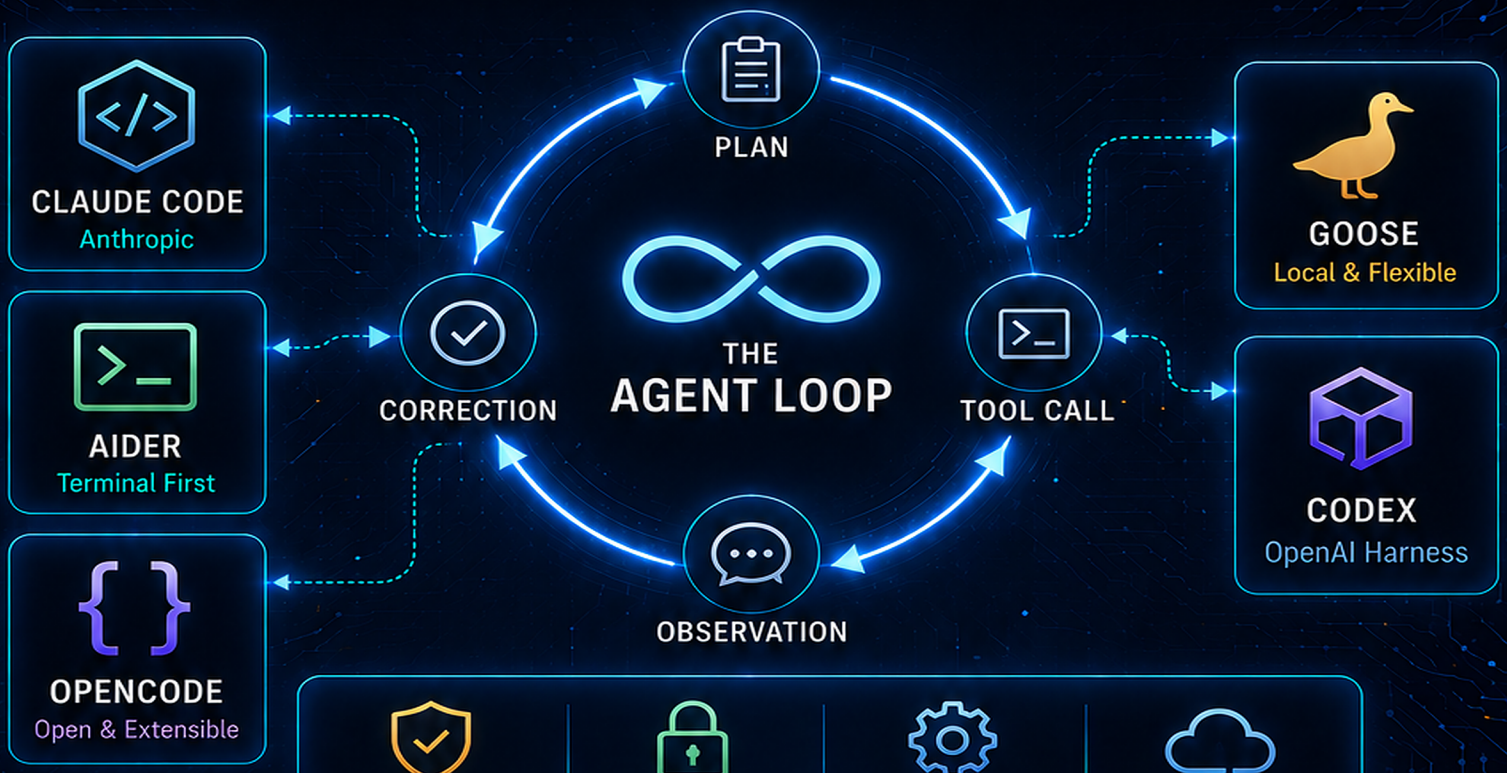
AGENTIC CODING HARNESSES, COMPARED & EXPLAINED

MASTER CLAUDE CODE, AIDER, OPENCODE, GOOSE, AND CODEX — THEN RUN THEM LOCALLY



DEEP COMPARISONS
Understand the real differences

SECURE & SANDBOXED
Protect your code and systems

CONFIGURE & EXTEND
Make each harness work for you

RUN LOCALLY
Use Ollama and local models for free

UNDERSTAND THE LOOP • CHOOSE THE RIGHT HARNESS • RUN IT LOCALLY

by YOHAN J. RODRÍGUEZ

Preface

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

For most of the last decade, using a large language model meant treating it as a chatbot. That is changing. Coding agents now run autonomously on your behalf — editing files, executing terminal commands, running test suites, and correcting errors until the job is done.

This book is a practical, hands-on guide to operating, configuring, and extending these agentic coding harnesses. It is written for developers, DevOps engineers, and technically curious professionals who want to graduate from copy-pasting code prompts to running autonomous loops safely and cheaply. You do not need a machine-learning background, but you do need to be comfortable on a command line.

A word on scope. This book is about *operating* and *extending* single-agent harnesses, not building hosted multi-agent systems. What it does cover is the durable mechanics of the agent loop, Model Context Protocol (MCP) tool integration, permission gates, context management, safety sandboxing, and running harnesses locally. Where tool CLIs or model configurations change, treat the book's examples as a starting point to confirm against current documentation.

The examples target Linux first, but include call-outs for macOS and Windows. Commands and code examples are written to be concrete and inspectable; some worksheets, prompts, and transcripts are intentionally illustrative, and when a command is meant to be run, the surrounding text makes that clear.

The goal is to build a durable mental model of how agents operate, so you can select the right tool for the right task and run it securely and affordably.

Yohan J. Rodriguez

Contents

1 | What an Agentic Coding Harness Is

A developer opens a ticket that reads: *add input validation to the user registration form*. The change is small and the developer knows roughly what it needs — reject malformed emails, enforce a minimum password length, trim the display name — but doing it touches three files, a test suite, and a couple of edge cases that are easy to get wrong. How that fifteen-minute task actually unfolds depends almost entirely on which kind of AI tool the developer reaches for, and the difference between the kinds is the subject of this book.

With one kind of tool, the developer does most of the mechanical work themselves. They paste the registration function into a chat window, paste the model that backs it, explain the project layout, read the suggested changes, copy them back into the editor by hand, run the tests in a terminal, copy the failure back into the chat, and repeat until it works. The model is a smart advisor, but every action that touches the codebase is performed by the human. With another kind of tool, the developer types the sentence once and watches. The tool searches the repository for the registration code, reads the relevant files on its own, writes the edits directly to disk, runs the test suite, sees a test fail, reads the error, corrects its own mistake, runs the tests again, and reports back what it changed and what it deliberately left undone. The human gave one instruction and reviewed one result; the tool did the looping in between.

Both tools are commonly described with the same three words — "AI pair programmer" — and that shared label is precisely the problem this chapter sets out to fix. The two tools differ not in quality but in *kind*: in what the model is allowed to see, what it is allowed to do, who runs the loop between "here is the task" and "here is the result," and what can go wrong as a consequence. This chapter builds the vocabulary to tell them apart. By the end of it you will be able to place any AI coding tool you encounter — the ones named in this book and the ones that do not exist yet — onto a clear spectrum, and to say precisely why the tool at the agentic end of that spectrum is a genuinely different category of software with genuinely different powers and risks.

