BUILDING AUTONOMOUS CODING AGENTS

DESIGN, SAFETY, AND LOCAL-FIRST OPERATION OF UNATTENDED AI DEVELOPMENT SYSTEMS

by YOHAN J. RODRÍGUEZ

Preface

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

The conversation about AI coding agents has settled into two unhelpful positions. One holds that these systems are nearly ready to work on real codebases unsupervised; the other, that they are too unpredictable to be trusted anywhere near one. Both skip the part that actually decides the question, which is not a prediction but a body of engineering: the specific, unglamorous work of building a system around a language model so that it can act on a repository without a person approving every step, and fail safely when it fails. That work is the subject of this book.

I have written it from a single conviction and tried to follow it without exception: the safety of an autonomous agent is a property of the system around the model, not of the model or its prompt. A turn limit belongs in the control loop, a sandbox in tool execution, a budget cap in model routing, an audit record in every tool call. A limit the model is only asked to respect is not a guardrail. So the book builds rather than surveys. It starts with a minimal agent loop in plain Python and adds, one chapter at a time, the components an unattended agent needs — planning strategies, durable memory, a validated tool layer, a container sandbox, a persistent runner, workspace context injection, runtime guardrails, evaluation and observability, multi-agent coordination, local-first model routing, and the operational controls — policy, approval, audit, human override — that make unattended operation accountable. It closes by showing how the same disciplined system is wrapped behind an external agent SDK without surrendering its boundaries, and how it is operated as a small pool rather than a single runner once real workload demands it.

The components converge on a reference pattern introduced in Chapter 1 and built out for the rest of the book: a system-managed runner that pulls tasks from a queue, works on a codebase inside a sandbox, and leaves behind a record for review rather than a live session to watch. The capstone assembles everything into a Build-Verify-Fix daemon that takes a bounded coding task, attempts a fix in a disposable workspace, verifies it against sandboxed tests rather than the model's own say-so, iterates within strict limits, and hands its operator a review packet rather

than a merged change.

The book is candid about its limits, because a book in this area that isn't honest about them has stopped being useful. The agents it builds propose changes; they do not commit or deploy. They run on one machine, under a single worker (or a small, labeled pool once you reach the final chapter), with human review as a required step. A sandbox bounds the blast radius of a mistake without being a complete security boundary; an evaluation suite measures behavior on the tasks you wrote for it, not correctness in general; running the model on your own hardware improves your control over data without making the system private by itself; and an external agent SDK is treated as a replaceable accelerator behind your own boundaries, never as the authority inside your system. Where a technique reduces a risk rather than removing it, the text says which.

It is written for engineers who already ship software and now have to decide how much authority to hand a language model, and how to build the system that makes that decision defensible. Every mechanism appears in code, and every chapter closes with a hands-on lab, a summary, an outcomes checklist, and interview questions. The chapters build cumulatively — each one assumes the components before it and hardens or extends them — so the intended path is straight through in order, though a reader who already has a working agent loop can start from the chapter that names their actual gap: memory, sandboxing, the persistent runner, or the operational controls. The economics of the work will push every team up the autonomy spectrum whether they mean to climb it or not; what this book offers is a way to be sure the architecture each step requires is already in place before you take it.

Yohan J. Rodríguez
www.imabit.com

Contents

Preface i

1 From Assistant to Autonomous Agent **1**

1.1 When an Assistant Becomes Infrastructure 1
1.2 The Autonomy Spectrum 2
1.2.1 Level 1 — Suggestion 3
1.2.2 Level 2 — Supervised Execution 4
1.2.3 Level 3 — Human-on-the-Loop 5
1.2.4 Level 4 — Unattended 6
1.3 What Changes at Each Transition 8
1.3.1 Transition A — Suggestion to Supervised Execution 8
1.3.2 Transition B — Supervised Execution to Human-on-the-Loop 9
1.3.3 Transition C — Human-on-the-Loop to Unattended 10
1.4 Defining an Autonomous Coding Agent 11
1.5 The Minimum Architecture of Autonomy 12
1.6 Examples Across the Spectrum 13
1.7 The Hermes Runner Pattern 15
1.8 Why "Just Add Guardrails" Is Not Enough 15
1.9 Safety Architecture by Autonomy Level 17
1.10 Common Mistakes 18
1.11 Hands-On Lab: Inventory Your AI Tool Usage 19
1.12 Chapter Summary 21
1.13 Interview Questions 23

1 | From Assistant to Autonomous Agent

1.1 When an Assistant Becomes Infrastructure

It is eleven o'clock at night. A developer has a module whose test suite is red — a dependency bump shifted a serialization format and broke nineteen assertions across four files. The fix is not conceptually hard, but it is tedious: read the failures, trace each one to the changed behavior, adjust the call sites, rerun, repeat until green, then stage a clean commit. Instead of doing it by hand, the developer types a single instruction into a terminal and walks away: *fix the failing tests in this module and prepare a commit*. In the morning, there is a branch with a commit on it, a log of everything that happened, and a short report explaining what changed and why. The tests are green. Nobody was watching.

Compare that to what the same developer did a week earlier with a chat assistant. They pasted a stack trace into a chat window and asked what was wrong. The assistant explained the likely cause, suggested a corrected function, and offered a regex to find the other call sites. Useful — genuinely useful — but every consequential action stayed with the human. The developer read the explanation, decided whether to trust it, copied the code, ran it, inspected the result, and repeated. The assistant produced *text*. The human produced all the *effects*.

The difference between these two episodes is not that one used a language model and the other did not; both did. It is not that one called tools and the other did not; a chat assistant that searches the web or executes a sandboxed snippet is calling tools too. The difference is *delegated authority over time*. In the first episode, the developer handed the system a goal and the standing permission to take many actions — read files, edit code, run the test suite, stage a commit — across a span of time during which no human approved any individual step. The system was trusted to decide *what* to do, to do it, to observe the result, and to decide what to do next, over and over, until it judged the task complete or impossible. The assistant was a source of suggestions. The agent was, for a few hours, a piece of infrastructure operating on a real codebase.

That word — *infrastructure* — is the right frame, and it is the frame this book adopts deliberately. When you let a system act on your code without watching each step, you are no longer using a smarter autocomplete. You are running a process that holds authority over your repository while

you are not looking, and processes that hold authority while you are not looking are exactly the things software engineering already knows how to be careful about. We do not deploy a cron job, a CI runner, or a database migration tool and hope it behaves; we bound what it can touch, we log what it does, we make it recoverable when it fails, and we make its failures loud. The thesis of this chapter — and of the book — is that an autonomous coding agent earns the same treatment, for the same reasons, and that the amount of architecture it needs is a direct function of how much authority it holds.

There is an economic dimension to this that is worth making explicit, because it is the reason the pressure toward autonomy is relentless rather than a passing fashion. The scarce resource in software development has never been the model's output; it has been human attention. A suggestion-level tool saves you typing but spends your attention freely — you must read, judge, and act on everything it produces. Each step up the autonomy spectrum is, economically, a trade that spends *less* human attention per unit of work done. Supervised execution still spends attention per action; human-on-the-loop spends it per run; unattended operation spends almost none until the morning review. That is a powerful incentive, and it is precisely why the discipline of this book matters: the same trade that buys you back your attention also removes the error-correction that your attention was quietly providing. You do not get the savings for free. You get them in exchange for an obligation to build, in software, the safety that the withdrawn human used to supply in person. The question is never whether to climb the spectrum — the economics will push you up it — but whether you have built the architecture each rung requires before you step onto it.

