Vibe Coding

Brian Edwards

November 2025

Contents

Vibe Coding

One Human, One AI, One Industry Disrupted

Brian Edwards

November 2025

brian.mabry.edwards@gmail.com

Preface

The conventional wisdom about artificial intelligence and software development goes something like this: AI tools are helpful assistants that can speed up certain tasks, autocomplete code snippets, and perhaps help developers work somewhat more efficiently. The prudent approach is to integrate these tools gradually into existing workflows, using them to augment human capabilities while maintaining the fundamental structures of software teams, development processes, and engineering practices that have evolved over decades. This is the safe view. It is also, I will argue, profoundly wrong.

What I am describing in this book is not an incremental improvement to software development. It is a rupture. The practices, team structures, and economic assumptions that have governed software engineering for the past thirty years are not being gently updated—they are being rendered obsolete. This is not a prediction about what might happen in five years. It is a description of what is already happening, documented through my own experience building production healthcare software over the past several months. The evidence suggests that a single developer working with current AI capabilities can accomplish in weeks what traditionally required teams of ten or more engineers working for a year or longer.

I recognize this sounds implausible. It sounded implausible to me before I experienced it. The purpose of this book is not merely to assert that this transformation is occurring, but to explain precisely how and why it works, to provide a detailed methodology for others to replicate these results, and to explore the broader implications for the software industry and beyond.

The ideas presented here are radical in the original sense of that word—they go to the root. They require rethinking fundamental assumptions about how software gets built. They are also opinionated in the extreme. I will not hedge my arguments or present "both sides" of questions that I believe have clear answers. Where the evidence from my direct experience contradicts conventional wisdom, I will say so plainly. This is not a balanced overview of AI-assisted development approaches. It is an argument for a specific methodology, grounded in specific evidence, with specific recommendations.

A word about timing: everything in this book is calibrated for late 2025. The specific tools, the particular capabilities, the concrete recommendations—all of these will change rapidly over the coming months and years. The underlying principles, I believe, will endure longer. But the reader should understand that this is a snapshot of a fast-moving situation, not a timeless treatise.

Part One: The Nature of the Change

Chapter 1: What Traditional Software Development Actually Looks Like

Before we can understand what is changing, we need to establish clearly what has existed. The way software gets built at successful technology companies is not arbitrary. It evolved over decades in response to real constraints, and understanding those constraints is essential to understanding why they no longer apply in the same way.

Consider a typical software team at a well-run technology company—I will draw on my own experience working at Atlassian on the Bitbucket team, which represented something close to industry best practices for distributed team development. The team consisted of roughly fifteen engineers spread across multiple time zones, supported by product managers, designers, and engineering managers. We followed a variation of agile methodology with two-week sprints, daily standups, regular retrospectives, and carefully maintained backlogs of work items prioritized through an elaborate process involving multiple stakeholders.

The question worth asking is: why? Why fifteen engineers instead of five or fifty? Why two-week sprints? Why the extensive ceremony around planning and coordination? The answers reveal the fundamental constraints that shaped traditional software development.

The first constraint is human cognitive bandwidth. An individual engineer can hold only so much context in their head at once. A codebase of any significant size exceeds what any single person can fully comprehend. Therefore, work must be divided into pieces small enough for individuals to reason about. But division creates coordination problems: the pieces must fit together, interfaces must be agreed upon, dependencies must be managed. The elaborate processes of traditional software development exist largely to solve coordination problems that arise from dividing work among humans with limited cognitive capacity.

The second constraint is human speed. Writing code is slow. Not the typing—the thinking. Understanding a problem, designing a solution, implementing it, testing it, debugging it—these activities consume hours and days. When a bug appears in production, the engineer must context-switch from whatever they were doing, rebuild the mental model of the relevant code, diagnose the issue, implement a fix, and verify it works. This might take an afternoon or several days. At Atlassian, we spent considerable effort on practices designed to reduce the cost of context-switching: comprehensive documentation, consistent coding standards, thorough code reviews. All of this overhead exists because human developers are slow and their attention is expensive.

The third constraint is human availability. Engineers get sick. They take vacations. They leave for other jobs. They have lives outside of work that legitimately take priority. A healthy software

organization must be resilient to individual unavailability. This means knowledge cannot be concentrated in single individuals—multiple people must understand each part of the system. It means documentation must be good enough that someone unfamiliar with a component can come up to speed. It means code must be written with future maintainers in mind, not just the original author. The "bus factor"—how many people would need to be hit by a bus before critical knowledge is lost—is a genuine concern that shapes how teams operate.

The fourth constraint is human fallibility. People make mistakes. They introduce bugs. They misunderstand requirements. They make poor design decisions under time pressure. Traditional software development includes numerous checkpoints to catch these errors: code reviews where other engineers examine changes before they're merged, automated tests that verify behavior hasn't regressed, staging environments where changes can be validated before reaching production, gradual rollouts that limit the blast radius of problems. Each of these practices adds time and overhead, but they exist because humans cannot be trusted to consistently produce error-free work.

Put these constraints together and you get something like the fifteen-person team I worked on at Atlassian. The elaborate processes, the coordination overhead, the careful division of work—none of this was bureaucracy for its own sake. It was a rational response to the genuine limitations of human developers working together on complex software systems.

And this brings us to the central question: what happens when those constraints no longer apply?

Chapter 2: What Has Actually Changed

The change I am describing is not that AI can help with coding. That has been true in some form for years, from simple autocomplete to more sophisticated suggestion systems. The change is that AI—specifically, the current generation of large language models exemplified by Claude—can perform the core intellectual work of software development. Not typing code faster, but understanding requirements, designing solutions, implementing them correctly, identifying and fixing bugs, writing tests, and maintaining large codebases over time. This is a difference in kind, not merely degree.

To understand why this matters so much, consider what actually takes time in software development. It is not typing. A proficient developer can type at 60 or 80 words per minute, and code is far more compact than prose. The bottleneck has never been keystroke speed. What takes time is understanding: reading existing code to understand how it works, comprehending requirements to understand what needs to be built, analyzing bugs to understand why they occur. What takes time is decision-making: choosing among possible approaches, evaluating tradeoffs, designing interfaces that will remain maintainable as requirements evolve. What takes time is verification: writing tests that meaningfully exercise the code, debugging when those tests fail, ensuring that changes don't break existing functionality.

Current AI systems can do all of this. Not perfectly—I will discuss limitations at length—but well enough to fundamentally change the economics. When I describe working with Claude on the Care Commons project, I am not describing an assistant that helps me code faster. I am describing a collaborator that reads thousands of lines of existing code, understands the architecture, implements new features that integrate correctly with what exists, writes comprehensive tests, and fixes its own mistakes when tests fail. The speed advantage is dramatic—tasks that would take a human developer an afternoon take Claude minutes—but the speed is not the fundamental point. The point is that the *nature* of the work I do has changed. I no longer implement features. I describe what needs to exist and verify that what gets built matches my intentions.

Let me be precise about the capabilities I am describing, because exaggeration in this domain is rampant and corrosive to clear thinking. Claude can: read and understand large codebases across multiple programming languages, maintain context about system architecture and business requirements across extended conversations, implement features that require understanding of complex business logic, write tests that meaningfully exercise the code being tested, debug failing tests by reasoning about what might have gone wrong, refactor code while preserving behavior, write documentation that accurately describes what code does, use command-line tools to interact with git repositories and deployment infrastructure, and learn from mistakes pointed out during review.

Claude cannot: make genuinely novel architectural decisions that go beyond patterns in its training data, understand requirements that are ambiguous or contradictory without human clarification, catch certain classes of subtle bugs that require deep domain expertise, replace human judgment about what should be built or whether a design is appropriate, operate without oversight in production systems with high stakes, or improve beyond its current capabilities through experience on a project.

This distinction matters because the appropriate use of AI in software development depends critically on understanding what it can and cannot do. The methodology I am advocating gives Claude tremendous autonomy in implementation while reserving certain decisions for human judgment. This division of labor reflects the actual distribution of capabilities, not aspirational claims about

AI or excessive caution born of unfamiliarity.

The constraints I described in the previous chapter—cognitive bandwidth, speed, availability, fallibility—look very different when the entity doing the work is an AI system rather than a human developer.

Cognitive bandwidth: Claude can hold vastly more context than a human. Not infinite, but vastly more. It can read through thousands of lines of code, maintain understanding of how different parts of a system interact, and recall details of earlier conversations. The need to divide work into small pieces, with all the coordination overhead that entails, is dramatically reduced.

Speed: Claude operates at a pace measured in seconds per feature, not hours or days. A 200-line React component that would take a human developer an hour or two takes Claude perhaps thirty seconds. This is not an approximation or an aspiration; it is what I observe repeatedly in practice. The compound effect of this speed advantage over an extended project is difficult to overstate.

Availability: Claude does not get sick, does not take vacations, does not leave for a job at another company. The knowledge it develops about a codebase is never lost to organizational turnover. There is no bus factor to worry about because the entity doing the work is always available.

Fallibility: Claude makes mistakes, but it makes different kinds of mistakes than humans do, and the checking systems that catch those mistakes can be automated. Humans forget. Humans get tired. Humans cut corners under deadline pressure. Claude does none of these things. It will consistently apply the same patterns, run the same checks, follow the same procedures every time. When it does make errors, they are often detectable through automated tests and easily fixed in subsequent iterations.

I want to be careful here because I am not arguing that AI is superior to humans in some abstract sense. I am arguing something more specific and more defensible: for the particular task of implementing software based on requirements that a human has specified, current AI systems have a set of characteristics that fundamentally change the optimal way to organize the work.

