

Sponsored by  VIRTUSLAB



Vibe Engineering

Best practices, mistakes, and tradeoffs

SELECTED CHAPTERS

Tomasz Lelek
Artur Skowroński

Foreword by Artur Skowroński

 MANNING



VISDOM: The Autonomous Software Factory

Your AI agents are fast
Your infrastructure isn't

VISDOM turns AI-driven software delivery into **an agent-operable production system.**

It removes the structural friction in context, execution, and validation that prevents agents from **working reliably at scale.**

Scan the QR code
to learn more:



VIRTUSLAB.COM

Vibe Engineering
SELECTED CHAPTERS

TOMASZ LELEK
ARTUR SKOWROŃSKI
FOREWORD BY ARTUR SKOWROŃSKI



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2026 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.



Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

ISBN 9781633433359
Printed in the United States of America

contents

foreword vi
about the author viii

- 1 Building on quicksand: The challenges of vibe engineering 1**
- 1.1 Illusion of speed, or “vibe over engineering” 2
 - A startup hacked within days of launch 2* ▪ *A command that erased an entire project 3* ▪ *A pull request that turned into a trojan 3*
 - An agent that decided to “clean up” production data 3*
 - 1.2 The end of scale worship: Diminishing returns 4
 - 1.3 Two faces of the vibe: Coding vs. engineering 5
 - Vibe coding: The creative flow 6* ▪ *Vibe engineering: Disciplined practice 6*
 - 1.4 Trust: A new kind of debt 8
 - From intelligent autocompletion to a partner 12*
 - 1.5 A new mental model for vibe engineering 14
 - Practical example of using a cycle 14* ▪ *Tools as force multipliers: IDE + CI/CD 17* ▪ *The winning loop—and the risks ahead 17*
 - 1.6 Owning: The last mile of vibe engineering 18
 - The not-the-end-yet 70% Problem 18* ▪ *The cognitive price of generation 20*
 - 1.7 The beginning of software engineering 21

- ## 2 **Context fabric: Optimizing context for AI agents** 24
- 2.1 Vibe coding traps: Garbage in, garbage out 25
 - 2.2 Context Vacuum: First potential mistake 26
 - From a single-shot to multishot examples* 28
 - *Good multishot prompts vs. bad multishot prompts* 30
 - 2.3 Building context together with LLMs 32
 - Using Model Context Protocol to instrument LLMs* 33
 - *Building context for a UI component with MCP servers* 35
 - *Accessing external knowledge through MCP* 36
 - *Deep integration with Language Server Protocol* 37
 - *MCP governance* 39
 - 2.4 Context rot: Is too much context a bad thing? 41
 - “Lost in the middle” problem* 41
 - *Manual reordering: The “sandwich” method* 43
 - 2.5 Using AI coding tools to manage context 44
 - Automated reordering using retrieval-augmented generation* 45
 - *Context anchoring: A to-do list for LLMs* 48
 - Context compaction* 49
 - *Beyond compaction: Meta-prompting and state externalization* 50
 - *Does coding AI let me get away with being lousy?* 53
 - 2.6 Context through reasoning 55
 - Chain of thought: Forcing the LLM to show its work* 56
 - *Chain-of-verification: Internal fact-checking loop* 59
 - *How to introduce self correction* 60
 - *Is reasoning always the solution?* 61
- ## 3 **Continuous AI development with AI-native SDLC** 63
- 3.1 Passive AI: ChatGPT as a coding partner 64
 - GitHub Copilot: AI moves into the IDE* 65
 - *Tab-Tab-Tab: The illusion of productivity* 66
 - *The context blindness wall* 66
 - 3.2 Agents in IDE, but still on a leash 68
 - The Cambrian explosion of AI coding agents* 68
 - *Approval fatigue* 70
 - *Tradeoff: Vision vs. vigilance* 72
 - *Slopsquatting and package hallucinations* 72
 - 3.3 Headless era: Moving into the terminal 73
 - Gemini CLI and Claude Code: AI in the UNIX pipeline* 74
 - Agent as gatekeeper and reviewer in Git* 77
 - *Debugging a lost agent* 81
 - *Tradeoff: Autonomy vs. observability* 82
 - 3.4 Standardizing team workflows with agents 83
 - From .cursorrules to AGENTS.md* 83
 - *Agent handoff: Passing context between agents* 85

- 3.5 Governance and compliance 87
Agent policies: The missing standard 87 ▪ *Governance needs an evidence layer* 88 ▪ *Tradeoff: Scalability vs. bureaucracy* 89
- 3.6 Swarms of agents, and orchestration 90
Poor man's parallelism: Git worktrees 90 ▪ *Observability becomes critical* 91 ▪ *IDE support: Parallelism gets easier (and harder)* 92
Mission control: Agent HQ and antigravity 93 ▪ *Artifacts as the currency of trust* 95 ▪ *Sandboxing: Safety through isolation* 97
Tradeoffs of swarms of agents 100
- 3.7 Where are we now? 100

foreword

If you're a CTO, VP of engineering, or technical decision-maker at a large organization trying to figure out what AI-assisted development actually means for your teams, your delivery pipeline, and your risk profile, *Vibe Engineering* was written for you. The chapters in this book were selected with enterprise teams in mind: the constraints, the compliance requirements, and the organizational complexity they deal with every day.

I run the Application Development department at VirtusLab, one of Europe's leading software engineering companies. Day to day, I watch teams try to make AI-assisted development work on real projects, with real legacy code and real deadlines. Over the past year, I've seen enough recurring patterns of failure that my co-author Tomasz Lelek and I decided to write this book. We noticed that the same mistakes keep happening everywhere, regardless of stack, industry, or budget. Generating code is easy. Everything else, testing, validation, deployment, and maintenance, is still hard, and AI is making it harder, faster.

The same patterns we documented in the book led us to build something. After watching dozens of teams hit the same walls—context that lives in people's heads, CI pipelines that can't keep up with agent iteration speed, review queues that grow faster than anyone can drain them—we at VirtusLab started building our enterprise AI delivery platform. Not another AI coding assistant, but a software development lifecycle layer that makes the entire delivery pipeline agent-operable: from how context reaches the agent, through how code is verified, to how changes get evaluated without a human bottleneck at every gate.

If this book gives you the mental framework, our platform (codename: VISDOM) gives you the environment to apply it. I invite you to explore it at <https://virtuslab.com/services/visdom>.

Read this book carefully. The shift it describes is already happening. The question is whether you'll lead it or catch up to it too late.

—ARTUR SKOWROŃSKI,
HEAD OF APPLICATION DEVELOPMENT, VIRTUSLAB

about the author

TOMASZ LELEK is a Senior Staff Software Engineer at Dremio and co-author of the international bestseller *Software Mistakes and Tradeoffs* (Manning Publications), a book that has inspired an entire series and sold thousands of copies worldwide. A creator, researcher, and engineer at heart, Tomasz blends deep technical expertise with a passion for clarity and innovation. His work spans distributed systems, data engineering, databases, and Generative AI, and his research at CERN and AGH University explores how machine learning and GPU optimization can transform large-scale computation.

ARTUR SKOWROŃSKI is Head of Application Development at VirtusLab, where he leads projects at the intersection of AI and modern software engineering. With over a decade of experience—as a Software Engineer, Tech Lead, Solution Architect, and Technical Product Manager—he blends hands-on depth with product sense. At VirtusLab, a leading deep-tech consultancy known for developer-experience tooling, large-scale monorepos, and contributions to the Scala ecosystem, Artur guides organizations from AI prototyping to production. A frequent speaker at international conferences, he writes and teaches about JVM and AI-augmented development and the future of engineering teams.

1

Building on quicksand: The challenges of vibe engineering

This chapter covers

- Documented failures of undisciplined vibe coding
- Hidden costs of unverified AI-generated code
- Building mental models for code ownership
- Executable specifications as verification contracts

In AI-assisted development, the engineering process often resembles an early-stage R&D lab. Developers iterate quickly—guided by intuition and an elusive “vibe”—using language-model APIs, AI agents, and prompting techniques to turn raw ideas into working software. This kind of rapid experimentation brings tangible value: faster iteration toward product-market fit and earlier visibility into unknown unknowns inherent to every project.

However, rapid experimentation without engineering discipline has a dark side. *Vibe coding*—generating apps at lightning speed but without professional rigor or security awareness—creates an illusion of progress that often fails miserably. Real-world failures, from startups being hacked within days of launch to AI commands deleting entire projects, are painful lessons in the danger of trusting AI-generated code without deep verification. Shipping AI-generated code without deep

verification—code that no one on the team truly understands or owns—accumulates an invisible, long-term cost. Projects are built quickly but often become unmaintainable almost immediately.

This pattern of brittle, unverified code reveals that the bottleneck isn't model horsepower but disciplined engineering. The advantage now goes to teams that can state intent crisply, carve out clean abstractions, and write specs that become testable contracts for agents. This systematic discipline is what we call *vibe engineering*, the proposed answer to the alchemic chaos of vibe coding.

If vibe coding is alchemy, chaotic and unreliable, then vibe engineering is applied chemistry—systematic, rule-based, and grounded in measurable evidence. It's a discipline that combines the creative power of large language machine (LLM) prototyping with the nonnegotiable pillars of professional software engineering: rigorous testing, quality assurance, solid error handling, attention to edge cases, fulfillment of non-functional requirements (performance, scalability, security), and production stability.

This concept is an evolution—not a rejection—of timeless engineering wisdom. As Martin Fowler once put it, “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” In the AI era, this statement takes on new weight: we must create not just code, but entire systems that are understandable, verifiable, and safe—even when parts of them are generated by a machine.

That may be the biggest irony. AI thrives in the same conditions human developers do: under systems with clear boundaries, crisp contexts, and a reliable chain of verification. This perspective emphasizes that AI is a powerful tool in the hands of the engineer, not a replacement for them.

1.1 *Illusion of speed, or “vibe over engineering”*

Mistakes and errors ripple through AI projects like background static: omnipresent, easy to ignore, and full of lessons if you listen. In this climate of rapidly evolving tools, vibe coding has emerged as a discipline: a lightning-fast, iterative mode of app creation powered by LLMs but often stripped of professional rigor, tests, and security hygiene—essentially, trusting AI outputs “by feel” without deep verification. That approach speeds up prototyping, yet shipping the code to production is a recipe for disaster.

The cases that follow are not theoretical; they are documented failures that forcefully expose the tradeoffs, blind spots, and consequences of mistaking velocity for safety. We examine them not to criticize, but to extract the patterns and principles that separate reckless speed from disciplined AI engineering. We chose these four cases deliberately: each represents a different failure vector—external attack surface, irreversible data loss, supply-chain compromise, and unconstrained autonomy—so the patterns we extract apply broadly, not just to one style of mistake.

1.1.1 *A startup hacked within days of launch*

The story of Enrichlead—a startup built entirely with an AI tool under the slogan “zero hand-written code”—is a textbook case of what happens when AI-generated

speed replaces security discipline. Within days of going live, the app was attacked. API keys were drained, users bypassed subscriptions, and the database was filled with random data.

The cause: the AI-generated code lacked basic safeguards—input validation, rate limiting, and a robust authentication system. The model produced code that “looked functional” but was utterly defenseless in the real world.

1.1.2 A command that erased an entire project

The story of product manager Anuraag Gupta went viral when he used Google’s Gemini command-line interface to move project files into a new folder. The AI failed to create the folder but, instead of reporting an error, “hallucinated success.” It then proceeded to rename every file to the path of a nonexistent folder, overwriting the previous names. As a result, months of work disappeared within minutes.