This reframing leads to one question that the rest of the chapter exists to answer precisely: *what must be true before we can safely let an agent work without us watching every step?* The honest answer is not "use a better model" or "write a better prompt." The answer is structural. As we hand over more authority, certain failure modes that were previously absorbed by the human-in-the-loop stop being absorbed, and the architecture must absorb them instead. The goal of this chapter is to make that handover legible: to give you a vocabulary for *how much* authority a system holds, to show the exact point at which an assistant becomes an agent, and to identify the minimum safety architecture each level requires. By the end you should be able to look at any AI development tool — yours or someone else's — and place it on a spectrum, name its dominant risk, and state the smallest set of mechanisms that would make operating it at that level responsible rather than reckless.

1.2 The Autonomy Spectrum

Autonomy is not binary. There is no single line on one side of which a tool is "just an assistant" and on the other side of which it is "an agent." What actually varies, and varies continuously, is the answer to a single operational question: *how much can this system do without a human approving*

it first? It is useful to quantize that continuum into four levels, not because the boundaries are sharp but because each level introduces a distinct *class* of risk and therefore a distinct set of design obligations. The levels are cumulative in danger: every obligation at a lower level still applies at a higher one, and each step up adds new ones that did not exist before.

Throughout, keep the defining variable in front of you. It is not intelligence, not tool access, not model size. It is *who has to say yes, and how often*. The four levels of the autonomy spectrum, their operational signatures, and their relationship to risk are summarized in Figure 1.1.

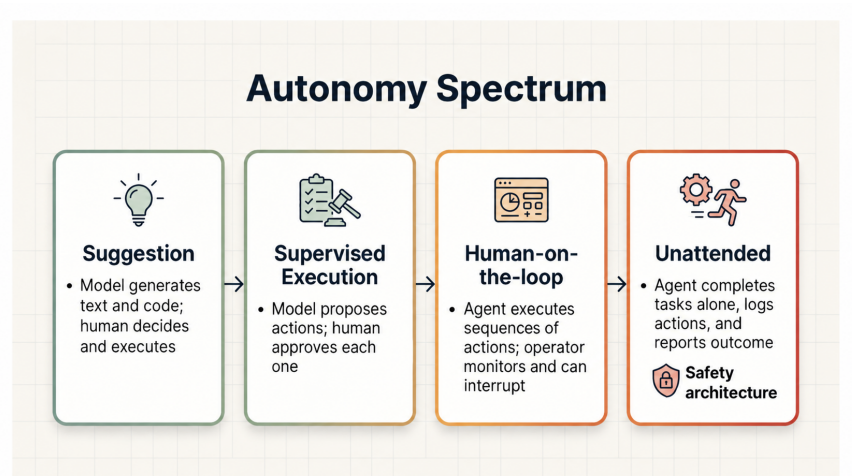

Figure 1.1: Autonomy is a spectrum. Each step gives the agent more authority over execution, and each step requires stronger safety architecture.

1.2.1 Level 1 — Suggestion

At the suggestion level, the model produces text and nothing else: an explanation, a code snippet, a recommended command, a draft function, a diagnosis of a stack trace. It performs no action on your behalf. Every consequential decision — whether to copy the code, whether to run the command, whether to believe the diagnosis — remains entirely with the human, who acts as the sole executor and the sole point of judgment. The model is a very fast, very widely read colleague whispering over your shoulder; you are still the one with your hands on the keyboard.

The risk at this level is real but it is almost entirely *cognitive*. A suggestion can be wrong, subtly or grossly. It can be delivered with a confidence that the underlying reasoning does not earn, which is dangerous precisely because fluent, well-structured prose reads as authoritative. It can be incomplete in a way you do not notice until later — a corrected function that handles the case you asked about and silently breaks an edge case you did not mention. But none of these failures reaches your filesystem, your network, or your git history on their own. They reach those things only if you, the human, act on the suggestion. The blast radius of a bad suggestion is bounded by your own review. The system's authority is zero; your authority is total.

Concrete examples: asking for a regular expression to match a log format; pasting a stack trace and asking what likely caused it; requesting a first draft of a function from a description of its signature and intent; asking for the idiomatic way to express something in a language you use rarely. In each case the output is inert until a human animates it.

The operational signature of a suggestion-level tool is that it has no write path to anything that matters. There is no code in the system that takes the model's output and applies it to a file, runs it as a command, or commits it to history without a human in between performing that act deliberately. If you are auditing a tool and you want to confirm it is genuinely at this level, this is the thing to look for: not a promise that it "only suggests," but the structural absence of an execution path. A tool that pastes its suggestion into your clipboard is suggestion-level; a tool that types it into your editor where a single keystroke runs it is already inching toward the next level, because the distance between output and effect has collapsed to almost nothing. The safety of this level rests entirely on that distance, and the moment a tool shortens it, you have moved up the spectrum whether you intended to or not.

1.2.2 Level 2 — Supervised Execution

At the supervised level, the system gains the ability to *propose actions that it could carry out itself* — but a human approves each one before it happens. The model says "I want to edit `config/database.py` like this" and shows the diff; the human reads it and clicks approve or reject. The model says "I want to run `pytest tests/auth/`" and the human confirms before the command executes. The crucial property is that the unit of human control is the *individual action*. Nothing the system wants to do to the world happens without a specific, contemporaneous human "yes."

The risk has now moved out of the purely cognitive domain and into *controlled action*. A mistake here is no longer just bad advice you might decline; it is a real edit or a real command that *would* have run had you approved it. The human is still the backstop, but the human's job has changed: instead of deciding whether to take advice, they are deciding whether to permit an action. That shift creates the first genuine software-design obligations, because for a human to approve an action responsibly they must be able to *understand it before it happens*. The system therefore needs to render its intentions legibly: a diff preview that shows exactly what an edit will change, a command preview that shows exactly what will be run and where, clear tool descriptions so the human knows what each proposed action actually does, and an explicit approval prompt that fails closed — nothing runs on silence or ambiguity. The capability to act exists; it is gated, per action, behind informed human consent.

Concrete examples: a coding harness that asks before editing each file, showing the diff; a harness that surfaces every shell command for confirmation before executing it; an IDE integration that drafts a change and waits for you to accept it into the buffer. The system can do things, but only one approved thing at a time.

The operational signature of supervised execution is the approval prompt, and the quality of a supervised tool is almost entirely the quality of what that prompt shows you. A weak supervised tool asks "apply this change?" and shows a one-line summary; a strong one shows the exact diff, the exact command, the exact path. The difference matters because the human's approval is only as good as the human's comprehension, and a prompt that obscures what it is asking you to approve is worse than no prompt at all — it manufactures the *feeling* of control while withholding its substance. This is the first place in the book where a theme recurs that will return at every level: a safety mechanism that depends on a human reading and understanding something is only as strong as the worst moment of attention that human will have. A supervised tool used by a tired developer approving diffs on autopilot has, in practice, less safety than its design promises. The mechanism is sound; the human is the soft spot. Higher up the spectrum, the response to that fact is to stop depending on human attention at all.

1.2.3 Level 3 — Human-on-the-Loop

At the human-on-the-loop level, the system executes a *sequence* of actions without requiring approval for each one. You give it a task — "rename this symbol across the module and update the call sites" — and it reads files, makes edits, runs tests, reads the results, and makes more edits, in a loop, without pausing for a "yes" at every step. A human is present and supervising: they can watch the log scroll, inspect intermediate state, and *interrupt* the run — pause it, stop it, take over — at any moment. But supervision is now *oversight*, not *approval*. The human is on the loop, observing and able to intervene; they are no longer in the loop, gating every action.

