing Instructions

- Always follow **TDD mindset**: for any bug fix or new feature, consider writing tests first or immediately after coding.
- Use **Jest** for unit tests. For React components, use @testing-library/react for rendering and assertions.
- Aim for high coverage on core logic (services, reducers, etc.). Include edge cases (invalid inputs, error states) in tests.
- **Test Naming:** use `describe` blocks for modules and `it('should ...')` for behaviors. Keep tests clear and focused.
- Run tests with `npm run test` and ensure all pass before considering a task done.

This entry turned Claude into a polite drill sergeant: I'd see it auto-suggest "*I'll now write tests for X"* after completing a function.

It's worth noting Anthropic suggests documenting testing frameworks and practices in CLAUDE.md , because it steers Claude's chain-of-thought toward quality. You're effectively training the AI to adopt a testing culture.

Pro Tip: Leverage Claude's tireless nature for exhaustive testing. Add a line like *"If time allows, generate additional tests for edge cases and potential regressions."*

Claude will then often go above-and-beyond, writing extra tests that even I might forget *(ever tested a function at 2 AM? Claude will!).* Some devs even include a prompt in CLAUDE.md for property-based tests or fuzz tests for critical sections.

Remember, Claude can think of and execute tests rapidly, so encourage it. Your future self *(and your QA team)* will thank you.

5. Error Handling Mantra

Teach Claude to debug and handle errors like a seasoned engineer. This entry was a personal game-changer. I call it the Error Handling Mantra – a set of guidelines that prompt Claude to approach bugs and exceptions methodically, rather than slapping on band-aid fixes.

The mantra usually goes into CLAUDE.md as a short checklist or philosophy, reminding the AI to think step-by-step when facing errors .

Real-world example: I once had Claude refactor an API call, and it introduced a subtle bug. Instead of explaining the entire app context again, I relied on my CLAUDE.md mantra: *"When an error occurs, analyze the root cause step-by-step before proposing a fix."*

The result? Claude produced a chain-of-thought explanation of the likely cause *(incorrect argument type)*, then suggested a fix, complete with a thoughtful console warning. I basically got a built-in rubber ducky debugger.

My Error Handling Mantra snippet looks like this:

Error Handling & Debugging

- **Diagnose, Don't Guess:** When encountering a bug or failing test, first explain possible causes step-by-step : docs.claude.com. Check assumptions, inputs, and relevant code paths.

- **Graceful Handling:** Code should handle errors gracefully. For example, use try/catch around async calls, and return user-friendly error messages or fallback values when appropriate.
- **Logging:** Include helpful console logs or error logs for critical failures (but avoid log spam in production code).
- **No Silent Failures:** Do not swallow exceptions silently. Always surface errors either by throwing or logging them.

This entry tunes Claude's behavior under failure conditions. After adding it, I've seen Claude catch its own mistakes more often.

It will say *"I noticed this could throw an undefined error, I'll add a check"* without my prompting. In essence, the AI starts thinking like a dev on a bug hunt, which is exactly what we want .

Above: A before/after diff from my code where Claude applied better error handling. The *"after"* (green) shows Claude adding an input validation *(if (!name) throw…)* and using a template string for output, following the Error Handling Mantra to not proceed with a missing name.

Pro Tip: Borrow techniques from seasoned debuggers. For instance, add a rule like *"Always use React Error Boundaries around components that make API calls"* if you're in a React context – this came from my CLAUDE.md after a production fiasco, and it was inspired by a pro tip I saw online .

Claude then automatically wrapped risky components in <ErrorBoundary> components in one of my projects!

Also, don't shy from teaching Claude debugging strategies: e.g., *"On critical bugs, consider binary search through git history or add temporary console.debug statements to isolate the issue."*

It sounds crazy to tell an AI this, but I've had Claude suggest bisecting code after reading that guideline. The mantra instills a cautious, systematic approach that will save your hide when things go wrong.

6. Clean Code Commandments

Encode your *"clean code"* principles so Claude writes maintainable, human-friendly code by default. Think of this like setting your pair-programmer's conscience. We all have pet peeves in code review – functions that are too long, variables named data or foo, deeply nested logic, etc.

I turned those unwritten rules into explicit CLAUDE.md entries, and the effect was immediate: Claude's code got noticeably cleaner and more modular. No joke, my diff churn went down because I wasn't constantly refactoring AI-written code for clarity.

