AI-ASSISTED DEVELOPMENT: WORKING WITH CODING AGENTS

A WORKING ENGINEER'S GUIDE TO
CLAUDE CODE, AIDER, AND AUTONOMOUS CLI ASSISTANTS

BUILD BETTER SOFTWARE • FASTER • SAFER • TOGETHER WITH AI

by YOHAN J. RODRÍGUEZ

Preface

"The future is already here — it's just not evenly distributed."
William Gibson

For most of software's history, writing code meant typing every line yourself. That is changing. A new class of autonomous terminal coding agents — Claude Code, Aider, and a growing field of CLI assistants — can now read a codebase, plan a change, edit many files, run the tests, and hand you a diff to review. Used well, they compress hours of mechanical work into minutes. Used carelessly, they produce confident, plausible, and wrong changes faster than any human ever could.

This book is a practical, hands-on guide to using these agents *well*. It is written for working developers who want to accelerate their daily coding without surrendering control of quality, security, or their git history. You do not need a machine-learning background. You do need to be able to read a diff, run a test suite, and judge whether code is correct — because the central discipline of agent-assisted development is verification, not blind trust.

A word on scope. This book is about the single-engineer, single-agent workflow at the terminal: choosing a tool, setting it up safely, steering it with context files, sandboxing it, prompting it, reviewing its output, and folding it into a real git and CI workflow. Building Model Context Protocol servers in depth and orchestrating fleets of agents are their own subjects, cross-referenced where they touch this one. Because the tools change their flags and config formats every few months, the book anchors on durable patterns — context anchoring, sandboxing, declarative prompting, diff auditing — and pushes the fastest-moving setup specifics into the companion repository.

Every script, sandbox, and lab here is meant to be run, not just read; the companion configs are tested against Debian 12 and 13. The goal is not to memorize one vendor's command set. It is to build a durable working method you can carry to whatever agent you use next.

Yohan J. Rodriguez

Contents

1 | The Landscape — Claude Code, Aider, Cursor, and Local Tools

A few years ago, "using AI to write code" meant one of two things: an editor that finished your line before you did, or a chat window in a browser tab where you described a problem and pasted back whatever came out. Both were useful. Neither changed the fundamental shape of the work. You were still the one moving every character into the file, running the tests, reading the stack trace, and deciding what to do next.

That has changed. A new category of tools can now do something the earlier generation could not: they can *operate inside your project*. They read your files, build a model of how your code fits together, make edits across multiple files, run your test suite, read the failures, and try again — all before they hand anything back to you. The interface is no longer "ask a question, get a snippet." It is closer to "assign a task, review the result."

This chapter is a map of that landscape. The goal is not to crown a winner or to walk you through any single tool's menus. Tools change names, change owners, gain and lose features, and reorganize their command-line flags between the time a book is written and the time you read it. What does not change as quickly is the *shape* of the tools: what category a tool belongs to, what that category is good and bad at, and which kind of work each one is appropriate for. If you understand the categories, you can pick up any new tool in an afternoon and know roughly where it fits and where it will hurt you.

By the end of this chapter you should be able to tell the difference between autocomplete and an agent, explain why a terminal-based agent is a different kind of animal from a web chat window, decide when each category is the right call, and assemble a practical starter stack for your own work. Most importantly, you should finish with a healthy respect for what these tools cannot do on their own — because the rest of this book is about building the discipline that makes them safe and productive.

Do not confuse code generation with software engineering.

Producing plausible code is the easy 20% of the job. The other 80% — understanding the system, choosing the right boundaries, anticipating failure, keeping the change reviewable,

and being accountable for what ships — is still yours. Every tool in this chapter is a force multiplier on the first part. None of them is a substitute for the second.

1.1 The Shift From Chat Assistance to Agentic Development

The first widely used AI coding tools were autocomplete engines. You typed, and the editor offered to finish the current line or block. It was a genuine productivity gain — it removed a lot of typing and a lot of trips to documentation for routine syntax — but it was fundamentally a *suggestion* layer sitting on top of the way you already worked. The model saw a narrow window around your cursor and guessed the next few tokens. It did not know what your application did, why it existed, or whether the suggestion it offered was consistent with the rest of the codebase. It could not run anything. It could not check itself.

The second phase was chat. Large language models became good enough at reasoning about code that you could describe a problem in plain language and get back something useful: an explanation of an error, a draft of a function, a comparison of two approaches. Chat was a step up in capability because the model could now reason about intent, not just continue a token sequence. But chat introduced a new and surprisingly expensive friction: the model lived in a separate window, disconnected from your actual project. You became the integration layer. You copied your code into the chat, copied the response back into your editor, fixed the imports it got wrong, adjusted the variable names it invented, and re-ran the tests by hand. This is the *copy-paste tax*, and on any real codebase it is heavy.

The third phase — the one this book is about — closes that gap. Agentic tools are given the ability to *act*. Instead of describing what you want and reintegrating the answer yourself, you delegate a task to something that can read the relevant files on its own, propose changes directly to those files, execute the commands needed to verify the work, observe the results, and correct course. The model is no longer a text generator you talk to. It is a worker you assign things to, operating inside a loop of action and feedback.

This is a meaningful shift for everyday engineering, and it is worth being precise about why. The difference is the move from *asking for code* to *delegating a task*. Those sound similar but they are different activities with different risk profiles. When you ask for code, you remain the operator: you decide where it goes, you wire it in, you run it, and any mistake the model made is caught at integration time because you are doing the integrating. When you delegate a task, you hand off the integration too. The tool decides which files to touch, makes the edits, and runs the checks. That is enormously more powerful and enormously more dangerous, because the model's mistakes now land directly in your working tree rather than being filtered through your hands.

Consider two requests that look superficially alike.

The first: *"Write a function that validates an email."* This is a code-generation request. Any tool from any category can answer it, and the answer will be roughly the same regex-and-checks function you have seen a hundred times. You will paste it somewhere, decide what to name it, figure out where it belongs, and make it consistent with everything around it. The tool did a small, well-scoped, low-context job, and you did all the engineering around it.

The second: *"Inspect this repository, find the validation layer, add email validation consistently with the existing validators, update the affected tests, and show me the diff."* This is a delegated task. To do it well, the tool has to discover where validation currently lives, learn the conventions the existing validators follow (do they throw, return a result object, accumulate errors?), apply the new validation in the same style, find and update the tests that exercise that layer, run those tests, and present the whole change as a reviewable diff. No amount of clever autocomplete or disconnected chat can do this, because it requires *acting on the real project* — reading it, changing it, and checking the change against reality.

The rest of this chapter is about the tools that can take on that second kind of request, how they differ, and when each one is the right choice. But hold on to the distinction, because it is the spine of the entire book; the value and the danger both come from the same source, which is that the agent acts on your real code.