Chapter 3: Why Claude, Why Now

This book focuses specifically on Claude as the AI system of choice for this methodology. This is not because Claude is the only capable system—there are several others worth mentioning—but because Claude represents the current state of the art for the particular kind of work I am describing, and because specificity is more useful than vague generalities about "AI assistants."

The other systems worth acknowledging include OpenAI's GPT-4 and its derivatives, Google's Gemini, various open-source models like Qwen and Llama variants, and the specialized coding models that have emerged from multiple organizations. Each of these has strengths. For simple, bounded tasks, many of them perform adequately. The differences emerge in sustained, complex work: maintaining coherent understanding across a large codebase, making consistent architectural decisions, recovering gracefully from errors, and operating with minimal human intervention over extended periods.

In my experience, Claude handles these challenges measurably better than the alternatives. I attribute this to several factors, though my analysis is necessarily speculative since I cannot see inside these systems. Claude appears to have been trained with particular attention to reasoning through complex problems rather than pattern-matching to likely completions. Its responses show evidence of something that functions like understanding rather than sophisticated prediction. When Claude encounters a failing test, it does not simply try variations until something works; it forms a hypothesis about why the test might be failing, investigates that hypothesis, and implements a targeted fix. This is what distinguishes a useful AI collaborator from a fast autocomplete.

The interface through which I work with Claude matters as much as Claude's underlying capabilities. The tool I use, called opencode, provides Claude with a persistent shell session, file system access, and the ability to invoke command-line tools. This is crucial because it gives Claude the ability to act, not merely advise. Claude can modify files, run tests, commit to git, create pull requests, and check deployment status—all without requiring me to copy-paste commands or manually execute its suggestions. The friction reduction is substantial: instead of Claude telling me what to do and me doing it, Claude simply does it and I review the results.

This combination—Claude's reasoning capabilities plus an interface that allows autonomous action—is what makes the methodology I describe in this book possible. Either element alone would be insufficient. A highly capable AI that could only suggest code for me to copy and paste would still leave me as the bottleneck. An autonomous interface connected to a less capable AI would produce a torrent of plausible-looking but subtly wrong output. The conjunction is where the transformation happens.

I want to acknowledge that this will change. The specific tools and systems I recommend today will likely be superseded within months. Claude's competitors are improving rapidly. The interfaces available for AI interaction will proliferate. The reader should understand this book as describing what works *now*, in late 2025, with the explicit recognition that the landscape will look different soon. The principles I articulate—giving AI maximal autonomy within defined guardrails, optimizing the development environment for AI effectiveness rather than human convenience, maintaining velocity through aggressive automation—will remain valid even as the specific implementations evolve.

Chapter 4: The Implications Are Uncomfortable

I have delayed discussing implications because I wanted to establish the factual basis first. Too many discussions of AI proceed from speculation about what might eventually be possible rather than observation of what already works. But having established that basis, the implications demand examination, and they are not comfortable.

The software industry employs millions of people worldwide. The economics of that industry rest on assumptions about how much human labor is required to produce and maintain software. If those assumptions are wrong by a factor of ten—if one developer with AI can do what ten developers did before—the consequences are severe for many of the people currently employed in software development.

I do not make this observation with any satisfaction. Many software developers are my friends and former colleagues. The Bitbucket team I worked with at Atlassian included skilled, dedicated people who took pride in their craft. The suggestion that much of what they did could now be accomplished by a single person with AI tools is not an insult to their abilities; it is a statement about how dramatically the technology has shifted.

The honest accounting looks something like this: in a traditional software organization, substantial engineering effort goes to coordination rather than creation. Meetings to align on requirements. Code reviews to ensure quality. Documentation to enable knowledge transfer. Sprint planning to divide work appropriately. These activities exist because human developers have the limitations I described earlier. Remove those limitations—or more precisely, substitute an entity that does not share them—and the coordination overhead largely evaporates.

Additionally, much of what individual engineers do is not creative problem-solving but implementation of well-understood patterns. Given clear requirements, implementing a CRUD API, building a React component, writing database migrations—these tasks do not require novel insight. They require careful attention to detail and familiarity with the relevant frameworks. AI excels at exactly this kind of work.

What remains for human developers? The parts that current AI cannot do well: understanding ambiguous requirements and asking clarifying questions, making judgment calls about what should be built, designing novel architectures for unprecedented problems, taking responsibility for systems that affect people's lives. These are valuable skills, and developers who possess them will remain valuable. But they constitute a smaller fraction of total software development labor than the industry has historically employed.

I want to be clear that I am describing a transition, not an overnight apocalypse. Organizations move slowly. Existing codebases require maintenance. Regulated industries have compliance requirements that slow adoption of new practices. The job losses will be gradual, uneven, and partially masked by continued growth in overall software demand. But the direction is unambiguous.

For individual developers, the strategic implications are significant. The skills that will remain valuable are not the same skills that were valuable five years ago. Deep expertise in a particular programming language or framework is less important when AI can work competently in any of them. The ability to coordinate large teams is less important when teams can be much smaller. What matters more is the ability to understand problems at a high level, to make good judgment calls about system design, to communicate effectively with AI systems, and to identify when AI output is wrong or inappropriate.

For organizations considering how to adapt, the key insight is that smaller teams moving faster will outcompete larger teams moving slower. The coordination overhead of traditional software organizations is a liability, not an asset. Companies that figure out how to operate with radically smaller engineering teams will have structural cost advantages that are difficult to overcome.

For society broadly, the implications extend beyond the software industry. Software developers have been among the most highly compensated knowledge workers. Their economic security has supported consumption, tax revenue, and a certain class-based stability. Disruption of that employment base will have ripple effects that are difficult to predict.

None of this is reason to slow down or pretend the change is not happening. But it is reason for honesty about what the change entails. Technological progress creates winners and losers. The responsible approach is to acknowledge this directly rather than hiding behind euphemisms about "augmentation" and "assistance."

Part Two: The Methodology

Chapter 5: The Foundation: Development Environment

The effectiveness of AI-assisted development depends critically on the development environment. This is not a matter of personal preference or minor optimization. The wrong environment creates friction that degrades AI effectiveness; the right environment removes friction and enables the AI to work at full capacity. The difference between these states is the difference between modest productivity gains and transformational change.

The foundation is a Unix-like operating system where the command line is a first-class interface. In practice, this means macOS or Linux. Windows can be made to work with the Windows Subsystem for Linux, but the experience is inferior. The reason is simple: AI systems like Claude are text-based. They process text input and produce text output. The most natural way for them to interact with a computer system is through command-line interfaces that follow the same paradigm. Graphical interfaces, which are optimized for human visual processing and motor control, add translation overhead that serves no purpose when the user is an AI.

The terminal emulator should support multiple panes and persistent sessions. I use iTerm2 with tmux, though alternatives exist. The critical capability is having several concurrent views into the system state—perhaps one pane showing Claude working, another showing test output, a third with log files—without the overhead of window management. tmux also provides session persistence: if the connection drops, the session continues running and can be reattached. This matters less for local development but becomes important when working with remote systems.

Every significant service in the development stack must have a command-line interface. This principle admits no exceptions because any service that cannot be operated from the command line cannot be operated effectively by AI. The services I use—GitHub for version control and project management, Vercel for web hosting and serverless deployment, Neon for PostgreSQL database hosting, Cloudflare for edge infrastructure—all provide comprehensive CLI tools. When I ask Claude to create a pull request, it runs `gh pr create`. When I ask it to check deployment status, it runs `vercel ls`. The AI's access to these systems is identical to my own access, mediated through the same command-line tools.

This principle extends to the development workflow itself. Tests run from the command line. Linting runs from the command line. Type checking runs from the command line. Builds run from the command line. Nothing critical happens through IDE buttons or graphical interfaces. The consequence is that Claude can verify its own work: after making a change, it can run the test suite and see whether the tests pass. This ability to close the feedback loop autonomously is what enables Claude to iterate toward correct solutions without human intervention at each step.

The choice of IDE matters less than one might expect, precisely because so much of the work happens through the command line. I use VS Code, but this is largely for my own benefit when reviewing Claude's work. Claude itself does not interact with the IDE; it reads and writes files directly. The editor is a viewing and occasional manual editing tool, not a locus of development activity.

Source control is git, hosted on GitHub. This is close to universal in modern software development, and for good reason: git's model of distributed version control, branches, and merges maps well to how software development actually works. The GitHub CLI tool (`gh`) provides programmatic access to pull requests, issues, actions, and repository management. Claude uses these extensively—creating branches, opening pull requests, checking CI status, commenting on issues. The GitHub web interface exists but is secondary to the command-line workflow.

The database hosting choice of Neon deserves specific mention. Neon provides PostgreSQL as a service with a feature critical for this workflow: database branching. Just as git allows branching code, Neon allows branching databases. This means Claude can test migrations against a copy of production data without risk to the production system. The Neon CLI provides commands to create branches, reset them, promote them to production. This integrates cleanly with the PR-based workflow: each pull request can have its own database branch where schema changes are tested before merging.

Finally, the AI interface itself. opencode provides Claude with persistent shell access, file system access, and tool invocation. The specific capabilities matter: Claude can read files, write files, execute commands, and wait for their output. It can make multiple tool calls in parallel when the tasks are independent. It maintains context across a conversation, remembering what files it has modified and what the state of the system should be. This is different from chat-based interfaces where each interaction is relatively independent; the persistence is what enables complex, multi-step development tasks.

Chapter 6: The Workflow in Practice

The previous chapter described the environment; this chapter describes how work actually flows through that environment. The methodology differs substantially from traditional software development processes, and understanding the differences requires seeing the workflow in action.