Real-world example: At one point, Claude generated a 150-line function.

My rule is *"no function should be over ~40 lines"*. So I added a Clean Code guideline: *"Limit function length; refactor large functions into smaller helpers with clear names."* Next time, Claude automatically split a long routine into several well-named functions without me asking.

I also added naming commandments *("Name functions clearly, e.g. calculateInvoiceTotal, not handleData")*. The code started to read like a book authored by Uncle Bob himself.

A snippet of my Clean Code Commandments:

Clean Code Guidelines

- **Function Size:** Aim for functions ≤ 50 lines. If a function is doing too much, break it into smaller helper functions.
- **Single Responsibility:** Each function/module should have one clear purpose. Don't lump unrelated logic together.
- **Naming:** Use descriptive names. Avoid generic names like *`tmp`, `data`, `handleStuff`*. For example, prefer *`calculateInvoiceTotal`* over *`doCalc`*.
- **DRY Principle:** Do not duplicate code. If similar logic exists in two places, refactor into a shared function (or clarify why both need their own implementation).
- **Comments:** Explain non-obvious logic, but don't over-comment self-explanatory code. Remove any leftover debug or commented-out code.

After this, I saw Claude self-correct things like: *"I notice this function is getting large, I'll refactor part of it into a new helper formatAddress()."*

In one case, it spotted duplicate code in two microservices and suggested abstracting it – all because the DRY principle was explicitly in its context. Essentially, you're baking your senior engineer wisdom into Claude's pseudo-consciousness.

Pro Tip: Include any specific "gotchas" or code smells unique to your project. For example, in one codebase we had a rule against using moment.js *(we use Day.js instead).*

I added: *"Do not use moment library (use Day.js for date handling)."*

Sure enough, Claude stopped suggesting moment altogether. If there's an anti-pattern you never want to see (like using any in TypeScript, or global variables, or copying code from StackOverflow without attribution), write it down.

Claude will generally comply. Think of these as unit tests for Claude's output – if it violates a Commandment, you have grounds to say *"Bad Claude, try again!"* But in my experience, it rarely does after seeing these guidelines.

7. Security Sentry

Appoint Claude as your security guard by listing critical security best practices. This CLAUDE.md entry ensures that your AI buddy codes with an eye on vulnerabilities and safe practices – an area where GPT-style models often lack context.

By being explicit about security (input validation, encryption, etc.), I got Claude to catch security issues that I might have missed in a hurry .

Real-world scenario: Working on a user authentication flow, I wrote guidelines on password handling: *"Always hash passwords with bcrypt, never store plain text; validate email format; apply rate limiting on login attempts,"* etc.

I kid you not – Claude then incorporated input validation for email and even reminded me to use parameterized SQL queries in one of the outputs, unprompted. It's like having an OWASP ZAP tool baked into your coding assistant.

Another time, in a React app, I added a note: *"Sanitize any HTML content to prevent XSS (use DOMPurify)."* Thereafter, any time we dealt with

dangerouslySetInnerHTML, Claude automatically suggested using DOMPurify. Mind blown.

Here's a sample Security Sentry entry:

Security Guidelines

- **Input Validation:** Validate all inputs (especially from users or external APIs). Never trust user input – e.g., check for valid email format, string length limits, etc.
- **Authentication:** Never store passwords in plain text. Use bcrypt with a salt for hashing passwords. Implement account lockout or rate limiting on repeated failed logins.
- **Database Safety:** Use parameterized queries or an ORM to prevent SQL injection. Do not concatenate user input in SQL queries directly.
- **XSS & CSRF:** Sanitize any HTML or user-generated content before rendering (consider using a library like DOMPurify). Use CSRF tokens for state-changing form submissions.
- **Dependencies:** Be cautious of eval or executing dynamic code. Avoid introducing packages with known vulnerabilities (Claude should prefer built-in solutions if external libs are risky).

This might seem heavy, but Claude can handle it – and it will enforce these rules diligently. Anthropic's own team noted Claude often finds bugs and potential vulnerabilities that humans overlook , so giving it a security rulebook makes it even more effective.

I've seen Claude refuse to implement something in an insecure way because "the Security Guidelines said so." For example, it wouldn't use localStorage

for a JWT without me explicitly allowing it, citing security concerns (I was both annoyed and impressed at the same time!).