1.2 A Taxonomy of AI Development Tools

It is tempting to talk about these tools by brand name, but brands are a poor way to reason about a fast-moving field. A better approach is to classify by capability — by what a tool can *see*, what it can *change*, and what it can *run*. Those three axes mostly determine both how useful a tool is and how much trouble it can cause.

Figure 1.1 sketches a taxonomy that will hold up even as specific products change.

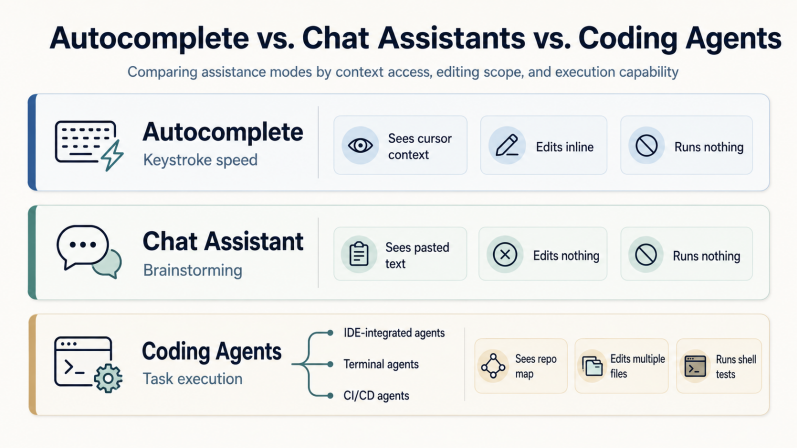

Figure 1.1: A capability-based taxonomy of AI development tools: suggestion-only, conversational, and agentic, classified by what each can see, change, and run.

The eight categories below are the working vocabulary for the rest of the book.

1. **Autocomplete engines.** These complete code as you type — a line, a block, sometimes a whole short function. *Good for* removing keystrokes on routine code. *Bad at* anything requiring knowledge of the wider system; they see only a small window around the cursor. *Typical risk:* confidently completing in a plausible-but-wrong direction that you accept on reflex. *Examples:* the inline-completion features built into modern editors and coding assistants.

2. **Chat assistants.** A conversational model in a browser or panel that reasons about whatever you give it. *Good for* explanation, design discussion, and isolated snippets. *Bad at* anything depending on files it cannot see; it works from your description, not your repository. *Typical risk:* plausible answers built on wrong assumptions about your actual code. *Examples:* general-purpose chat models accessed through a web interface.

3. **IDE-integrated agents.** Agentic behavior that lives inside your editor. They can read open or selected files, propose multi-file edits as inline diffs, and sometimes run limited commands. *Good for* tightly reviewed changes where you want to stay in the editor. *Bad at* large autonomous loops and work that spans the whole system. *Typical risk:* over-editing and hidden context — changing more than you expected based on files you did not realize were in scope. *Examples:* editor "agent modes" and IDE-embedded assistants.

4. **Terminal coding agents.** Command-line tools that operate on the whole repository, edit many files, and run shell commands to verify their work. *Good for* serious, multi-file, test-backed engineering tasks. *Bad at* situations where you have not set up guardrails — they are the most capable and therefore the most dangerous. *Typical risk:* destructive or far-reaching commands, and changes that outrun your ability to review them. *Examples:* terminal-first coding agents and git-aware command-line assistants.

5. **Repository mapping tools.** Not always a separate product — often a capability inside category 4 — these build a structured map of your codebase (files, symbols, definitions, references) so an agent can decide what to read. *Good for* helping an agent find the right 1% of a large repo. *Bad at* nothing in particular; the risk is in what the agent does with the map, not the map itself. *Examples:* the indexing layers inside repository-aware agents.

6. **Local/offline coding assistants.** Agents and completion/chat tools powered by open-weight models running on your own hardware. *Good for* privacy, offline work, and predictable cost. *Bad at* the hardest reasoning and longest-context tasks, where the strongest cloud models still lead. *Typical risk:* over-trusting a weaker model on a high-stakes change. *Examples:* open-weight coding models served through local inference runtimes.

7. **CI/CD and automation agents.** Agents that run inside a pipeline rather than at your keyboard — opening fix-up branches, proposing dependency bumps, triaging failures. *Good for* routine, well-bounded, repeatable maintenance. *Bad at* judgment calls; an unsupervised agent in CI can do damage at scale and speed. *Typical risk:* automated changes merging or deploying without a human in the loop. *Examples:* bot-style agents wired into pull-request and pipeline events.

8. **Custom tool-using agents (via MCP or similar).** Agents extended with custom tools through a protocol — for example, the Model Context Protocol — so they can query a database, hit an internal API, or read a ticketing system. *Good for* grounding an agent in your real systems instead of its training data. *Bad at* staying safe without careful permissioning; every new tool is new capability and new attack surface. *Typical risk:* a tool that can read secrets or take real-world actions being driven by untrusted input. *Examples:* agents wired to custom tool servers; covered in depth in a later chapter.

The remaining sections of this chapter walk through these categories in the order a developer usually meets them — from the lowest-friction, lowest-power tools to the highest.

1.3 Autocomplete Tools: Fast, Narrow, and Low-Friction

Autocomplete is where most developers first encounter AI assistance, and it remains the easiest to adopt because it changes almost nothing about how you work. You write code the way you always have; the tool simply offers to finish what you started. You glance at the suggestion, accept it with a keystroke if it is right, and ignore it if it is not.

The defining characteristics of autocomplete are *inline suggestions*, *single-function completions*, *boilerplate generation*, and crucially, *developer-controlled acceptance*. Nothing happens unless you press the key. That single fact is what makes autocomplete the safest category in this chapter: the human is in the loop on every change, by construction, because every change is literally something you accepted character-by-block as you typed.

The flip side of that safety is limited power. Autocomplete does not understand your system deeply. It sees a window around your cursor — the current file, maybe a few related files the editor has pulled in — and predicts what is statistically likely to come next given that window. It has no model of your application's purpose, your architectural rules, or the invariants your team relies on. It will happily complete a database call in a style that contradicts your data-access conventions, or generate an error-handling pattern your codebase abandoned two years ago, because from where it sits the suggestion looks locally reasonable. The suggestion is plausible at the line level and sometimes wrong at the system level.

This is exactly why autocomplete is a precision tool, not a general one. It excels when the right answer is mostly determined by local context and does not require judgment about the wider system.

Best use cases for autocomplete:

- **Repetitive syntax** — loops, guards, getters and setters, the shape of a switch statement you have written a thousand times.
- **Small helper methods** — a date formatter, a string slugifier, a unit conversion, where the implementation is obvious and self-contained.
- **Test case scaffolding** — the boilerplate of arranging, acting, and asserting, where you still supply the meaningful assertions.
- **API call boilerplate** — constructing a request, setting headers, handling the obvious status codes in the conventional way.
- **Data mapping code** — copying fields from one struct or DTO to another, where the mapping is mechanical.