Work originates as GitHub issues. This is not merely a project management choice; it establishes a persistent record of intent that survives context window limitations and session boundaries. When I identify something that needs to be done—a feature to implement, a bug to fix, an improvement to make—I create an issue describing it. The description includes enough context that Claude can understand what is needed without reference to prior conversations. This discipline pays off when sessions restart: Claude can read the issue and proceed without me reconstructing verbal context.

Issues proceed through a lifecycle: open issues represent the backlog, issues assigned to a milestone represent current work, closed issues represent completed work. Claude can query this state through the `gh` CLI and prioritize accordingly. I do not maintain a separate sprint planning process or elaborate prioritization framework. The simplicity is deliberate: overhead that serves coordination in larger teams serves no purpose when the team is one human and one AI.

Implementation proceeds through pull requests. Claude creates a branch, makes changes, opens a pull request, and pushes commits. Each pull request runs through CI—the automated checks I will describe shortly—and I review the results. If CI passes and the implementation looks correct, I merge. If not, Claude iterates: reading the error messages, diagnosing the problem, implementing a fix, pushing another commit. This cycle—implement, check, fix—often completes multiple times before I am involved at all.

The pull request description matters more than in traditional development. Because Claude writes these descriptions, and because they form the record of what changed and why, the quality of these descriptions reflects directly on the system's ability to maintain coherent history. I have configured Claude to write substantive PR descriptions that explain not just what changed but why the change was made. This documentation is generated automatically as part of the workflow, not added as an afterthought.

Code review in this context has a different character than traditional code review. In a traditional team, code review serves multiple purposes: catching bugs, ensuring consistency with coding standards, spreading knowledge across the team, mentoring junior developers. When the developer is an AI, some of these purposes do not apply. Claude does not need mentoring. Knowledge does not need to spread because Claude already understands the entire codebase. What remains is verification: confirming that the implementation actually does what it should do, catching errors that slipped past the automated checks, ensuring that the approach makes sense at a higher level.

My review process therefore focuses on different things than traditional code review. I spend less time examining syntax and style—Claude is consistent about these, and linting catches most issues anyway. I spend more time examining logic and architecture: does this approach make sense? Will it scale appropriately? Does it handle edge cases correctly? Are there security implications I should consider? These are judgment questions that automated systems cannot currently answer, and they are where human review adds genuine value.

The frequency of merges is much higher than in traditional development. Where a traditional team might merge a few PRs per day, Claude might open a dozen and I might merge several per hour during active development. Each PR is typically smaller—a single feature, a single bug fix,

a single refactoring—because the overhead of creating and merging PRs is negligible when it is automated. Smaller PRs are easier to review, easier to revert if something goes wrong, and create a cleaner history. The traditional argument for batching work into larger PRs—reducing coordination overhead—does not apply when there is no coordination overhead to reduce.

Deployment follows a staging model: changes merge to a development branch, which deploys to a preview environment where I can verify behavior. Promotion to production is a separate step, gated on verification in preview. This is conventional practice, but the speed is not: the cycle from merge to preview deployment to production deployment might take fifteen minutes in total, most of that consumed by build and deployment automation rather than human decision-making.

A word about working hours and pace. One of the adjustments I have had to make is accepting that Claude can work while I am not present. If I identify a batch of issues before ending a session, Claude can work through them. I return to find PRs waiting for review, some already merged after passing CI. This is disorienting at first—the code is progressing without me—but it is the logical extension of giving AI meaningful autonomy. The constraint is my review bandwidth, not Claude's implementation bandwidth.

Chapter 7: Automated Verification: The Guardrails

I have mentioned CI—continuous integration—several times without fully explaining what it contains. This explanation is overdue because the automated verification systems are not mere conveniences; they are the foundation that makes AI autonomy possible. Without robust automated checks, AI autonomy would be reckless. With them, AI autonomy is transformative.

The principle is simple: no code reaches production without passing automated verification. But the implementation is extensive, and the specific checks matter greatly.

The first layer is formatting and linting. These tools enforce consistency in code style—indentation, spacing, naming conventions, import ordering. They catch trivial errors like unused variables or missing semicolons. Modern linting tools also catch some classes of substantive errors: detecting potential null pointer exceptions, identifying security vulnerabilities in common patterns, enforcing type safety in dynamically-typed languages. Claude can and does make formatting errors; the linter catches them immediately.

The second layer is type checking. Care Commons uses TypeScript throughout, which provides static type analysis. The type system catches a large class of errors at compile time: calling functions with wrong argument types, accessing properties that do not exist, returning values that do not match declared return types. Type checking is particularly valuable for AI-generated code because the type system serves as executable documentation—the types describe what the code should do, and the type checker verifies that the code actually does it.

The third layer is automated testing. Tests verify behavior at multiple levels. Unit tests verify that individual functions work correctly in isolation. Integration tests verify that components work correctly together. End-to-end tests verify that the system as a whole behaves correctly from a user's perspective. Each test represents an assertion about what the software should do, and each test failure indicates either a bug in the code or an outdated assumption in the test.

I maintain high test coverage requirements—typically above 80% of code exercised by tests. In traditional development, such requirements are sometimes criticized as arbitrary targets that encourage writing low-value tests to hit a number. This criticism has merit for human developers, who must balance test-writing time against other priorities. It has less merit for AI developers, who can write tests quickly and rewrite them easily when requirements change. The tests serve not just as verification but as executable specification: they document what the code is supposed to do in a form that can be automatically verified.

The fourth layer is security scanning. Automated tools check for known vulnerabilities in dependencies, scan for security-sensitive patterns in code, and verify that authentication and authorization are implemented correctly. These checks are particularly important for healthcare software, where security failures can have serious consequences.

The fifth layer is build verification. The code must not only pass all checks but actually compile and build successfully. This catches errors that the other layers might miss: dependency conflicts, build configuration problems, deployment issues.

All of these checks run automatically on every pull request. They also run as pre-commit hooks—checks that execute before code can even be committed to git. This catches errors earlier, before they enter the PR workflow at all. The pre-commit hooks are mandatory; they cannot be bypassed. This might seem draconian, but it ensures that every commit represents code that passes basic verification.

The effect of these guardrails is to provide fast, reliable feedback on every change. Claude makes a change, runs the checks, and knows immediately whether the change is acceptable. If a test fails, Claude can see the failure message, diagnose the cause, and fix it. If the linter complains, Claude can see what rule was violated and correct it. This feedback loop enables autonomous iteration: Claude can work through a complex change, adjusting and refining until all checks pass, without human intervention at each step.

Without these guardrails, AI autonomy would be dangerous. Claude would produce code that looks correct but contains subtle bugs. The bugs would reach production and cause problems. I would lose trust in Claude's output and find myself reviewing everything in detail, negating the productivity benefits. The guardrails make trust possible.

A note on the tradeoff involved: maintaining extensive automated checks requires ongoing investment. Tests must be updated when requirements change. Linting rules must be tuned to avoid false positives. Type definitions must be maintained as the codebase evolves. This maintenance cost is real, but it is smaller than it would be for human developers because Claude can do most of it. When I change a requirement and a dozen tests need updating, Claude updates them. The investment pays off through the autonomy it enables.

Chapter 8: Git Workflow and Branch Strategy

Version control strategy interacts closely with the AI-driven workflow in ways worth examining carefully. The choices I have made reflect the characteristics of AI development rather than tradition or personal preference.

The repository has three long-lived branches: `develop`, `preview`, and `production`. These correspond to three environments: development, preview, and production. Code flows in one direction: from develop to preview to production. Each promotion is a deliberate step, typically a pull request that I review and merge.

There is no `main` branch. This is deliberate and sometimes confuses people familiar with common git workflows. The `main` branch convention assumes a single source of truth for what is "current." In this workflow, there are three sources of truth for three different purposes: develop is current for development, preview is current for pre-production verification, production is current for live users. The three-branch model is more explicit about this reality.

Feature branches are short-lived. Claude creates a branch, implements a feature, opens a PR, and I merge it—often within an hour. The branch is deleted after merge. There is no concept of long-running feature branches that accumulate changes over days or weeks. Long-running branches create merge conflicts; merge conflicts require human judgment to resolve; avoiding merge conflicts reduces friction. The rapid merge cadence prevents branches from diverging far enough to create significant conflicts.

Every commit that reaches any of the long-lived branches has passed CI. Branch protection rules enforce this: pull requests cannot be merged unless CI passes. This means I can have confidence that any commit on develop, preview, or production represents tested, verified code. This confidence is what enables rapid promotion between environments.

Claude has full push access to develop. It can create branches, push commits, open pull requests, and even merge them if CI passes. This level of access might seem concerning, but consider the safeguards: CI must pass, which means all automated checks must succeed. I can see every PR and every commit in the GitHub interface. Nothing reaches production without my explicit action to promote it. The autonomy is real but bounded.

Direct commits to develop are acceptable for small changes. This differs from many team workflows, which require PRs for everything. The reasoning: PRs exist to enable code review, and code review exists to catch errors that the author missed. When the author is AI and comprehensive automated checks exist, the marginal value of a PR for a small change is limited. I still use PRs for anything substantial, but a trivial fix—correcting a typo, updating a version number—can go directly.

Commit messages follow a conventional format: present tense, imperative mood, brief description of what changed. This format was not chosen arbitrarily; it is the format that appears most often in git documentation and well-maintained open source projects, which means it is the format Claude is most comfortable producing. I could insist on a different format, but the cost of deviation from Claude's natural inclinations is not worth the benefit.