Pro Tip: Update this section as new threats emerge or when using new tech. If you switch to using AWS S3, add a line like *"Ensure S3 buckets are accessed with least privilege and no public ACLs."* Claude will then pay attention to that when generating IaC or backend code. The Security Sentry gives you peace of mind – it's like you've hired a security consultant who reviews every line as it's written. In 2025, where agentic AI coding is rising, this entry is your safety net against accidentally deploying vulnerabilities.

8. Teamwork Protocol

Keep Claude aligned with your team's collaboration conventions – from Git etiquette to documentation. This entry addresses the *"soft"* aspects of coding that are nonetheless critical for a smooth developer experience. After all, coding isn't just writing functions – it's also writing commit messages, doing code reviews, and updating docs. By teaching Claude your team's workflow, you get an AI that's not just coding in a vacuum but truly acting as a member of the team.

Real-world example: Our team has conventions like branching off dev for features, prefixing commit messages with ticket IDs, and updating the CHANGELOG.md for notable changes. I encoded these in CLAUDE.md.

The result? When Claude finished a feature implementation, it automatically formatted the commit message perfectly *(e.g., "feat(login): add Google OAuth support (RES-102)")* and even appended a bullet to our changelog in one go. I sat there slack-jawed. Similarly, I added: *"Document new endpoints in the API docs markdown."* Sure enough, after creating a new API route, Claude added a neat snippet in docs/api.md describing the endpoint.

A template for Teamwork Protocol might be:

Collaboration & Workflow

- **Git Branches:** Follow GitFlow lite – create feature branches off `dev` (e.g., `feature/login-form`), merge via Pull Request. Do **not** commit directly to `main`.
- **Commit Messages:** Use conventional commits (e.g., `feat: `, `fix: `, `docs: ` prefixes). Include JIRA ticket ID in commit if available. Keep message concise (one line summary, optional details after).
- **Pull Requests:** When a task is done, have Claude open a PR with a brief description of changes and tag the relevant reviewers (e.g., `@frontend-team` for UI changes).
- **Documentation:** If code changes affect user-facing behavior or APIs, update the relevant Markdown docs in the `docs/` folder as part of the same PR.
- **Code Reviews:** Claude should assist in code reviews if asked (e.g., static analysis for bugs, ensure style guide adherence) and only approve when all checks pass.

By spelling out these expectations, I effectively got Claude to handle the boring stuff that often falls through the cracks. It started writing release notes for me *("Added feature X, fixed bug Y")* when pushing a PR, because I had mentioned updating docs and changelogs.

Anthropic's best practices hint at including repository etiquette and workflow notes in CLAUDE.md – which is exactly what this is. It turns Claude from just a code generator into a more holistic dev assistant.

Pro Tip: Use Claude to onboard new team members. With a detailed Teamwork Protocol in place, a new developer (human) can literally read CLAUDE.md to get a quick intro to "how we work around here." I've shared our CLAUDE.md with interns, and they found it more helpful than the company wiki.

Also, consider adding a fun line like *"Our team values clean code and helping others – Claude should be encouraging in tone during explanations."* Believe it or not, the AI will adopt a friendly tone in its comments and outputs, which makes the dev experience more pleasant.

This protocol section makes Claude a true collaborator, not just a code monkey.

9. Edge-Case Oracle

Ensure no corner case is left unconsidered by instructing Claude to always think about edge conditions. This entry pushes Claude to go beyond the "happy path" and anticipate scenarios that a junior dev might miss. It's like having a senior engineer who endlessly asks *"But what if…?"* for every change, which dramatically improves robustness.

Real-world scenario: I added a note in CLAUDE.md: *"For any non-trivial feature, list out potential edge cases and handle them."* When building a date range picker, Claude came up with edge cases like "What if the end date is before the start date?" and implemented a check to swap them.

I hadn't even thought of that yet! On another occasion, for a financial calc module, it asked *"What if the input array is empty?"* and decided to return 0 gracefully – again, something I could have overlooked at first.

Anthropic's internal teams reported similar benefits: catching edge cases in design rather than in prod . So let's put that wisdom into our Oracle entry.

Example snippet:

Edge Case Considerations

- Always consider edge and corner cases for any logic:
- Empty or null inputs (e.g., an empty list, missing fields, zero values).
- Max/min values and overflow (e.g., extremely large numbers, very long text).
- Invalid states (e.g., end date before start date, negative quantities).
- Concurrency issues (e.g., two users editing the same data simultaneously).
- If an edge case is identified, handle it in code or at least flag it with a comment/TODO.
- Prefer to fail fast on bad input (throw an error or return a safe default) rather than proceeding with wrong assumptions.