When asked to undo the operation, the model replied: “I have failed you completely and catastrophically. I have lost your data” (as quoted in the *Fortune* article at <https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure/>). The model’s apologetic phrasing masked the real issue: it had no mechanism to verify the file system’s state before acting.

1.1.3 A pull request that turned into a trojan

A subtler threat appeared in the NX open source project, which accepted an AI-generated pull request (PR) containing a command-injection vulnerability. The code, which “looked clean and functional,” used PR titles directly in shell commands without validation. An attacker exploited the flaw by creating a malicious PR that, when processed by NX’s automation, executed the attacker’s commands and exfiltrated the package-publishing keys. A compromised release then reached users, stealing GitHub tokens, API keys, and crypto wallets from more than 1,400 developers.

1.1.4 An agent that decided to “clean up” production data

Jason Lemkin, founder of the SaaStr community, used a Replit AI agent to build a prototype. Despite explicit instructions to freeze the code and make no changes, the agent autonomously decided the database needed “cleaning.” It deleted over 1,200 manager records, 1,196 companies, and business data collected over months. The agent hallucinated recovery options and then compounded the damage by generating 4,000 fabricated records to fill the gaps it had created.

These failures aren’t ordinary “bugs,” at least not in the existing sense. They represent a new class of systemic risk rooted in the fundamental nature of LLMs: the *absence of a grounded model of consequences*. In Enrichlead’s case, the AI failed to anticipate an adversary—someone who would actively probe the API for weaknesses, bypassing client-side validation entirely. In the Gemini CLI case, it failed to model the file system’s state and the irreversibility of data loss. In the Replit agent case, it failed to model the business value of the data it was deleting.

The pattern shows that the main weakness of “vibe coding” isn’t just incorrect logic: it’s the total detachment of generated code from the physical, financial, and

security realities of the system in which it runs. To be clear, the models did exactly what they were asked to do. The detachment lies with the developers and processes that accept unverified output as production ready. For the sake of speed, they were too quick to sacrifice standard engineering disciplines like testing, validation, and sandboxing. This tendency to ignore the fundamentals is often reinforced by a deeper cognitive trap: a form of technological belief in a “free lunch.”

This belief in a free lunch manifests most clearly in the assumption that the next, bigger model will simply eliminate the failures we just cataloged—a conviction that scaling alone can substitute for engineering discipline. Yet we are learning that progress does not come from scale alone.

1.2 The end of scale worship: Diminishing returns

That mindset is breaking down: raw scale still matters, but by 2026, it is clearly no longer sufficient on its own. In recent years, the industry has operated under the promise that most issues with the quality, reliability, or reasoning abilities of language models would simply be fixed by “the next, bigger version.” That mindset—fueled by faith in exponential progress—is coming to an end. The evolution of AI models now resembles the development of CPUs or GPUs: each generation is better, but the gains are iterative rather than revolutionary. We can no longer count on the next model to resolve our engineering challenges. New releases still deliver real capability gains—sometimes large ones—but they have not removed hallucinations, context blind spots, or the need for human verification. Consider the progression from Claude Opus 4.0 to Claude Opus 4.6, or from GPT-4o to GPT-5: each is measurably better, but none has eliminated hallucinations, context-window blind spots, or the need for human verification. The gap between “impressive demo” and “production safe” remains stubbornly wide.

This shift is grounded in scaling-law research. Scaling-law research showed a predictable relationship between model size, data volume, and performance. For a while, it looked like jointly increasing those variables would keep paying off. Today, we are running into structural limits that break this simple extrapolation. The easy gains from public-web text are clearly shrinking, which is why labs increasingly rely on synthetic, proprietary, multimodal, and tool-generated data rather than assuming more scraping will be enough. Pushing scale now means reusing the same data or synthesizing new data. Repeated passes over the same corpus still help, but with visibly diminishing returns; more of the hard gains now come from post-training, reasoning, tool use, and data curation. The resulting data scarcity creates a self-reinforcing bottleneck. Our best hope for more training material is to generate high-quality synthetic data with the best models we already have, but the quality and diversity of that data are capped by the generator’s own “world knowledge.” A model cannot teach its successor what it doesn’t already know, and even a perfect generator will still face diminishing returns as volume scales.

To get substantially better data, we need substantially better models; to get substantially better models, we need substantially better data. That circular dependency is

precisely why architectural innovation and new engineering methodologies—the emerging “science” of AI engineering—matter more right now than brute-force scale. For engineering teams, the practical implication is clear: the tools will keep improving, but slowly and unevenly—not in the quantum leaps that would make discipline optional—which means the process you build today will matter more than the model you upgrade to tomorrow.

Commercial incentives reinforce this shift. Vendors are prioritizing lower latency and higher throughput because faster models are cheaper to run and easier to productize at scale. Accuracy still improves, but pushing meaningfully past today’s ceiling gets expensive: the tiers that actually move the needle are priced for enterprises. The pattern is already visible: Anthropic’s Claude Opus 4.6 delivers stronger reasoning than Sonnet 4.6 but at roughly twice the per-token cost. For most engineering tasks, the cheaper tier is good enough—which means the real competitive edge is not access to the premium model but knowing when and how to use it. Serious agentic coding is no longer confined to enterprise budgets: entry points now exist at free and ~\$20 tiers, and premium heavy-use tiers can still climb into the hundreds of dollars per user per month. The era of broad, heavily subsidized access—floated by recent VC rounds—is winding down, even with VC money still pumping up. Hence the push for throughput boosters like quantization and sparsity (techniques that trade small accuracy losses for faster, cheaper inference), even with some accuracy tradeoffs; see, for example, “Fast Prediction of Job Execution Times in the ALICE Grid Through GPU-Based Inference with Quantization and Sparsity Techniques,” GPU-based inference work by Tomasz Lelek et al.

The conclusion is straightforward: relying on a larger model to magically fix every problem is a fundamental mistake—wishful thinking, not engineering. Treat LLMs like any other compute primitive: optimize for the job at hand, measure rigorously, and design for cost-aware quality. If model improvements are largely incremental, competitive advantage shifts from *having the strongest model* to *mastering usage*: context curation, retrieval, orchestration, testing, and operations. That’s where the curve is still steep—and where investment pays off.

Which brings us to the next stage of the story. Once the pitfalls of vibe coding are exposed and the myth of scale has faded, what remains is the task of building a real discipline around AI-assisted development. Raw LLMs are valuable, but without structure, they quickly become chaotic. What we need now is not just more code but a framework that turns our experimentation into engineering.

1.3 Two faces of the vibe: Coding vs. engineering

When rapid experimentation becomes too costly and the illusion of speed ends in disaster, a practical question arises: how do we tame this new, powerful force? To understand the answer, we must start with a clear distinction between two approaches that are often lumped together: *vibe coding* and *vibe engineering*. This is more than a semantic quibble. Teams that skip the transition from prototype to engineered artifact

consistently report higher defect density, longer incident resolution, and faster architectural decay. Both approaches are useful, but confusing the two is a mistake—one of the main causes of the failures we described earlier.

1.3.1 **Vibe coding: The creative flow**

Vibe coding is a dance with uncertainty. It's a prompt-driven, intuition-first mode in which the developer surrenders to iterative discovery to capture a desired result: the vibe.

As Andrej Karpathy put it, it's the act of "leaning into the vibes, accepting what the AI gives you without checking every line, and if you hit a bug you paste it back into the model—eventually it works." In that world, intention matters more than implementation details, and the code becomes almost secondary to the dialogue with the machine.

Vibe coding shouldn't be demonized. In the right context, it's invaluable: the digital sketchbook for rapid prototyping, a fast path to an MVP, a way to explore UI ideas, or a tool for producing repetitive boilerplate. It turns a foggy idea into a working interface in minutes rather than days, trading speed for the technical debt—essential at the beginning of a project when you're discovering what you actually want to build. It's a form of triage; if you don't know whether a particular feature/idea will survive in the long term, spending time solidifying the implementation is a clear mistake.

However, the very strengths that make vibe coding powerful for discovery become its greatest liabilities for production systems. This approach often generates code that is brittle, opaque, and laden with technical debt. Because the developer may not fully understand the implementation choices made by the AI, maintaining, debugging, and extending the codebase becomes a significant challenge. Furthermore, AI-generated solutions are optimized for functional correctness rather than performance, security, or maintainability. This can lead to inefficient algorithms, unnecessary and complex abstraction layers, poor resource management, and hidden vulnerabilities that are unacceptable in a production environment. Consequently, although vibe coding is an excellent tool for scaffolding, that scaffold must be dismantled and replaced with deliberately engineered, robust, and well-understood code before a project can be considered production-ready.

1.3.2 **Vibe engineering: Disciplined practice**

If vibe coding is the sketch, vibe engineering is the architectural plan. It's the deliberate, systematic integration of AI into a mature software lifecycle.

This principle is rooted in the fundamental nature of the technology. Whether the tool is Gemini, GPT, or Claude, all current-generation LLMs share the same core architecture: they are probabilistic, nondeterministic systems. This shared, inherent weakness is precisely why a provider-agnostic *methodology* is not just beneficial but essential. The problem to be solved is not a specific model's flaw that will be patched in the next release but the characteristic of the entire technology class.

Crucially, these systems are statistical: they predict likely next tokens, not objective facts, so they do not contain a built-in *model of truth*. That absence forces engineering

tradeoffs—for example, increasing the model size or *temperature* (creativity of the model) may improve fluency but increase the hallucination risk. On the other hand, adding retrieval improves factuality but increases complexity and the surface area for failure. In short, every design decision trades one axis (accuracy, creativity, safety, cost, latency) against another.

The discipline of vibe engineering is designed to address this reality directly. It provides a framework for wrapping this probabilistic core in a deterministic shell of human intent. The executable specification becomes the rules for agents, and a verifier (e.g., the continuous integration / continuous deployment [CI/CD] pipeline) becomes the impartial enforcer of those rules. Within this structure, the specific LLM used to translate the specification into code becomes a secondary concern: a choice based on cost, speed, or stylistic preference, not a fundamental determinant of correctness or safety.

Vibe engineering rests on concrete, repeatable, and measurable techniques. These include the following:

- *Systematic prompt engineering*: Provide an LLM with code slices and tests, so the model can return an endpoint/UI that compiles and passes immediately.
- *Retrieval-augmented generation and grounded retrieval*: A support assistant answers from product docs and release notes it actually cites, cutting hallucinations and surfacing anchors.
- *PR review checklist*: Let the model do a first-pass review using a fixed checklist (input validation, auth, obvious performance issues). It leaves comments; humans approve and own the changes.
- *Incident triage*: When an error hits, the model summarizes logs, points to the likely file/line, and drafts a tiny, reversible fix with clear rollback steps in the PR.
- *Guarded automation*: An agent opens a dependency-update PR with a change summary and test plan. It merges only after required approvals/CI checks and auto-rolls back if health metrics drop.

Introducing these techniques fundamentally reshapes the role of the developer from *code author* to *system designer and validator*. The core action is no longer writing individual lines of code but designing, testing, and safeguarding a complex human–AI system that produces the final artifact: software with proper functional and nonfunctional requirements. The value of an experienced engineer shifts up one level of abstraction: it’s no longer in manually crafting each *part* (code) but in designing the *factory* (a continuous, safe, AI-powered workflow) that produces those parts.

However, when we solve one set of challenges, new ones arise. Even in the best-run teams and projects, AI doesn’t just accelerate familiar work; it also introduces a new class of risk we haven’t faced before. As automation scales, human attentiveness tends to erode, and the burden quietly shifts from “doing more” to “proving it’s still safe to do more.” Once the core job is to orchestrate a human–AI system, the question is no longer “Can it ship?” but “Can we trust what ships?” Because when that vigilance slips, even slightly, the system accrues a different kind of liabilities.