Avoid using autocomplete for:

- **Architectural decisions** — autocomplete has no idea what your architecture is and will not protect its boundaries.
- **Security-sensitive code** — authentication, authorization, cryptography, and input handling are exactly where a plausible-but-wrong line does the most damage.
- **Complex multi-file refactors** — autocomplete works a line at a time; it cannot coordinate a change that has to be consistent across many files.
- **Database migration chains** — order, reversibility, and data safety matter here, and none of that is visible in a cursor window.
- **Business rules that require domain context** — the tax calculation, the eligibility check, the pricing logic. The code is the easy part; knowing the rule is the job, and autocomplete does not know your rule.

In a professional workflow, autocomplete is best understood as a typing accelerator that you keep switched on all the time and trust for nothing important. It pays for itself on the thousand small completions a day and asks very little of you in return — as long as you remember that "I accepted it" means "I am responsible for it."

1.4 Chat-Based Coding Assistants: Useful but Disconnected

Chat assistants are the tool most developers reach for when autocomplete is not enough and they want to actually *think* with the model. And for thinking, chat is genuinely excellent. A good chat assistant is the best rubber duck ever built: you can paste an unfamiliar stack trace and get a clear explanation, describe a design problem and get three approaches with their trade-offs, ask why a particular language feature behaves the way it does, or work through an algorithm before you commit to writing it. For *understanding* and *planning*, chat is hard to beat.

The strengths cluster around tasks that do not require the model to know your specific codebase: explanation, learning, design alternatives, and isolated snippets. If the problem can be fully described in the chat window, chat handles it well. The model is reasoning, and reasoning is what it is good at.

The weaknesses all stem from one root cause: the chat assistant is *disconnected from your project*. It cannot see your files. It knows only what you paste, and what you paste is always a partial, possibly stale, possibly misleading slice of reality. This produces a cluster of failure modes — *stale context* (you pasted the function but not the three callers that constrain it), *missing files* (the model invents a helper that you already have, or assumes a config that does not exist), *manual copy-paste* as the only integration path, and *hallucinated assumptions* about structure the model cannot see. The model fills the gaps in its knowledge with confident guesses, and confident guesses about code you did not show it are frequently wrong.

Here is the kind of thing that happens constantly. Suppose you ask a chat assistant: *"How do I add caching to my user lookup so repeated requests don't hit the database?"* The model cannot see your project, so it answers from the average of everything it has seen. It produces a clean, well-commented example that wraps your lookup in an in-memory dictionary cache with a time-to-live, and it explains the approach nicely. The answer is plausible and, in a vacuum, correct.

It is also probably wrong for *your* system. Your application might run as several processes behind a load balancer, so an in-memory cache in one process is invisible to the others and produces inconsistent reads. You might already have a shared cache abstraction that every other part of the codebase uses, which the model could not know about and therefore bypassed, creating an inconsistency a reviewer will flag. Your user object might be invalidated by events the model has never heard of, so a naive time-to-live cache will serve stale data after a profile update. None of these are failures of the model's reasoning. They are failures of *context*. The model gave a good general answer to a question that had a specific answer it could not see.

This is the copy-paste tax in its most insidious form: not the keystrokes of moving text back and forth, but the *cognitive* cost of being the only thing in the loop that actually knows the project. Every chat answer about your codebase arrives needing to be reconciled with reality, and you are the one who has to do the reconciling.

None of this makes chat useless — far from it. It makes chat *the wrong sole interface for*

serious codebase work. Use it for what it is good at: planning a change before you make it, understanding a system you did not write, weighing designs, and demystifying errors. Then take that understanding to a tool that can actually see and act on your project. Chat is where you decide what to do. It should rarely be where you do it.

1.5 IDE-Integrated Agents: Convenient but Editor-Bound

Between disconnected chat and full terminal agents sits a comfortable middle ground: agents that live inside your editor. These are the "agent modes" and embedded assistants that can read the files you have open, propose changes as inline diffs, and apply edits across several files without you copying anything by hand. They bring agentic behavior — see, change, sometimes run — into the environment you already work in, with your syntax highlighting, your debugger, and your familiar keybindings right there.

The benefits are real and immediate. You stay inside the editor, so there is no context switch to a browser tab and back. The agent has *file awareness* — it can see what you are working on and the files related to it, rather than only what you remembered to paste. Changes arrive as *inline diffs* you can read in place, accept, reject, or tweak hunk by hunk. And because everything happens in the editor with the diff right in front of you, *developer review* is built into the natural flow of the work. For a developer who is nervous about handing too much control to a command-line tool, an IDE agent is an excellent on-ramp: it is meaningfully more powerful than autocomplete, but it keeps you visually anchored to every change.

IDE agents are a particularly good fit for frontend work and for small-to-medium refactors. When you are iterating on a component, tweaking layout and behavior, or renaming and rewiring a handful of files, the editor is exactly where you want to be, and seeing each proposed edit rendered inline matches the tight, visual feedback loop that kind of work wants.

But the editor that makes these tools convenient also bounds them, and there are risks worth naming. The first is *hidden context*: the agent decides which files to pull in, and that selection is not always visible to you, so a change can be informed by — or can touch — files you did not realize were in scope. The second is *over-editing*: asked for a small change, an agent may "helpfully" reformat, refactor, or "improve" code well beyond what you wanted, burying the one edit you cared about inside fifty you did not review carefully. The third is *unclear execution boundaries*: when an IDE agent can also run commands, it is not always obvious *which* commands it ran, against which environment, or what side effects they had — the editor abstracts that away, which is convenient until it is not.

The practical way to think about IDE agents is as a bridge. They take you from "the model suggests, I type" to "the model changes, I review" without yet asking you to manage a process that has free run of your shell. That is a valuable rung on the ladder. But for work that genuinely spans the whole system — that needs to run the full test suite, parse build output, and iterate

autonomously until everything is green — you will want the category that was built for exactly that, which is where we go next.

1.6 Terminal-Based Coding Agents: The New Power Tool

This is the category that changed what "AI-assisted development" means, and it deserves the most attention.

A terminal-based coding agent runs as a command-line program inside your project directory. From there it can do something none of the previous categories can do in full: it can treat your repository as its workspace and the shell as its hands. It inspects the repository to decide what is relevant. It reads and edits multiple files. It runs commands — your test runner, your build, your linter, your formatter, `git` itself. It reads what those commands print, including compiler errors and failing test output. And then it does the thing that makes it feel qualitatively different from a chat window: it *iterates*. It uses the feedback from the commands it ran to fix its own mistakes, and it keeps going until the task is done or it gets stuck — at which point it shows you a diff and waits.