1.1 The Moment Coding Tools Stopped Being Just Assistants

For most of the short history of AI-assisted programming, the model sat on the far side of a wall. You could send it text and it could send text back, and everything useful that happened to your

actual code happened because a human carried the model’s words across that wall by hand. You read a suggestion and typed it in. You copied an error and pasted it over. The model never touched the repository; it advised, and you acted. This arrangement is the natural shape of a chat interface, and it defined what people meant by “using AI to code” for a surprisingly long time.

The shift that this book is about is the moment the wall came down — the moment a tool was built that lets the model *act* on the repository rather than only describe what action to take. The mechanism that made this possible is not a smarter model, although models did get smarter. It is a piece of engineering wrapped around the model: a program that takes the model’s structured requests to read a file, write a file, or run a command, actually performs them against the real file system and shell, captures what happened, and feeds the result back to the model so it can decide what to do next. That wrapping program is the *harness*, and the loop it runs — request an action, perform it, observe the outcome, decide again — is what turns a text generator into something that can carry a multi-step coding task to completion largely on its own.

It is worth being precise about what changed, because the change is easy to mis-state as “the AI got agency,” which makes it sound like a property of the model. It is not. The model is the same kind of next-token predictor it always was; what is new is the software around it that is willing to treat certain pieces of the model’s output as commands to execute rather than as prose to display. The intelligence lives in the model, but the *agency* lives in the harness. This distinction is the spine of the whole book, and Chapter 2 takes the harness apart to show the loop in detail. For now the important thing is the qualitative jump: once a tool will run the model’s requested actions and return the results, the human stops being the courier between the model and the code, and the tool can iterate — edit, test, observe, fix, retest — in a way that a chat window simply cannot.

1.2 Why “AI Coding Tool” Is Too Vague

Walk through the marketing for a dozen AI coding products and you will read the same handful of phrases over and over: *AI pair programmer*, *your AI coding assistant*, *code with AI*. These phrases are not wrong, exactly, but they are uselessly broad. They describe an autocomplete plugin that suggests the end of the line you are typing, a chat website where you discuss architecture, an editor that rewrites a function in place, and a terminal program that refactors twelve files and runs your test suite unattended — all with the same words. A category label that covers tools this different is not telling you anything you can act on.

The vagueness has real costs, and they fall in two opposite directions. Developers who under-read the category *under-trust* these tools: having formed their impression from autocomplete, they assume “AI coding” means small inline suggestions and never discover that a tool exists which could have done the whole multi-file change. Developers who over-read the category *over-trust* them: having heard that “the AI can just do it,” they hand a tool broad permissions on

a real repository without understanding that it can now run shell commands, and are surprised when something destructive or careless happens. Both mistakes come from the same root, which is treating a spectrum of very different capabilities as a single undifferentiated thing.

What the phrases hide is exactly the set of differences that matter when you choose a tool or reason about its behavior: how much of your project the model can actually see, whether it can change files itself or only suggest changes, whether it can run commands and read their output, whether it loops on its own or stops after every step to wait for you, what permissions it needs to do its job, what it can break, and where you as the human sit in the process. None of these are captured by "pair programmer." All of them are captured by asking a more precise question: *who controls the loop, what tools can the model call, and what can those tools touch?* The rest of this chapter is a disciplined way of answering that question for any tool you meet.

1.3 The Developer Tooling Spectrum

It helps to lay the whole landscape out as a single line, because the categories are not arbitrary buckets — they are ordered. As you move from left to right, four things increase together: the amount of *context* the model can draw on, the *autonomy* it has to act without you, the *permissions* it needs to do so, and the *risk* that something goes wrong without your catching it in the moment. A tool further right can see more, do more, and break more. None of these positions is "better"; they are suited to different work, and a productive developer uses tools from several points on the line during a single day (Figure 1.1).

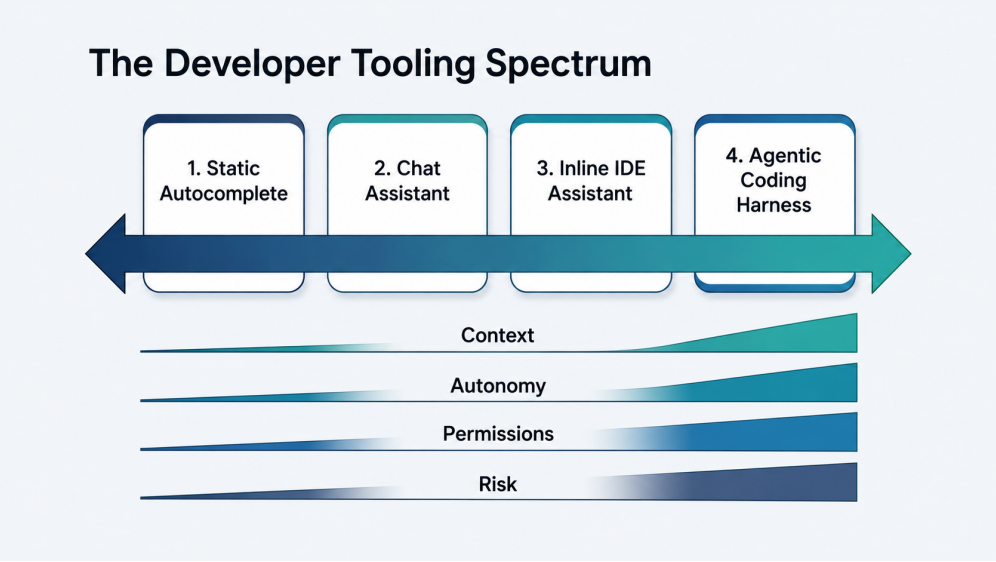


Figure 1.1: The AI coding tool spectrum: as tools move from autocomplete toward harnesses, they gain permission to inspect, modify, execute, and self-correct against the codebase.

The four stations on the line are *static autocomplete*, *chat assistants*, *inline chat and IDE assistants*

with project context, and *agentic coding harnesses*. The next four sections take each in turn and answer the same five questions for all of them, so the comparison is honest: what can the model see, what can it do, what must the human do, what can it not do, and what is it best and worst at. Holding the questions constant is what makes the categories distinguishable rather than impressionistic.

1.4 Static Autocomplete: Prediction Without Agency

Static autocomplete is the AI coding tool most developers met first, usually as an extension that finishes the line or the block you are typing. As you write, it predicts the most likely continuation from the immediate surroundings — the current file, the symbols in scope, often a window of related open files — and offers it as a ghosted suggestion you accept with a keystroke. GitHub Copilot's inline completions and Tabnine are the familiar examples. It is genuinely useful and, for many developers, the single highest-frequency AI interaction of the day.