Pull request descriptions are more substantial than commit messages. A good PR description explains the motivation for the change (why), the approach taken (how), and any notable decisions or tradeoffs. Claude generates these descriptions as part of opening the PR. The quality has improved over time as I have given feedback on what makes a useful description.

I rebase rather than merge when bringing in changes from upstream. This keeps the git history linear, which makes it easier to understand the sequence of changes and to bisect when tracking down bugs. Rebasing can be more complex than merging when there are conflicts, but conflicts are rare given the short-lived nature of branches.

The overall effect is a version control system that provides full traceability—every change is recorded with its context and motivation—while minimizing the overhead traditionally associated with rigorous version control. The sophistication is in the automation, not in the process.

Chapter 9: When Claude Gets Stuck

The previous chapters might suggest that AI-driven development proceeds smoothly from requirement to implementation without difficulty. This is not the case. Claude gets stuck. Claude makes mistakes. Claude sometimes produces code that is subtly wrong in ways that pass automated tests but fail in production. Understanding when and why these problems occur is essential to working effectively.

The most common cause of difficulty is ambiguous requirements. If I ask Claude to "make the search faster," the request is ill-defined: faster by how much? faster under what conditions? at what cost in complexity or resource usage? Claude will make assumptions to fill in the gaps, and those assumptions may not match my intent. The solution is specificity. "Add an index on the user_id column to improve query performance for the user search endpoint. Verify with EXPLAIN that the query uses the index." This is unambiguous. Claude can implement it, verify it, and know that it has succeeded.

The second common cause is insufficient context. Claude's understanding of the codebase comes from reading it, but reading a large codebase is not the same as having built it over months or years. There are patterns and conventions that are implicit, not written anywhere. There are historical reasons why things are done certain ways. There are constraints from external systems that are not documented in the code. When Claude lacks this context, it may make changes that are locally sensible but globally inappropriate. The solution is to make context explicit. Document the patterns. Explain the constraints. When reviewing code that seems off, explain why and add documentation so the issue does not recur.

The third common cause is hitting the limits of AI capability. Some problems require genuine novelty—approaches that do not appear in Claude's training data. Some problems require deep domain expertise that Claude lacks. Some problems require understanding the needs and preferences of actual users, which Claude cannot know. For these problems, AI assistance has less value. Human judgment must drive the solution, with AI helping to implement decisions that the human has made.

When Claude does get stuck—repeatedly failing to fix a test, or producing code that doesn't work—the worst response is to keep asking it to try again. Repeated attempts without new information are unlikely to succeed. Instead, I try to diagnose why Claude is stuck. Is the requirement unclear? I clarify it. Is context missing? I provide it. Is this a problem beyond Claude's capability? I take over that aspect and let Claude help with the parts it can handle.

There is also the phenomenon I think of as "confident wrongness." Claude can produce code that looks correct, that passes tests, but that has a subtle flaw. Perhaps it misunderstood a requirement. Perhaps it implemented a standard pattern that does not quite fit the situation. Perhaps it optimized for the wrong thing. These errors are harder to catch because they do not trigger obvious failures. The defense is careful review of logic, especially for code that deals with critical paths: authentication, authorization, financial transactions, health data. Trust but verify, and verify most carefully where errors would be most consequential.

An important mindset shift: when Claude makes errors, the appropriate response is usually to improve the systems that should have caught the error, rather than to reduce Claude's autonomy. If a bug reaches production, the question is not "why did Claude make this mistake?" but "why did the tests not catch it?" The tests can be improved. Claude will then be less likely to make similar mistakes in the future because the tests will catch them earlier. This investment in verification systems has compounding returns.

Chapter 10: The Human Role

If Claude does the implementation, what does the human do? This question has no single answer because the human role varies depending on the problem, the human's strengths, and the current state of the project. But there are patterns.

The human sets direction. What should be built? Why does it matter? What are the priorities when tradeoffs are necessary? These are judgment questions that depend on understanding users, markets, and values. Claude can implement a feature, but choosing which features to implement requires understanding what will be valuable to users and viable as a business. This remains human work.

The human provides domain expertise. I understand home healthcare—the regulations, the workflows, the pain points that agencies face—because I have studied it and talked to people who work in the industry. Claude knows what is in its training data, which includes general information about healthcare but not the specific context of this project. When I explain that Texas has different EVV requirements than Florida, or that medicaid reimbursement works in a particular way, I am providing context that Claude cannot have derived from first principles.

The human makes architectural decisions that go beyond pattern matching. For standard problems, Claude's architectural instincts are good—it will suggest approaches that are common because they work well. For novel problems or unusual constraints, the human must think through the tradeoffs and provide guidance. The division is roughly: Claude handles tactical implementation decisions, the human handles strategic architectural decisions.

The human reviews for correctness at a level beyond what automated tests can verify. Tests check that code does what the tests expect. They do not check that the tests expect the right thing. A human reviewing a PR should be asking: does this actually solve the problem? Is this the right approach? Are there edge cases that the tests miss? This kind of review requires understanding the problem at a deeper level than "does the code pass tests."

The human handles interactions that require human judgment or relationships. Talking to potential users. Negotiating with partners. Making decisions about pricing and positioning. Writing content that requires voice and personality. These are not implementation tasks; they are the business activities that surround the software.

The human sets and enforces standards. What level of test coverage is acceptable? How should the code be organized? What practices are required versus optional? Claude operates within whatever constraints it is given; the human chooses the constraints. This is more powerful than it might seem because good constraints enable better AI performance. A strong type system catches errors that tests would otherwise need to catch. Consistent coding standards reduce cognitive overhead for both AI and human reviewers.

And the human takes responsibility. This is perhaps the most important point. When software fails, someone must be accountable. When decisions turn out to be wrong, someone must adjust course. When ethical questions arise, someone must make judgment calls. Claude is a tool, however sophisticated. The human is the moral agent who bears responsibility for how that tool is used.

The distribution of work I have described—Claude implementing, human directing and reviewing—is optimized for the current state of AI capabilities. As those capabilities advance, the boundary will shift. Claude will be able to make more decisions autonomously, handle more complex situations without guidance, perhaps eventually exercise something like judgment. But that future is not here

yet. For now, the human role remains substantial, and developers who imagine they can simply hand off problems to AI and walk away will be disappointed.

Chapter 11: Parallel Operations with Multiple AI Agents

A single human working with a single AI instance is already more productive than a traditional developer. But the model extends further: a single human can work with multiple AI instances in parallel, each operating on a different task. This multiplies the productivity gain again, though it introduces new challenges.

The setup I use involves multiple instances of Claude, each working in a separate terminal session, each focused on a different part of the codebase. They share the git repository through normal git mechanisms—branches, commits, pulls, pushes—but otherwise operate independently. Each has the same capabilities: file system access, command execution, ability to create and modify code.

Coordination is minimal because the work is divided to minimize dependencies. One instance might be working on frontend features while another works on backend API changes. As long as the interfaces between their areas are stable, they can proceed in parallel without stepping on each other. This is not different in principle from how human developers work on a team, but the scale is different: I can have several AI instances running simultaneously, each potentially completing tasks that would take a human developer hours.

The human role in parallel operations is primarily traffic control. Which AI instance should work on which problem? Are there dependencies between tasks that require sequencing? When one instance's work affects another's, who needs to pull updates and when? These are coordination questions, and while they are much simpler than the coordination problems on a large human team, they are not zero.

The challenges of parallel operation are mostly around merge conflicts and race conditions. If two instances modify the same file simultaneously, their changes may conflict. The git workflow handles this—the second one to push will need to rebase—but it adds friction. The solution is to divide work so that instances operate on largely separate parts of the codebase, only coordinating at well-defined interfaces.

There is also a cognitive overhead for the human. Monitoring multiple parallel streams of work, reviewing multiple PRs, keeping track of what each instance is doing—this is more mentally demanding than working with a single instance. The productivity gain from parallelism is real, but it is not linear. Two instances are not twice as productive as one instance because the coordination overhead grows.

I use this approach selectively rather than all the time. For complex, interconnected changes, a single instance working carefully is more effective than multiple instances creating coordination problems. For independent tasks that happen to need doing around the same time, parallel operation makes sense. The judgment about when to parallelize is itself a skill that develops with practice.

The current implementation of this—multiple terminal sessions, each with opencode connected to Claude—is somewhat crude. I expect better tooling to emerge that makes multi-agent coordination more seamless. But even the crude version demonstrates the principle: human plus AI is productive, human plus multiple AIs is more productive still.

Part Three: The Economics and Ethics

Chapter 12: What Software Costs Now

The economics of software development have been remarkably stable for decades. Yes, tools have improved. Yes, languages have become more expressive. Yes, cloud infrastructure has reduced operational costs. But the fundamental equation—skilled labor, applied over time, to produce software—has not changed. A person-year of software development has cost roughly what a skilled professional's labor costs, adjusted for whatever market and geography apply.

This stability is ending. The cost of producing software is falling dramatically, and the fall has only begun.

Consider what it cost to build software like Care Commons using traditional methods. The core platform—multi-tenant architecture, role-based access control, HIPAA-compliant audit logging, regulatory compliance features, web and mobile interfaces—represents perhaps 100,000 lines of code across the various packages. A traditional estimate might be 20-30 lines of production code per developer per day, accounting for design, testing, debugging, meetings, and other overhead. Call it 4,000 developer-days, or roughly 15-20 person-years of effort.

At fully-loaded costs for senior software developers—salary, benefits, equipment, office space, management overhead—perhaps $200,000 per person-year. The traditional cost to build this software would be $3-4 million, plus ongoing maintenance.