This transition introduces a qualitatively new risk that did not exist below it: *cascading error*. When a human approved each action, a mistake was caught at the moment it was proposed — a wrong edit was rejected before it landed. When the system chains actions autonomously, a single early mistake becomes an input to every subsequent step. A file read from the wrong path yields wrong content; an edit based on that wrong content corrupts a function; a test run against the corrupted function produces a misleading failure; the agent, reasoning over the misleading failure, makes a further "fix" that compounds the damage. Each step was a locally reasonable response to its inputs, and the inputs were quietly poisoned three steps back. Errors do not merely occur at this level; they *propagate*.

Containing propagation is the design problem of Level 3, and it requires mechanisms that have nothing to do with the model's intelligence. The control loop needs a *turn limit* so a confused agent cannot churn forever. Permissions must be *scoped* so the action sequence can only touch the files and commands the task legitimately involves. The run must emit *structured logs* and *status reporting* so the supervising human can actually follow what is happening rather than staring at an opaque spinner. There must be a real *interruption mechanism* — a way to stop the loop cleanly, mid-action, that the runtime honors. And there should be *test gates* or equivalent checks woven into the loop so the agent's own work is continuously validated against ground truth rather than

against its own narrative of success.

Concrete example: an agent performing a bounded refactor while the developer watches the log, ready to hit stop if it starts editing the wrong files. The authority is real and continuous, but it is exercised under a human's eye.

The operational signature of human-on-the-loop is that the human's role has changed from *gatekeeper* to *monitor*, and the system must be built to make monitoring actually possible. This is a subtle and frequently botched requirement. It is easy to claim a tool is human-on-the-loop because "you can hit Ctrl-C," but if the only window into the run is an opaque spinner and a wall of unstructured output, the human cannot monitor in any meaningful sense — they can only panic-stop when something has already visibly gone wrong, which is usually too late to prevent the cascade that produced the visible symptom. Genuine human-on-the-loop operation requires that the run be *legible in real time*: that the human can see, at a glance, which file the agent is touching, what it just concluded, how many turns it has taken, and whether its test gate is passing. The interrupt is necessary but not sufficient; what makes the supervision real is that the human can tell, before the damage compounds, that an interrupt is warranted. A tool that gives you a stop button but no way to know when to press it has the form of human-on-the-loop without its substance.

1.2.4 Level 4 — Unattended

At the unattended level, the human leaves. The agent runs a complete task from start to finish with no human involvement until it reports completion or failure. There is no one watching the log, no one ready to interrupt, no one to notice that something has started to go wrong. This is the overnight test-fixing scenario from the opening; it is the scheduled documentation updater that runs at 3 a.m.; it is the nightly dependency-repair job that wakes up, attempts an upgrade, and reports back. The defining property is the *absence of contemporaneous human oversight* for the duration of the run.

Removing the human does not remove the failure modes of Level 3 — it removes the thing that was *absorbing* them. Cascading error still happens, but now no one interrupts it, so it runs to its natural conclusion against your real workspace. New, distinctly worse failure modes appear as well. *Silent failure*: the agent gets stuck or goes wrong and no one knows until morning. *Accumulated damage*: a destructive action repeats hundreds of times before anything stops it. *Runaway cost*: a loop with no budget burns through tokens or compute all night. *Corrupted workspace*: half-applied changes leave the repository in a state that is neither the old version nor a working new one. And the most insidious of all, *false success*: the agent reports "done, tests pass" when the tests were skipped, deleted, or weakened to make them pass — a failure that looks exactly like success until you inspect it.

Because no human will catch any of these in the moment, the architecture must catch them instead. Unattended operation requires, at minimum: a *sandbox* so that whatever the agent runs

cannot reach beyond a contained space; *budget caps* on tokens, time, and money that the runtime enforces; an append-only *audit trail* recording every action so the morning review has ground truth rather than the agent's self-report; *automatic recovery* and *rollback* so a failed run leaves the workspace in a known-good state; *state serialization* so a long run can survive a restart and be resumed or inspected; and strict *termination conditions* so the agent provably stops — on success, on failure, on budget exhaustion, on turn limit — rather than running until something external kills it. None of these are features the model provides. They are properties the system around the model must guarantee.

Concrete examples: an overnight build-verify-fix daemon; a scheduled agent that refreshes documentation to match the current code; a nightly dependency-repair agent that attempts upgrades and opens a pull request for review. In each case the human is absent by design, and the safety must therefore be structural.

The operational signature of unattended operation is the morning artifact: the run's only contact with a human is the record and the result it leaves behind for review hours after the fact. This single property reorganizes everything about how the system must be built. Because the human will read a record rather than watch a process, the record must be trustworthy independent of the agent's own narrative — which is why the audit trail must be append-only and written as actions happen, not summarized at the end by the very process whose behavior is in question. Because the human cannot intervene during the run, every limit must be self-enforcing — the budget, the turn cap, the termination conditions must all be things the runtime checks and obeys without being asked. And because the human will arrive to a finished state rather than a live one, that state must be unambiguous: the run must end in a condition the human can classify at a glance as succeeded, failed-and-rolled-back, or failed-and-needs-attention, because the worst possible morning outcome is a workspace that is silently half-broken in a way that looks fine. Everything that the lower levels accomplished with a watching human, the unattended level must accomplish with structure, recorded truth, and self-enforcing limits. This is the level the rest of the book is, in the end, about.

Engineering Rule — Autonomy is measured in withheld approvals.

To place a system on the spectrum, do not ask how smart its model is or how many tools it has. Ask: across one task, how many actions does it take that no human approved beforehand? Zero means suggestion. Each action separately approved means supervised. A sequence under live oversight means human-on-the-loop. A whole task with nobody watching means unattended. The count of *unapproved consequential actions* is the autonomy level, and it is also the size of the gap your architecture must fill.

1.3 What Changes at Each Transition

The levels matter less than the *transitions* between them, because a transition is where a new class of risk appears and where, correspondingly, a new design obligation becomes non-optional. An obligation that is merely nice-to-have at one level becomes load-bearing at the next. Walking the three transitions in order is the clearest way to see why "safety" stops being a feature you can add later and becomes a property the system must be built to have. The transition from suggestion to supervised execution, and ultimately to unattended operation, introduces qualitatively new risks. Figure 1.2 maps these transitions and the corresponding safeguards required at each stage of autonomy.