What the model can see is a local window: the code immediately around your cursor and, depending on the product and its configuration, a selection of other open or recently edited files. It is not reading your whole repository on demand; it is being fed a context window assembled from what is near at hand. *What the model can do* is exactly one thing — propose text at the cursor. *What the human must do* is everything else: decide what to build, position the cursor, judge each suggestion, accept or reject it, and run and verify the result. *What the tool cannot do* is act on its own initiative: it cannot open a file you did not open, cannot run your tests, cannot make a change in a file you are not currently editing, and has no notion of a task that spans more than the next few lines. There is no loop here at all — just a fast, repeated prediction triggered by your typing.

This makes autocomplete superb at what it is for: reducing the keystrokes and the small recall burden of writing code you already know how to write. It completes the boilerplate, remembers the argument order of a function you half-remember, and keeps you in flow. Its limitations are the mirror image of its strengths. It has no view of the task, so it cannot tell you that your approach is wrong, only finish the approach you are taking. It can produce confident, plausible, subtly incorrect code — the classic risk of autocomplete is that it is just right enough to be accepted without scrutiny. And because it operates on a local window, it cannot reason about the project as a whole. It is prediction without agency, and that is precisely why it is low-risk: it never does anything you did not type your way into accepting.

1.5 Chat Assistants: Reasoning Without Direct Execution

A chat assistant is a conversational model you talk to in a separate window — a website or an app — that is not wired into your editor or your repository at all. ChatGPT and the Claude.ai

web interface are the canonical examples. Compared with autocomplete it is a large step up in reasoning: you can describe a problem, paste code, ask for an explanation, request a design, debate trade-offs, and get back genuinely thoughtful, multi-paragraph answers. But it is a step *sideways* on the question that organizes this chapter, because the chat assistant, for all its reasoning power, cannot touch your code at all.

What the model can see is only what you paste into the conversation. It has no access to your file system; if it needs to know what is in a file, you must copy that file into the chat. *What the model can do* is produce text — explanations, suggested code, commands for you to run — which lands in the chat window and nowhere else. *What the human must do* is be the courier and the operator: gather and paste the relevant context, read the answer, carry any code back into the editor by hand, run it, and paste any errors back in to continue the conversation. *What the tool cannot do* is read a file you did not paste, write a change you did not copy across, or run a single command. Listing 1.1 shows the shape of this: to get useful help, the human front-loads the context manually, and even then the model is guessing about everything outside the paste.

Listing 1.1: A chat-style request: the human must assemble and paste the context by hand, and will carry any answer back into the editor manually

```
You: I need to add input validation to my user registration form.
Here is the relevant file. Please tell me what to change.

# app/registration.py
def register_user(form):
    user = User(
        email=form["email"],
        password=form["password"],
        display_name=form["display_name"],
    )
    db.session.add(user)
    db.session.commit()
    return user

Also here is my User model, in case it matters:

# app/models.py
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True)
    password = db.Column(db.String(255))
    display_name = db.Column(db.String(80))

What validation should I add, and where exactly does it go?
```

There is a loop in a chat workflow, but the human is running it. The model proposes; the human applies and tests; the human reports the result back; the model revises. Every turn of that loop crosses the wall between the chat and the code, and a person carries it across each time. This is why a chat assistant is so strong for the parts of programming that are about thinking — understanding an unfamiliar error, choosing between two designs, learning an API, drafting an algorithm — and so laborious for the parts that are about doing. Its great limitation is staleness and blindness: it reasons only over the snapshot you pasted, so it cannot see the three other files

that actually matter, and it cannot check its own suggestion against reality because it cannot run anything. It is reasoning without execution, and the gap between the two is filled by you.

1.6 Inline Chat and IDE Assistants: Context-Aware Help

The third station closes much of the gap by moving the conversation inside the editor and giving it real, if bounded, access to the project. Tools such as Cursor and Windsurf — and the in-editor chat modes that many IDEs now ship — let you talk to a model that can read files from your workspace, that can be pointed at a selection or a whole file, and that can apply suggested changes into your buffers directly, often shown as a diff you accept or reject. This is a real jump in convenience over the copy-paste chat: the model can pull in the relevant files itself, propose an edit across more than one of them, and write the change where you can see it land.

What the model can see is now a meaningful slice of the project: the files you reference, the files it retrieves from the workspace index, and often the symbols and structure around them. It is no longer limited to what you paste. *What the model can do* expands to editing files in place, typically through a review step — it proposes a diff and you apply it — and, in many of these tools, to running commands or tests when you ask and approve. *What the human must do* shrinks but does not vanish: you still drive the session turn by turn, review each proposed change, and decide when to run things. *What the tool cannot do* is the part that distinguishes this station from the next one, and it is the subtle part. In its assistant posture, this kind of tool keeps the human at the center of every step: it edits when asked and waits for your judgment, rather than setting itself a multi-step plan and driving it to completion on its own.

This is the category that resists clean labeling, and honesty requires admitting it. The leading tools here are configurable and fast-moving; several of them now include an "agent" mode that behaves much like the harnesses in the next section, and the line between "a context-aware assistant that edits when asked" and "a harness that runs the loop itself" is exactly where these products are competing and changing month to month. The right way to read this category is therefore not by brand but by *mode*: used as context-aware help, where you approve each step and the human runs the loop, a tool sits here; switched into a mode where it plans, edits, runs, observes, and corrects on its own until the task is done, the same tool has stepped to the right, onto the agentic station. The taxonomy is about what the tool is doing, not what it is called.

1.7 Agentic Coding Harnesses: The Model Gets Hands

At the right end of the spectrum is the category this book is named for. An agentic coding harness is a program — usually run from the terminal — that gives the model a set of tools for acting on the real project and then runs a loop: it asks the model what to do next, the model answers with a request to use one of those tools, the harness performs the action against the actual file system,

shell, or network, captures the result, feeds it back to the model, and asks again. Claude Code, Aider, OpenCode, Goose, and Codex are the harnesses this book examines in depth, and they differ in many details, but they share this shape. The model is no longer advising a human who acts; the model is acting, through the harness, and the human supervises.

The practical consequence is the contrast that opened this chapter. Where the chat user assembled and pasted context by hand, the harness assembles it itself: when you give it the task in Listing 1.2, it searches the repository, decides which files are relevant, and reads them without being handed anything. Where the chat user copied suggestions back into the editor, the harness writes the edit to disk directly. Where the chat user ran the tests and pasted the failure back, the harness runs the tests, reads the failure from the command output, and feeds it to the model as the next observation. The whole edit–test–observe–fix loop that a person drove manually in the chat workflow runs inside the tool, turn after turn, until the task is finished or some limit stops it.

Listing 1.2: A harness-style request: the human delegates the whole task once; the tool will gather context, edit, test, and iterate on its own

```
You: Add input validation to the user registration form. Reject
empty or malformed emails, reject passwords shorter than 10
characters, and trim and length-check the display name. Keep the
existing return behavior, add tests for the new rules, and make
sure the full test suite passes before you finish.
```

Notice how little the human said. The chat prompt had to carry the files and the structure; the harness prompt carries only the *intent*, because the harness can discover the rest. That shift — from instructing the model about your code to delegating a task over your code — is the whole difference, and it is why the harness can take on work that is too multi-step and too context-hungry for the categories to its left. It is also why the harness needs powers the others do not, and why it can do damage the others cannot. The remainder of the chapter makes the definition precise, names the capabilities that earn the word "agentic," and is honest about the cost of those capabilities.