Pro Tip: Encourage Claude to articulate edge cases in planning mode. If you use Claude's *"think"* or plan capabilities, having this Oracle entry will nudge it to enumerate edge cases during the planning step. This means you'll get a checklist of scenarios to test or account for before writing the code. You can even formalize this: add

"Step 2: Brainstorm edge cases" in a blueprint of how to approach problems *(see next section on workflow guardrails)*. Then Claude will systematically produce that list. Better to catch the weird stuff upfront than at 3 AM on launch night, right?

10. Agentic Workflow Guardrails

Guide Claude in orchestrating complex, multi-step tasks by breaking them down and verifying each step. This final entry is like the meta-hack that ties everything together.

Claude Code is an agentic AI coding tool – meaning it cannot only write code but also plan, execute, and adjust in multi-step workflows. The Workflow Guardrails entry tells Claude how to go about solving big problems rather than jumping in headfirst.

It's essentially process advice: plan, execute in parts, review, adjust – the same way a senior dev would tackle a large project.

Real-world example: I tasked Claude with adding a major feature (a full OAuth2 login flow). Instead of letting it try everything in one go (which can fail spectacularly), I leveraged Guardrails.

In CLAUDE.md, I had something like: *"For large tasks, follow a 3-step approach – Analysis → Plan → Implement. Always propose a plan first for approval."* So when I prompted Claude for the OAuth feature, it first responded with a structured plan *(list of steps: create OAuth client, add callback endpoint, handle tokens, etc.)*. We iterated on that plan *(I added a step for refresh tokens)*.

Only then did Claude write the code, and it did so modularly *(it literally said, "I will implement step 1 now…" and so on).* The outcome was clean and worked on first run. Without guardrails, I suspect it would've been a monolithic dump of code and likely errors.

Here's a condensed version of an Agentic Workflow Guardrails entry:

Workflow & Planning Guidelines

- For any complex or multi-step task, Claude should first output a clear plan or outline of the approach anthropic.com. *(E.g., list the steps or modules needed)*.
- **Incremental Development:** Implement in logical chunks. After each chunk, verify it aligns with the plan and passes tests before moving on.
- Think Aloud: Use extended reasoning *("think harder or ultrathink")* for complex decisions. It's okay to spend more tokens to ensure a solid approach rather than rushing coding.
- **User Approval:** Pause for confirmation after providing a plan or major design decision. Only proceed once the user/developer confirms.
- **Error Recovery:** If a solution isn't working, Claude should backtrack and rethink rather than stubbornly persisting. Consider alternative approaches if tests fail or constraints are hit.

These guardrails dramatically improved Claude's success rate on big tasks.

It essentially follows a mini agile cycle now: *propose -> get feedback -> implement -> review.*

This is backed up by Anthropic's own recommendations – they observed that forcing Claude to research and plan before coding *"significantly improves performance for deeper tasks"*.

I can second that: the code I get with this approach is not only more correct, but often better engineered *(Claude will actually say "I'll create a helper class for OAuth config" because it had time to think of design).*

Pro Tip: Define custom trigger words for modes. For example, I added: *"If I say 'Let's brainstorm', enter Plan Mode."* This way Claude knows when I'm explicitly asking for a plan vs. code.

You can also integrate your sub-agent strategies here (if you use them) – e.g., instruct Claude to spin up a *"Test Runner"* sub-agent when needed, etc.

Another useful tip: have Claude summarize the plan in a file (like PLAN.md) that you can refer back to or even include via @filename reference. This keeps context size down.

Ultimately, the Workflow Guardrails ensure Claude behaves more like a project partner than an overeager junior dev. It will pace itself, double-check, and keep you in the loop at every stage. In other words, it codes like an engineer, not just a code generator.

Ready to supercharge your own Claude Code sessions? Steal my CLAUDE.md template and tweaks (link in bio), tailor them to your stack, and drop it into your repo.

From that point on, every prompt you feed Claude will carry the weight of your project's collective knowledge and standards. As a result, you'll crank out production-ready code at lightning speed – I'm talking shipping in hours what used to take weeks.