What did it actually cost? My direct expenses over several months of development: roughly $200 per month in API costs for Claude, hosting costs approaching zero on free tiers, and my own time. If I were to value my time at market rates for senior developers, the total would still be under $50,000. The cost reduction is not 20% or 50%; it is 95% or more.

This is not a temporary anomaly that will correct as AI vendors raise prices. The marginal cost of AI inference continues to fall as hardware improves and competition intensifies. The floor is well above zero—electricity and silicon are not free—but nowhere near the cost of human labor.

The implications for software businesses are profound. Companies that derive value from proprietary software face a problem: their moat is evaporating. Any competent developer with AI tools can replicate most software functionality in a fraction of the time and cost it originally required. The defensibility shifts from code to other factors: network effects, data assets, brand, regulatory compliance, integration partnerships.

For software buyers, the implication is that prices should fall. If software costs 95% less to produce, competitive pressure should eventually transmit that cost reduction to customers. The transition

will be messy—existing vendors will resist, switching costs are real, enterprise sales cycles are long—but the direction is clear.

For society, the implication is that software might become something closer to a commodity or public good. When production costs fall enough, the economic case for proprietary software weakens. Why pay for something that could be produced cheaply and shared freely? This argument has always existed in the open source community, but it has been countered by the practical reality that producing high-quality software required significant investment. That counterargument is now much weaker.

I want to be careful not to overstate the case. Not all software becomes cheap to produce at the same rate. Software that requires novel research, deep domain expertise, or massive data collection remains expensive. The AI tools I am using are themselves expensive to create—Anthropic has raised billions in capital. The cost structure is shifting, not collapsing to zero.

But for a very large category of software—business applications, web services, mobile apps, internal tools—the economics have changed fundamentally. This category includes most of the software that most developers work on. The implications for employment, for industry structure, for what gets built and by whom, are only beginning to unfold.

Chapter 13: Why Open Source, Why Now

Given the economics I have described, why give software away? If Care Commons represents hundreds of thousands of dollars of value, why publish it under an open source license? This question deserves a direct answer.

The first reason is that the moat does not exist. If I can build this software in months, someone else can too. Any attempt to profit through proprietary licensing would face competition from developers who could replicate the core functionality using the same tools I used. The competitive advantage, if there is one, cannot lie in the code.

The second reason is that open source accelerates adoption. For Care Commons specifically, the target users are home healthcare agencies, many of which are small businesses operating on thin margins. They are unlikely to adopt expensive proprietary software from an unknown vendor. They are more likely to try open source software that they can evaluate without commitment, deploy without ongoing licensing costs, and modify if their needs differ from the default behavior.

The third reason is ethical. Home healthcare serves vulnerable populations: elderly people, people with disabilities, people who need care to live independently. The agencies that serve these populations are often underresourced. If the cost of producing this software has fallen dramatically, those savings should flow to the people who need care, not be captured as excess profit by a software vendor.

The fourth reason is that open source enables different business models. The software is free. Services around the software are not. Hosted deployments for agencies who do not want to operate their own infrastructure. Enterprise support for organizations with complex requirements. Customization and integration for specific use cases. Training and consulting. A marketplace for add-on modules developed by third parties. These revenue streams exist whether the core software is open source or proprietary.

There is a pattern here from earlier in my career. At Zenoss, we built commercial open source enterprise software—open core model, with a free community edition and paid enterprise features. The open source base enabled distribution and adoption that proprietary software would have struggled to achieve. The commercial layer enabled the company to sustain development and support. This model is not new. What is new is that AI makes the open source portion vastly cheaper to produce, shifting the economics further toward open models.

I do not claim that open source is the right choice for all software or all situations. Proprietary software will continue to exist where defensible advantages exist, where network effects create winner-take-all dynamics, where the target market will pay for convenience. But for an increasing range of software, the economics now favor open approaches. The cost of production has fallen enough that giving away the artifact and monetizing the service around it makes sense.

This has broader implications. When software becomes cheap to produce and openly available, the limiting factor shifts from capital to distribution and expertise. Who can get the software to users who need it? Who can help those users successfully adopt it? These are still valuable activities, but they are different activities than building proprietary software and defending it from competition.

The gold rush metaphor is apt. Anthropic is selling shovels—the AI tools that enable this transformation—and they will profit accordingly. The prospectors using those shovels face a different situation. Individually, we may strike gold. Collectively, we are flooding the market with gold,

driving down its price, making it available to people who could never afford it before. Whether this makes individual prospectors rich depends on factors other than how much gold they can dig.

Chapter 14: The Disruption of Software Teams

I have deferred a direct discussion of what this transformation means for software teams and the people who work on them. The deferral was not because the question is unimportant but because I wanted to establish the factual basis first. Having done so, the implications are stark.

The typical software team exists because the work of building software exceeds what any individual can do in a reasonable timeframe, and because the limitations of human cognition require dividing and coordinating work. A team of ten engineers can do more than ten times what one engineer can do, because specialization and parallel work overcome some individual limitations—but also less than ten times, because coordination overhead consumes substantial effort. Teams are a tradeoff, accepted because the alternative of solo development was not viable for substantial software.

That alternative is now viable. The numbers are not precise, but directionally clear: a solo developer with AI can do the work of something like five to ten traditional developers. The coordination overhead that consuming half or more of a large team's effort disappears. The consistency and availability advantages of AI over human teammates add further gains.

The obvious implication is that teams will shrink. The same output will require fewer people. Some of that labor will be redeployed to expanded output—more software getting built, new projects that would not have been economical before—but not all of it. The net effect on employment is negative for software developers as a category.

I want to be clear that I take no pleasure in this observation. I have spent my career on software teams, first as an individual contributor and later in leadership roles. The Bitbucket team at Atlassian was one of the best teams I have worked with—smart, collaborative, dedicated people doing good work. The suggestion that teams like that might be rendered unnecessary is not a celebratory observation but a sober one.

But sober observations are still observations. If the economics and capabilities have changed as I have described, the industry will adjust, whether anyone wants it to or not. Companies that figure out how to operate with smaller teams will have lower costs. They will outcompete companies clinging to larger teams out of habit or sentiment. The adjustment might take years, and it will be uneven across different types of software and different geographies, but the direction is determined by economics.

For individual software developers, the strategic implications are significant. The skills that commanded premium compensation—deep expertise in a specific language or framework, ability to work effectively on large teams, experience with established development processes—are less valuable now. The skills that are becoming more valuable are: ability to communicate effectively with AI systems, judgment about what software should do and how it should be designed, expertise in domains where AI has less training data, and the entrepreneurial ability to create value independently rather than as part of a large organization.

This is not an indictment of traditional software engineering skills. Knowing how to write good code, design clean architectures, and debug complex systems remains important. But it is no longer sufficient. The developers who will thrive are those who combine traditional skills with effective use of AI tools—who become force multipliers rather than replaceable units of labor.

For organizations, the implication is that team structure requires rethinking. The fifteen-person team with multiple roles—junior developers, senior developers, tech leads, engineering managers, product managers, designers—made sense when human labor was the primary input. It may not

make sense when AI can do much of the implementation work. Smaller teams with flatter structures, where every person is exercising judgment rather than implementing decisions made by others, may be more effective.

I do not know exactly what the new equilibrium looks like. The transformation is underway, not complete. But I am confident that software organizations in five years will look very different from software organizations today, and that the change will not be gentle for those unprepared for it.

Chapter 15: The Path Forward

I have argued that a fundamental transformation is underway in how software gets built. I have described the methodology that enables that transformation. I have explored the economic and social implications. What remains is to discuss what to do about it.

For developers who want to adopt this methodology, the path is straightforward: set up the environment I have described, start using AI tools for real work, and develop the skills of effective AI collaboration. The learning curve is real but not steep. Someone experienced in traditional software development can become productive with this approach in weeks, not months. The main barrier is not technical but psychological—accepting that the AI can do more than one initially expects, and adjusting one's role accordingly.

Start with a greenfield project if possible. Applying this methodology to an existing codebase works, but the AI's lack of historical context creates friction. A new project allows the AI to build understanding from the beginning, establishing patterns and conventions that it understands because it created them. Care Commons was a greenfield project for exactly this reason.

Be aggressive with test coverage from the start. Tests are the foundation that enables AI autonomy. Every test is a guardrail that prevents classes of errors from reaching production. The investment in testing pays off not eventually but immediately, in the form of AI that can iterate to correct solutions without human intervention at each step.

Set up CI/CD early. The automation infrastructure is not a nice-to-have; it is essential. If the AI cannot verify its own work, you are the bottleneck. If the AI can verify its own work, you are free to do higher-value activities while it iterates.

Develop the habit of explicit communication. When something does not work as you expected, do not just ask the AI to fix it. Explain why it does not work, what you expected, and what was wrong about the AI's assumptions. This information improves future work on the same codebase. The AI learns about your project through what you tell it; incomplete communication produces incomplete understanding.

Accept that the AI will do things differently than you would. This is fine. Unless there is a technical reason to prefer your approach, let the AI use the patterns it knows best. Your preferences are not optimized for AI productivity; the AI's preferences are. The exception is when your preferences reflect genuine architectural requirements or domain knowledge—then insist, and explain why.

For organizations considering how to adapt, the key insight is that this transformation cannot be ignored or adopted halfway. Organizations that add AI tools to existing processes without changing the processes will see modest gains. Organizations that redesign their processes around AI capabilities will see transformative gains. The choice is not whether to change but how much.

Pilot projects are a reasonable starting point. Take a bounded piece of work, assign it to a small team or single developer with AI tools, and compare the results to traditional approaches. The data will speak for itself. Let the evidence drive broader adoption rather than mandating change from above.