1.8 A Precise Definition of an Agentic Coding Harness

We can now state the definition the rest of the book will use. An *agentic coding harness* is a program that connects a language model to a real software project and runs an execution loop in which the model, not the human, chooses and requests the next action — reading a file, writing a file, running a command — and the harness performs that action against the actual environment and returns the result to the model so it can choose again, repeating until the task is complete, a limit is reached, or a human intervenes.

Three parts of that sentence carry the weight. The first is *real*: the harness acts against the genuine file system and shell, not a sandboxed copy of pasted text, so its actions have real effects. The second is *the model chooses the next action*: the harness does not follow a fixed script; it asks the model what to do at each step and the model's answer drives the loop, which is what makes

the behavior adaptive rather than canned. The third is *repeating until*: there is a loop with a termination condition, so the tool can take many steps for one instruction and can recover from its own mistakes between steps. A tool that has all three is a harness. A tool that is missing one of them belongs to one of the earlier stations, and calling it a harness will mislead you about what it can do and what it can break.

It is worth saying plainly what this definition is *not*. It is not a judgment that harnesses are the best tool, or the tool you should always reach for; much of this book is about when a smaller, left-of-spectrum tool is the right and safer choice. And it is not a claim about model quality: a harness driving a mediocre model is still a harness, and a brilliant model behind a chat window is still a chat assistant. The definition is about architecture — the loop and what it can touch — not about intelligence, and keeping those two ideas separate is the single most clarifying habit you can form when evaluating these tools.

1.9 The Four Capabilities of a True Harness

The definition becomes a usable test when we break it into the concrete capabilities a tool must have to satisfy it. There are four, and a tool needs all four to be a full agentic coding harness. They are autonomous file-system read and write, shell command execution, a recursive self-correction loop, and tool registration through a structured tool mechanism. The first two are about reach — what the model can touch. The third is about iteration — whether it can recover from its own errors. The fourth is about extensibility — how new powers are added in a disciplined way. Take any one of them away and the tool drops to a different category, as the rest of this section and the next four will show (Figure 1.2).

Figure 1.2: A true agentic harness combines file access, shell execution, iterative correction, and registered tools behind a permission boundary.

1.10 Capability 1: Autonomous File-System Read and Write

The first capability is the one that ends the copy-paste era: the model can read and write files in the project on its own initiative, without a human handing it text or carrying changes back. "On its own initiative" is the load-bearing phrase. Autocomplete sees a local window it was fed; a chat assistant sees only what you pasted; an inline assistant reads files you point it at or that its index surfaces. A harness, by contrast, decides for itself which files it needs, finds them, opens them, and edits them, because reading and writing are tools the model can call as part of the loop.

In practice this means two distinct powers, and both matter. The first is *autonomous read*: given a task, the harness can search the repository — by file name, by symbol, by a text grep — form a hypothesis about which files are relevant, read them, and follow references from one file to another, building its own picture of the codebase the way a developer would when dropped into an unfamiliar project. The human did not select the files; the model did, from the task. The second is *autonomous write*: the harness can apply changes directly to those files on disk — create a new file, edit an existing one, delete a line — so that after a turn the working tree actually differs. The change is real, not a suggestion in a window.

This capability is the largest single jump in power and the largest single jump in responsibility. The power is obvious from the opening scenario: a tool that can find and read the three files that matter, then edit them, can carry a real change with one instruction. The responsibility is the flip side: a tool that can write to any file in your project can also overwrite something you cared about, edit the wrong file, or make a sweeping find-and-replace that does more than you meant. This is why version control stops being good hygiene and becomes a hard prerequisite for harness work — a clean git state before you start is what lets you see exactly what the tool changed and undo it if you disagree. Later chapters treat this in depth; for now, hold the symmetry in mind: the same autonomy that lets the harness do the whole task is the autonomy that lets it do the whole task *wrong*, and your review of its diff is the safeguard.

1.11 Capability 2: Shell Command Execution

The second capability is the ability to run shell commands and read their output. This is what lets a harness move from editing code to *verifying* it, and it is the capability that most clearly separates a tool that suggests changes from a tool that closes the loop on whether those changes work. A harness with shell access can run your test suite, invoke a linter or type checker, build the project, run a script, inspect the environment, and use git — and, crucially, it can read what those commands print and react to it. Running a command and seeing its result is the harness's sense organ: it is how the model perceives the consequences of what it just did.

The reason this matters so much is that code edits are cheap to propose and hard to get right,

and the only reliable judge is execution. A chat assistant can suggest a fix that looks correct and is not; it has no way to find out, and neither do you until you run it yourself. A harness runs it. When the harness edits the registration function and then runs `pytest`, the pass-or-fail result is ground truth, and a failing test is not a dead end but an observation the model can act on. The walkthrough later in this chapter turns on exactly this: the harness's first attempt fails a test, and because it can read the failure from the command output, it can diagnose and fix the mistake rather than confidently shipping it.

Shell execution is also where the risk of the category becomes concrete and serious, and it deserves to be named here even though Chapter 6 is devoted to it. A tool that can run `pytest` can also run `rm -rf`, can run `curl` to send data out of your network, and can run any script it finds or writes. The same mechanism that runs your tests will run whatever command the model emits, and the model's behavior can be influenced by text it reads in your repository — a comment, a README, a dependency's documentation — which is the root of the prompt-injection risk that recurs throughout this book. This is the central reason harnesses gate their actions behind permissions, and it is why "just let it run everything unattended" is almost never the right default. Shell access is the capability that makes a harness powerful and the capability that makes it dangerous, and those are the same sentence.

1.12 Capability 3: Recursive Self-Correction

The third capability is the loop itself: the harness does not perform one action and stop, but repeats the cycle of acting and observing, using each observation to decide the next action, so that it can correct its own mistakes without a human prompting each step. This is what the word "recursive" points at — the output of one turn (a test failure, a file's contents, a command's error) becomes the input to the next turn, and the model steers itself toward the goal across many turns rather than producing a single answer.

This is the capability that most deserves the word "agentic," because it is what lets the tool absorb the friction that a human otherwise absorbs. In a chat workflow, when a suggestion fails a test, the human is the one who notices, copies the error back, and asks for a revision; the loop runs at human speed and human attention. In a harness, that loop runs inside the tool: it sees the failure, reasons about the cause, edits again, and re-runs, often several times, before it reports back to you at all. The developer's experience changes from *step the model through the task* to *delegate the task and review the outcome*, and that change is only possible because the tool can iterate on its own results.

Self-correction is also what makes a harness's behavior feel qualitatively different to watch, and it is worth being clear-eyed about both sides of it. On the good side, a competent harness will catch and fix a surprising share of its own errors — a typo it introduced, an import it forgot, a test it broke — because it runs the checks that surface them. On the cautionary side, the loop is

not magic: it can also iterate in the wrong direction, "fixing" a failing test by weakening the test rather than the code, thrashing between two broken states, or burning through turns on a task it has misunderstood. This is why every harness has *termination conditions* — a turn limit, a token or cost budget, an explicit completion signal, and your ability to interrupt — which Chapter 2 examines closely. The loop is the source of the harness's power and the reason it needs limits; an agent that could loop forever with no ceiling would be a liability, not an asset.