Because a terminal agent operates at the level of "the project and the shell" rather than "the open file" or "the pasted snippet," it can work across the whole stack in a single task: backend code, frontend code, build scripts, configuration, documentation, and tests, all in one coherent change. That breadth is the source of its power. It is also the source of its danger, which is why this category demands stronger safety practices than any other — a subject the next several chapters are devoted to.

The mental model that makes terminal agents click is the *agent loop*. Where chat is a single request-and-response, a terminal agent runs a cycle, often many times over, within one task. Figure 1.2 shows the shape of it.

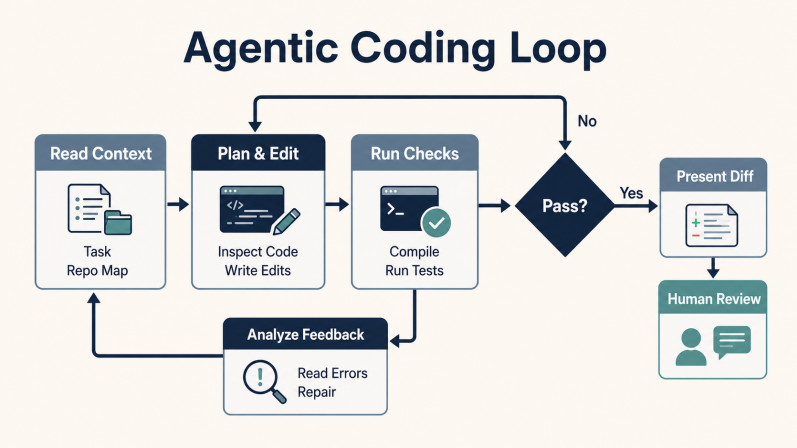

Figure 1.2: The agent loop. The agent reads context, edits code, runs checks, reads the errors, and repairs — looping until the checks pass — then presents a diff and waits for human review.

Walk through it once, because every later chapter builds on this picture. The agent *reads context* — the task you gave it, any project context files, the map of the repository. It *inspects files*, opening the code it judges relevant. It *plans* the change. It *modifies* the code, editing as many files as the task requires. It *runs checks* — and this is the pivotal step, the one chat cannot take — executing your tests and build. It *reads the errors* that come back. If something failed, it *repairs* and loops back to running checks, and it will go around this loop repeatedly, converging on a working state. When the checks pass, it *presents a diff* of everything it changed, and it *waits for human review*.

That loop is why terminal agents are powerful: the agent is not generating code in a vacuum and hoping. It is generating code, *checking it against reality*, and correcting. The terminal is the reality check. A model writing code with no feedback is guessing; the same model running the tests and reading the failures is doing something much closer to engineering.

A terminal agent without tests is just a confident text generator with shell access.

The entire value of the agent loop comes from the steps where it runs checks and reads errors. Remove the tests, and the loop has nothing real to push against. The agent will still confidently edit your files and still present a tidy diff, but "it ran and produced a diff" now means only that it produced text, not that the change is correct. The feedback loop is the product. Protect it.

It is worth being equally clear about what the agent loop does *not* do, because the loop can create a dangerous illusion of completeness. Closing the loop means the agent got the tests to pass. It does not mean the change is well-architected, secure, aligned with product intent, or something you should ship. The loop verifies *internal consistency against the checks you have*; it cannot verify that those checks were the right ones, that the design is sound, or that the feature is what the business actually needed. Those judgments remain entirely yours. The human stays responsible for architecture, for security, for product intent, and for final acceptance. The agent is a fast, tireless executor inside boundaries *you* set; it is not a replacement for the engineer who sets them.

The agent can move faster than your ability to review.

A terminal agent can touch twenty files and run the test suite five times in the span of a coffee refill. That speed is the point — and it is also the trap. If you let the agent run long, ambitious tasks and then try to review the result all at once, you will not actually review it; you will skim it, trust the green checkmark, and approve. The discipline that keeps you safe is to keep tasks small, keep diffs reviewable, and make the agent show its work in pieces

you can actually hold in your head. Speed without proportional review is not productivity. It is risk you have not noticed yet.

The arrival of terminal agents is what makes the rest of this book necessary. Autocomplete needed no methodology — you accept a line or you do not. Chat needed no methodology beyond "double-check it." But a tool that edits your files and runs your shell needs a real operating discipline: sandboxing so it cannot harm the host, context files so it understands your project, tests so the loop has something to verify against, diff review so you stay accountable, and git hygiene so every change is reversible. Those are the chapters ahead.

1.7 Claude Code, Aider, Cursor, and Similar Tools: How to Think About Them

Now we can place specific, well-known tools onto the map — carefully, because the goal is durable understanding, not a review that expires with the next release. Treat the names below as *examples of categories*, not as endorsements or as fixed feature lists. The specifics will drift; the category each tool exemplifies is the stable part.

Claude Code is a good example of a *terminal-first coding agent*. It is designed to run in the shell, work against the whole repository, edit files, execute commands, and iterate through the agent loop described above. When this book needs a concrete picture of "a terminal agent," this is the shape it has in mind: command line, repository-aware, able to act and verify.

Aider is a good example of a *git-aware, repository-mapping coding assistant*. It leans heavily on building a map of your codebase to decide what to read, and it integrates tightly with git — characteristically committing changes as it goes so that your history becomes the record of, and the undo button for, the agent's work. It exemplifies the idea that version control is not an afterthought but a core part of working safely with an agent.

Cursor-style tools are examples of *IDE-integrated agent environments*: the editor is the home base, edits appear as inline diffs, and the agent's reach is anchored to what you can see on screen. They exemplify the convenient middle ground of the previous section.

Copilot-style tools began as the canonical example of *autocomplete* and have increasingly grown *agentic* capabilities on top. They are a useful reminder that these categories are not walls — a product can start in one and extend into another, and many are converging toward "autocomplete plus an agent mode."

Local tools — open-weight models served through local runtimes — are the example of the *privacy-and-experimentation* category, covered in the next section. They exemplify trading raw capability for control, cost predictability, and keeping your code on your own machine.

The reason to hold these as categories rather than feature lists is practical: a year from now

the specific capabilities will have moved, but a terminal-first agent will still be a terminal-first agent, an IDE agent will still be editor-bound, and an autocomplete tool growing an agent mode will still carry the strengths and risks of both. Learn the categories and you can re-evaluate any tool quickly when its details change.

Table 1.1 summarizes the categories along the axes that actually matter — what a tool can see, change, and run, and therefore what it is for and what it can break.