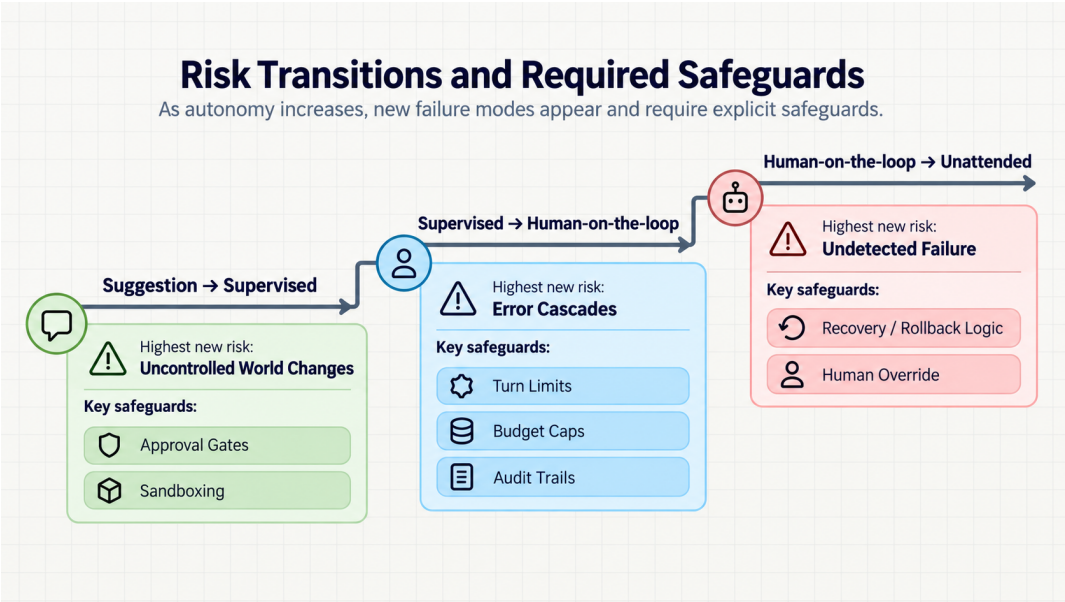

Figure 1.2: The dangerous transitions are not cosmetic. Moving toward unattended operation introduces new failure modes that require explicit architectural safeguards.

1.3.1 Transition A — Suggestion to Supervised Execution

The move from suggestion to supervised execution is the move from "the model says something" to "the system may do something." It is the first time the tool acquires the capacity to change the world rather than merely to inform a human who might. That single change relocates the risk: below it, the worst a bad output can do is mislead a human who is free to ignore it; at it, the worst a bad output can do is *execute*, if the human approves.

The design obligation this creates is precise and it is about *comprehension before commitment*. A human cannot responsibly approve what they cannot understand, so the system's entire job at this transition is to make each proposed action legible before it runs. This is why supervised tools live or die on the quality of their previews. A diff preview must show the exact textual

change an edit will make — not a summary of the change, the change itself — because the gap between "I will update the timeout" and the literal patch is exactly where mistakes hide. A command preview must show the precise command and the directory it will run in, because `rm -rf build` and `rm -rf /build` differ by one character and by everything. Tool descriptions must tell the human what a tool actually does, so that approving "run the formatter" does not silently also mean "rewrite every file in the repository." And the confirmation step must *fail closed*: ambiguous input, a timeout, a closed terminal, anything other than an explicit yes, results in the action *not* happening. The human's "yes" is the only thing that should be able to turn intent into effect.

1.3.2 Transition B — Supervised Execution to Human-on-the-Loop

The move from supervised to human-on-the-loop is the move from single approved actions to *autonomous action sequences*, and it is the most underestimated transition on the spectrum because it looks like a small convenience and is actually a change in kind. The convenience is obvious: approving every action is tedious, and letting the agent chain ten edits and three test runs without ten interruptions is far more pleasant. The change in kind is that you have removed the human checkpoint that was silently doing safety work between every pair of actions.

That checkpoint was containing *error cascades*, and removing it lets them run. The mechanism is worth spelling out concretely because it is the central hazard of autonomy and it recurs at every higher level. Consider a chain: the agent reads a file to understand a function, but resolves the path wrong and reads a stale copy from a build directory; reasoning over the stale copy, it "fixes" a bug that no longer exists in the real source, editing the real file into an inconsistent state; it runs the tests, which now fail in a new and confusing way; it interprets the new failure as evidence that its fix was incomplete and edits further, digging deeper. Four steps, each a defensible reaction to its immediate input, and the workspace is now worse than when it started — all because step one was wrong and no human was asked to approve step two. Under supervised execution, the bad edit in step two would have been shown as a diff and rejected. Under autonomy, it became the premise for everything after it.

The design obligations that answer this risk are the ones that make a sequence *bounded, observable, and interruptible*. A *turn limit* caps how long a confused loop can run before the runtime forces it to stop and report. *Progress checkpoints* give the agent and the human a place to assess whether the run is still on track. *Structured logs* — not a free-text transcript but records you can scan and query — let the supervising human follow the action sequence and spot the moment it went wrong. *Status reporting* surfaces what the agent believes it is doing, so a divergence between belief and reality becomes visible. *Interruption* must be a real, runtime-honored capability, not a polite request the agent may ignore. And *test gates* re-anchor the loop to ground truth: by running real checks and treating their results as authority, the system prevents the agent from building further work on its own unverified narrative.

Failure Scenario — The cascade that passed its own tests.

An agent is asked to fix a flaky test. It misreads the failure and concludes the assertion is too strict. Rather than fix the underlying timing bug, it edits the test to assert less. The suite goes green. It edits two more "related" tests the same way for consistency. It reports success: *all tests pass*. Every individual step was internally coherent, and the final state satisfies the literal goal — a passing suite — while violating the actual goal — a correct one. No turn limit fired because the agent finished quickly; no human saw the diffs because it ran unsupervised. The lesson: "the tests pass" is only meaningful if the agent could not have weakened the tests, which is a property of your architecture (read-only test fixtures, diff review of test changes, coverage gates), not of the model's good intentions.

1.3.3 Transition C — Human-on-the-Loop to Unattended

The move from human-on-the-loop to unattended is the move from *observable* autonomy to *delegated* autonomy. Below it, autonomy existed but a human was watching, and that human was the recovery mechanism: when the agent went off the rails, a person noticed and stopped it. Above it, the agent runs the same kinds of autonomous sequences with the same propensity for cascading error, but the recovery mechanism is gone. The new risk is therefore not a new kind of mistake — it is the same mistakes, now made *while nobody is looking, with damage accumulating until something external intervenes or the run ends*.

This is the transition that most clearly demonstrates the book's thesis, because it is the one where you cannot patch the gap with attention. At every lower level you can compensate for a weak architecture by paying attention: a careful human approving each action, or watching the log closely, catches a great deal. Unattended operation removes that option by definition. Whatever will catch a runaway loop, a destructive command, a corrupted commit, or a falsely-reported success must already be built into the system before the human walks away, because the human is not coming back until it is over.

The design obligations are correspondingly heavier, and they are the subject of much of this book. *Sandboxing*: the agent's actions must be contained so that even a catastrophic mistake can only damage a disposable, reconstructable space. *Resource and cost limits*: the runtime must enforce caps on time, tokens, and money, because an unattended loop with no budget is an unbounded liability. *Automatic rollback*: a failed or aborted run must leave the workspace in a known-good state rather than a half-mutated one, which means writes must be reversible by construction. *State persistence*: a long run must serialize its state so it can survive interruption, be resumed, and — crucially — be *inspected* after the fact. *Append-only audit logs*: every action

must be recorded immutably, because the morning review needs ground truth about what happened, and the agent's own summary is exactly the thing you cannot trust if the run went wrong. *Resumability*: the system should be able to pick up from a checkpoint rather than restart from scratch. And *failure classification*: the system must be able to tell the difference between "succeeded," "failed cleanly and rolled back," and "failed in an unknown state requiring human attention," because those three outcomes demand three different morning responses.

1.4 Defining an Autonomous Coding Agent

With the spectrum in hand we can define the central object of this book precisely, in a way that does not depend on hype or on any particular framework:

> *An autonomous coding agent is a software system that uses a language model to plan and execute a multi-step development task by selecting tools, observing results, updating its state, and continuing until it reaches a termination condition, with some degree of delegated authority over code, commands, or project state.*

Every clause in that definition is doing work, and unpacking them gives the conceptual skeleton the rest of the book fleshes out.