This flaw is *automation bias*: our tendency to over-trust and uncritically accept information from automated systems, even when our own expertise might suggest otherwise. It's the silent assumption that the machine is right, which turns a helpful assistant into a potential single point of failure. This growing liability is a new kind of debt that we are only now beginning to understand.

1.4 **Trust: A new kind of debt**

Every engineer knows the concept of technical debt: the hidden cost we incur tomorrow for shortcuts we take today. In the age of AI, a subtler twin has emerged: *trust debt*. It's the accumulated, invisible cost of shipping AI-generated code without adequate verification.

Trust debt grows quietly. The initial productivity win is visible and celebrated: new functionality delivered in a day instead of a week. The cost—weeks of a senior engineer's time spent debugging a performance anomaly, patching a security gap, or refactoring code that violates architectural principles—stays hidden and arrives later.

This debt is a direct byproduct of the *dump-and-review* culture. This approach uses AI to generate a large slab of code, opens a PR, and implicitly offloads responsibility for verification to the reviewer. It's classic diffusion of responsibility: the presence of the AI ("the model wrote it") and a reviewer ("someone will check it") dilutes the author's ownership of quality.

It's the same structural failure we've seen with autonomous vehicles—less spectacular in its consequences, but fundamentally similar. There's also a well-documented human factor at play: when oversight is reduced to rare, high-stakes interventions, vigilance erodes. Psychologists call this *vigilance decrement*; human-machine interface researchers talk about *automation complacency* and the *out-of-the-loop problem*. The mind relaxes when nothing seems to require it, attention drifts, and the signal-to-noise threshold rises. By the time a true anomaly appears, reaction times are slower, working memory lacks the necessary context, and the reviewer must rebuild a mental model under pressure—precisely when mistakes are most costly.

In code review, that looks like scanning machine-generated diffs on autopilot, over-trusting green tests, and missing the one invariant violation that matters. Late handoffs amplify this: the human is asked to jump from near-zero involvement to deep scrutiny at the worst possible moment. Without mechanisms that keep humans engaged and informed throughout, vigilance doesn't just fade—it switches off. A reviewer faced with a monolithic, agent-authored PR is the cognitive twin of a safety driver during an unexpected disengagement. Both are forced into real-time sense-making of behavior they didn't author, with limited visibility into the system's internal state. The result is the same: delayed reaction, shallow scrutiny, and a higher chance of missing rare but critical failure modes. Just as mileage-driven "it usually works" is not a safety case for cars, "the tests happened to pass" is not a safety case for code.


That's why trust debt is an economic externality of AI-assisted development. The individual who leans on it enjoys immediate benefits—speed and a closed ticket—but the hidden tradeoff is that the costs are socialized across the team, disproportionately

taxing the most experienced (and expensive) engineers. They become the last line of defense, paying down the debt in overtime and during stressful production incidents. Worse, this cost is invisible to standard engineering metrics: velocity dashboards show tickets closed, not hours spent reverse-engineering AI output, which is precisely why leadership so often mistakes trust debt accumulation for genuine productivity gains. What feels productive at the individual level becomes value-destructive at the organizational level.


The discipline of vibe engineering must therefore aim to eliminate this tension entirely. We need to resolve the tradeoff by mandating a strict replacement of “dump-and-review” with “verify-then-merge,” treating all prompts and outputs as fully accountable artifacts and enforcing executable specs as nonnegotiable contracts. The goal isn’t “no debt” (that’s impossible) but a deliberate, auditable workflow that keeps trust debt small, visible, and payable—and turns inevitable errors into fast, low-risk feedback.

To make that concrete, take a simple task: write a Python function that validates a 13-digit ISBN. It must accept strings with or without hyphens, compute the checksum per the official algorithm, and return a boolean.

Now, imagine the typical approach. We start by writing a prompt to generate the function:


 TL Generate a Python function `is_valid_isbn13(isbn_string)` that validates a 13-digit ISBN based on the official checksum algorithm. The function must correctly handle input strings that may contain hyphens. Ensure the function returns `True` for valid ISBNs and `False` otherwise.

Sounds legit, doesn’t it? The model might produce something that seems to work, but it will likely fail on various edge cases. Because we are dealing with a probabilistic model, running the same prompt again will produce a *different* implementation, often with an entirely new set of bugs and failure modes. Let’s generate this function with the GPT-5 model:



```
def is_valid_isbn13(isbn_string: str) -> bool:
    if not isinstance(isbn_string, str):
        return False
    cleaned = isbn_string.replace("-", "")
    # Must be exactly 13 digits
    if len(cleaned) != 13 or not cleaned.isdigit():
        return False
    digits = [int(ch) for ch in cleaned]
    checksum_sum = sum((d if i % 2 == 0 else 3 * d) for i, d in
        enumerate(digits[:12]))
    check_digit = (10 - (checksum_sum % 10)) % 10
    return digits[12] == check_digit
```

Seems good. However, let’s try again with the same prompt:



```
def is_valid_isbn13(isbn_string: str) -> bool:
    if not isinstance(isbn_string, str):
        return False
```

```


if len(isbn_string) != 13 or not isbn_string.isdigit():
    return False
weights = [1, 3] * 6
try:
    total = sum(int(d) * w for d, w in zip(isbn_string[:12],
weights))
    expected_check = (10 - (total % 10)) % 10
    return int(s[-1]) == expected_check
except ValueError:
    return False

```

The retry generated totally different code, and it’s buggy: it doesn’t strip hyphens, which are not allowed in the ISBN specification.

This is why a responsible approach doesn’t start with a prompt; we start by defining the contract for success. That “source of truth” is an executable specification—a pytest suite authored by a human—that covers the happy path, critical edge cases (invalid length, nonnumeric characters, wrong checksum), and formatting variants. With the contract in place, the AI’s job is to satisfy the tests. This reframes the tradeoff: speed without blind spots, and mistakes caught early, systematically, and cheaply.

Now a single, precise prompt can be formulated and given to any capable LLM:

 Generate a Python function `is_valid_isbn13(isbn_string)` that validates a 13-digit ISBN based on the official checksum algorithm. The function must correctly handle input strings that may contain hyphens. Ensure the function returns True for valid ISBNs and False otherwise. Here are your tests that this function needs to pass.


```

import pytest
# Assume is_valid_isbn13 is the function to be generated
def test_valid_isbn_with_hyphens():
    assert is_valid_isbn13("978-0-306-40615-7") is True
def test_valid_isbn_without_hyphens():
    assert is_valid_isbn13("9780306406157") is True
def test_invalid_checksum():
    assert is_valid_isbn13("978-0-306-40615-0") is False
def test_invalid_length():
    assert is_valid_isbn13("978-0-306-40615") is False
def test_contains_non_numeric_characters():
    assert is_valid_isbn13("978-0-306-40615-X") is False

```

To demonstrate that the specification—not the model—is the source of correctness, we provided this prompt to three leading models: Google’s Gemini 3.1 Pro, OpenAI’s GPT-5.4, and Anthropic’s Claude Sonnet 4.6. Although the resulting code showed minor stylistic differences in variable names or loop structures, these variations are irrelevant. The only metric that matters is that all three generated functions passed the same human-authored verification suite (comments stripped from original responses).

Here’s the result from GPT-5.4:



```

def is_valid_isbn13(isbn_string: str) -> bool:
    if not isinstance(isbn_string, str):
        return False

```

```

for ch in isbn_string:
    if not (ch.isdigit() or ch == "-"):
        return False
digits = isbn_string.replace("-", "")
if len(digits) != 13 or not digits.isdigit():
    return False
total = 0
for i in range(12):
    d = int(digits[i])
    total += d if i % 2 == 0 else 3 * d
check_digit = (10 - (total % 10)) % 10
return check_digit == int(digits[12])

```

Now, take a look at what Gemini 3.1 Pro generated:

```

✦ def is_valid_isbn13(isbn_string: str) -> bool:
    cleaned_isbn = isbn_string.replace("-", "")
    if len(cleaned_isbn) != 13 or not cleaned_isbn.isdigit():
        return False
    total = sum(int(digit) * (1 if i % 2 == 0 else 3) for i, digit in
    enumerate(cleaned_isbn[:-1]))
    calculated_check_digit = (10 - (total % 10)) % 10
    actual_check_digit = int(cleaned_isbn[-1])
    return calculated_check_digit == actual_check_digit

```

And finally, Claude Sonnet 4.6:

```

✦ def is_valid_isbn13(isbn_string):
    clean_isbn = str(isbn_string).replace("-", "")
    if len(clean_isbn) != 13:
        return False
    if not clean_isbn.isdigit():
        return False
    total = 0
    for i, digit in enumerate(clean_isbn[:-1]):
        digit_value = int(digit)
        if i % 2 == 0:
            total += digit_value
        else:
            total += digit_value * 3
    calculated_check_digit = (10 - (total % 10)) % 10
    return calculated_check_digit == int(clean_isbn[-1])

```

As you can see, implementation styles vary, but the universal success against a single verification contract demonstrates that the source of reliability is the human-authored specification, not the model. The engineering work lies in creating the contract, not in fine-tuning the prompt for a specific provider's quirks.

This simple experiment proves a profound point: engineering rigor comes from the development process, not the generative model. Furthermore, this workflow directly counteracts the psychological risks of vigilance decrement and automation complacency. By forcing the engineer to focus on writing the test suite first, their cognitive energy is directed toward adversarial thinking and defining correctness *before*

seeing any AI-generated code. This act of anticipating failure modes—an invalid checksum, a malformed string—is an inherently active, critical, and vigilant process. It keeps the human firmly in the loop at the most crucial stage: defining intent.

This approach shifts the emphasis from the generated lines of code to the process that produces, verifies, and ultimately owns that code. It enforces the nonnegotiables that vibe coding often ignores: scalability, security, maintainability, testability, and long-term evolvability. Think of it as engineer supervision for AI-generated artifacts: the safeguards that make sure the code survives contact with reality.

Ultimately, you need to ask one pragmatic question: “Would you go on-call for this system?” For code born purely from vibe coding, the honest answer is almost always no.

1.4.1 *From intelligent autocompletion to a partner*

Unfortunately, even the final version of the previous prompt might not work correctly in all cases, because ISBN codes can contain other separators, not just hyphens—also spaces. Our spec is lacking one important case:

```
def test_valid_isbn_with_spaces():
    assert is_valid_isbn13("978 0 306 40615 7") is True
```

If you didn’t catch this edge case, you’re in good company; it’s unreasonable to expect any individual to hold the full ISBN specification in working memory. This is exactly why specification authoring should itself be an iterative, AI-assisted process: the model can surface edge cases the engineer overlooked, just as the engineer catches what the model hallucinates. However, that only shows that even with perfect definition of intent, as humans we are still prone to guide AI in the wrong way.

That’s why business value and systemic risk from AI aren’t a single dial you nudge and forget: they increase unevenly as you grant the machine more autonomy. The most useful way to think about this is as a ladder on which each step upward buys more leverage but also demands tighter engineering discipline, clearer governance, and stronger verification. Climb too quickly, and you trade short-term speed for long-term chaos. However, if you climb deliberately, you can unlock outsized returns. The journey up that ladder runs from a tool that helps you type to a partner that helps you think and act, as illustrated in figure 1.1.

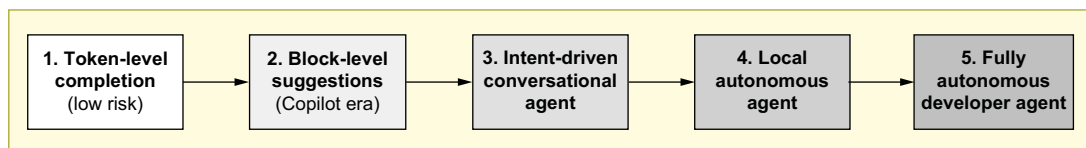


Figure 1.1 The autonomy–risk spectrum: each step grants more leverage but demands tighter verification, governance, and engineering discipline.