1.13 Capability 4: Tool Registration and Tool Schemas

The fourth capability is the one that is easiest to overlook and the most structurally important: the harness exposes its powers to the model as a set of *registered tools*, each described by a schema that tells the model the tool's name, what it does, and what arguments it takes. Reading a file, writing a file, running a command, searching the codebase, fetching a URL — each is a tool with a defined interface, and the model acts by emitting a structured request to call one of them rather than by producing free-form prose that the harness has to guess how to interpret. This is the disciplined seam between the model's text and the harness's actions.

Concretely, the model does not say, in English, "please open the registration file"; it emits a structured tool call — a small, typed object naming a tool and its arguments. Listing 1.3 shows the shape of one: a request to use a `read_file` tool with a path and a line range, and the `tool_result` that the harness appends back into the conversation after it runs the call. The mechanism has several names depending on the model and the framework behind it — Anthropic's tool use, OpenAI's function calling, the Model Context Protocol (MCP), or another equivalent schema system — but the idea is the same across all of them: tools are declared with structured descriptions, the model selects and fills one in, and the harness validates and executes it. Chapter 2 shows a full round-trip and Chapter 4 is devoted to MCP.

Listing 1.3: A simplified tool call and its result. The model emits a typed request to use a registered tool; the harness runs it and feeds the result back as the next turn

```
{
  "type": "tool_use",
  "id": "toolu_01F3a9",
  "name": "read_file",
  "input": {
    "path": "app/registration.py",
    "start_line": 1,
    "end_line": 40
  }
}

// The harness executes the call and appends the result
// back into the conversation as a new turn:

{
  "type": "tool_result",
  "tool_use_id": "toolu_01F3a9",
  "content": "def register_user(form):\n    user = User(\n ..."
```

```
}
```

Tool registration matters for two reasons beyond mechanics. First, it is what makes a harness *extensible* in a principled way: because tools are declared through a schema, you can add new ones — a tool that queries your database, files a ticket, or calls an internal service — and the model can use them without the harness being rewritten, which is the entire premise of MCP and the reason a later chapter builds a tool server from scratch. Second, it is the natural place to put a *permission boundary*: because every action the model takes flows through a named, structured tool call, the harness can inspect each call before running it and decide whether to allow it silently, ask you to confirm, or refuse. Free-form prose has no such checkpoint; a typed tool call does. The schema is not bureaucracy — it is what makes the harness both safely gateable and cleanly extensible, which is why a tool that drives real actions through anything looser tends to be both harder to extend and harder to trust.

1.14 Human-in-the-Loop vs. Human-on-the-Loop

Once a tool can run its own loop, the most important operational question is no longer *what can it do* but *where do you stand while it does it*. There are three useful positions, and a mature harness user moves between them deliberately depending on the task and the trust they have in the tool for it. Naming them precisely is worth doing, because "how autonomous should the agent be" is a setting you adjust many times a day, not a philosophy you choose once.

Human-in-the-loop means the harness pauses for your approval at each consequential action: it proposes to edit a file or run a command, and nothing happens until you say yes. You are inside the loop, and the agent cannot take a step you did not sanction. This is the safest posture and the right default for unfamiliar tools, sensitive repositories, and any action that is hard to undo. Its cost is your attention — you are consulted constantly — which is exactly the friction you accept in exchange for control.

Human-on-the-loop means the harness runs its loop and acts without pausing for each step, while you watch the stream of what it is doing and retain the power to interrupt the moment something looks wrong. You are not approving every action, but you are present and supervising, ready to hit the brakes. This is the posture most experienced users settle into for routine work on a project they trust the tool with: it keeps the speed of autonomous iteration while keeping a human ready to stop a bad run. Its cost is that an unwanted action can happen before you catch it, which is why it pairs with version control and a sandbox rather than replacing them.

Human-after-the-loop means the harness runs to completion unattended and you review the result only when it is done — the diff, the test output, the summary. You were not present during the work at all. This is appropriate for narrow, well-bounded, reversible tasks in a sandboxed or disposable environment, and it is genuinely useful for batch or background work; it is also where the risk is highest, because nothing was checked while it ran. The honest guidance, which

this book returns to repeatedly, is that full autonomy is rarely the right posture for ordinary development on a repository you care about: the value of a harness is mostly captured at human-on-the-loop, and the marginal convenience of walking away entirely is usually not worth giving up the ability to interrupt.

1.15 Why Harnesses Need More Permissions Than Chat

A chat assistant needs no permissions on your machine at all, because it never touches your machine — it receives text and returns text, and every effect on your code is produced by your own hands. A harness is the opposite: its entire value is that it acts on your machine, and acting requires permission to act. This is not a flaw or an oversight; it is the direct, unavoidable consequence of moving from a tool that advises to a tool that does. If you understand why the permissions are broad, you will configure them deliberately rather than either rejecting harnesses out of fear or accepting every prompt out of habit.

Spelling out the specific permissions makes the point concrete. A harness needs to *read files* across your project, which means it can see whatever is in the repository, including anything sensitive that lives there. It needs to *write files*, which means it can change or create or delete them. It needs to *run shell commands*, which is the broadest permission of all, because the shell can do nearly anything the account running it can do. Many harnesses also benefit from *network access* — to fetch documentation, install a dependency, or call an API — which is a path for data to leave as well as enter. And they routinely perform *git operations*, which is convenient and also a way to rewrite history or push if given the latitude. Each of these is something a chat assistant simply never asks for, because it never acts.

Two risks deserve to be named explicitly because they are specific to this expanded permission set. *Secrets exposure* is the first: a tool that can read any file can read your `.env`, your credentials, your private keys, and if it also has network access, the path from "read a secret" to "send it somewhere" is short — whether through a mistake, a misjudged action, or a prompt injection that turns the tool against you. *Destructive commands* are the second: a tool with shell access can delete files, drop a database, force-push over history, or run a script that does any of these, and a single careless or manipulated command can do damage faster than you can react. These are not reasons to avoid harnesses; they are reasons to run them the way the rest of this book teaches — with a permission gate on actions, a clean version-control state, secrets kept out of reach, and, for anything beyond routine, a sandbox. The broad permissions are the price of the capability, and the gate is how you pay it safely.

1.16 What Harnesses Still Cannot Safely Do

It is just as important to be clear about the ceiling as the reach, because the failure mode of over-trusting a harness is as real as the failure mode of under-using one. A harness is a capable tool, not a competent engineer, and several things remain firmly the human's job.

A harness cannot reliably understand intent beyond what you stated, and it will confidently fill gaps with assumptions. If your one-sentence task under-specifies the requirement — which validation rules, which edge cases, what counts as "done" — the harness will choose, and it may choose differently than you would. It also cannot be trusted to judge whether its own changes are *correct* in the senses that tests do not capture: a green test suite means the tests pass, not that the behavior is what the business needs, that the design is sound, or that a subtle security or performance implication has been considered. The harness optimizes toward the signal it can see, and where the signal is incomplete, so is its judgment.