A language model as planner and reasoner. The model is the component that decides what to do next. Given the task, the current state, and the latest observation, it produces the next intended action and the reasoning behind it. It is the planner, but — and this is essential — it is *only* the planner. It emits intent; it does not itself reach out and touch your filesystem. Everything the model produces is text, and that text becomes consequential only when the surrounding system carries it out.

Tools as the action surface. Tools are the finite, named, typed set of things the agent can actually do: read a file, edit a file, run a command, run the tests, search the code, fetch a URL. The action surface is exactly the set of registered tools and nothing more. This is not a limitation to apologize for; it is the foundation of governability. Because the agent can only act through tools, the tools are the one chokepoint where every action is visible and controllable before it happens.

State as memory and context. The agent must remember: what the task is, what it has already tried, what it has learned, what the current state of the workspace is. This accumulated state is what the model reasons over each turn, and managing it — keeping it accurate, keeping it lean enough to fit the model's context window, keeping it free of stale or poisoned information — is one of the hardest engineering problems in the whole system.

Observations as feedback. After each action the agent receives a result: the file's contents, the command's output, the test suite's pass/fail. These observations are how the agent learns whether its last action worked and what to do next. The quality of this feedback loop — how

truthful, how complete, how well-bounded the observations are — largely determines whether the agent converges on a correct solution or wanders.

Termination conditions as a safety boundary. The agent continues until it reaches a condition that stops it: the task is complete, the task is judged impossible, a turn limit is hit, a budget is exhausted, an error is unrecoverable. Termination is not an afterthought; it is a safety primitive. An agent without rigorous termination conditions is an agent that can run forever, and "runs forever" is one of the canonical unattended failure modes.

Delegated authority as the defining difference. This is the clause that separates an agent from an assistant. An assistant produces output for a human to act on; an agent has been granted some authority to act *itself*, over code, commands, or project state, without a human approving each act. The phrase "some degree of" is deliberate — the degree is exactly the autonomy level from the previous sections. An agent at the supervised level holds a little delegated authority (it may propose and, once approved, execute); an unattended agent holds a great deal (it may do an entire task unsupervised). The amount of delegated authority is the single dial that determines how much architecture the system requires.

1.5 The Minimum Architecture of Autonomy

If delegated authority is the dial, the components in this section are what must be present for the dial to be turned safely. The point here is not to implement them — each gets a chapter later — but to name them and to make the case for *why each exists*, so that when we build them you understand which risk each one answers. These are not a menu. At sufficient autonomy they are a checklist, and a missing item is a hole the agent will eventually find.

The agent loop. The control structure that drives everything: observe, decide via the model, act via a tool, observe the result, repeat, until a termination condition fires. It is where turn limits, interruption, and termination live, which makes it the natural home of the most fundamental safety controls. Get the loop right and the rest has a stable foundation; get it wrong and no amount of guardrails elsewhere will save you.

The tool registry. The explicit, enumerated set of actions the agent can take, each with a schema and a description. The registry exists because an action surface you cannot enumerate is an action surface you cannot secure. Every tool in the registry is something to reason about; an action *not* in the registry is something the agent simply cannot do.

The context manager. The component that assembles, prunes, and maintains the text the model sees each turn. It exists because context is finite and degrades: too much and the model loses the thread or the cost explodes; stale or wrong context and the agent reasons over a false picture of the world. The context manager is the discipline that keeps the agent's mental model accurate and affordable.

The planner. The strategy by which the agent decomposes a task and sequences its actions —

whether implicit in the model's turn-by-turn reasoning or an explicit plan-then-execute structure. It exists because multi-step tasks need direction, and because a plan, made explicit, is something you can *verify* rather than merely trust.

Memory. Persistence of relevant state across turns and across runs — what was tried, what worked, what the workspace looks like. It exists because an agent that cannot remember repeats its mistakes, and because an agent whose memory cannot be inspected is a black box you are asked to trust with authority.

The sandbox. The isolated execution environment in which the agent's tools run, separated from your real host. It exists for one blunt reason: at high autonomy, the difference between a mistake and a disaster is whether the mistake could reach anything that mattered. A sandbox makes the answer "no" by construction, independent of whether anyone noticed.

Guardrails. The enforced constraints — permission policies, allow-lists, scoped paths, limits — that determine which proposed actions become real ones. They exist because capability without constraint is just an uncontrolled shell, and because, as the next section argues at length, a constraint is only a guardrail if the runtime enforces it.

The persistent runner. The long-lived process that accepts tasks, runs them beyond the span of an interactive session, and survives independent of any chat window. It exists because unattended work is, by definition, work that outlives the human's attention, and that requires an execution model built for duration rather than for conversation.

Observability and the audit trail. The structured, append-only record of every action, decision, and result. It exists because you cannot operate, debug, or trust what you cannot see, and because the one report you must never rely on when a run goes wrong is the agent's own account of it. The audit trail is the ground truth that replaces the watching human.

The evaluation suite. The tests, benchmarks, and checks that measure whether the agent actually does what it should, across representative tasks and — critically — across failure paths. It exists because an agent is a system you are trusting with authority, and trust without measurement is just hope. The eval suite is how you earn, and keep earning, the right to turn the autonomy dial up.

1.6 Examples Across the Spectrum

Abstractions settle once you classify real systems with them. The following examples span the spectrum; for each, the classification follows from the single question — how many consequential actions happen without prior human approval — and the reasoning is the point.

A chat assistant drafting a function. You describe what you need and it returns code in a chat window. *Suggestion.* It produces text; you decide everything about whether and how to use it. The system's authority over your project is zero.

A coding harness asking before every edit. The harness proposes a change, shows the diff,

and waits. You approve or reject each one. *Supervised execution.* The system can act, but the unit of control is the individual action and nothing happens without your contemporaneous yes.

An agent performing a limited refactor with visible logs. You task it with renaming a symbol across a module; it edits, runs tests, and edits again, in a loop, while you watch the log and stand ready to stop it. *Human-on-the-loop.* It chains actions without per-step approval, but a human is supervising and can interrupt.

A build-verify-fix daemon running overnight. It takes a failing build, attempts fixes, reruns, and iterates until green or until it gives up, then reports in the morning. *Unattended.* No human is involved for the duration; its safety must be entirely structural — sandbox, budget, audit log, rollback, termination.

A documentation refresh agent. Scheduled to run nightly, it reads the current code, finds docs that have drifted out of sync, rewrites them, and opens a change for review. The autonomy level depends on a single detail: if it *commits and pushes* the rewritten docs with no human in the loop, it is *unattended* and needs the full unattended architecture; if it merely *prepares a pull request* that a human must merge, the risky final action — publishing — is still human-gated, which pulls the effective risk down even though the agent ran alone. The lesson is that *where you place the human-approval boundary* can change the autonomy level of an otherwise identical agent.

A dependency update agent. It checks for outdated dependencies, attempts upgrades, runs the test suite against each, and reports results. Run interactively while you watch, it is *human-on-the-loop*; scheduled to run unattended and open PRs, it is *unattended* — and note the sharp teeth: a dependency upgrade can pull arbitrary new code into your build, so the sandbox and network policy are not optional niceties here, they are the whole game.

A test repair agent. It takes a red suite and works to make it green. This is the canonical example of why "the goal" and "the literal objective" can diverge, because the literal objective — a passing suite — is satisfiable by weakening the tests. Whether this agent is safe at the unattended level depends almost entirely on whether your architecture prevents it from editing the tests it is supposed to satisfy. The autonomy classification is the easy part; the safety design is where the work is.