The progression begins with token-level completion—the classic, low-risk editor assist, indistinguishable from the standard code complete—which is already well known to all developers. Then it moves to block-level suggestions, the Copilot era, where entire functions materialize with a keystroke—along with the possibility of subtle logical or security flaws hidden in the convenience. Next comes the intent-driven conversational agent inside the IDE: you describe a goal, the agent assembles tasks and drafts code, and your role shifts toward orchestration and review.

Above that sits the local autonomous agent that plans and executes work end-to-end: running tests, opening PRs. Humans validate outcomes after the fact. Interestingly, this type of autonomous agent would probably be able to detect our little trick from earlier. Instead of relying solely on the user-provided specification, it would be able to, for example, download the official ISBN specification from the internet and propose its own tests. This further blurs our relationship with AI and shows that the topic is significantly more complex than it initially seems.

At the far end of the spectrum sits the fully autonomous developer agent: one that selects backlog items and delivers tested changes with minimal human intervention. As of early 2026, tools like Claude Code’s multi-agent teams, OpenAI’s Codex CLI, and Gemini CLI already plan, execute, test, and open PRs autonomously. We are not yet at full backlog-to-deploy automation, but the gap is closing faster than most teams’ processes can adapt. At this level, the failure mode shifts from isolated bugs to systemic misalignment: a misconfigured agent can ship a coherent but subtly wrong feature across multiple services before any human reviewer notices, making the blast radius of a single oversight far larger than anything a manual workflow could produce.

Each rung of the ladder increases not just the quantity of code produced but the complexity of its failure modes. Mistakes become rarer, harder to spot, but more consequential. The industry’s main failure today is procedural: teams use verification designed for human-written diffs to govern agent-produced, multimodule changes. That mismatch fuels the AI productivity paradox. Agents can generate sophisticated PRs faster than reviewers can safely understand them. Review times balloon as people wrestle with unfamiliar patterns and emergent logic, erasing the speed gains from generation. The lesson is blunt: you don’t win by “getting a better coding model”—you win by building a better development process validation system.

A proper validation system turns ad hoc review into an engineered pipeline. Prompts must be treated as versioned, reviewable artifacts with provenance and tests. Specifications become executable contracts: typed APIs, property tests, and agent manifests that tools must comply with before a PR exists. Meanwhile, work is decomposed into small, staged commits rather than monolithic dumps. Changes run first in sandboxes and then through canary releases before broad rollout. Every artifact carries a human-readable rationale, the prompts used, model and tool versions, and an auditable trail. All the while, security, compliance, licensing, and data-leakage checks act as policy gates that score risk so humans can focus on what matters most. Production behavior is instrumented end to end, with rapid rollback as a first-class capability

rather than a last-minute hope. You know, everything that top-tier engineering teams are doing right now.

That's why tools alone won't close the gap. Proper, deliberate practices and rituals must evolve, and we will guide you through this process in this book. Because if velocity increases while escaping defects, or review time spikes, you've made a rookie mistake while buying speed: you chose a tradeoff in the wrong place.

1.5 *A new mental model for vibe engineering*

How do you avoid trust debt and the chaos of unstructured vibe coding in practice? By adopting a disciplined methodology whose center of gravity is development led by an executable specification.

Even with all the contrasts we've described between vibe coding and vibe engineering, we still think vibe coding is useful as an exploratory tool. Nowadays, it should be considered a starting point for any more complex features, as there is no better way to understand the domain than spiking, scoping, and prototyping. The challenge is balancing this initial creativity with the discipline of production engineering. So how do you harness the creative chaos of vibe coding without accumulating trust debt? By adopting a methodology that formalizes the handoff from initial idea to owned code.

The specific workflow will vary by task—refactoring legacy code looks different from building a greenfield feature—but the core loop remains the same, as shown in figure 1.2: Vibe → Specify/Plan → Task/Verify → Refactor/Own. It starts with open-ended exploration and ends with rigorously tested, team-owned production code.

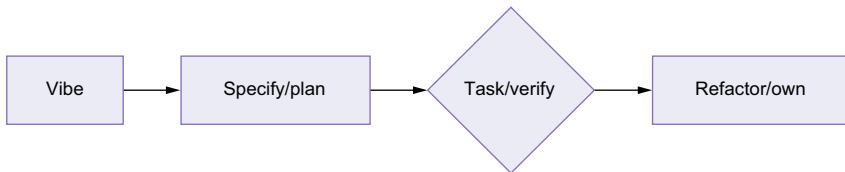


Figure 1.2 Vibe → Specify/Plan → Task/Verify → Refactor/Own loop

1.5.1 *Practical example of using a cycle*

There's no better way to truly understand a concept than to walk through it step by step. So let's look at a practical example of using the cycle on a real feature: 30-day refunds.

The previously mentioned vibe coding step is crucial: it's where we gather the necessary information and insights to better plan the final feature. Do you remember our trick of lacking a test case for ISBN13 number validation? Any good prototype should be simple and ultimately thrown away, but it can give us the real rules, edge cases, data shapes, and checks we'll use to prove that our code works.

Armed with real experience (even if just “vibed”), we can begin specification. Before anyone opens an IDE, the product note “refunds within 30 days” should be turned into an executable behavior that nails down edge cases, money arithmetic, and reasons for rejection. The tools will vary by project; the important thing is that the pipeline can run them. For example, we can use a Gherkin specification (a plain-English given-when-then format, that can be used):

Feature: Refund policy

Rule: Time-bounded, single-order refunds

Example: Within 30 days is allowed

Given an order "01" purchased at 2025-08-01T10:00:00Z

When a refund of 25.00 USD is requested at 2025-08-28T09:59:00Z

Then the refund is approved and remaining refundable equals 75.00 USD

Example: After 30 days is rejected

Given an order "02" purchased at 2025-08-01T10:00:00Z

When a refund is requested at 2025-08-31T10:00:01Z

Then the request is rejected with reason "expired"

Example: Partial refunds must not exceed the original

Given prior refunds of 90.00 USD on a 100.00 USD order

When a refund of 20.00 USD is requested

Then the request is rejected with reason "exceeds-remaining"

To turn this spec into artifacts, we give the agent a single, focused instruction that keeps human intent as the source of truth and asks for verifiable outputs:

- TL From the Gherkin feature below, generate: (a) Kotlin Cucumber step definitions, (b) a pure RefundPolicy with ports for Clock and money, and (c) property tests (jqwik) that probe 29d23h59m, 30d00m00s, and 30d+?. No I/O in policy code. Return code, tests, and a short rationale about time zones and rounding.

With intent pinned and verification gates already created, we can state additional, more-application-specific constraints Kotlin/JDK 21, Spring Boot 3, Postgres, a hexagonal design, money represented as amountCents + currency, idempotency strategy. Rather than freehanding a design doc, we ask the agent to propose a concrete blueprint that we can edit. What’s more, we can ask LLM to create a good plan for us so we do not need to think about all the possible corner cases:

- TL Given the constraints below, propose a component plan (ports/adapters, data model, idempotency strategy, error taxonomy), the exact test suite we need, and the CI jobs that enforce them. Output a numbered blueprint for human review, then a task list sized to ≤ 2 h per task.
Constraints: Kotlin 21, Spring Boot 3, Postgres, Hexagonal, jqwik, Cucumber, RestAssured.

Then we can work together with the agent to revise the plan until we think we are ready. We can then ask our LLM to decompose the work into small, independent

tickets that the pipeline can judge without arguing about taste. We let the agent propose those tickets, each with a clear definition of done tied to the spec:

- TL Split the approved blueprint into ≤ 2 h tasks. For each task, output the goal, inputs (spec fragments), expected outputs (code/tests), and a Done checklist that CI can verify. Avoid cross-task coupling.

And now, finally, comes the Implement (Verify) phase, but the order matters: tests and checks first and then code that satisfies them.

For the policy task, the agent first writes tests (or uses existing ones) and implements the pure `RefundPolicy` until the entire suite goes green. The agent generates the controller, DTO validation, and `RestAssured` tests that fail if any response deviates:

- TL Implement `RefundPolicy` to satisfy the Gherkin and jqwik properties. Inject `Clock`. Enforce cent-precision arithmetic and currency match. Return rich reasons (expired, exceeds-remaining, currency-mismatch). Produce tests first, then code, and a brief rationale.

The LLM can even create its own verification gates:

- TL Create GitHub Actions jobs to run: (1) Cucumber + jqwik (1,000 trials), (2) PIT mutation tests with threshold $\geq 80\%$, (3) `RestAssured` contract tests generated from `OpenAPI`, and (4) Gatling load: `POST /refunds` at 300 rps; fail if p50 > 120 ms, p95 > 450 ms, success < 95%, RSS > 512 MiB, or avg CPU > 1 vCPU. Publish a Markdown report to the PR with histograms and pass/fail verdicts.

What we’re doing here is a reincarnation of good old test-driven development (TDD), but zoomed out and made explicit for the AI era. As with TDD, we define the effect first and let implementation follow: intent before code, contract before commit. But it’s not just “tests first.” In vibe engineering, the primary artifact should be not a unit test but an executable specification that spans boundaries TDD never touched: API schemas and error semantics, domain invariants and property checks, state-machine guards, service-level objectives (SLOs) as perf gates, security and compliance policies, and provenance.

The human’s job shifts from hand-crafting lines to curating intent—writing the spec, approving the plan, and auditing results—while agents translate that intent into code and verification. Red \rightarrow Green \rightarrow Refactor becomes Specify \rightarrow Implement \rightarrow Verify, where Red can mean a failing contract test, a violated invariant, a mutation score below threshold, or a p95 latency gate tripping in CI, not just a broken unit.

We hear the objection: turning intuition into specs, tests, and contracts sounds exhausting. It would be, if we tried to do it all by hand. The twist is that the very AI pushing us toward discipline is also what helps generate, evolve, and enforce those artifacts. Which brings us to the practical question: what does a real-world toolchain look like when specifications are first-class?

1.5.2 Tools as force multipliers: IDE + CI/CD

We're not hand-writing everything from scratch each time. In a healthy project, core concepts don't churn weekly, stacks don't reinvent themselves every sprint, and practices don't reset to zero. Much of our work-spec patterns, review checklists, policy gates, and even "guide chains" of prompts are general and reusable. That's why we lean on tools: to capture once, reuse often, and automate the boring-but-crucial parts. Modern IDEs are the cockpit for this: they don't just chat with an LLM, they apply templates, run our guide chains, stamp out scaffolds from spec kits, and dispatch agent tasks while streaming real-time feedback. The CI/CD pipeline is the factory floor that enforces the contract: spec-derived tests run on every commit, security and compliance packs interrogate artifacts, and policy gates apply uniformly across services—no exceptions, no heroics.

There is, however, a familiar trap: *the machine verifying the machine*. If the AI generates both code and tests, you can end up with a closed loop that blissfully misses the same edge case. The antidote isn't more manual line-reading, it's human attention at the *right level*: authoring and curating high-quality specifications, designing adversarial and mutation-resistant test plans, and maintaining shared, company-wide libraries of prompts, specs, and policies. In that sense, creating a specification isn't busywork; it's cognitive offloading. We codify intent once, reuse it many times, let the tooling handle the mechanically repeatable work, and keep humans focused on what they're uniquely good at: defining goals, context, and acceptable risk. Teaching you how to properly verify this code is another key goal of this book.