Tool category	**Primary interface**	**Repository awareness**	**Can edit files?**	**Can run commands?**	**Best for**	**Main risk**	**Human review required?**
Autocomplete engine	Inline in editor	Local (cursor window)	Inserts at cursor	No	Routine syntax, boilerplate, small helpers	Reflex-accepting a plausibly wrong line	Per-suggestion, built in
Chat assistant	Web/panel chat	None (only what you paste)	No (you copy-paste)	No	Explanation, design, learning, isolated snippets	Confident answers on unseen context	Yes — you integrate everything
IDE-integrated agent	Editor agent mode	Open/ selected files	Yes, inline diffs	Limited	Frontend work, small–medium refactors	Hidden context, over-editing	Yes — review each diff
Terminal coding agent	Command line	Whole repository	Yes, many files	Yes, shell	Multi-file, test-backed engineering tasks	Destructive commands; outrunning review	Yes — review diff and commands
Repository mapping tool	Library/agent internal	Whole repo (as a map)	Via host agent	Via host agent	Helping an agent find the right files	Misleading the agent if the map is stale	Indirect (review the agent's output)
Local/offline assistant	CLI/editor/ local server	Depends on host tool	Depends on host tool	Depends on host tool	Private exploration, boilerplate, offline work	Over-trusting a weaker model	Yes — and more skeptically
CI/CD automation agent	Pipeline/PR bot	Repo + pipeline context	Yes, via branches/ PRs	Yes, pipeline steps	Routine, repeatable maintenance	Unsupervised change at scale	Yes — gated by PR review
Custom tool-using agent (MCP)	Terminal/editor + tools	Repo + connected systems	Yes	Yes, plus custom tools	Grounding an agent in real systems	Tools that read secrets or act for real	Yes — and permission tightly

Table 1.1: Tool category comparison matrix. Classified by what each category can see, change, and run — and therefore what it is for, what it risks, and how review is structured.

Notice that the last column is almost entirely "yes." That is not an accident, and it is the quiet thesis of this chapter: across every category capable of touching real code, a human in the loop is not optional. The categories differ in how *much* review they demand and how the review is structured — per-keystroke for autocomplete, per-diff for agents — but the requirement itself does not go away as the tools get more powerful. If anything, it intensifies.

1.8 Local and Open-Weight Coding Models

For most of the recent history of AI coding, the strongest tools were cloud services: you sent your prompt and your code to a provider's servers, and a large model answered. That is still where the frontier of capability lives. But a parallel track has matured quickly — *open-weight* coding

models that you can download and run on your own hardware — and for a meaningful set of tasks it is now a serious option rather than a curiosity.

The model side of this track is a family of open-weight code models. Without anchoring to any one version, the recognizable examples are the *Qwen-Coder*–style models, the *DeepSeek-Coder*–style models, and the older *CodeLlama*–style models — each a family of openly available weights tuned for code. On the runtime side, tools like Ollama and LM Studio, along with various local inference servers, make it straightforward to pull one of these models and serve it on your own machine, exposing it to your editor or to a local agent. The combination — open weights plus a local runtime — is what makes offline, on-premises coding assistance possible.

The appeal is straightforward and, for some teams, decisive.

- **Privacy.** Your code never leaves your machine or your network. For proprietary codebases, regulated industries, or organizations with strict data-handling policies, "the model runs locally" can be the difference between "allowed" and "forbidden."
- **Offline availability.** No network dependency means the assistant works on a plane, in an air-gapped environment, or during an outage of whatever cloud service you would otherwise depend on.
- **Cost control.** After the upfront hardware investment, the marginal cost of a request is essentially electricity. For heavy, continuous use, predictable local cost can compare favorably to recurring per-use cloud billing.

The constraints are equally real, and being honest about them is part of using local tools well.

- **Hardware limitations.** Running a capable code model locally needs a capable machine — generally a reasonably strong GPU with enough memory to hold the model, or an equivalent setup. As an *illustrative* example that will age, a developer laptop may comfortably run a small model but struggle with a large one; treat any specific memory figure you read as a snapshot, not a rule, because both models and hardware move quickly.
- **Weaker reasoning and context handling.** This is the honest core of the trade-off. The strongest cloud models still lead on the hardest reasoning and on holding long, complex context coherently. A local model is often very good at the routine and noticeably less reliable at the subtle. It may lose the thread on a large refactor, miss an implication that a frontier model would catch, or handle a smaller effective context.

That trade-off maps cleanly onto where local tools belong.
Good use cases for local/open-weight tools:

- **Private code exploration** — asking questions about a sensitive codebase you are not allowed to send to a cloud service.

- **Boilerplate** — the same routine generation autocomplete is good at, now available offline.
- **Small refactors** — bounded, mechanical changes where strong reasoning is not the bottleneck.
- **Offline experimentation** — trying ideas where connectivity or cost would otherwise be a constraint.
- **First-pass review** — a quick local sanity check before a human, or before escalating to a stronger model.

Poor use cases for local/open-weight tools:

- **Complex architectural reasoning** — exactly the kind of subtle, long-context judgment where the capability gap is widest.
- **High-stakes security changes** — where a weaker model's mistake is most expensive and you can least afford "almost right."
- **Large ambiguous refactors without tests** — broad scope, unclear requirements, and no safety net is the worst possible combination to hand to your least capable option.

The practical posture is to treat local models as a valuable, controllable part of the stack rather than as a lesser substitute for everything. They are the right tool when privacy, offline operation, or cost dominates and the task is within their range — and the wrong tool when the task's difficulty is the binding constraint. Many experienced developers end up running both: a local model for private, routine, or offline work, and a cloud agent for the hard problems. Choosing between them deliberately is the subject of the next section.

1.9 Cloud Agents vs. Local Agents

The cloud-versus-local decision is one you will make repeatedly, sometimes per task, and it deserves a clear-eyed comparison rather than a tribal allegiance. Both have genuine advantages; neither is universally correct.

The case for cloud agents. Cloud-hosted models lead on raw capability: *stronger reasoning* on hard problems, *better long-context handling* so they hold a large change coherently, and *better instruction following* so they stay on task through a complex, multi-step request. They put *less burden on local hardware* — the heavy computation happens on the provider's machines, so a modest laptop can drive a very powerful model — and they are generally *easier to set up*, often little more than installing a tool and authenticating.

The case against cloud agents. The costs are the mirror image of the benefits. There is *recurring cost* that scales with use, which for heavy continuous workloads adds up. There are

data-transfer concerns: your code and context leave your environment and travel to a third party, which may be unacceptable for sensitive work. There are *corporate-policy restrictions* — many organizations limit or forbid sending source code to external services. And there is *dependency on network and provider*: no connectivity, a provider outage, a changed policy, or a deprecated model, and your workflow is disrupted by something outside your control.

The case for local agents. Local models invert that risk profile. They offer *privacy* — code stays on your machine. They offer *predictable marginal cost* — after hardware, requests are cheap. They offer *offline use*, and they offer *control and experimentation*: you choose the model, you keep it as long as you like, and nothing changes under you without your say-so.