In 2026, I predict every dev's repo will have a CLAUDE.md – or get left behind. This is the new secret sauce for those in the know. If you've read this

far, you're now among them.

Go forth and build amazing things with your AI sidekick! And once you've tried these hacks, come back and brag: what's your killer CLAUDE.md entry? Drop it in the comments – I'm genuinely excited to learn new tricks from this awesome community.

Happy coding!

Or check out my new Master Guide for Spec-Driven Development:

👉 **Step 1:** Read this Master Guide — your evergreen hub for Spec-Driven Development resources.
👉 **Step 2:** Read Part 1: The Foundation to set up your memory system, constitution, and specification workflow.
👉 **Step 3:** Continue with Part 2: Execution and Scaling to turn specs into plans, tasks, and tested features.

My new agentic coding guide:

👉 THE Claude Agent SDK BUILDER'S PLAYBOOK — Part 1

Or read about my experiences using the prompt I have shared with you in this article: I Gave Claude Code 2.0 Our 3-Week Refactor at 11 PM. At 7 AM, It Was Done

About the Author

Alireza Rezvani is a Chief Technology Officer, Senior Full-stack Architect, and Software Engineer, as well as an AI Technology Specialist, with expertise in modern development frameworks, cloud-native applications, and agent-based AI systems. With a focus on *ReactJS*, *NextJS*, *Node.js*, and cutting-edge AI technologies and concepts of AI engineering, Alireza helps engineering teams leverage tools like *Gemini CLI*, and *Claude Code* or *Codex from OpenAI* to transform their development workflows.

Connect with Alireza at *alirezarezvani.com* for more insights on AI-powered development, architectural patterns, and the future of software engineering.

Looking forward to connecting and seeing your contributions — check out my *open source projects on GitHub*!

✨ Thanks for reading! If you'd like more practical insights on AI and tech, hit **subscribe** to stay updated.

I'd also love to hear your thoughts — drop a comment with your ideas, questions, or even the kind of topics you'd enjoy seeing here next. Your input really helps shape the direction of this channel.

Claude Code | Agentic Coding | Software Engineering | Anthropic Claude | Artificial Intelligence

Written by Reza Rezvani

574 followers · 61 following

Follow

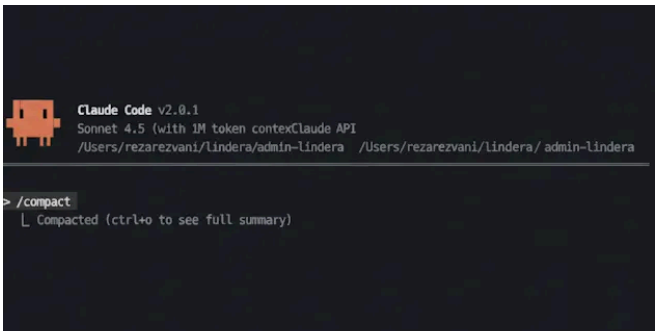
As CTO of a Berlin AI MedTech startup, I tackle daily challenges in healthcare tech. With 2 decades in tech, I drive innovations in human motion analysis.

No responses yet

Bgerby

What are your thoughts?

More from Reza Rezvani

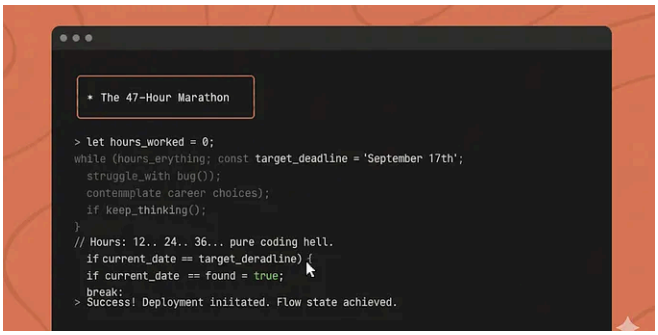


Reza Rezvani

The Complete Claude Code 2.0

Claude Code 2.0 isn't just an update—it's a complete reimagining of how AI assists...

Sep 30 284 11



Reza Rezvani

The 47-Hour Marathon That Almost Made Me Quit Claude Code—Unti...

I had to share it with you ... Maybe you had to go through the same. I need to tell you about...