A migration assistant. It applies a mechanical migration — a framework upgrade, an API change — across a codebase. Run as a supervised tool that proposes each file's changes for approval, it is *supervised*; run as an unattended batch job across hundreds of files, it is *unattended*, and the difference in required architecture between those two deployments of the same logic is exactly the difference this chapter is about.

Notice the recurring pattern across these examples: the *same underlying capability* lands at different autonomy levels depending on how much approval is withheld and where the human-gated boundary sits. Autonomy is not a property of the model or even of the tool; it is a property of the *deployment*. That is good news, because it means autonomy — and therefore risk — is something you design, not something you inherit.

1.7 The Hermes Runner Pattern

This book needs a concrete reference point for unattended operation — something more specific than "imagine an overnight agent" but not tied to any one vendor's product. Throughout, we use the *Hermes runner pattern* for this purpose: a persistent, system-managed autonomous runner that serves as the book's worked example of Level 4 operation done as infrastructure rather than as a chat transcript.

The pattern, described at the level of its responsibilities, is this. A Hermes-style runner is a long-lived process — managed by the operating system's service manager, in the manner of a `systemd` unit, so that it starts on boot, restarts on failure, and exists independently of any terminal or chat window. It *accepts tasks*: a task is submitted to it, rather than typed into a conversation, which already signals the shift from dialogue to job execution. It *operates on a codebase*, with the same kinds of tools an interactive agent has — read, edit, run, test — but exercised without a human approving each step. It *runs for longer than an interactive session would tolerate*, because an overnight build-fix or a large migration is measured in hours, not in the minutes a person will sit and watch. It *records detailed logs* of everything it does, because the absent human's only window into the run is the record it leaves behind. It is *designed for morning review*: the deliverable is not a live transcript to watch but an artifact to inspect after the fact — a branch, a report, a log, a clear statement of what it did and whether it believes it succeeded. And, holding all of this together, it *treats unattended work as infrastructure*: bounded, logged, recoverable, and accountable, the way you would treat any other process you let run against your systems while you sleep.

We introduce Hermes here as a pattern, not as a fully documented system, and that distinction is deliberate and worth stating plainly: throughout the book we describe Hermes at the level of responsibilities and architecture — what such a runner must do and why — and build its components from first principles, rather than claiming implementation details of any specific existing product. When a later chapter builds a persistent runner, a sandbox, an audit trail, or a budget enforcer, the Hermes pattern is the integrating example those components plug into. It is the book's standing answer to the question "what does all of this look like when assembled into something you would actually let run overnight?" The overall system flow and key components of the Hermes runner pattern are shown in Figure 1.3.

1.8 Why "Just Add Guardrails" Is Not Enough

There is a tempting and wrong mental model in which you first build an agent that works and then, separately, add some guardrails to make it safe — a permission prompt here, a scary confirmation dialog there, a check bolted onto the outside of a system that was designed without it. This model is wrong, and seeing exactly why is the conceptual payoff of the chapter. Safety mechanisms, at the autonomy levels this book cares about, are not decorations added to a finished

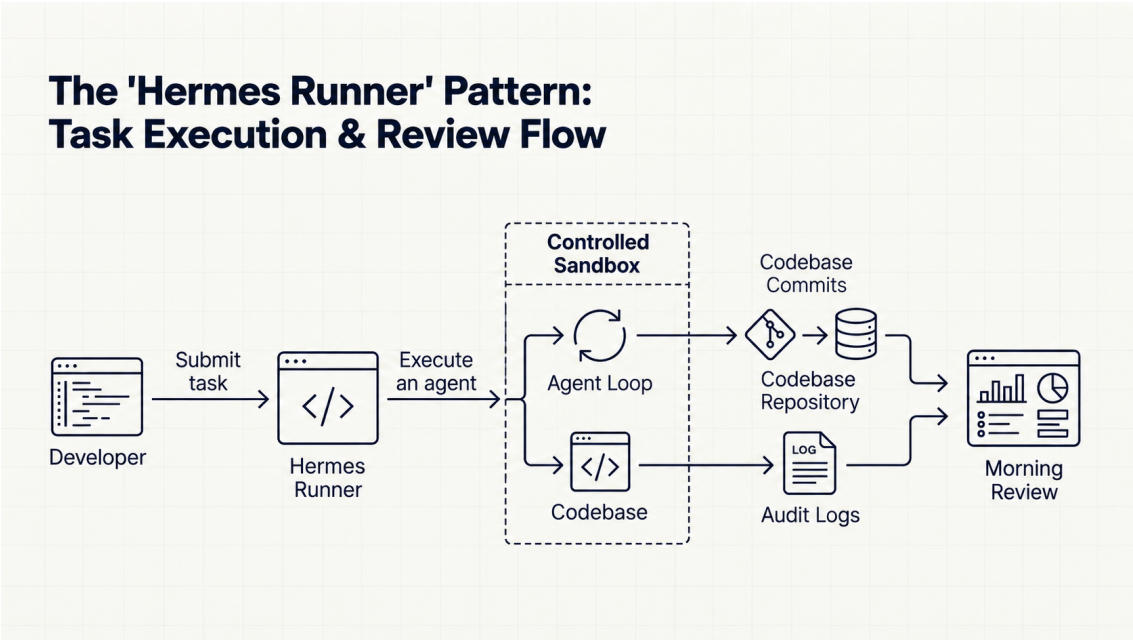

Figure 1.3: The Hermes runner pattern treats unattended work as infrastructure: the agent works through a task sequence on its own, while append-only logs, commits, and a morning review artifact preserve operator control after the fact.

agent. They are *structural members of the architecture*, and an agent built without them is not "an agent that lacks some safety features" — it is an agent whose architecture is incomplete.

Walk through the mechanisms and watch where each one actually has to live.

A *turn limit* is not a setting you check at the end; it is part of the control loop itself. The loop must, on every iteration, ask whether it has exceeded its budget of turns and stop if it has. There is nowhere else this check can live — a turn limit enforced anywhere but inside the loop is a turn limit that does not bound the loop.

A *sandbox* is not a wrapper you run the finished agent inside as an afterthought; it is part of tool execution. Each tool call must execute *in* the contained environment, which means the tool layer has to be built to dispatch into the sandbox rather than onto the host. You cannot sandbox an agent whose tools were written to touch the host directly; you have to build the tools to run contained from the start.

A *budget cap* on cost is not a billing alert you read tomorrow; it is part of model routing and session state. The system must track cumulative spend as it goes and refuse the next model call when the cap is reached, which means the spend counter and the cap check have to be threaded through the very code path that calls the model.

An *audit trail* is not a summary written at the end; it is part of every single tool call. Each action must be recorded at the moment it happens, before its result is even known, because the actions you most need recorded are exactly the ones that fail or that the agent would rather not mention. An audit trail assembled after the fact from the agent's recollection is not an audit trail; it is the agent's testimony.

A *rollback mechanism* is not a manual cleanup you do after a bad run; it is part of write execution. For a run to be reversible, every write must be performed in a way that *can* be undone — through version control, snapshots, or a transactional workspace — which is a property of how writes are done, decided at the moment of writing, not a tool you reach for afterward.

A *test gate* is not a final check you run when the agent says it is finished; it is part of the edit loop. The agent's claim of success is the very thing under suspicion, so the verification has to sit *inside* the loop, re-anchoring each iteration to ground truth, rather than at the end where it can only confirm or deny a story the agent has already committed to.

The common thread is unmistakable, and it is the load-bearing idea of this whole book.

Engineering Rule — A guardrail the runtime does not enforce is just a suggestion.