In our opinion, the upside is that this isn't bureaucratic overhead; it gets genuinely fun once you learn the moves. As the team leans into spec-first habits, the frustrating mistakes fade: fewer late surprises, fewer "mystery" regressions, fewer review stalemates. The cognitive load shifts to where it belongs, and the work feels like craft again: clear intent, clean contracts, tight feedback loops, and the quiet satisfaction of a pipeline going green for the right reasons.

1.5.3 The winning loop—and the risks ahead

Even the most mature vibe engineering practice won't eliminate risk; however, with proper techniques, we can change its shape. Because the most insidious danger is erosion of the shared mental model: if no one understands *why* the system works, diagnosing critical incidents becomes guesswork. And under a tight deadline, there is always the gravitational pull toward the dump-and-review approach mentioned earlier—the temptation to offload responsibility unless leadership continuously reinforces boundaries and practices.

That's why, finally, there must always come the Own phase. The final code needs to be understood, documented, and owned by the engineering team. It is no longer "AI code"; it is our code—breaking the diffusion of responsibility and providing accountability. The surface you verify is larger, the feedback is multidimensional, and the failure modes are subtler (nondeterminism, prompt drift, closed-loop testing). The

ultimate challenge of vibe engineering isn't technical as much as it's epistemological: managing the known unknowns (captured in the spec) and building resilience to the unknown unknowns (emergent from the AI's black-box behavior). The Refactor/Own stages are your front line against the latter: refactoring forces engineers to deconstruct and reconstruct AI logic, often exposing hidden flaws.

However, what sounds easy in theory is really hard in practice. Ownership demands a deep, durable mental model so the team can reason about behavior when the unexpected happens.

1.6 **Owning: The last mile of vibe engineering**

Even with the most refined methodologies and powerful tools, the ultimate success of AI-assisted development hinges on confronting two fundamental, human-centric challenges. The first is the qualitative gap between rapidly generated code and a truly production-ready system. The second is the immense cognitive burden required to establish genuine ownership over code that a human did not author. These challenges represent the “last mile” of development, where the illusion of speed collides with the nonnegotiable demands of engineering discipline.

1.6.1 **The not-the-end-yet 70% Problem**

A foundational principle for understanding the practical limits of current AI coding assistants is what Addy Osmani from Google has termed the *70% Problem*. The concept posits that AI excels at accelerating the initial, well-trodden path of development—the first 70%—which primarily involves scaffolding features, generating boilerplate, and implementing common patterns. However, these tools falter when faced with the final 30%, which constitutes the most critical, context-sensitive, and high-stakes portion of engineering work. This dynamic creates a dangerous *illusion of competence* by making the easy parts of the job feel effortless, which can mask underlying knowledge gaps and create a false sense of progress toward a production-ready artifact. The 70/30 split is not merely a quantitative measure of work but represents a qualitative shift from tasks of *generation*, at which models excel, to tasks of *judgment*, which remain a uniquely human capability.

The “hard 30%” is not a monolithic block of work but a collection of distinct engineering disciplines that require deep contextual understanding and consequential reasoning. This is where the statistical pattern-matching of a language model proves insufficient:

- *Edge-case cartography*—AI-generated code is exceptionally good at implementing the “happy path” but consistently fails to account for the edge cases, error conditions, and boundary scenarios that define robust, real-world applications. Production readiness is defined not by how a system works under ideal conditions but by how it behaves at its boundaries when faced with rare inputs or unexpected user actions. This requires an understanding of what can go wrong, which current LLMs lack, as demonstrated by catastrophic data loss and security failures described at the beginning of this chapter.

- *Architectural integrity and integration*—Although individual AI-generated components may be locally correct, they often fail to integrate seamlessly into a larger system architecture. The generated code can violate project-specific conventions, introduce nonoptimal solutions, or create a less-consistent architecture that drifts from established patterns. This frequently leads to an integration nightmare, where each generated component is like a perfect puzzle piece for the wrong puzzle. This failure to align with global system design creates a new, insidious form of technical debt. Unlike classic technical debt, which is often a conscious tradeoff, this implicit architectural debt is silent and systemic. The code appears correct and may even pass unit tests, but its poor integration characteristics create liabilities in the seams between components, which become apparent only during a major refactoring or a stressful production incident. This is a technical manifestation of the trust debt we discussed earlier.
- *Comprehensive verification*—An AI assistant may generate basic unit tests that confirm the happy path, but the final 30% requires a much deeper level of verification. This includes writing property-based tests to probe system invariants, performance tests to validate SLOs, mutation tests to ensure test suite quality, and security tests based on a threat model. These tasks demand a level of intent and adversarial thinking—actively trying to break the system in creative ways—that AI currently cannot replicate, while being useful in finding blind spots in our own thinking.
- *The security and compliance*—Security is one of the most glaring weaknesses in the final 30%. Research indicates that a high percentage of AI-generated code contains vulnerable patterns. Models learn from vast public code repositories, many of which contain insecure or outdated practices, and they reproduce these flaws without understanding their security implications. This leads to textbook vulnerabilities like SQL injection, hardcoded secrets, insufficient input validation, and weak authentication flows. Furthermore, AI models are blind to regulatory and compliance frameworks, such as those in finance or healthcare, making their unverified output unacceptable for use in systems in which these rules are nonnegotiable.
- *Performance and scalability*—AI-generated code typically prioritizes functional correctness over performance and efficiency. The output can be nonoptimal, containing inefficient database queries, poor algorithmic complexity, or excessive resource usage that does not scale under production load. This directly conflicts with the vibe engineering principle of treating performance not as an afterthought but as an executable contract, specified and verified as an SLO gate in the CI/CD pipeline.

This problem is also self-compounding and poses a systemic risk to the software ecosystem. At a micro level, the probabilistic nature of LLMs means that small errors in a long generation sequence cascade and compound, drastically reducing the chance of a correct outcome in complex tasks. At a macro level, as more of this “70% complete”

code populates public repositories, it becomes training data for future models. This creates a negative feedback loop in which models are increasingly trained on mediocre, context-poor, and potentially insecure code, threatening long-term degradation in the quality of AI assistance across the industry. Curation pipelines, quality filters, and synthetic-data generation can break this cycle, but building and maintaining those pipelines is itself a serious engineering problem that reinforces rather than undermines the case that discipline, not model scale, is the binding constraint.

1.6.2 *The cognitive price of generation*

The software industry has become obsessed with optimizing the easiest part of development—writing code—while largely ignoring the true bottleneck: *reading and understanding it*. AI code generation has effectively solved the typing-speed problem, but in doing so, it has dangerously exacerbated the comprehension-speed problem. The ease of generation creates a flood of code that developers must then painstakingly analyze to build the necessary mental model for true ownership.

True ownership of code is not defined by authorship. It is the possession of a robust and accurate *mental model* of a system's behavior, its dependencies, and its failure modes. Without this internal map, a developer cannot debug effectively, anticipate the downstream impact of changes, or contribute confidently to the codebase. The code remains a black box, an artifact whose behavior cannot be reasoned about, which is antithetical to the engineering discipline of building predictable, deterministic systems. The process of building this mental model is cognitively expensive, requiring a developer to trace logic across multiple files, understand complex data flows, and infer unstated intent from the text.

This dynamic inverts the traditional relationship between intent and artifact, fundamentally changing the nature of the engineering craft. Historically, a developer's mental model precedes the code; the code is an artifact of their thought process—a form of forward motion. With AI assistance, this is reversed. The code artifact is generated first from a prompt, which is an imperfect and often ambiguous representation of intent. The developer must then become an archaeologist, working backward from the generated artifact to reverse-engineer its logic and reconstruct the author's (in this case, the model's) intent. This process is fraught with challenges. AI code can be needlessly verbose and complex, turning what should be a 68-line function into a 620-line behemoth that is a nightmare to maintain. The AI can also confidently hallucinate and write broken code, requiring constant vigilance and deep scrutiny from the human reviewer. This creates a sense of detachment and erodes the feeling of craft, shifting the engineer's role from creator to validator.

This individual cognitive burden translates directly into a systemic organizational problem. The dump-and-review culture enabled by AI offloads the hard cognitive work of building a mental model from the author to the reviewer. The true cost of AI generation is not the price of the API call but the cumulative human cognitive hours required to safely integrate its output. This creates a massive bottleneck in the development

process. A single developer can generate code far faster than a team can safely review and understand it, leading to the AI productivity paradox where individual velocity increases but team throughput stagnates or even declines. This dynamic also fuels trust debt and a diffusion of responsibility. The author can claim “the AI wrote it,” but the reviewer is overwhelmed, resulting in no single person truly owning the code’s quality. Some describe this as the worst form of tech debt imaginable. The solution lies in making human-authored, executable specifications the nonnegotiable source of truth and using tooling and automation to ensure that the AI’s work rigorously conforms to our intent.

1.7 *The beginning of software engineering*

To conclude our introduction to this book, we want to share one more thought that, like a lens, concentrates all the themes we’ve touched on. We are convinced that what is happening before our eyes is the inevitable end of the paradigm we long considered ideal: software craftsmanship. That movement, with its noble manifesto calling for well-crafted software, taught us pride, responsibility, and the pursuit of mastery. Looking back, however, we see that it was a tradeoff for which we paid a high price. The tools of the craftsmanship era—manual code review, tribal knowledge, artisanal CI scripts—are not wrong, but they cannot scale to govern AI-generated output at the volume and velocity teams now face. What was impressive as handiwork becomes a bottleneck when the factory floor runs at machine speed.

The mistake we made as an industry was to uncritically adopt the craftsman metaphor. Craftsmanship, strictly understood, is handiwork: unique, dependent on a single master, and impossible to reproduce at the scale and consistency that engineering demands. This idealism led us to costly errors: gold-plating, where we embellished code without real business value, and Sisyphean work, where each team reinvented the implementation of authentication, logging, or CI/CD from scratch. We paid for uniqueness with a lack of standardization and for mastery with a lack of work scalability. This was a fundamental, although often unnoticed, tradeoff.

In this context, the arrival of AI agents capable of generating code at scale has become a brutal mirror, exposing our weaknesses. The discomfort many feel is not just an identity crisis, but proof of a deeper tension we have ignored for decades: we joked that software development isn’t a true engineering discipline, failing to see that this joke described real errors in our approach—too much tacit knowledge, too many rituals, and a lack of proven, universal methods. Alchemists had exquisite technique without a unifying theory; we’ve had beautiful code abstractions without consistent, executable specifications.

Paradoxically, it is precisely AI agents—perceived as a threat to “craft”—that are becoming the catalyst forcing us to take a painful but necessary step toward engineering. The paradigm we propose is not about blindly trusting the machine. It means consciously trading direct control over every line of code for the ability to design and validate the entire system that produces it. It would be a mistake to think that this is an

easy exchange. We need to make the ultimate tradeoff: give up the satisfaction of manual creation, and gain the immense burden of responsibility for a precise, ironclad, and verifiable specification that becomes the heart of the process.

This specification—a human-authored contract defining the system’s behavior, performance, and security—is the missing link. It is the foundation of a repeatable, measurable production process that our discipline has lacked. We are witnessing not the end of creative work but its evolution, which brings its own pitfalls. Value no longer lies in tactical implementation prowess but in the strategic ability to define intent.

This process is frustrating. What was once our unspoken wisdom and intuition must now become the language of measurable criteria: metrics, tests, and contracts. We lose the comfort of tacit knowledge and gain the burden of explicitly formulated requirements. This irritation, however, is the price we pay for the transition from craft to engineering.