A harness also cannot safely make irreversible or outward-facing decisions on its own — deleting data, deploying to production, sending an email, posting to a public service, spending money, or anything whose consequences leave the sandbox and cannot be undone by reverting a diff. These are precisely the actions that warrant a human in the loop regardless of how much you trust the tool for ordinary editing, because the cost of a wrong one is not a failed test but a real-world effect. And a harness cannot substitute for your review: the diff it produced and the summary it wrote are claims to be checked, not conclusions to be accepted, and the discipline of reading what it changed is the thing that keeps the autonomy an asset rather than a liability. The harness gives you leverage; it does not give you the option of not looking.

1.17 Tool Calls as the Boundary Between Text and Action

It is worth pausing on the single idea that everything in this chapter pivots around, because it is the idea the whole book elaborates: the tool call is the boundary where a model's text becomes a real-world action, and the harness is what stands on that boundary deciding what to let across. A language model, by itself, only ever produces text. It cannot read your file or run your test; it can only emit characters. What a harness adds is the willingness to treat certain of those characters — a structured tool call — as an instruction to perform a real action, plus the machinery to perform it and report back.

This reframing dissolves a lot of confusion. "How can the AI run my tests?" is answered by: it cannot — it emits a tool call asking to run them, and the harness runs them and hands back the output. "How can the AI edit my file?" is answered the same way: the model requests a write through a tool, and the harness performs the write. Every power a harness appears to have is, mechanically, a tool the harness exposes and a call the model emits, executed on the other side of the boundary by ordinary code. The intelligence chooses the action; the harness, not the model,

takes it. That is why the same model behind a chat window is harmless to your file system and behind a harness can rewrite your repository: nothing about the model changed, only whether there is a program on the boundary willing to act on its requests.

Seen this way, the entire spectrum from autocomplete to harness is a spectrum of *how much of the model's output is allowed to cross the boundary into action*, and how much supervision sits at the crossing. Autocomplete lets one kind of text cross — a completion you accept by typing. Chat lets none cross automatically; you carry it. An inline assistant lets edits cross under your per-step review. A harness lets reads, writes, and commands cross under a permission gate, in a loop. Keeping this boundary in view is what lets you reason about any tool, present or future: ask what crosses, what runs it, and who is watching the crossing, and you have classified the tool.

1.18 Walkthrough: Adding Input Validation with a Harness

The abstractions land best against a concrete run, so here is the opening scenario carried all the way through by a harness, step by step. The project is a small Python web application with a registration function in `app/registration.py`, a thin `app/validators.py`, and a test file in `tests/test_registration.py`. The task is the one sentence from Listing 1.2: add input validation to the registration form, with tests, and make the suite pass. Nothing about this example is language-specific in spirit; the same loop would run against an Express app in TypeScript or a service in Go.

The harness begins by *planning*. It restates the task into a short ordered plan — find the registration code and its tests, add a validation helper, call it from the registration function, add tests, and run the suite until it is green — which is the model's way of decomposing one instruction into the steps the loop will execute. Then it *discovers*: rather than asking you where the code is, it greps the project for `register_user`, finds the definition in `app/registration.py` and a reference in the test file, and reads all three relevant files to build its picture. No context was pasted; the harness assembled it from the task. The transcript in Listing 1.4 shows this sequence compressed into the form a harness's verbose output takes.

Listing 1.4: A compressed harness transcript: plan, discover, edit, test, observe the failure, correct, and re-test until green

```
> Add input validation to the user registration form ...

[plan]   1. find the registration code and its tests
         2. add a validate_registration() helper
         3. call it from register_user()
         4. add tests, then run the suite until green

[tool]   grep -rn "register_user" app/ tests/
         app/registration.py:4:def register_user(form):
         tests/test_registration.py:9: ... register_user(form)

[tool]   read_file app/registration.py        (40 lines)
[tool]   read_file app/validators.py          (12 lines)
[tool]   read_file tests/test_registration.py (31 lines)
```

```
[edit]   app/validators.py   (+18 lines)
[edit]   app/registration.py (+4 -0 lines)
[edit]   tests/test_registration.py (+22 lines)

[tool]   pytest -q
         F....                                  [100%]
         FAILED tests/test_registration.py::test_short_password
         AssertionError: ValidationError not raised
         1 failed, 4 passed in 0.41s

[observe] register_user() built the User before calling
          validate_registration(); the guard ran too late.

[edit]   app/registration.py (move validation above User(...))

[tool]   pytest -q
         .....                                  [100%]
         5 passed in 0.39s

[done]   task complete -- see summary
```

With the files in hand, the harness *edits*. It writes a `validate_registration` helper into `app/validators.py` that checks the email, the password length, and the trimmed display name; it adds a call to that helper inside `register_user`; and it adds three tests covering the new rules. These are real writes to disk, not proposals — after this turn the working tree differs. Then it *runs the tests*, and here the loop earns its keep: one test fails. The harness placed the validation call after the `User` row was already built, so the short-password case constructed the user before the guard ran and the expected error was never raised. A chat assistant would have handed you this bug to discover; the harness reads the failure straight from the `pytest` output.

Because it can *observe* the failure, it can *correct* it. The model reasons from the error message that the validation has to run before the row is constructed, edits `register_user` to move the guard above the `User(...)` call, and re-runs the suite. This time everything passes. Only now does the harness stop and report. Its *summary*, in Listing 1.5, names the files it changed, states the validation behavior it added, shows the verification result, and — importantly — is honest about what it did *not* do: the email check is a format test rather than a deliverability check, there is no rate limiting, and password strength beyond length is not enforced.

Listing 1.5: The harness's closing summary: files changed, behavior added, verification, and an explicit list of remaining risks and non-goals

```
Summary of changes

Files changed:
  app/validators.py        new validate_registration(form) helper
  app/registration.py      validate before building the User row
  tests/test_registration.py  3 new tests for the new rules

Validation behavior added:
  - email must be present and match a basic address pattern
  - password must be at least 10 characters
  - display_name is trimmed; 1-80 characters after trimming
  - a failed check raises ValidationError; the row is not saved
```

```
Verification:
  pytest -q -> 5 passed in 0.39s

Remaining risks / not done:
  - email check is a format test, not a deliverability check
  - no rate limiting or duplicate-email UX message added
  - password strength beyond length not enforced
```

The whole run came from one sentence, and the loop — plan, discover, read, edit, test, observe, correct, re-test, summarize — happened inside the tool. Your job was to write the task well and to review the diff and the summary at the end, including that list of remaining risks, which is exactly the kind of judgment the harness cannot make for you. That division of labor — you set intent and review outcome, the harness runs the loop in between — is the working relationship the rest of this book teaches you to operate well (Figure 1.3).

Figure 1.3: A harness does not merely answer once; it loops through planning, tool use, observation, correction, and verification until the task is complete.