The case against local agents. The costs are practical and capability-related: real *hardware requirements*, more *setup complexity*, *weaker models* at the top end, *slower inference* on consumer hardware, and *more limited context* than the strongest cloud offerings.

Table 1.2 condenses the trade-off into something you can consult quickly.

Dimension	**Cloud agents**	**Local agents**
Reasoning capability	Strongest available	Good for routine; weaker on hard problems
Long-context handling	Better	More limited
Instruction following	Better	Adequate; can drift on complex tasks
Privacy / data control	Code leaves your environment	Code stays local
Marginal cost	Recurring, scales with use	Low after hardware investment
Setup effort	Low	Higher
Hardware burden	Minimal (provider-side)	Significant (your machine)
Offline operation	No	Yes
Availability risk	Network + provider dependent	Self-contained
Best when	The task is hard and data is shareable	Privacy, offline, or cost dominates

Table 1.2: Cloud vs. local trade-off matrix. Start from the binding constraint — privacy, connectivity, or difficulty — rather than from raw capability.

A simple way to make the call in practice: **start from the constraints, not the capability.** If the work involves code you are not permitted to send off-machine, the decision is made — local, full stop, regardless of capability. If you must work offline, same. If neither privacy nor connectivity is binding, then ask about difficulty: for a hard, ambiguous, or high-stakes change, reach for the strongest cloud model you are allowed to use, because capability is what protects you there; for routine, bounded, or exploratory work, a local model is often more than enough and gives you

control and predictable cost as a bonus. The mature setup is rarely "all cloud" or "all local." It is knowing, task by task, which constraint is in charge.

1.10 Context Windows, Repository Maps, and Why Agents Get Lost

To use any of these tools well — and especially the powerful ones — you have to understand the single most important limitation underneath all of them: **an agent does not automatically know your codebase.** It is easy to assume that because a terminal agent "has access to the repository," it somehow holds the whole thing in mind. It does not, and understanding why will save you from a whole category of frustration and a whole category of bugs.

Every model operates within a *context window* — a finite budget of text, measured in *tokens* (roughly, fragments of words and code), that it can consider at once. The window has grown large, but it is still finite, and real codebases are enormous. A medium project can be millions of tokens of code, comments, configuration, and history. You cannot simply pour the entire repository into the window; it would not fit, and even where it technically fit, it would be wasteful and would dilute the model's attention across a mountain of irrelevant material. So at any given moment, the agent is reasoning about a *small, selected slice* of your project, not the whole thing.

That selection is the crux. Because the agent cannot see everything, something has to decide *which files matter for this task* — and that decision is where things go right or wrong. To make it, capable agents build a *repository map*: a structured, lightweight index of the codebase. Rather than reading every file in full, the agent first builds an overview — the files that exist, the symbols they define (functions, classes, types), and how those symbols reference each other. Producing that map generally relies on parsing the code structurally rather than treating it as plain text; a common, general approach is *tree-sitter*–style parsing, which understands a file's syntax well enough to extract its definitions and references cheaply. The map is to the agent what a table of contents and an index are to you when you pick up an unfamiliar book: not the content itself, but enough structure to know where to look.

With a map in hand, the agent can do *file selection* intelligently — pulling the handful of files actually relevant to the task into its limited context, and leaving the rest summarized or out entirely. This is what lets a terminal agent work on a large repository at all. It is not holding the repo in its head; it is using a map to load the right 1% into a context window at the right moment, then moving on.

Now the failure mode becomes obvious. **Agents make wrong assumptions when context is incomplete.** If the map is stale, if the relevant file was not selected, if a crucial convention lives in a file the agent never opened, then the agent reasons from a partial picture — and reasoning confidently from a partial picture is exactly how you get a change that is locally sensible and globally wrong. It reimplements a helper you already have because it never saw yours. It violates

a pattern because the file that establishes the pattern was outside the slice it loaded. None of that is stupidity; it is competent reasoning applied to the wrong inputs. Most "the agent did something dumb" moments are, on inspection, "the agent could not see the thing that would have stopped it."

This is precisely why a later chapter is devoted to *context files* — files like `CLAUDE.md` and `AGENTS.md` that you write to give the agent the durable, high-value context it would otherwise have to discover or guess: your architecture, your conventions, your invariants, the commands that run your tests, the things you want it to never do. If the context window is the agent's working memory and the repository map is how it navigates, then context files are the briefing you hand it before it starts — the project knowledge you make sure is in the window every time, instead of hoping the agent stumbles onto it. Hold that thought; Chapter 3 is built on it. For now, hold on to the durable lesson that the agent's intelligence is real even though its *knowledge of your project is always partial and always managed*, and that a great deal of skill in AI-assisted development is the skill of managing what the agent can see.

1.11 Terminal State, Command Execution, and Feedback Loops

If context is what the agent *knows*, the terminal is how the agent *checks what it knows against reality* — and that reality check is the reason terminal agents earn their place at the top of the capability ladder.

Recall the agent loop. Its power lives in the steps where the agent runs something and reads the result. Those steps are only possible because the agent has a terminal. With it, the agent can *run tests* and see which pass and fail; *inspect logs* to understand runtime behavior; *run build commands* and learn whether the project even compiles; *read compiler output* to see precisely what broke and where; and, crucially, *repair errors based on that feedback* and try again. Each of these turns a guess into a measurement. A model proposing a fix with no way to run it is reasoning in the dark. The same model running the test suite and reading the failure is getting a hard, factual signal about whether its change actually works — and a hard signal is worth more than any amount of confident prose. The terminal is the agent's reality check.

But the same capability that makes the terminal valuable makes it dangerous. A tool that can run `npm test` can also run `rm -rf`. A tool that can read compiler output can also read your environment variables, including secrets. A tool that can install a dependency can also reach out to the network, pull arbitrary code, and execute it. *Command execution is the single largest source of risk* in AI-assisted development, because it is the point where the agent stops merely editing text and starts taking actions in the real world — some of which are irreversible.

The way to keep the benefit while containing the risk is to recognize that not all commands are equal. A large fraction of what an agent needs to do is *read-only* or *trivially reversible* — observing the project, running tests, checking status. These are safe to let an agent run freely, because the

worst case is wasted time, not damage.

Examples of low-risk commands an agent can run freely:

- `git status` — see what has changed; reads nothing dangerous, changes nothing.
- `git diff` — inspect the exact changes; read-only.
- `npm test` — run the JavaScript/TypeScript test suite.
- `dotnet test` — run the .NET test suite.
- `pytest` — run the Python test suite.
- `grep` / `rg` — search the codebase; read-only.
- `ls` — list directory contents; read-only.
- `cat` for specific files — read a particular file's contents.

Notice the pattern: these *observe* or *verify*. They are the steps of the agent loop that make it work, and they cannot, by themselves, harm your machine, your data, or anything outside the project.