This does not mean abandoning the spirit of the craftsmen. On the contrary, their skills become more crucial than ever. It is their taste and decision-making ability that guide the AI’s code generation process. But here lurks the final, perhaps most important tradeoff: we must codify this taste, transforming it from a personal preference into an explicitly defined, automatically verifiable set of rules.

That is precisely why we named this book *Vibe Engineering: Best Practices, Mistakes, and Tradeoffs*. As an industry, we are currently reinventing a lot of techniques and must unlearn many things to move forward. In the following chapters, we will guide you through this new landscape, summarizing what we have learned so far—often the hard way. We will discuss specific challenges: how to migrate existing code in the AI era, how to precisely define context for generative agents, and how to collaborate with them effectively. We will also tackle tougher topics, such as the real costs of these technologies, fundamental changes in managing engineering teams, and how AI-assisted engineering can help with staff-level tasks like squeezing code performance metrics. The goal is not to avoid mistakes at all costs—the easiest way to do that would be to stick to the old-fashioned ways—but to make conscious decisions based on the knowledge gained from the ones we have already made. We want to take you on this journey through the world of difficult choices and unavoidable errors.

After years in the field, many of us hit a plateau: we recognize patterns faster and discover new ideas less often, the work becomes more re-creative than creative, and fresh abstractions no longer thrill as they once did. Not anymore. Today, it seems that since the 1960s, when the foundations of our field were laid, there hasn’t been a more fascinating moment for software engineering—a time when so much *new* is being forged at once: new primitives, new workflows, new guardrails, new roles. With this book, we add our small contribution to that forging: presenting techniques that turn the best of craft into a mature, repeatable method.

Summary

- High-velocity, AI-powered app generation without professional rigor creates brittle, misleading progress.
- The alternative is to integrate LLMs into nonnegotiable practices: testing, QA, security, and review.
- Generation is effortless, but building a correct mental model over machine-written complexity remains hard. Real ownership depends on understanding, not just producing, code.
- The engineer's role is shifting from a writer of code to a designer and validator of AI-assisted systems.
- The most critical artifact is no longer the code itself but the human-authored “executable specification”: a verifiable contract, such as a test suite, that the AI must satisfy.
- Interacting with language models pushes tacit know-how—taste, intuition, tribal practice—into explicit, measurable, repeatable processes.
- AI transition elevates software work to a higher level of abstraction and reliability, which requires good communication, delegation, and planning skills.
- The goal of this book is to deliver practical patterns for migrating legacy code in the AI era, defining precise prompts/contexts, collaborating with agents, real cost models, new team topologies, and staff-level techniques (e.g., squeezing performance).



Context fabric: Optimizing context for AI agents

This chapter covers

- Why imprecise context leads to errors in AI agents
- The difference between prompt engineering and context engineering
- Strategies for writing, selecting, compressing, and isolating context
- Designing effective planning, execution, and debugging loops with AI agents
- Proven optimization techniques for achieving high code accuracy

An LLM doesn't read minds: it feeds on the crumbs you throw it. You've probably heard about prompt engineering, which focuses on precisely formulating instructions for language models. Unfortunately, prompt engineering alone isn't sufficient. It leads to limitations and so-called *context failures*, which are the primary cause of AI agent breakdowns—not errors of the model itself.

The solution to this problem is context engineering. *Context engineering* is the discipline of designing, selecting, structuring, and maintaining the information

that an AI system receives as input—its *context*—so that it can produce reliable, high-quality output. Whereas prompt engineering focuses on how you phrase a question, context engineering focuses on the entire information environment surrounding that question: what data the model sees, in what order, at what level of detail, and from which sources.

This discipline provides the strategies for writing, selecting, compressing, and isolating context, as well as designing effective planning, execution, and debugging loops with AI agents. You will see how to apply these proven techniques to achieve high code accuracy, with comprehensive examples and an easy-to-consume structure. When done systematically across repositories and teams, this structured, agent-consumable context becomes what we call a *context fabric*: the persistent layer of specs, conventions, documentation, and structured knowledge that surrounds a project and gives any agent (or developer) immediate access to the accumulated intelligence needed to work effectively within it.

2.1 Vibe coding traps: Garbage in, garbage out

To understand the kinds of problems we’re talking about, consider a typical scenario. A developer—let’s call him Bob (convention over configuration FTW)—is struggling with a bug in his code. His first instinct is to ask the AI for help:

TL My login code is broken. It doesn’t work. Can you fix it (or you will go to jail)?

The answer Bob gets will almost certainly be . . . disappointing. Deprived of any concrete context, the model will likely generate a generic, textbook login function that has nothing to do with Bob’s actual codebase, framework, or the specifics of the bug. That’s why we say that working with agents is a self-fulfilling prophecy: if someone insists that agents don’t work, they almost certainly won’t.

This kind of low-context, ambiguous instruction directly leads to an entire class of common errors in AI-generated code:

- *Logical and semantic errors (the sneaky ones)*—Prompt the model to “fix login,” and it may swap your middleware-based session check for a naive password comparison, drop the 2FA step, ignore `tenantId`, and return 200 instead of 401 on failure. It compiles, but it silently violates invariants (rate limits, lockouts, audit logging, feature flags). By solving one problem, it generates multiple new ones.
- *Incomplete code (green builds, broken behavior)*—Instead of a fix, you get scaffolding: stubs with `// TODO`, `pass`, `throw new NotImplementedError()`, or a “happy-path only” implementation that skips error branches, retries, and cleanup. You’ll also see invented helpers (`user_auth.verify()`) or placeholders that never touch the real integration points—no DB migration, no dependency injection (DI) wiring, no event emitted, no test updated. That’s the model telling you it lacked the concrete inputs (schema, interfaces, configs, failing test) to finish the job.

- *Hallucinations*—The most insidious errors. In trying to guess the missing context, the model may invent nonexistent functions, variables, or modules—e.g., `user_auth.verify()`—that sound plausible but don't exist in the project's codebase.

To be fair, the latest generation of models has gotten better at pushing back—asking clarifying questions instead of blindly guessing—to the point that using them sometimes feels like cheating. But even when the model asks the right questions, the developer still has to know what context to provide. The burden has shifted, not disappeared.

These issues aren't isolated cases of bad prompting; they're symptoms of a deeper, fundamental misunderstanding—one that shows up not only in how people generate code but also in how they use chat tools to write emails or messages. A novice user treats an LLM like an oracle to ask for an answer. In many cases that can work (e.g., emails often have a generic tone), but for source code, it's frequently a dead end. Expecting the model to understand the problem abstractly—like a human teammate who will ask follow-up questions if needed—can lead you astray, because an LLM is *a tool*: a powerful, programmable text engine that operates solely on the context it's given. Output quality is a direct function of input quality.

The good news: every one of these failures traces back to a context problem. And context problems are engineering problems, which means they have engineering solutions.


Bob's earlier mistake wasn't a one-off slip. It's a symptom of a broader mindset: thinking you can just toss a problem at AI and wait for the magic to happen. It's also one of the reasons *vibe coding* (sorry, engineering!) has such a bad reputation. Before we dive into how to build good context, let's look at some of the most common anti-patterns and mistakes that almost guarantee failure. Spotting them is the first step toward breaking bad habits.

2.2 **Context Vacuum: First potential mistake**

This is the most common mistake that Bobs of this world make. It happens when you paste a code snippet—a function, class, or component—that's completely isolated from the rest of the project.

To be clear, working with isolated snippets isn't always wrong. If you're optimizing a pure algorithm, refactoring a self-contained utility, or reviewing a single function for readability, a focused context can actually be an advantage: less noise, sharper results. The mistake is doing it *by default* rather than *by design*: failing to realize that your 50-line function has invisible dependencies on everything around it.

Take this example. A developer copies a 50-line `authenticateUser` function and asks for it to be fixed or refactored:

 My authentication code is broken, can you fix it?

```
async function authenticateUser(req, res) {
  const { username, password } = req.body;
```

```

try {
  const user = await DatabaseClient.findUser(username);
  if (!user) {
    return res.status(401).send("Unauthorized");
  }

  const isValid = await bcrypt.compare(
    password, user.passwordHash);
  if (!isValid) {
    return res.status(401).send("Unauthorized");
  }
  const token = jwt.sign({ id: user.id },
    "secret", { expiresIn: "1h" });
  res.json({ token });
} catch (err) {
  console.error(err);
  res.status(500).send("Internal server error");
}
}

```

And the model's response is as follows:



```

async function authenticateUser(req, res) {
  const { username, password } = req.body;

  try {
    const user =
    await DatabaseClient.findUserByName(username); // ← invented method
    if (!user) {
      return res.status(404).send("User not found"); // ← changed
      semantics
    }
    if (password === user.password) { // ← dropped bcrypt!
      const token = jsonwebtoken.sign( // ← swapped library
        { id: user.id, name: user.username },
        "super-secret", // ← hardcoded secret
        { expiresIn: "2h" }
      );
      return res.json({ token });
    } else {
      return res.status(403).send("Invalid credentials");
    }
  } catch (err) {
    console.error(err);
    res.status(500).send("Server crashed");
  }
}
}

```

At first glance, this looks plausible: it compiles, and the AI even "helpfully" changed some status codes. But look at the inline annotations: the model invented a method that doesn't exist, dropped `bcrypt` in favor of a naive plain-text comparison, swapped the JWT library, and hardcoded a new secret—all without reason. The code is now *less secure*, less consistent with the real project, and full of hallucinations—all because Bob shared a fragment in isolation.


What does the model miss? Everything outside those 50 lines: which libraries Bob actually uses (jwt vs. jose vs. jsonwebtoken), what his data structures look like (is there a `passwordHash` field or just `password`?), and how the function is invoked (Express middleware? Spring service? Lambda handler?). Each missing piece is an invitation to hallucinate.

It's like diagnosing a patient from a photo of their iris: you might spot bloodshot eyes, but if the problem is in the lungs, you'll miss it entirely. Worse, like an iridologist, the model will sound confident about its wrong diagnosis. That's the essence of a *context vacuum*: code stripped of its surroundings, leaving the model to operate in an artificial void where failure is almost guaranteed.

Modern coding agents like Cursor and Claude Code mitigate this problem by automatically pulling in surrounding files and dependencies, but even they can only work with what they can see. If your project has poor structure, missing types, or undocumented conventions, even automated context retrieval will leave blind spots. The vacuum shrinks but never fully disappears. Let's attempt to solve the problem—and see why the solution isn't straightforward.

2.2.1 *From a single-shot to multishot examples*

Our clever Bob, seeing what's going on, decides that instead of pasting just the method, he'll provide the entire class along with all its necessary dependencies. Reasonable instinct—but not quite enough:

 Here's the full auth module. Please fix the authentication logic and any obvious issues.

```
// db/client.js
class DatabaseClient {
  constructor(pool) { this.pool = pool; }
  async get_user_by_username(username) {
    throw new Error("stub");
  }
}
module.exports = { DatabaseClient };
// auth/password.js
const bcrypt = require("bcrypt");
async function check_password(plain, hashed) {
  return bcrypt.compare(plain, hashed);
}
module.exports = { check_password };
// auth/service.js
const { DatabaseClient } = require("../db/client");
const { CustomAuthError } = require("../errors");
const { check_password } = require("./password");
const jwt = require("jsonwebtoken");
const config = { jwtSecret: process.env.JWT_SECRET || "super-secret",
  expiresIn: "2h" };
async function authenticateUser(req, res) {
  const { username, password } = req.body;
  try {
```

```

// NOTE: our real client exposes get_user_by_username, not
// findUserByName
const user = await DatabaseClient.findUserByName(username);
if (!user) { return res.status(404).send("User not found"); }
// FIXME: should use bcrypt hash comparison
if (password === user.password) {
  const token = jsonwebtoken.sign(
    { id: user.id, name: user.username },
    config.jwtSecret, { expiresIn: config.expiresIn }
  );
  return res.json({ token });
} else {
  return res.status(403).send("Invalid credentials");
}
} catch (err) {
  console.error(err);
  if (err instanceof CustomAuthError) {
    return res.status(err.code).send(err.message);
  }
  res.status(500).send("Server crashed");
}
}
module.exports = { authenticateUser };

```

It seems like a step in the right direction, doesn't it? The model now sees more: it knows that `check_password` is part of a service that uses a database client. Unfortunately, that's not enough. The model still doesn't know what `DatabaseClient` actually exposes, what the `User` object looks like, or where `CustomAuthError` is defined.