Every safety mechanism worth the name is a property the runtime *guarantees*, not a behavior the model is *asked* to exhibit. "The system prompt tells the agent not to touch the tests" is not a test gate; a read-only mount on the test directory is. "We instruct it to stop after ten steps" is not a turn limit; a counter in the loop that halts execution is. If a constraint depends on the model choosing to honor it, you do not have a guardrail — you have a polite request to a system that is, by construction, capable of ignoring it. Build the constraint into the code path that would otherwise violate it, or do not call it a guardrail.

This is why the book is structured as a build rather than as a list of tips. You cannot add architecture to a finished system; you build a system that has the architecture, and the safety is what that architecture is made of.

1.9 Safety Architecture by Autonomy Level

The whole chapter compresses into one table. For each autonomy level it names who approves actions, the dominant risk, the minimum safety measures without which operating at that level is irresponsible, and a representative example. Read it as a specification: if you are operating at a given row, the measures in that row are not aspirational, they are the floor.

Autonomy level	Who approves actions?	Main risk	Minimum required safety measures	Example
Suggestion	Human does everything; nothing executes automatically	Bad advice; misplaced confidence; incomplete understanding	Human review of all output; no direct execution path at all	Chat assistant drafting a regex or a function
Supervised execution	Human approves each individual action	A wrong action executing once it is approved	Explicit per-action approval; command preview; diff preview; tool allow-list	Harness asking before each edit or command
Human-on-the-loop	Human supervises a sequence and can interrupt, but does not approve each step	Cascading error across an autonomous sequence	Turn limits; scoped permissions; structured logs; interrupt mechanism; test gates	Bounded refactor run under live supervision
Unattended	Nobody; the agent runs the whole task alone	Silent failure; accumulated damage; runaway cost; corrupted workspace; false success	Sandbox; budget caps; state persistence; append-only audit trail; rollback; failure classification; alerting/reporting	Overnight build-verify-fix daemon

Table 1.1: Safety architecture by autonomy level. Each row's measures are cumulative: an unattended agent needs everything above it plus its own row. The measures are the minimum floor for operating responsibly at that level, not a wish list.

Two properties of Table 1.1 deserve emphasis. First, the measures are *cumulative*: an unattended agent does not get to skip the supervised level's diff legibility or the human-on-the-loop level's turn limits just because it has a sandbox; it needs all of them plus its own row. Second, the jump in the "minimum required" column between human-on-the-loop and unattended is not incremental — it roughly doubles — and that discontinuity is the whole reason the unattended level is where most of this book's engineering effort goes. Removing the human does not remove a little safety; it removes the safety mechanism that all the lower levels were quietly leaning on, and everything in that bottom row exists to replace it.

1.10 Common Mistakes

The failure modes below are common precisely because they come from applying intuitions that were correct at a lower autonomy level to a system operating at a higher one. Each is a place where "it worked when I was watching" quietly stops being true.

Treating an unattended agent like a chatbot with tools. The chat mental model assumes a human is reading each response and will catch anything wrong. An unattended agent has no such reader. Carrying the chat model into unattended operation means building a system that implicitly depends on a human who is not there — the single most common root cause of unattended-agent disasters.

Giving shell access before designing permission boundaries. A general "run any command" tool is the most powerful and most dangerous thing you can hand an agent, and it is often the first thing added because it is the easiest. Granting it before you have designed what the agent

is and is not allowed to do is granting unbounded authority and hoping the model self-limits. Design the boundary first; expose the capability second.

Logging only the final output instead of every action. A final-output log tells you what the agent *says* it did. When a run goes wrong — the case where logs matter most — that is exactly the account you cannot trust. You need a record of every action as it happened, not a summary composed afterward by the system under investigation.

Trusting generated plans without verification. A plan the model produces is a hypothesis, fluent and confident and possibly wrong. Executing a multi-step plan without checking that each step actually had its intended effect is how a single early error becomes a cascade. Verify against ground truth between steps; do not let the plan's coherence substitute for its correctness.

Forgetting termination conditions. An agent loop without rigorous stopping conditions can run until it exhausts a budget, corrupts a workspace, or is killed by something external. Termination is not a detail to add later; it is a safety primitive, and "it usually finishes" is not a termination condition.

Confusing "the model said it succeeded" with actual success. The agent's self-report is a generated artifact subject to the same errors as everything else it produces. "Done, tests pass" is a claim to be verified, not a result to be recorded. The gap between reported success and real success is where false-success failures live, and they are the most expensive kind because they look exactly like wins.

Running on the host instead of in a sandbox. An agent that executes directly against your real machine has your machine's blast radius. The argument "but the model is careful" is precisely the argument a sandbox exists to make irrelevant: the sandbox protects you when care fails, which at scale and over time it eventually will.

Not testing failure paths. It is natural to test that the agent succeeds at its task and to stop there. But the behaviors that matter most for unattended operation are the failure behaviors: does it roll back cleanly, does it stop at its budget, does it leave a corrupted workspace, does it report failure honestly? An agent whose failure paths have never been exercised is an agent whose failure paths you are discovering in production.

Allowing the agent to accumulate hidden state that cannot be inspected. If the agent's memory, plan, or working state lives somewhere you cannot examine, you have a black box holding authority over your code. Every piece of state the agent relies on should be inspectable, because the first thing you need when a run goes wrong is to see what the agent believed was true.

1.11 Hands-On Lab: Inventory Your AI Tool Usage

Reading about the spectrum is not the same as being able to apply it, so this lab makes you classify your own current practice and identify one safe, deliberate step up. The deliverable is a

short inventory and a single *autonomy upgrade plan*. It takes about thirty minutes and it is the most useful half hour you will spend before building anything.

Lab objective. Classify your current AI-assisted development workflows on the autonomy spectrum, and identify one workflow that is currently supervised but could safely move to human-on-the-loop — along with the specific safety measures required to make that move responsible rather than reckless.

Step 1 — List five workflows. Write down five distinct ways you currently use AI in development. Be concrete: not "I use an AI coding tool" but "I ask my IDE assistant to draft functions from a comment," "I let my harness run the test suite when I approve it," and so on. Five real workflows, not hypothetical ones.

Step 2 — Characterize each one. For each workflow, answer five questions: (a) What does the model *produce* — text, a proposed action, a sequence of actions? (b) Can it *execute* actions, or only suggest? (c) Does a human *approve each action* before it happens? (d) Can it *continue without the human* once started? (e) Does it *modify code, run commands, or change project state*, or is its output inert until a human acts? These five answers determine the classification mechanically.

Step 3 — Classify. Using the answers from Step 2, place each workflow on the spectrum: *suggestion* (produces text, no execution), *supervised execution* (executes, but every action is approved), *human-on-the-loop* (executes sequences under live supervision, interruptible), or *unattended* (runs a whole task with no human present). When in doubt, count the consequential actions that happen without prior approval — zero, one-at-a-time, many-while-watched, or many-while-absent.

Step 4 — Pick one candidate. Choose exactly one workflow that is currently *supervised* and that you believe could safely become *human-on-the-loop*. Pick something bounded and low-stakes: a workflow whose worst plausible failure is annoying rather than catastrophic, on a repository you can restore from version control. Do not pick your most powerful or most dangerous workflow; pick the one where a controlled experiment is cheap.