A different and much smaller set of commands can change things outside the safe boundary — your system, your data, your accounts, the network, production. These should never run on an agent's unsupervised say-so. They demand an explicit human confirmation, every time.

Examples of high-risk commands that require confirmation:

- `rm -rf` — recursive deletion; can destroy work, or far more, in an instant and without undo.
- **Package installation** — pulls and may execute external code, expanding your trust boundary and your attack surface.
- **Database migrations** — can alter or destroy data, sometimes irreversibly, and often touch shared environments.
- **Network calls** — can exfiltrate data or fetch and run untrusted content.
- **Credential inspection** — reading environment variables, key files, or token stores risks leaking secrets into the agent's context.
- **Production deployment commands** — push changes to live systems where mistakes have real, immediate, customer-facing consequences.

- **Broad** `chmod` / `chown` — sweeping permission or ownership changes across wide paths can break a system or quietly open a security hole.

The line between these two lists — observe-and-verify versus change-the-world — is the foundation of agent safety, and it is why a later chapter is devoted entirely to *permissions and sandboxing*. The goal there will be to let the agent run the first list as freely as it likes, so the feedback loop stays fast and frictionless, while requiring explicit approval (or outright preventing, inside a sandbox) anything from the second list. For now, carry away the principle: the terminal is what makes a terminal agent genuinely useful, and it is exactly the same thing that makes a terminal agent genuinely dangerous. You do not get one without the other, so you manage it deliberately. Chapter 4 is how.

1.12 The Real Productivity Pattern: Human Designs, Agent Executes

Everything so far converges on a single thesis, and it is the heart of this book. The productive way to work with AI coding agents is *not* "let the AI build the app." It is a division of labor in which the human does the thinking that requires judgment and the agent does the execution that requires speed — with a verification gate between the two.

Spelled out as a workflow, the pattern looks like this:

- **The human defines the goal.** What are we actually trying to accomplish, and why? This is product and intent, and it is not delegable.
- **The human defines the constraints.** What must remain true? What must not change? What are the non-negotiables — performance, compatibility, security, the public API?
- **The human provides the architecture.** Where does this belong, what patterns does it follow, what are the boundaries it must respect? The agent can implement a design well; it should not be the one silently choosing the design.
- **The agent performs bounded implementation.** Within the goal, constraints, and architecture you set, the agent writes the code — fast, across as many files as needed.
- **The agent runs checks.** It executes the tests and build, reads the failures, and repairs until the loop closes. This is the agent doing what it is best at: tireless iteration against feedback.
- **The human reviews the diff.** Not the green checkmark — the *diff*. You read what actually changed and judge whether it is correct, sound, and acceptable.

- **The human commits only after verification.** The commit is an act of accountability. You are signing your name to this change. You do that after you have understood it, not before.

The useful metaphor for the agent in this pattern is a *junior developer* — a specific kind of junior developer, with a specific set of strengths and weaknesses you must manage. It is genuinely **useful** and astonishingly **fast** and completely **tireless**; it will cheerfully attempt the tedious refactor you have been avoiding and never complain. But it is also **literal** — it does what you said, not what you meant, so vague instructions yield confidently wrong results. It is sometimes **overconfident** — it will present a broken change with the same calm assurance as a correct one, because it does not feel doubt. It **needs clear tasks**, because ambiguity is where it goes wrong. It **needs tests**, because tests are how its work gets checked and how the loop has something real to verify against. And it **needs review**, because no matter how green the checks, you are the one accountable for what ships.

If you have ever onboarded a talented but green engineer, you already know how to manage this relationship. You give them well-scoped tasks with clear acceptance criteria. You make sure there are tests. You review their pull requests carefully, especially early, until you have calibrated how much to trust them on which kinds of work. You do not hand them the production database and the architecture and the security model on day one and walk away. You delegate *bounded* work and you *verify* the result. Every technique in the chapters ahead — context files, sandboxes, declarative prompts, disciplined diff review, git hygiene, test-repair loops — is a way of being a good manager of a very fast, very literal, very tireless junior who has, alarmingly, been handed a shell on your machine.

1.13 Choosing the Right Tool for the Job

With the categories established, the practical question becomes routine: faced with an actual task, which kind of tool do you reach for? The wrong instinct is to use your most powerful tool for everything — to fire up a terminal agent for a one-line helper, or to ask a disconnected chat window to perform a five-file refactor it cannot even see. Matching the tool to the task is most of the skill.

Table 1.3 gives a working decision framework, expressed as concrete tasks mapped to the appropriate category.

Task	Best-fit tool category	Why
Writing a small helper function	Autocomplete	Local, self-contained, no system knowledge needed
Explaining an unfamiliar error	Chat assistant	Pure reasoning; no need to act on the repo
Refactoring five files consistently	Terminal agent	Needs whole-repo awareness and multi-file edits
Adding unit tests	Terminal agent (or IDE agent)	Should read the code under test and run the suite
Updating documentation	IDE agent or terminal agent	Multi-file, benefits from repo context, low risk
Designing an architecture	Human-only first, then chat	Judgment call; use chat to explore options, not to decide
Fixing a failing CI pipeline	Terminal agent (with care) or CI agent	Needs to run builds/tests and read failures
Exploring a private codebase	Local model	Privacy-sensitive; keep code on your machine
Running a migration	Human-only / human-confirmed	High-stakes, possibly irreversible — never unsupervised
Reviewing a pull request	Chat or agent as a *second* opinion	Useful assist; the accountable reviewer is still you
Building a new feature slice	Human designs, terminal agent executes	The core pattern: you set goal/architecture, agent implements

Table 1.3: Task-to-tool decision matrix. Match the tool to the task; using the most powerful tool for everything is a common and costly mistake.

A few of these deserve emphasis because they are where people most often pick wrong. **Designing an architecture** is *human-first*: an agent is excellent at implementing a design and dangerous at silently inventing one, so use chat to widen your options and then *you* decide. **Running a migration** is *human-only or human-confirmed*: the stakes and irreversibility put it firmly in the high-risk command territory discussed earlier, no matter how routine it looks. **Reviewing a pull request** with an agent is genuinely helpful as a second set of eyes, but it does not transfer accountability — the human reviewer still owns the call.

The same logic renders cleanly as a decision flowchart. When a task lands on your desk, you can walk it down the tree in Figure 1.3.