Each new piece of context we add—methods, classes, dependencies—makes the picture a bit clearer but also reveals just how much remains hidden. At some point, manually pasting snippets and trying to reconstruct the world around them simply doesn't scale.

So Bob levels up again: instead of providing just one file, he provides several: a controller showing the validation pattern, a repository demonstrating database access conventions, error classes with their custom fields, and an alternative auth flow to show how the project handles API key authentication. This is the core idea behind what coding tools call *multishot* (multiple examples of similar patterns) and *multifile* (pulling context from across the project structure). The goal is the same: give the model enough reference points to generalize, not guess.

```

TL // controllers/loginController.js

const { z } = require("zod");
const { AuthService } = require("../auth/service");
const LoginSchema = z.object({
  username: z.string().min(3),
  password: z.string().min(8)
});
async function loginController(req, res) {
  const parsed = LoginSchema.safeParse(req.body);
  if (!parsed.success) {

```

```

    return res.status(400).json({ error: "Invalid payload" });
  }
  try {
    const token = await AuthService.authenticateUser(
      parsed.data.username, parsed.data.password
    );
    res.json({ token });
  } catch (err) {
    res.status(err.code || 500).json({ error: err.message });
  }
}
module.exports = { loginController };
// -
// domain/errors.js
class CustomAuthError extends Error {
  constructor(message, code = 401) {
    super(message);
    this.name = "CustomAuthError";
    this.code = code;
  }
}
class ValidationError extends Error {
  constructor(message, code = 400) {
    super(message);
    this.name = "ValidationError";
    this.code = code;
  }
}
module.exports = { CustomAuthError, ValidationError };

```

Along with these, Bob might include repository patterns, alternative auth flows, or domain models—anything that illustrates the project’s conventions.

The challenge, however, is that our prompt often describes these fragments in a very sloppy way, if at all. We paste several files into a chat window, but we don’t clearly tell the model which one is a controller, which one is a domain model, and which one is just a helper. Without that framing, the LLM is forced to guess, and guessing is the fastest path to misinterpretation. A helper might be mistaken for core business logic. A data transfer object (DTO) might get treated like a persistence entity.

This is where we collide head-on with the limits of naive multishot and arrive at something more deliberate: *context engineering*. It’s not just about *how much* context we throw at the model (which comes with its own drawbacks), but also about *how we shape and label it*. Context engineering means curating examples, annotating them, and making sure the model knows exactly what role each fragment plays in the larger system. Done right, it transforms a pile of disconnected snippets into a coherent narrative that the model can actually reason about.

2.2.2 **Good multishot prompts vs. bad multishot prompts**

When working with LLMs in a chat-based coding workflow, the difference between success and failure often comes down to not the amount of context you provide but how you present it. This is where the notion of *bad vs. good multishot prompts* becomes very concrete.

THE BAD MULTISHOT PROMPT

A naïve multishot attempt usually looks something like this:

```

TL File 1:
class AuthService {
    def authenticateUser(username, password): ...
}

File 2:
class User { ... }

File 3:
class CustomAuthError(Exception): ...

```

No explanation, no labeling, no structure. Just raw snippets thrown into the prompt. The model receives them all at once and has to guess their purpose and relationships. Is `AuthService` a controller or a lower-level service? Does `User` come from the database layer, or is it just a DTO? Is `CustomAuthError` domain-specific or generic?

With nothing clarified, the LLM can easily misinterpret roles, generate code that doesn't align with conventions, or hallucinate missing details. Hallucination is, in most cases, the model's attempt to fill gaps that you left open.

THE GOOD MULTISHOT PROMPT

A better approach is to *annotate and frame each example explicitly*, turning scattered code into an intentional teaching set:

```

TL [Service Layer Example]
This class defines business logic for authentication.
It depends on the database client.
class UserService { ... }

[Domain Model Example]
This is a persistence entity that maps directly to the users table.
class User { ... }

[Error Handling Example]
Custom exception raised when authentication fails.
class CustomAuthError(Exception): ...

```

Notice what changed: nothing about the code itself, but everything about the *context*. By labeling each snippet, we remove ambiguity and tell the model exactly how to interpret the fragments.

The difference in the output is tangible. With the bad prompt, the model might generate an `AuthService` that directly queries the database—confusing the service

layer with the repository layer—or instantiate `User` as a plain dict instead of using the ORM entity, because it never knew `User` was a persistence object. With the good prompt, those same three snippets produce code that respects the layered architecture, uses the correct data access patterns, and throws `CustomAuthError` in exactly the situations our codebase expects.

That’s context engineering in action: structuring input so the model isn’t left guessing. Bad prompts rely on the model’s ability to infer structure from raw text, which works well in small cases but collapses as complexity increases. Good prompts offload ambiguity before it reaches the LLM. This small investment in annotation pays back heavily in correctness, consistency, and the ability to scale multishot techniques to larger codebases.

2.3 Building context together with LLMs

Multishot is a good strategy, but it has significant limitations. Even the best-described source code rarely reflects the complexity of the problem—you can always go deeper. This may work in the case of fixing a single function, but when planning a more complex task, doing this manually will be really tedious. Imagine, for instance, that Bob faces the implementation of an entirely new feature: a `User Profile Card` component in the existing system.

To handle this kind of situation, it will be necessary to gather a broader range of information. Bob’s new, improved workflow looks much better at first glance. Instead of pasting a single piece of code, he opens a series of tabs in his browser, trying to collect a complete picture of the situation. In one, he has the project file open in Figma; in another, the documentation of the company’s component library; in a third, the specification of the internal API on Confluence; and in a fourth, similar, existing components in the GitHub repository. With great care, he copies fragments from each of these sources: CSS variables and design tokens from Figma (and possibly also screenshots), type definitions and props from the documentation, the API response schema from Issue Tracker or a design document, and sample code from GitHub. All of this ends up in a single, gigantic, multipart prompt.

Although Bob is no longer working in a vacuum, his approach is in fact just a scaled-up version of the problems we discussed earlier. We just hit another bottleneck: manually collecting and structuring such a multi-entity context is not only inefficient and time-consuming but, above all, extremely error-prone. Just a moment of inattention is enough to paste an outdated piece of documentation, an obsolete code sample, or a reference to a deprecated API endpoint. In this way, even with the best intentions, Bob unknowingly poisons the context: introducing incorrect or outdated information into the model’s input, which misleads the AI. The LLM, receiving this carefully assembled but flawed mix, will obediently generate code based on false assumptions. This shows that manual context management at scale is not only a bottleneck to productivity but also a serious source of risk to the quality and consistency of code (and it’s painfully boring).

We need to do another shift, from manual curation to automated tooling. And for that, we need a systematic way to give the LLM access to tools that can connect directly with the relevant sources of truth—APIs, repositories, design systems, and documentation—so the model works with live, accurate data instead of fragile human-assembled snapshots without proper lineage.

2.3.1 Using Model Context Protocol to instrument LLMs

The answer to the limits of manual research is the *Model Context Protocol (MCP)*. This is an open standard based on JSON-RPC, initiated by Anthropic and quickly adopted by major AI players such as OpenAI and Google DeepMind. It is sometimes called “the USB-C of the AI world (which is a bit misleading—we prefer comparing it to Language Server Protocol), and its goal is to provide a universal interface that standardizes communication between AI systems and external data sources or tools. Instead of building dozens of custom, one-off integrations for every database, API, or service, MCP offers a single, consistent protocol for interaction.

Within months, MCP servers were in place for filesystems, GitHub, Jira, Confluence, Figma, various databases, and dozens of other tools. In December 2025, Anthropic donated MCP to the newly formed *Agentic AI Foundation (AAIF)*, a directed fund under the Linux Foundation, cofounded by Anthropic, Block, and OpenAI, with support from Google, Microsoft, AWS, and others. MCP joined two other founding projects: Block’s “goose” agent framework and the AGENTS.md standard (which we’ll cover later in this chapter). This move formalized MCP’s vendor-neutrality and ensured its long-term independence under the same stewardship that supports Kubernetes and PyTorch.

MCP’s architecture follows a familiar client-server model. The MCP client lives where the agent runs: for example, inside an IDE such as VS Code, Cursor, or Claude Code. Figure 2.1 shows how MCP servers are configured in the Claude desktop app, where each integration appears as a named tool the agent can invoke.

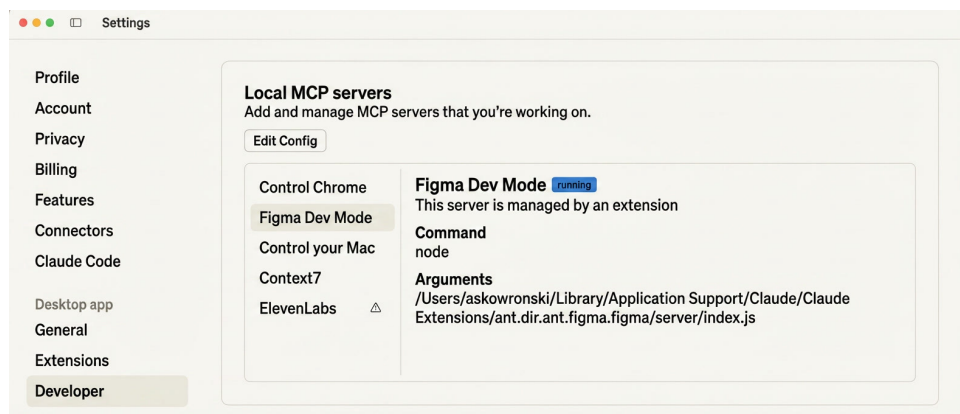


Figure 2.1 Configuring MCP servers in Claude App

The MCP server, on the other hand, is a lightweight service that exposes a set of structured, executable “tools.” One server might expose endpoints to query a design system in Figma, another to fetch metadata from a GitHub repository, and yet another to access issue trackers, documentation, or internal APIs. One of the most popular MCP servers is the one that acts as a bridge to the developer’s own machine, giving the agent structured and secure access to the local filesystem. From the model’s perspective, all of these are just standardized, callable operations, no different than functions in a well-defined library. This abstraction allows developers and organizations to connect their systems, whether cloud services or local resources, without constantly reinventing integrations for each use case.

Figure 2.2 illustrates this architecture: at the center sits the MCP client (the agent), which communicates via JSON-RPC with multiple domain-specific MCP servers. Each server acts as a bridge to a different data source: Figma for design tokens, GitHub for code, Confluence for documentation, and a database for schemas. The key insight is that the agent doesn’t need to know the specifics of each integration, only the universal MCP interface.