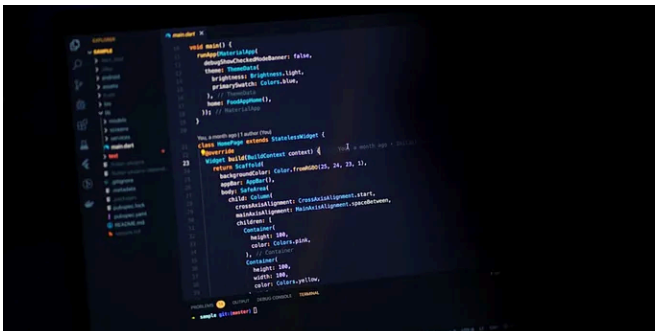
Sep 18 278 13

Reza Rezvani

Mastering Claude Code: A 7-Step Guide to Building AI-Powered...

From Chaos to Code: How I Reduced Development Time by 70% Using Claude...

Sep 9 108 4



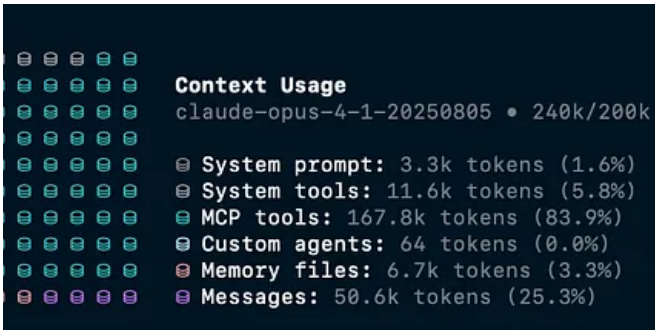
In nginity by Reza Rezvani

The Flutter Architecture That Saved Our Team 6 Months of...

Sep 14 382 13

See all from Reza Rezvani

Recommended from Medium



Sevak Avakians

Claude Code Limit Hit on Max Plan?! What to do next:

$200/m plan blocks me for a week!

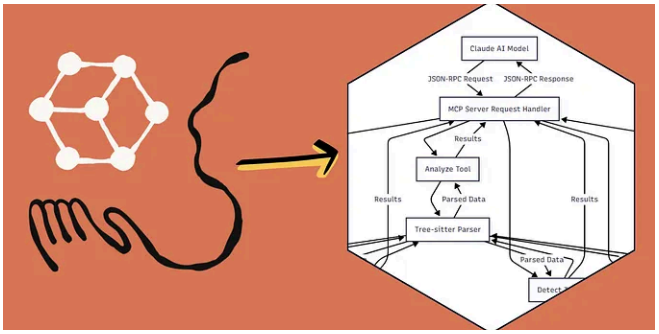
5d ago 50 8

Reza Rezvani

The ultimate Code Modernization & Refactoring prompt for your...

Transform your legacy codebase chaos into a strategic modernization roadmap with this...

3d ago 63



Joe Njenga

I Asked Claude 4.5 to Build Me a Complex MCP Server (It Was So...

If you want to build your custom MCP server, stop wasting time thinking, coding, or trying...

In Vibe Coding by Alex Dunlop

This Just Became the Most Important Tool in Frontend (Here'...

The goldmine I've always wanted solved

3d ago 359 6

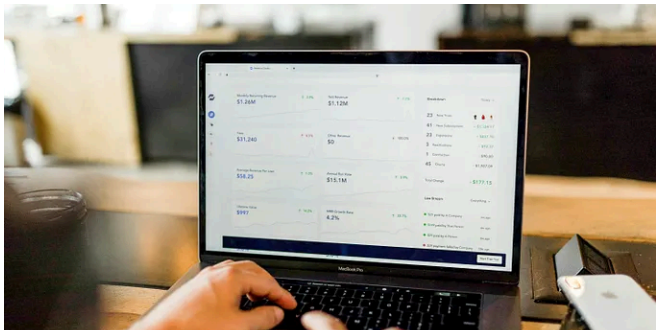
In Artificial Intelligence in Plain E... by Prithwish N...

5 Essential MCP Servers That Give Claude & Cursor Real Superpower...

Transform Claude & Cursor into web scraping, browser-controlling automation engines. 5...

Sep 27 158 1

6d ago 66 3



In Coding Nexus by Civil Learning

How I Used Claude to Validate My SaaS Idea in 10 Minutes.

A few months ago, I had too many SaaS ideas stored in Notion.

6d ago 68 7

See more recommendations