Consider how role definitions need to evolve. If AI does the implementation, what do developers do? The answer is not that developers become unnecessary but that developers do different things: more design, more review, more judgment, less typing. Job descriptions, performance expectations, and career paths all need updating.

For society broadly, the implications are beyond any individual or organization to address. But they are not beyond discussion. The changes I have described will affect employment, industry structure, the economics of software, and the accessibility of technology to solve problems. Some of these effects are positive, some are negative, and many are uncertain. Engaging seriously with these implications, rather than ignoring them or dismissing them as hype, is itself a contribution.

What I have described is not science fiction. It is not a vision of what might be possible someday. It is what I am doing, right now, documented through a real project that anyone can examine. The transformation is underway. The question is not whether to participate but how.

Part Four: The Practice

Chapter 16: Setting Up the Development Environment

The previous chapters established why this methodology works and what its implications are. This chapter and those that follow turn to practical details of implementation. They are written to be useful to someone setting up this kind of development environment for the first time.

The operating system should be macOS or Linux. The command-line interface is not merely a preference; it is fundamental to how AI systems interact with development tools. The alternative—Windows with Windows Subsystem for Linux—works but adds friction and complexity. If you have a choice, choose a Unix-native system.

The terminal emulator should support multiple panes and persistent sessions. iTerm2 on macOS is excellent. The key capabilities are: splitting the terminal into multiple views that can be observed simultaneously, scrollback buffer large enough to review AI output, and integration with tmux for session persistence. Install tmux and configure it according to your preferences; the defaults are reasonable starting points.

Install the following command-line tools:

Git, for version control. This is likely already present but should be the current stable version.

The GitHub CLI (`gh`). This provides command-line access to GitHub features including repositories, pull requests, issues, and actions. Authenticate it with `gh auth login` and verify with `gh auth status`.

Node.js version 22 or later, managed through nvm to allow switching between versions if needed. Node is the runtime for the JavaScript ecosystem that much of modern web development depends on.

The Vercel CLI (`vercel`). This provides deployment and management of Vercel-hosted applications. Authenticate with `vercel login`.

The Neon CLI (`neon` or `neonctl`). This provides management of Neon PostgreSQL databases. Authenticate with `neon auth`.

The Wrangler CLI for Cloudflare Workers if you use Cloudflare infrastructure. Authenticate with `wrangler login`.

These tools follow a pattern: they are official CLIs provided by the services, they authenticate through their respective platforms, and they provide programmatic access to capabilities that would otherwise require web interfaces. The pattern is what matters; the specific services can be substituted if you use different hosting providers.

For the AI interface itself, install opencode. This is the tool that provides Claude with shell access and file system capabilities. Installation is straightforward with npm or the method described in the opencode documentation. Verify the installation by running it and checking that you can interact with Claude.

Configure git with your identity and preferred settings. At minimum:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
git config --global init.defaultBranch develop
```

The default branch name is `develop` rather than `main` because our workflow uses `develop`, `preview`, and `production` as the three long-lived branches.

Create a new directory for your project and initialize it as a git repository:

```
mkdir my-project
cd my-project
git init
```

Create a GitHub repository and link it:

```
gh repo create my-project --public --source=. --remote=origin
```

The repository is now ready for development. The following chapters describe how to structure the codebase, configure automation, and begin working with Claude.

Chapter 17: Project Structure and Configuration

The structure of a project affects how easily Claude can understand and modify it. This chapter describes a structure that has worked well for Care Commons and explains the reasoning behind the choices.

The root directory contains configuration files that apply to the entire project: package.json for Node.js dependencies and scripts, tsconfig.json for TypeScript configuration, .gitignore for files that should not be committed, and similar project-wide settings. These files are read frequently by Claude and should be kept clear and consistent.

The source code is organized into packages following a monorepo pattern. A monorepo contains multiple related packages in a single repository, managed together but independently versioned. This structure has several advantages for AI-driven development: all code is in one place for Claude to read, dependencies between packages are explicit, and shared tooling configuration applies across the entire project.

The packages directory contains the main application packages:

- `packages/core` for shared domain logic, database access, and utilities
- `packages/app` for the backend API server
- `packages/web` for the frontend web application
- `packages/mobile` for the React Native mobile application

The verticals directory contains domain-specific modules that implement particular features:

- `verticals/scheduling-visits` for visit scheduling
- `verticals/time-tracking-evv` for electronic visit verification
- `verticals/billing-invoicing` for billing features

This separation keeps the core packages focused on infrastructure while the verticals contain business logic. Claude can work on a vertical without needing to understand all the others.

Each package has a consistent internal structure:

- `src/` for source files
- `src/__tests__/` for test files
- `dist/` for compiled output (gitignored)
- `package.json` for package-specific dependencies
- `tsconfig.json` extending the root configuration

TypeScript is used throughout. The benefits for AI-driven development are substantial: the type system serves as machine-checkable documentation, type errors catch whole categories of bugs at compile time, and the IDE support enables better code understanding. The strict TypeScript configuration is enabled—no implicit any, strict null checks, and similar settings that catch errors early.

The `.gitignore` file excludes:

- `node_modules/` - installed dependencies
- `dist/` - compiled output
- `.env` - environment variables containing secrets
- Coverage reports and other generated files

Nothing in gitignore should be needed to understand or build the project. Everything necessary is committed.

Configuration files that Claude needs to understand should be in standard formats: JSON or YAML rather than custom formats. The more conventional the configuration, the better Claude understands it.

A file named `CLAUDE.md` or `AGENTS.md` in the root directory contains instructions for Claude. This file describes:

- Project structure and conventions
- How to run tests and other verification
- Coding standards and patterns to follow
- Domain knowledge specific to the project
- Things to avoid or be careful about

Claude reads this file at the start of each session and uses it to guide behavior. Maintaining this file is an ongoing task: as the project evolves, the instructions should evolve too.

Chapter 18: Automated Verification Configuration

The automated verification systems are the guardrails that make AI autonomy possible. This chapter describes how to configure them.

The package.json in the root directory defines scripts that run verification:

```
{
  "scripts": {
    "lint": "turbo run lint",
    "typecheck": "turbo run typecheck",
    "test": "turbo run test",
    "build": "turbo run build"
  }
}
```

Turbo is a build system for monorepos that runs tasks across packages in parallel while respecting dependencies. Each package defines its own versions of these scripts, and turbo orchestrates running them.

ESLint is configured for linting. The configuration should be strict: enable rules that catch real problems, disable rules that create noise. A starting point:

```
{
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  "rules": {
    "no-unused-vars": "error",
    "@typescript-eslint/no-explicit-any": "error",
    "@typescript-eslint/no-floating-promises": "error"
  }
}
```

The specific rules matter less than consistency and comprehensiveness. The goal is that any code passing lint is free of common errors.

TypeScript type checking is configured through tsconfig.json. The strict setting enables all strict mode checks:

```
{
  "compilerOptions": {
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true
  }
}
```

Strictness catches more errors at compile time, which is particularly valuable for AI-generated code.

Testing uses Vitest, a fast test runner compatible with the modern JavaScript ecosystem. Configuration in vitest.config.ts:

```
export default {
  test: {
    coverage: {
      reporter: ['text', 'json', 'html'],
      threshold: {
        lines: 80,
        functions: 80,
        branches: 80,
        statements: 80
      }
    }
  }
}
```

The coverage thresholds are enforced: if coverage drops below 80%, the test run fails. This ensures that test coverage is maintained as the codebase grows.

Pre-commit hooks run verification before code can be committed. Using Husky:

```
{
  "husky": {
    "hooks": {
      "pre-commit": "npm run lint && npm run typecheck && npm run test"
    }
  }
}
```

The hooks cannot be bypassed without explicitly disabling them, which should never be done. Any commit that reaches the repository has passed local verification.

GitHub Actions configuration runs verification on every pull request. A workflow file at `.github/workflows/ci.yml`:

```
name: CI
on: [pull_request]
jobs:
  verify:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '22'
      - run: npm ci
      - run: npm run lint
      - run: npm run typecheck
      - run: npm run test
      - run: npm run build
```

Branch protection rules require CI to pass before merging:

```
gh api repos/{owner}/{repo}/branches/develop/protection -X PUT \
  -f required_status_checks='{"strict":true,"contexts":["verify"]}' \
  -f enforce_admins=true
```

The effect of this configuration is that no code reaches the protected branches without passing all verification steps. This creates the foundation for AI autonomy: Claude can implement changes and verify them locally, confident that if verification passes, the changes are acceptable.

Chapter 19: Working with Claude: Practical Patterns

Having established the environment and configuration, this chapter describes practical patterns for working with Claude effectively.

Begin each session by orienting Claude to the current state of the project. Claude has persistent knowledge of the codebase from reading files, but it does not have memory across sessions. A starting prompt like:

"I'm continuing work on Care Commons. Please read AGENTS.md and familiarize yourself with the current state. Then check `gh issue list` to see what needs to be done."

This primes Claude with the project-specific context it needs.

Assign tasks at the right level of abstraction. Too high-level and Claude may make assumptions you do not intend. Too detailed and you are doing work that Claude could do. The sweet spot is something like:

"Implement the endpoint for creating a new visit. It should validate that the caregiver is assigned to the client, that the visit time does not conflict with existing visits, and that the service type matches the client's care plan. Write tests for the validation logic."

This specifies what the feature should do without specifying how to implement it. Claude can choose the implementation approach and verify its own work through tests.

When Claude's implementation is not what you expected, explain why rather than just asking for changes. If Claude used the wrong pattern:

"The validation logic should be in the service layer, not the route handler, because we want to reuse it from the mobile API. Please move it to VisitService and update the route handler to call the service."

This explanation helps Claude understand the architectural reasoning and apply it to future decisions.

Review diffs rather than full files when reviewing Claude's work. Claude produces a lot of output. Reading all of it in detail is not feasible and not necessary. The diffs show what changed, which is what matters for review.