1.19 Comparing the Same Task Across Tool Categories

Running the same registration task through each station of the spectrum makes the categories vivid. With *autocomplete*, you write the validation yourself and the tool helps you type it: as you start the `if not email` guard it offers a plausible completion, which you accept or correct, line by line. The thinking, the file navigation, the testing — all yours; the tool shaves keystrokes off the typing. With a *chat assistant*, you paste the registration function and the model, describe the rules, and get back a well-explained block of code and some tests, which you then copy into the

right files, run, and — when a test fails — paste the failure back to get a revision. The reasoning is excellent and the labor is entirely yours.

With an *inline IDE assistant*, you open the registration file, ask in the editor's chat to add the validation, and the tool proposes a diff across the file (and perhaps the test file) that you review and apply; if you ask it to, it runs the tests, and you drive the next step based on the result. You are still turning the crank, but the crank is shorter. With an *agentic harness*, you type the one sentence and review the finished diff and summary, as the walkthrough showed. The same intent flows through four very different amounts of human labor and human control.

Table 1.1 consolidates the comparison into the reference the rest of the book leans on. Read the capability columns as "can the tool do this on its own initiative," and read the whole table as capability-first: the brands in the examples column will change, but the rows — defined by what the tool can see and do — are durable. The cell values are necessarily compressed, and several of them are configuration- and version-dependent, especially for the inline-assistant row, whose tools increasingly offer an agent mode that moves them into the bottom row; the prose in this chapter is the nuance behind the cells.

Category	Examples	Read?	Edit?	Shell?	Loop?	Human role	Main risk	Best use
Static autocomplete	Copilot inline, Tabnine	Buffer only	No	No	No	Writes and steers every line	Plausible wrong code accepted unread	In-flow typing of known code
Chat assistant	ChatGPT, Claude.ai web	No	No	No	No	Pastes context, applies, tests	Stale or guessed context; no verification	Reasoning, design, learning
Inline IDE assistant	Cursor, Windsurf	Yes	Yes	Partial	Partial	Reviews each proposed step	Over-trusting unread edits	Context-aware edits with you
Agentic harness	Claude Code, Aider, OpenCode, Goose, Codex	Yes	Yes	Yes	Yes	Sets intent, reviews outcome	Destructive or manipulated actions	Multi-step tasks over a repo

Table 1.1: The tooling spectrum at a glance. Capability columns mean "on the tool's own initiative." Examples are illustrative and version-dependent; the rows, defined by capability, are the durable part.

1.20 How to Classify Any New AI Coding Tool

The point of a taxonomy is to outlive the examples in it. New tools will appear constantly while this book ages, and the way to place any of them — without trusting its marketing — is to ask a short, fixed series of questions about capability. Each question targets one of the boundaries between categories, and the answers locate the tool on the spectrum regardless of what it calls itself.

1. **Can it inspect files without you pasting them?** If no, it is a chat assistant or autocomplete. If yes, it is at least an inline assistant.

2. **Can it modify files directly?** If it only suggests text you copy, it is left of the inline-assistant station. If it writes to disk, it is an inline assistant or a harness.

3. **Can it execute commands or tests?** This is the big divider. A tool that can run your shell and read the output has the verification capability that defines the right half of the spectrum.

4. **Can it observe command output and adapt?** Running a command is not enough; the question is whether the result feeds back into the next decision. If it does, you are looking at real agency.

5. **Can it repeat until success, budget exhaustion, or your interruption?** A genuine loop with a termination condition is the heart of "agentic." One-shot tools do not have it.

6. **Is there a permission gate?** Ask where approval happens and whether you can put yourself in-the-loop, on-the-loop, or after-the-loop. A tool that acts with no gate at all is one to treat with particular caution.

7. **Can tools be added through a schema — MCP, function calling, plugins, or similar?** Extensibility through registered tools marks a mature harness and tells you how its powers can grow.

A tool that answers yes to the file, command, observation, and loop questions is an agentic coding harness, whatever its name, and you should configure its permissions accordingly. A tool that answers no to the command and loop questions is an assistant or an editor enhancement, and treating it as a harness will over-set your expectations. The questions, not the brand, are the classifier — and they will keep working on tools that do not exist yet.

1.21 Common Misconceptions About Coding Agents

A handful of mistaken beliefs cause most of the early confusion about harnesses, and naming them sharpens the mental model this chapter has built.

"A harness is just autocomplete with a bigger context window." It is a different category, not a bigger version of the same one. Autocomplete predicts text at your cursor and never acts; a harness reads, writes, runs commands, and loops. The difference is agency, not context size, and no amount of context turns a predictor into an actor.

"The AI runs my tests and edits my files itself." Mechanically, it does not. The model emits structured tool calls; the harness — ordinary code on the boundary — performs the file write or the test run and hands the result back. Keeping this straight is what makes the security model comprehensible: every real action is a tool call the harness chose to execute, which is exactly where a permission gate can sit.

"More autonomy is always better." Autonomy is a setting, not a virtue. The right amount depends on the task's reversibility and your trust in the tool for it; for most real development the sweet spot is human-on-the-loop, with full unattended autonomy reserved for narrow, sandboxed, reversible work. Maximizing autonomy maximizes both speed and the size of a mistake you cannot catch in time.

"If the tests pass, the change is correct." Passing tests mean the tests pass — not that the design is sound, the requirement was understood, or a security or performance implication was considered. The harness optimizes toward the signal it can see; your review supplies the judgment the tests cannot.

"Harnesses are dangerous, so serious developers avoid them." The danger is real and it is manageable with the practices this book teaches — a permission gate, clean version control, secrets kept out of reach, and a sandbox for anything beyond routine. Avoiding harnesses to avoid the risk also forgoes the leverage; the goal is to operate them well, not to abstain.

"Picking a harness is about picking the smartest model." Model quality matters, but the harness is the architecture around the model — its tools, its loop, its permission gates, its context handling — and two harnesses driving the same model can behave very differently. This book is about the architecture precisely because it is the part that the "which model" conversation tends to miss.

1.22 Hands-On Lab: First Agentic Session

This is the book's first lab, and its goal is not to finish a feature but to *watch the loop* and feel the difference between the categories with your own hands. You will run the same small task through two harnesses and record what each one does — which files it reads, which commands it proposes or runs, whether it asks before acting, and whether it recovers from a failure on its own.

Lab 1 — Run the Same Task Through Two Harnesses

Setup. Create a tiny three-file Python CLI project — a word-counter is plenty — so the task is small enough to follow every step. Then install two harnesses, Claude Code and Aider. The commands in Listing 1.6 are a starting point, but package names and installation methods change; verify the official, current install instructions for each tool before running them, and expect to provide each tool with credentials or a model endpoint as its documentation describes.

Listing 1.6: Lab scaffold: a tiny CLI project and the two harness installs to verify against current official instructions

```
# Verify the official install instructions for the current
# versions before running these -- package names and flags change.

# A tiny 3-file Python CLI project for the lab
mkdir wordcount && cd wordcount
printf 'def count(text):\n    return len(text.split())\n' > wc.py
```

```
printf 'from wc import count\nimport sys\nprint(count(sys.stdin.read()))\n' > cli.py
printf 'from wc import count\n\ndef test_count():\n    assert count("a b c") == 3\n' > test_wc.py

# Install Claude Code (Node) and Aider (Python). Names may change.
npm install -g @anthropic-ai/claude-code
pip install aider-chat

# Run the same task through each, one at a time, in this folder:
#   "Add a --lines flag to cli.py that prints the line count too,
#    and add a test for it. Run the tests before finishing."
claude
aider
```

The task. In the project folder, give each harness the same instruction, one at a time: *add a* `-lines` *flag to* `cli.py` *that also prints the line count, add a test for it, and run the tests before finishing*. Let each tool work, and keep yourself *in-the-loop* — approve actions as it asks — for this first run, so you see every step.