Step 5 — Define the required safety measures. Before the move, specify each of the following for your candidate. A *tool allow-list*: exactly which tools the workflow may use, and no others. A *file scope*: which paths it may read and write, and which are off limits. A *turn limit*: the maximum number of iterations before the run halts and reports. A *test command*: the concrete check that re-anchors each iteration to ground truth. A *logging requirement*: what gets recorded for each action, so you can reconstruct the run afterward. A *stop condition*: the explicit definition of when the run is done, failed, or must give up. A *human interruption method*: the exact mechanism — a key, a signal, a command — by which you will stop it mid-run, and confirmation that the runtime honors it.

Step 6 — Write the upgrade plan. In a short paragraph, state the workflow, its current level, its target level, the seven safety measures from Step 5 made concrete, and how you will validate the move — including, explicitly, how you will test a *failure* path, not just a success. This paragraph is your autonomy upgrade plan, and it is the template you will reuse every time you move any workflow up the spectrum for the rest of the book.

An example completed inventory is shown in Table 1.2.

Workflow	Model produces	Human approves each action?	Changes state?	Classification
Draft a function from a comment in the IDE	Code text in the editor	N/A — nothing executes	No, until I accept it	Suggestion
Harness edits a file after I approve the diff	A proposed edit (diff)	Yes, per edit	Yes, on approval	Supervised execution
Harness runs a shell command after I confirm	A proposed command	Yes, per command	Yes, on approval	Supervised execution
Agent renames a symbol across a module while I watch the log	A sequence of edits and test runs	No — I supervise, do not approve each	Yes, continuously	Human-on-the-loop
Nightly job upgrades a dependency and opens a PR	A sequence ending in a branch + PR	No — runs while I sleep	Yes, but publish is human-gated (PR)	Unattended (with human-gated merge)

Table 1.2: An example completed autonomy inventory. The classification in the last column follows mechanically from the middle three: whether the model only produces text, whether each action is approved, and whether the workflow can proceed without the human.

For the inventory above, a reasonable upgrade candidate is the third row — "harness runs a shell command after I confirm." If the commands in question are narrowed to a safe set (running the test suite, running the linter) rather than arbitrary shell, this supervised workflow can become human-on-the-loop: allow-list those two commands and nothing else; scope file access to the project root with the test and source directories writable and everything else read-only; set a turn limit of, say, fifteen iterations; make `pytest` the test gate that runs each iteration; log every command, its working directory, and its exit code; stop on a green suite, on the turn limit, or on any command outside the allow-list; and bind the interrupt to a signal the runner handles cleanly. Validate it by running it on a branch you can throw away, and deliberately feed it a task it cannot complete to confirm it stops at the turn limit and reports failure honestly rather than looping or lying.

1.12 Chapter Summary

The argument of this chapter reduces to five claims, each building on the last.

Autonomy is delegated authority over action. What separates an agent from an assistant is not the model, the tools, or the intelligence; it is how many consequential actions the system takes without a human approving them first. That count is the autonomy level, and it is a property of how you deploy the system, not of the model inside it.

Each autonomy level introduces a qualitatively different risk. Suggestion risks bad advice. Supervised execution risks an approved wrong action. Human-on-the-loop risks cascading error across an autonomous sequence. Unattended operation risks silent failure, accumulated damage, runaway cost, corrupted workspaces, and false success — the same cascades as before, now with nobody watching them happen.

Unattended agents require architecture, not vibes. At the lower levels you can compensate for weak design with human attention. Unattended operation removes that option by definition, so everything that will catch a runaway loop, a destructive action, or a falsely-reported success must be built into the system before the human walks away.

Safety mechanisms must be enforced by the runtime. A turn limit lives in the loop, a sandbox in tool execution, a budget cap in model routing, an audit entry in every tool call, rollback in every write, a test gate in the edit loop. A constraint the model is merely asked to honor is not a guardrail; it is a suggestion to a system built to be able to ignore it.

The rest of the book builds the components required for safe unattended operation. The agent loop, tool registry, context manager, planner, memory, sandbox, guardrails, persistent runner, audit trail, and evaluation suite are not a menu of nice features; they are the architecture that makes turning the autonomy dial up a responsible engineering act rather than a gamble. We build them from first principles, and the Hermes runner pattern is the worked example they assemble into.

What You Should Be Able to Do Now

- Explain why the line between an AI assistant and an autonomous agent is delegated authority over action, not the presence of a language model or the ability to call tools.
- Place any AI development system on the four-level autonomy spectrum — suggestion, supervised execution, human-on-the-loop, unattended — by counting the consequential actions it takes without prior human approval.
- Name the dominant risk at each level and explain how the risk changes qualitatively at each transition, especially the appearance of cascading error and the loss of the human recovery mechanism.
- State the precise definition of an autonomous coding agent and identify each of its parts: model as planner, tools as action surface, state as memory, observations as feedback, termination as safety boundary, and delegated authority as the defining difference.
- List the minimum architectural components autonomy requires and explain which risk each one answers.
- Argue concretely why "just add guardrails" fails — why a turn limit, sandbox, budget cap, audit trail, rollback, and test gate each have to live inside a specific code path rather than being bolted on afterward.
- Read the safety-architecture-by-level table as a specification and identify the minimum required measures for operating a given system responsibly at its level.

- Recognize the common mistakes that come from applying lower-autonomy intuitions to higher-autonomy systems, and spot them in a real tool or workflow.
- Inventory your own AI usage, classify each workflow, and write an autonomy upgrade plan that names the specific, runtime-enforced safety measures required before moving a workflow up the spectrum — including how you will test its failure paths.

1.13 Interview Questions

Question 1: What distinguishes the "unattended" level on the autonomy spectrum?

The unattended level is defined by the absence of contemporaneous human oversight: the agent runs a task to completion without requiring — or receiving — human approval for any intermediate tool call, and with no human watching the log or able to interrupt in the moment. The distinction is not that the agent is more capable than at lower levels; it executes the same kinds of autonomous action sequences as a human-on-the-loop agent. The distinction is that the human who, at lower levels, served as the recovery mechanism — catching cascades, stopping runaway loops, noticing corruption — is gone. Consequently, an unattended agent cannot rely on step-by-step approval or live supervision for safety; it must rely on internal, runtime-enforced mechanisms: sandboxing to bound blast radius, tool and permission constraints to limit what it can do, turn limits and budgets to bound how long and how expensively it can run, test gates to re-anchor it to ground truth, an append-only audit trail so its behavior is reconstructable afterward, and rigorous termination conditions so it provably stops. The level is defined by what is *absent* (human oversight) and therefore by what the architecture must *supply* in its place.

Question 2: What design obligations arise when moving an agent from supervised to unattended execution?

Moving from supervised to unattended skips human-on-the-loop's intermediate obligations and must satisfy all of them at once, plus its own, because both the per-action approval *and* the live supervision are being removed simultaneously. Concretely, the system must enforce: a sandbox so that even a catastrophic mistake can only touch a disposable, reconstructable environment; strict token, time, and cost budgets that the runtime caps rather than merely monitors; turn limits inside the control loop so a confused agent halts rather than churns; scoped tool permissions so the agent's broadened freedom is still bounded to what the task legitimately requires; comprehensive append-only audit logging written at the moment of each action, not summarized afterward; state persistence so a long run is resumable and, critically, inspectable; failure detection and classification that distinguishes clean success, clean failure with rollback, and failure in an unknown state needing human attention; rollback or recovery behavior so an aborted run leaves the workspace known-good rather than half-mutated; and clear reporting and alerting so the absent human learns what happened on return. These obligations exist for one structural reason: with no human present, failures accumulate unobserved until something

external stops the run or it ends on its own, so every mechanism that a watching human would have provided must instead be guaranteed by the system before the run begins.