```
git diff HEAD~1
```

Look for changes that do not make sense, patterns that are inconsistent with the rest of the codebase, and anything that suggests Claude misunderstood the requirement.

Use the test suite as your primary verification. If tests pass, the implementation is likely correct. If tests fail, the failure messages tell you what is wrong. You should not be manually verifying that code works—that is what tests are for.

When Claude gets stuck, provide more context rather than more instructions. Claude's errors usually stem from missing information: an implicit requirement that was not stated, a pattern that exists elsewhere in the codebase but was not mentioned, a constraint from the domain that Claude does not know. Adding that information is more productive than giving step-by-step instructions.

Let Claude commit and push its own work. This might feel uncomfortable initially, but it is the logical extension of giving Claude autonomy. With CI verification, there is no risk: bad commits

will not reach protected branches because CI will fail. The benefit is that you are not a bottleneck for routine commits.

End sessions cleanly. If there is work in progress, have Claude commit it to a branch:

"Commit your current progress to a branch called feature/in-progress-visit-endpoint and push it. Add a description in the commit message of what is done and what remains."

This preserves state for the next session and creates a reference point if something goes wrong.

Chapter 20: Bot Accounts and Multi-Instance Coordination

For larger projects or faster velocity, you may want multiple instances of Claude working in parallel. This chapter describes how to set up and coordinate multiple AI agents.

Each Claude instance should have its own GitHub identity. This is accomplished through bot accounts—GitHub accounts that exist for automation rather than individual users. Create separate accounts for each instance you plan to run:

```
Account: my-project-bot-1
Email: my-project-bot-1@example.com
```

For each bot account, create a Personal Access Token with permissions for repositories, pull requests, and issues. Store these tokens securely.

Configure git in each terminal session to use the appropriate bot identity:

```
git config user.name "my-project-bot-1"
git config user.email "my-project-bot-1@users.noreply.github.com"
```

Authenticate the GitHub CLI with the bot's token:

```
echo $BOT_TOKEN | gh auth login --with-token
```

Now commits and GitHub API calls from that session appear to come from the bot account. This enables tracking which instance made which changes.

Divide work to minimize overlap. If one instance is working on frontend features and another on backend API changes, they can proceed in parallel with minimal coordination. The division should follow natural boundaries in the codebase.

Each instance should pull from the shared repository before starting significant work:

```
git pull --rebase origin develop
```

This ensures they are starting from the current state, reducing merge conflicts.

When conflicts do occur—two instances modified the same file—resolve them promptly. The instance that pushed second will need to rebase and resolve. Claude can usually handle merge conflicts, but they add friction. Better division of work reduces their frequency.

For coordination that requires real-time awareness—one instance waiting for another to complete a dependency—use GitHub issues or pull request comments. Instance A can create a PR, and instance B can watch for that PR to be merged before proceeding with dependent work.

The human role in multi-instance coordination is primarily monitoring and traffic direction. Which instance should work on which problem? Are there dependencies between tasks that require sequencing? When problems arise, which instance should stop and which should continue?

A practical pattern: dedicate one terminal pane to monitoring. Watch the PR list, review as things come in, merge what passes CI. Other panes show individual instance work. This allows observation of multiple streams while maintaining ability to intervene when needed.

The gains from parallelism are real but diminishing. Two instances are not twice as productive as one because coordination overhead grows. In my experience, two to three parallel instances is a

sweet spot for a single human to effectively monitor. More than that and the cognitive overhead of tracking everything becomes the bottleneck.

Chapter 21: The CI/CD Pipeline in Detail

Continuous integration and continuous deployment are central to this methodology. This chapter describes a complete CI/CD configuration suitable for production use.

The CI pipeline runs on every pull request and push to protected branches. It consists of several jobs that run in parallel where possible:

Setup job: Installs dependencies and caches them for subsequent jobs.

```
setup:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: '22'
        cache: 'npm'
    - run: npm ci
    - uses: actions/cache@v4
      with:
        path: node_modules
        key: node-modules-${{ hashFiles('package-lock.json') }}
```

Lint job: Runs ESLint across all packages.

```
lint:
  needs: setup
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/cache@v4
      with:
        path: node_modules
        key: node-modules-${{ hashFiles('package-lock.json') }}
    - run: npm run lint
```

Type check job: Runs TypeScript type checking.

```
typecheck:
  needs: setup
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/cache@v4
      with:
        path: node_modules
        key: node-modules-${{ hashFiles('package-lock.json') }}
    - run: npm run typecheck
```

Test job: Runs the test suite with coverage reporting.

```yaml
test:
  needs: setup
  runs-on: ubuntu-latest
  services:
    postgres:
      image: postgres:16
      env:
        POSTGRES_PASSWORD: test
      ports:
        - 5432:5432
  steps:
    - uses: actions/checkout@v4
    - uses: actions/cache@v4
      with:
        path: node_modules
        key: node-modules-${{ hashFiles('package-lock.json') }}
    - run: npm run test -- --coverage
    - uses: codecov/codecov-action@v4
      with:
        files: ./coverage/lcov.info
```

Build job: Verifies that production builds succeed.

```yaml
build:
  needs: [lint, typecheck, test]
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/cache@v4
      with:
        path: node_modules
        key: node-modules-${{ hashFiles('package-lock.json') }}
    - run: npm run build
```

The deployment pipeline runs when code is merged to specific branches:

Preview deployment: Triggers when code merges to the `preview` branch. Deploys to a preview environment where the full application can be tested.

```yaml
deploy-preview:
  if: github.ref == 'refs/heads/preview'
  needs: build
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - run: vercel pull --yes --environment=preview
    - run: vercel build
    - run: vercel deploy --prebuilt
```

Production deployment: Triggers when code merges to the `production` branch. Deploys to the production environment with appropriate safeguards.

```yaml
deploy-production:
  if: github.ref == 'refs/heads/production'
  needs: build
  runs-on: ubuntu-latest
  environment: production
  steps:
    - uses: actions/checkout@v4
    - run: vercel pull --yes --environment=production
    - run: vercel build --prod
    - run: vercel deploy --prebuilt --prod
```

The `environment: production` setting in the production deployment job enables GitHub's environment protection rules. These can require manual approval before deployment proceeds, add waiting periods, or limit who can approve production deployments.

Branch protection rules tie everything together:

```
gh api repos/{owner}/{repo}/branches/develop/protection -X PUT \
  -f required_status_checks='{"strict":true,"contexts":["lint","typecheck","test","build"]}' \
  -f required_pull_request_reviews='{"required_approving_review_count":0}' \
  -f enforce_admins=true
```

Note that required approving reviews is set to zero. In a solo developer workflow, requiring approval from another person does not make sense. The CI checks provide the verification; human review is optional but not required for the system to function.

Chapter 22: Database Migrations and Schema Management

Database schema changes require particular care because they affect persistent data. This chapter describes how to manage migrations safely in an AI-driven workflow.

Use a migration framework that tracks which migrations have been applied. The specific choice—Knex, Prisma, Drizzle, or plain SQL files with a tracking table—matters less than consistency. Care Commons uses Knex migrations.

Each migration is a file with an up function (applies the change) and a down function (reverses it):

```
export async function up(knex: Knex): Promise<void> {
  await knex.schema.createTable('visits', (table) => {
    table.uuid('id').primary();
    table.uuid('client_id').references('clients.id');
    table.uuid('caregiver_id').references('caregivers.id');
    table.timestamp('scheduled_start').notNullable();
    table.timestamp('scheduled_end').notNullable();
    table.timestamps(true, true);
  });
}

export async function down(knex: Knex): Promise<void> {
  await knex.schema.dropTable('visits');
}
```

Migration files are named with timestamps to ensure ordering:

```
20251101120000_create_visits_table.ts
20251101130000_add_visit_status_column.ts
```

When Claude needs to modify the database schema, it creates a new migration file rather than modifying existing ones. Applied migrations are immutable—changing them would create inconsistency between environments that have run them.

Database branching enables safe testing of migrations. With Neon:

```
neon branch create --name feature/new-schema
```

This creates a copy of the database that can be modified without affecting the original. Claude can run migrations against the branch, test that they work correctly, and then the branch can be promoted or deleted.

The workflow for schema changes:

1. Claude creates a migration file
2. Claude runs the migration against a database branch
3. Claude runs tests to verify the application works with the new schema
4. The migration is committed along with application code changes
5. When merged, CI runs migrations against the preview database
6. Production migration runs when code is promoted to production

Migrations should be backwards-compatible when possible. Adding a new column with a default value is backwards-compatible: old code ignores the new column, new code uses it. Removing a

column is not backwards-compatible: old code that references it will break.

For non-backwards-compatible changes, use a multi-step process:

1. Deploy code that handles both old and new schema
2. Apply the migration
3. Deploy code that uses only the new schema
4. Remove the compatibility code

This is more complex than single-step changes, but it avoids downtime and rollback complications.

Claude should include comments in migrations explaining the purpose:

```
/**
 * Add status tracking for visits.
 * Status values: scheduled, in_progress, completed, cancelled
 * Existing visits default to 'completed' for backwards compatibility.
 */
export async function up(knex: Knex): Promise<void> {
  await knex.schema.alterTable('visits', (table) => {
    table.string('status').defaultTo('completed');
  });
}
```

These comments become documentation of how the schema evolved.

Chapter 23: Security Considerations

Security in AI-driven development requires specific attention because the attack surface differs from traditional development. This chapter addresses the key considerations.

Secrets must never appear in code or logs. API keys, database credentials, authentication tokens—all of these must be managed through environment variables or secrets management systems. Claude knows this principle and generally follows it, but verification is essential. Any PR that might contain secrets should be reviewed carefully.

Environment variables are managed through .env files locally and through the hosting platform's secrets management for deployed environments:

```
# .env (gitignored, never committed)
DATABASE_URL=postgresql://user:password@host/database
API_SECRET=your-secret-key

# Set in Vercel
vercel env add DATABASE_URL
```

Claude should never have access to production credentials. The credentials Claude uses for development and testing should be for non-production environments only. This limits blast radius if something goes wrong.

Dependency security requires ongoing attention. The npm ecosystem has had security vulnerabilities in popular packages. Automated scanning helps:

```
npm audit
```

This should run as part of CI. High-severity vulnerabilities should block merging until addressed.

The code Claude produces may contain security vulnerabilities, just as human-written code might. The categories are familiar: injection attacks, authentication bypass, information disclosure. Automated security scanning tools can catch some of these:

```
- name: Security scan
  uses: github/codeql-action/analyze@v2
  with:
    languages: javascript
```

But automated tools do not catch everything. Human review should pay particular attention to:

- Authentication and authorization logic
- Input validation and sanitization
- Data access patterns (is data properly scoped to authorized users?)
- Error messages (do they reveal sensitive information?)

Rate limiting protects against abuse. Any public-facing API should limit request rates:

```
import rateLimit from 'express-rate-limit';

app.use('/api/', rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // limit each IP to 100 requests per windowMs
}));
```

HTTPS is non-negotiable for production. Vercel and similar platforms provide this automatically. For custom deployments, certificates from Let's Encrypt are free and can be automated.

Audit logging records security-relevant events: who accessed what data, when, and from where. For healthcare software like Care Commons, this is not just good practice but a regulatory requirement. Claude can implement audit logging, but the requirements should be specified by a human who understands the compliance context.

Security reviews should happen periodically even when nothing has obviously changed. Schedule time to review authentication flows, authorization checks, and data access patterns. Look for patterns that seemed correct when implemented but might have become problematic as the codebase evolved.

Chapter 24: Monitoring and Operations

Software in production requires monitoring. This chapter describes the operational aspects of running AI-developed software.

Health checks verify that the application is running correctly. A simple health endpoint:

```
app.get('/health', async (req, res) => {
  const dbHealthy = await checkDatabase();
  const cacheHealthy = await checkCache();

  if (dbHealthy && cacheHealthy) {
    res.status(200).json({ status: 'healthy' });
  } else {
    res.status(503).json({
      status: 'unhealthy',
      db: dbHealthy,
      cache: cacheHealthy
    });
  }
});
```

External monitoring services can poll this endpoint and alert when it returns unhealthy status.

Structured logging enables analysis and debugging:

```
import pino from 'pino';

const logger = pino({
  level: process.env.LOG_LEVEL || 'info'
});

logger.info({ userId, action: 'login' }, 'User logged in');
```

Logs should include enough context to understand what happened without including sensitive data. User IDs are fine; passwords are not.

Error tracking services capture exceptions in production:

```
import * as Sentry from '@sentry/node';

Sentry.init({ dsn: process.env.SENTRY_DSN });

// Errors are automatically captured
// Additional context can be added:
Sentry.setUser({ id: userId });
```

When errors occur, these services provide stack traces, context, and frequency information that helps diagnose and prioritize fixes.

Performance monitoring tracks response times and identifies slowdowns:

```
app.use((req, res, next) => {
```

```
  const start = Date.now();
  res.on('finish', () => {
    const duration = Date.now() - start;
    logger.info({
      method: req.method,
      path: req.path,
      status: res.statusCode,
      duration
    }, 'Request completed');
  });
  next();
});
```

Patterns in this data reveal performance problems before users complain.

Alerting should be configured for conditions that require attention:

- Health check failures
- Error rate spikes
- Response time degradation
- Resource utilization thresholds

The alerting threshold should be calibrated to avoid alert fatigue. Too many alerts and they get ignored; too few and real problems go unnoticed.

Deployment rollback capability is essential. When a production deployment causes problems, you need to be able to revert quickly:

```
vercel rollback
```

This reverts to the previous deployment. The previous deployment still exists and can be restored instantly, without rebuilding.

Database rollback is more complex. If a migration has modified data, rolling back the code does not roll back the data changes. This is why backwards-compatible migrations are preferred: rolling back the code leaves the database in a state that the old code can still work with.

Incident response procedures should exist before incidents occur. Who gets notified? What are the escalation paths? What information should be gathered? Having these decisions made in advance reduces response time when problems occur.

Conclusion: What Comes Next

I began this book with a claim that traditional software development is being disrupted. Having described the methodology, the evidence, and the implications, I want to conclude by looking forward.

The transformation I have described is in its early stages. The tools are improving rapidly. What is difficult today will be routine tomorrow. The capabilities that seem remarkable now will be baseline expectations within a few years. Anyone who assumes the current state represents a stable equilibrium will be surprised.

For developers, the immediate priority is adaptation. Learn to work effectively with AI tools. Develop the judgment skills that complement AI implementation capabilities. Position yourself as a force multiplier rather than a replaceable implementer. This is not optional career advice; it is survival guidance for a profession undergoing fundamental change.

For organizations, the priority is experimentation. The companies that figure out how to operate with AI-augmented development will have structural advantages that compound over time. Waiting to see how others do it first is a strategy, but it cedes initiative to more aggressive competitors.

For the software industry broadly, we are entering a period of significant disruption. Employment patterns, business models, and competitive dynamics are all shifting. Some of this will be painful. Some of it will create opportunities that did not exist before. The distribution of pain and opportunity will not be even or fair.

I have tried in this book to be clear-eyed about both the capabilities and the implications of AI-driven development. The capabilities are real and substantial. The implications are uncomfortable for many people and organizations that have built careers and businesses on assumptions that no longer hold. Neither the capabilities nor the implications should be ignored or minimized.

The methodology I have described—Vibe Coding—is one approach to this new reality. It is the approach that has worked for me, documented through the concrete evidence of Care Commons. Other approaches will emerge. Some will be better. The specific recommendations in this book will date quickly as tools evolve.

What will not date quickly is the fundamental insight: AI can do the core intellectual work of software development, and this changes everything about how software should be built. The details of how to work with AI will evolve. The fact that AI is a capable collaborator rather than just a faster autocomplete—that is the durable insight.

I do not know exactly what software development looks like in five years. But I am confident it does not look like software development in 2020 or even 2023. The transformation is underway. The

evidence is visible to anyone who looks. The only question is whether we engage with it intentionally or let it happen to us.

Choose to engage.

Brian Edwards November 2025

brian.mabry.edwards@gmail.com

Care Commons: github.com/neighborhood-lab/care-commons

Appendix A: Tool Installation Reference

This appendix provides installation commands for the tools referenced throughout the book.

Homebrew (macOS package manager)

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Git

```
brew install git
```

GitHub CLI

```
brew install gh
gh auth login
```

Node.js via nvm

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
nvm install 22
nvm use 22
```

Vercel CLI

```
npm install -g vercel
vercel login
```

Neon CLI

```
npm install -g neonctl
neon auth
```

Wrangler (Cloudflare)

```
npm install -g wrangler
wrangler login
```

tmux

```
brew install tmux
```

iTerm2

```
brew install --cask iterm2
```

opencode

```
npm install -g @anthropic/opencode
```

Appendix B: Sample AGENTS.md File

This is a condensed example of the instructions file that Claude reads at the start of each session:

```
# Project: Care Commons

## Overview
Open-source home healthcare management platform. Multi-tenant SaaS with
HIPAA compliance, EVV support, and state-specific regulatory configurations.

## Structure
- packages/core: Shared domain logic and database
- packages/app: Express API server
- packages/web: React frontend
- packages/mobile: React Native mobile app
- verticals/: Domain-specific feature modules

## Commands
- npm run dev: Start development servers
- npm run test: Run test suite
- npm run lint: Run linting
- npm run typecheck: Run type checking
- npm run build: Production build
- ./scripts/check.sh: Run all checks

## Git Workflow
- develop: Integration branch, deploys to GitHub Pages
- preview: Pre-production, deploys to Vercel preview
- production: Live system, deploys to Vercel production

## Coding Standards
- TypeScript strict mode
- ESM imports with .js extensions
- Tests required for new functionality
- 80% coverage minimum

## Domain Notes
```

```
- Texas EVV: 100m geofence + GPS accuracy
- Florida EVV: 150m geofence + GPS accuracy
- HIPAA: All PHI access must be audit logged
- Multi-tenant: All queries must scope to organization_id
```

Appendix C: Recommended Reading

For those interested in deeper exploration of topics touched on in this book:

On AI Capabilities: - Anthropic's research publications on Claude's capabilities - Papers on large language model reasoning and tool use

On Software Engineering: - "The Mythical Man-Month" by Frederick Brooks (still relevant) - "Accelerate" by Nicole Forsgren et al. on development practices

On Open Source Economics: - "The Cathedral and the Bazaar" by Eric Raymond - Research on commercial open source business models

On the Future of Work: - Economic analyses of AI impact on employment - Historical studies of technological disruption

---

This book was written in November 2025. Given the pace of change in AI capabilities, specific recommendations may become outdated quickly. The principles should remain relevant longer.