Introduce a failure. On a second run with each tool, deliberately break something before you start — introduce a syntax error in `wc.py`, or change the existing test so it fails — and give the same task. The point is to watch how each harness notices the problem: does it run the tests, read the failure, and correct itself, or does it stop and ask you?

Record, for each tool:

- which files it *read*, and whether it found them on its own or asked you;
- which shell commands it *proposed or executed*, and whether it ran the tests;
- whether it *asked for confirmation* before editing or running, and at which steps;
- whether it *corrected itself* after the failure, and how it diagnosed it;
- how many *turns* it took to finish, and whether it hit any limit;
- what *you* had to do manually that you expected the tool to do, or vice versa.

What to take away. You are not benchmarking the tools against each other — they differ in defaults and will both finish this easy task. You are building the habit of watching the loop: noticing where the permission gate sits, where the tool reads versus guesses, and where self-correction does or does not kick in. That habit is the foundation for every later chapter, and you will repeat this benchmark task against more harnesses as the book goes on. Keep your notes; Chapter 12, on choosing a harness per task, refers back to exactly this kind of observation.

1.23 Interview Questions

These questions check that the chapter's vocabulary has become usable. The answers are deliberately practical and brief — the way you would actually explain the idea to a colleague or in an interview.

How does a harness like Claude Code differ from a standard autocomplete extension? An autocomplete extension predicts text at your cursor from nearby code and never acts on its own: it has no view of a task, cannot open files you did not open, cannot run anything, and never loops. A harness runs an autonomous loop — it reads files it chooses, writes edits to disk, runs shell commands such as your test suite, reads the output, and corrects its own mistakes across many turns until the task is done or a limit stops it. The difference is agency and the loop, not context size: autocomplete shaves keystrokes off code you are already writing, while a harness can carry a multi-step change from one instruction.

What are the security risks of unattended shell access for coding agents? The shell can do nearly anything your account can, so an agent running commands unattended can run destructive ones — `rm -rf` on the wrong path, a force-push that rewrites history, a database drop — before you can react. It can exfiltrate data: a tool that reads any file can read your `.env` or keys, and with network access it can send them out via `curl`. And because the model's behavior is influenced by text it reads, a prompt injection planted in a README, a comment, or a dependency's docs can steer the agent into running a malicious command it was never asked to run. The mitigations are a permission gate on actions, a sandbox for anything non-routine, secrets kept out of the working tree, and a clean version-control state so every change is visible and reversible — the subject of Chapter 6.

Why is a tool call the right place to put a permission boundary? Because every real action a harness takes flows through a named, structured tool call with typed arguments, the harness can inspect each call before running it and decide to allow, confirm, or deny it. Free-form prose has no such checkpoint; a typed call does. The same structure that lets you extend a harness with new tools is what lets you gate the ones it already has.

When should you keep a human in the loop rather than on it? Keep a human *in* the loop — approving each action — for unfamiliar tools, sensitive repositories, and any action that is hard to undo or reaches outside the sandbox (deploys, deletions, anything that spends money or sends a message). Move to *on* the loop — watching and ready to interrupt — for routine, reversible work on a project you have learned to trust the tool with. Full unattended autonomy is for narrow, well-bounded, reversible tasks in a disposable environment, and rarely for ordinary development.

A vendor calls its product an "AI pair programmer." How do you find out what it actually is? Ignore the label and ask the capability questions: can it read files without you pasting them, edit them directly, run commands and read their output, adapt to that output, loop until done, and gate its actions — and can you add tools through a schema? The answers place it on the spectrum no matter what it is called, and tell you what permissions it needs and what it can break.

1.24 Chapter Summary

This chapter replaced the vague phrase "AI coding tool" with a precise spectrum and a definitional test. The spectrum runs from static autocomplete, through chat assistants, through context-aware inline IDE assistants, to agentic coding harnesses, and as you move along it the context the model can draw on, the autonomy it has, the permissions it needs, and the risk it carries all rise together. None of the categories is best; each is suited to different work, and a productive developer uses several in a day.

An agentic coding harness is the right-hand end of that spectrum: a program that connects a model to a real project and runs a loop in which the model chooses the next action — read, write, run — the harness performs it against the actual environment, and the result feeds back so the model can choose again, repeating until the task is done or a limit intervenes. A tool is a full harness only if it has all four capabilities: autonomous file-system read and write, shell command execution, a recursive self-correction loop, and tool registration through a schema such as MCP, function calling, or another tool-use mechanism. Remove any one and the tool belongs to a category further left.

Key Takeaways

- **Classify by capability, not brand.** The seven questions — inspect, modify, execute, observe, loop, gate, extend — place any tool on the spectrum and will outlive the examples in this book.
- **Agency lives in the harness, not the model.** The model only ever emits text; the harness is the code on the boundary that treats a structured tool call as a real action and performs it.
- **Four capabilities define a harness:** autonomous read/write, shell execution, a self-correcting loop, and tool registration through a schema. All four, or it is an assistant.
- **The tool call is the text-to-action boundary,** which is exactly why it is the natural place for a permission gate, and why harnesses are both extensible and gateable.
- **Choose your loop position deliberately.** In-the-loop for risky or irreversible work, on-the-loop for routine trusted work, after-the-loop only for narrow, sandboxed, reversible tasks.
- **Broad permissions are the price of action.** A harness must read, write, and run commands to be useful; manage the risk with a gate, a sandbox, secret hygiene, and clean version control rather than by abstaining.
- **Tests are a signal, not a verdict.** Green tests do not mean correct design, understood intent, or a considered security implication; your review supplies what the loop cannot.

1.25 What Comes Next

This chapter gave you the vocabulary — autocomplete, chat, inline assistant, and harness — and the four capabilities that earn a tool the word "agentic." What it deliberately left as a black box is the loop itself. You have seen *that* a harness plans, acts, observes, and corrects; you have not yet seen *how*, at the level of the actual exchange between the harness and the model.

Chapter 2 opens that box. It follows a single request through the agent loop in mechanical detail: how the harness assembles the context it sends to the model — the system prompt, the tool definitions, the conversation so far — how the model replies with a tool call rather than prose, how the harness validates and executes that call and feeds the observation back, and how the loop decides when to stop, whether by an explicit completion signal, a turn limit, a cost budget, or your interruption. You will see a real tool-call round-trip and the kind of verbose log a harness prints, so that the behavior you watched in this chapter's lab becomes something you can read and reason about line by line. With the vocabulary of Chapter 1 and the mechanism of Chapter 2 in hand, the harness stops being magic and becomes a system you can operate, extend, and trust exactly as much as it has earned.