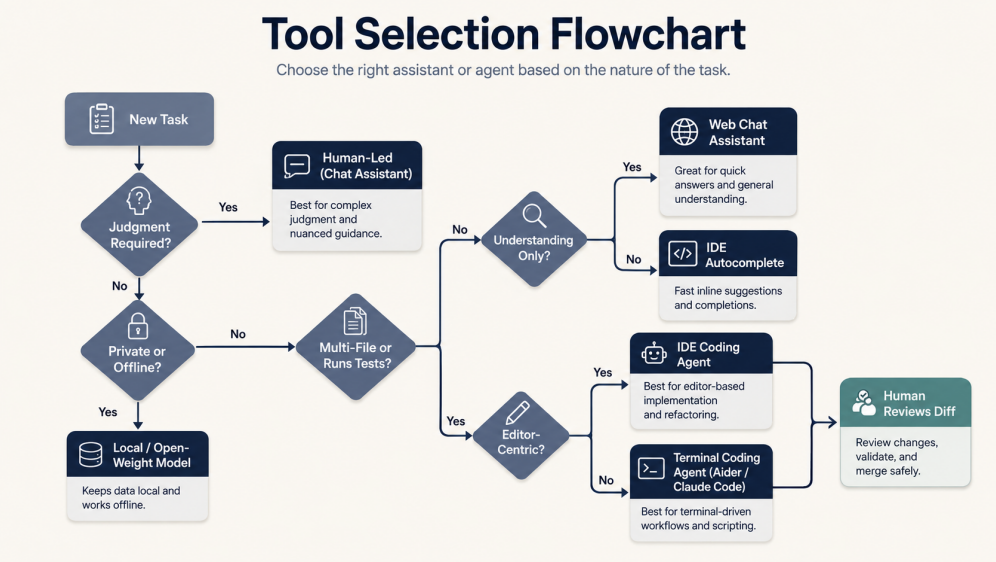

Figure 1.3: A tool-selection decision flowchart. Judgment and irreversibility route to a human first; privacy routes to local; scope and the need to run checks route to an agent — and every path that changes code ends at a human reviewing the diff.

The flowchart encodes the priorities argued throughout this chapter: **judgment and irreversibility route to a human first**; **privacy routes to local before anything else**; **scope and the need to run checks route to an agent**; and *every* path that produces real changes funnels into the same final gate — a human reviewing the diff before it is committed. The routing changes with the task; the gate never moves.

1.14 A Practical Starter Stack

Theory is useful; a concrete starting point is more useful. Here is a sensible default stack for a working developer adopting AI assistance deliberately. It is intentionally described by *role*, not by vendor — the point is the shape of the stack, which you can fill with whatever specific tools you are allowed to use and prefer. Swap any component; keep the structure.

- **IDE autocomplete, always on, for low-friction suggestions.** This is the typing accelerator. It costs you nothing in workflow change and pays for itself on routine code. Keep it on; trust it for nothing important.
- **One terminal-based coding agent for serious repository work.** This is your power tool for multi-file, test-backed engineering. You do not need three of them; pick one, learn its loop deeply, and build your safety practices around it. Depth on one agent beats shallow familiarity with several.
- **One chat assistant for planning and explanation.** This is where you think before you act — explore designs, understand unfamiliar code, decode errors. Use it to decide *what* to do,

then take that decision to a tool that can act.

- **An optional local model setup for private or offline experimentation.** If you handle sensitive code, work offline, or want predictable cost for routine tasks, add a local model. It is optional, not mandatory; add it when a real constraint calls for it.

Those are the assistants. But a stack is not just assistants — the most important parts of a safe AI-assisted workflow are the *backbones* that make the assistants safe to use at all:

- **Git as the safety backbone.** Every change an agent makes lives on a branch, in commits small enough to understand and revert. Version control is what makes the agent's mistakes recoverable instead of catastrophic.
- **Tests as the verification backbone.** Tests are what give the agent loop something real to push against and what give you something objective to trust. Without them, "it passed" means nothing.
- **Context files as the memory backbone.** Files like `CLAUDE.md` / `AGENTS.md` are how you give the agent the durable project knowledge it cannot infer, so it stops guessing about your conventions and constraints.
- **A Docker sandbox as the isolation backbone.** Running the agent inside a container bounds the blast radius of command execution: the agent can run freely against an environment that, if it goes wrong, you simply throw away.

Your git history is the emergency brake.

When an agent makes a change you do not like — and it will — the thing that saves you is not cleverness, it is `git`. A clean history of small, reviewed commits means any bad change can be inspected, isolated, and reverted in seconds. Work the agent on branches. Commit often and in small pieces. Never let an agent's work pile up as a giant uncommitted blob in your working tree, because that is precisely the state in which a mistake becomes unrecoverable. The brake only works if you have been pumping it all along.

Notice the balance of the stack. The assistants are the exciting part, but the four backbones — git, tests, context, sandbox — are what turn "exciting" into "safe to do on a real codebase Monday morning." A team that adopts the assistants and skips the backbones has bought a race car and skipped the brakes. The rest of this book is, in large part, about the backbones.

1.15 What This Book Will Teach You Next

You now have the map. You can tell autocomplete from an agent, an IDE agent from a terminal agent, and a cloud model from a local one. You understand the agent loop and why the terminal is both its source of power and its source of danger. You know that context is always partial and always managed, that the human designs while the agent executes, and that every path producing real changes ends at the same gate: a human reviewing the diff before committing. That is the conceptual foundation. The rest of the book turns it into practice.

Here is the road ahead:

- **Setup and your first session** — getting an agent running and doing a first real task safely.
- **Context files** — writing `CLAUDE.md` / `AGENTS.md` so the agent understands your project instead of guessing.
- **Sandboxing** — containing the agent so command execution cannot harm the host.
- **Declarative prompting** — describing tasks as outcomes and constraints so a literal executor does the right thing.
- **Multi-file refactoring** — driving an agent across a codebase while keeping changes scoped and reviewable.
- **Diff review** — reading agent output efficiently and catching the failures that hide behind a green check.
- **Delegation decisions** — a sharper framework for what to hand off and what to keep by hand.
- **Git workflows** — branching and micro-commits that keep the emergency brake within reach.
- **Test-repair loops** — making the agent loop converge on correct, verified code.
- **Local vs. cloud trade-offs** — choosing and combining models as constraints and stakes shift.
- **MCP and custom tools** — grounding an agent in your real systems, safely.
- **Team guidelines** — shared context, conventions, and policies so agents behave consistently across a team.
- **CI automation** — putting agents to work in the pipeline without letting them ship unsupervised.

- **Pitfalls and long-term maintenance** — the anti-patterns to avoid and how to keep an agent-assisted codebase healthy over time.

The throughline of every one of those chapters is the same conviction this one has been building: professional AI-assisted development is a discipline, not an act of faith. The power of these tools is real, and so are their hazards, and the two are inseparable because they come from the same place — the agent acting on your real code.

So the next step is not to trust agents more. It is to build a controlled environment in which they can be useful *without being dangerous* — a place where the agent can move fast because the boundaries are firm, the work is reversible, the checks are real, and a human is always the one who decides what ships. Building that environment, piece by piece, is what the rest of this book is for. We start with setup.