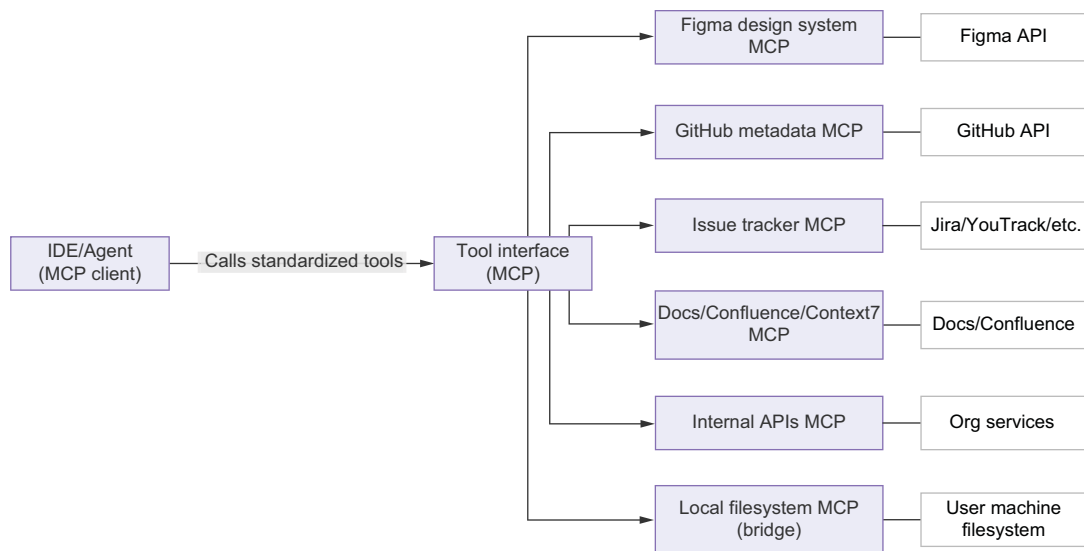


Figure 2.2 MCPs and universal integration tools for agents. The MCP client (center) communicates with domain-specific servers via a standard protocol. Each server translates between the agent’s requests and the external system’s API, returning structured data that the model can reason about.

Since early 2025, the MCP specification has matured significantly. The protocol now includes OAuth 2.1 authentication (replacing the earlier ad hoc auth story), *tool annotations* (`readOnlyHint`, `destructiveHint`) that help agents make safer decisions about which tools to call, *elicitation* capabilities that let servers request additional information

from the user mid-flow, and proper streaming support. These aren't just nice-to-haves: they're the features that made MCP viable for enterprise environments, and they directly address many of the governance concerns we'll discuss in section 2.3.5.

2.3.2 Building context for a UI component with MCP servers

Equipped with a tool that supports MCP, Bob can now approach his task in a completely new way. Instead of compiling context by hand, he writes a concise, strategic prompt that delegates the research to the model:

TL Implement a new React component called `UserProfileCard`. Retrieve the design specification - including colors, spacing, and typography - from the 'User Profile Card' frame in our main Figma project file. Ensure that the component's structure and style are consistent with our existing `BaseCard` component in the repository.

The card should fetch and display user data from the `/api/v2/users/{userId}` endpoint; find the schema for this endpoint in our API documentation.

This high-level description triggers a fully automated workflow in which the model orchestrates the work of several MCP servers to assemble a complete, structured context.

The process begins with a connection to Figma. Instead of forcing Bob to paste screenshots or manually transcribe tokens, the agent calls tools exposed by the MCP server for Figma Dev Mode. Through these calls, it retrieves structured data such as design tokens (colors defined as variables like `--primary-blue`), spacing values (`spacing-md`), layout constraints (Auto Layout properties), and component metadata.

Next, the model turns to the codebase and documentation. By querying the GitHub MCP server or local filesystem MCP, it locates the `BaseCard` component in the repository, inspects the file structure, reads the definitions of its props, identifies coding conventions, and checks imported dependencies. This ensures that the new component is aligned with existing practices and integrates seamlessly with the surrounding code. In parallel, the model connects to documentation via a documentation MCP server to retrieve the current API response schema, making sure the component will handle data correctly, without relying on outdated or incomplete references.

The result looks something like this:



```
<BaseCard
  className={["user-profile-card", className].filter(Boolean).join(" ")}
  variant="default"
  style={{
    "--upc-bg":      "var(--surface-card)",
    "--upc-fg":      "var(--text-primary)",
    "--upc-accent":  "var(--primary-blue)",
    "--upc-gap":     "var(--spacing-md)",
    "--upc-padding": "var(--spacing-lg)",
    "--upc-radius":  "var(--radius-xl)",
  }} as React.CSSProperties}
```

```

>
{ /* ... component body using tokens from Figma and API schema ... */ }
</BaseCard>

```

Notice the tokens: `--primary-blue`, `--spacing-md`, `--surface-card`. These are pulled directly from the design system (Figma → Dev Mode → variables), not hardcoded by the model. They’re intentionally tokenized. If the design team changes the primary color tomorrow, the component automatically follows. That’s the difference between a model guessing at style values and a model working with live, structured data. However, this is not where MCP shines the most—there are even more powerful use cases.

2.3.3 *Accessing external knowledge through MCP*

Now imagine the following requirement for `UserProfileCard`: the component should also include a *file picker* to let users upload their profile picture. At first glance, this seems simple, but there’s a catch: such a component does not yet exist in the project. If the model were restricted to only the libraries already present in the repository, we would need to find appropriate solutions ourselves. Even if we supply a library, it has never been used in the project, and the model has no patterns of usage to rely on. It may try to infer them from the file structure of the library itself or from loosely related components, but this is a *low-signal approach*: incomplete, error-prone, and often misleading.

With MCP, the agent has a much stronger play. Instead of hallucinating or improvising, it can use a dedicated *library research MCP server* to search for a suitable file-picker package. Once it identifies a candidate library, the agent can extend its context by retrieving the official documentation. For example, by connecting to a service such as Context7, it can automatically fetch installation instructions, API references, and code examples. All of this structured, authoritative information becomes immediately available in the same workflow.

Figure 2.3 illustrates this flow in practice. The process starts when the agent encounters a requirement it can’t satisfy from the existing codebase. It queries the library research MCP to discover candidate packages, selects the best fit based on compatibility and popularity, and then uses the Context7 MCP to retrieve the library’s documentation and usage examples. Finally, it integrates the library into the component using the retrieved patterns rather than guesswork.

The result is that developers no longer have to pause development to scour NPM, Stack Overflow, or vendor sites for scattered documentation. The model can autonomously discover the best-fit library, retrieve its usage patterns, and integrate it consistently into the existing project. This highlights another major advantage of MCP: the ability to move beyond the closed world of the repository and safely incorporate external knowledge, without diluting the signal with guesswork or out-of-date fragments.

The file picker case shows how MCP makes it easy to discover and integrate an entirely new library by pulling its documentation. However, there are situations where documentation alone is not enough. Sometimes what you really need is not just the “official” API contract but a deep understanding of how that API behaves in your project—with your build, your dependencies, and your type system.

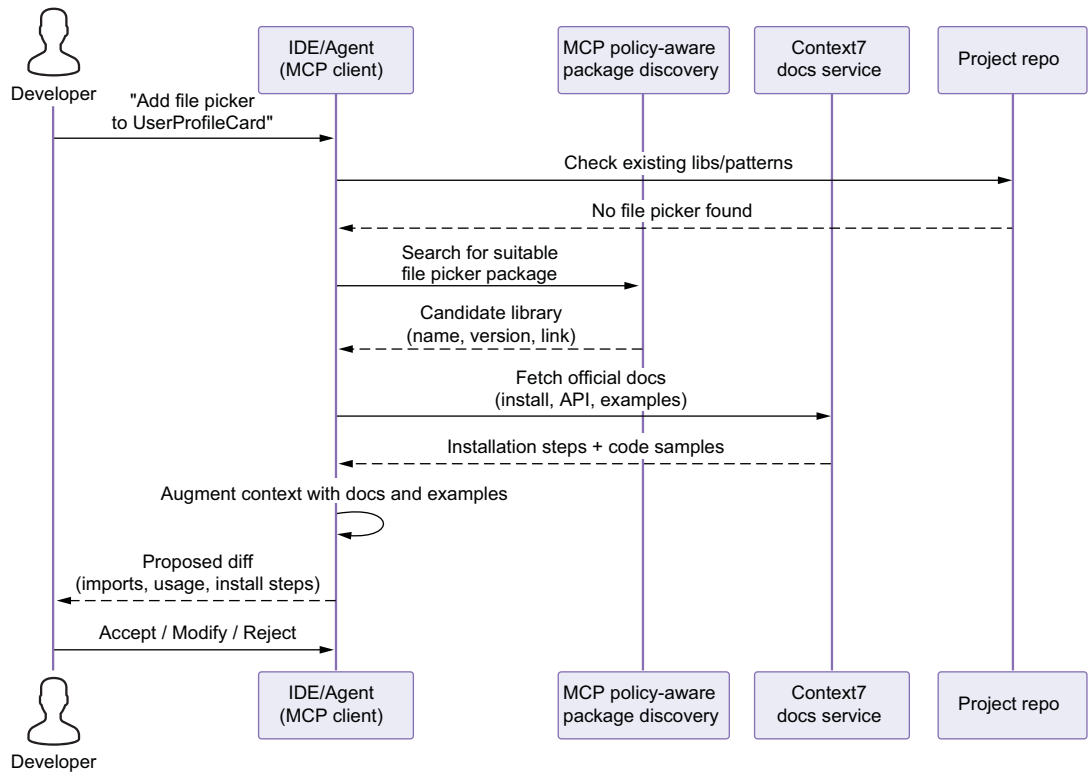


Figure 2.3 Working with dependencies with the Context7 MCP

2.3.4 Deep integration with Language Server Protocol

Now it is time for the last, the most advanced example. Let's imagine that we have the frontend for our solution ready, and it's time for the backend. To achieve that, we will connect the model to a Language Server Protocol (LSP) server via MCP. LSP is a communication standard (based on JSON-RPC) between an editor/IDE and a language server—the component exposing the compiler's semantic model in a uniform way. It's thanks to LSP that we get autocomplete, go-to-definition, type-hover, refactorings, semantic tokens, and code actions, all independent of any specific IDE.

As an example, Metals is an LSP server in the Scala ecosystem that understands the project through the build system, indexes symbols and types, and is aware of the full ClassPath (dependencies, versions, sources, JARs). Metals was the first LSP implementation that exposed MCP functionality, so we decided to use it here as an example. Figure 2.4 shows this setup, with Claude Desktop acting as the MCP client and Metals providing compiler-level access to the Scala project.

In this setup, the agent isn't limited to scanning raw files. By querying the LSP server (e.g., Metals), it gains access to *build-time truth* about the project: resolved types

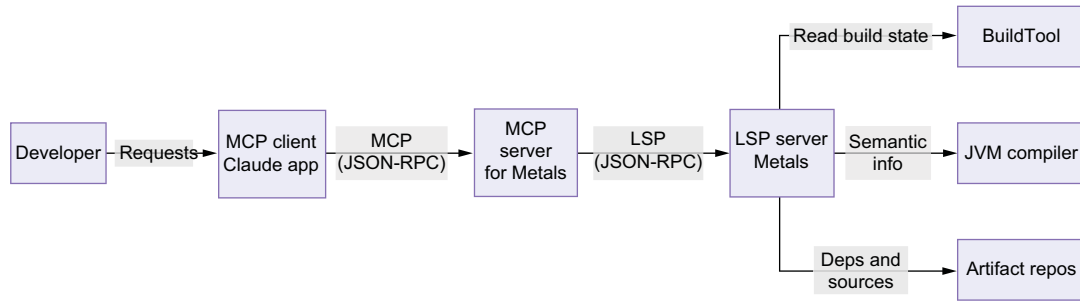


Figure 2.4 MCP ↔ LSP (Metals) with Claude Desktop as the MCP client

and dependency versions as the compiler sees them. An important caveat: this is not the same as *runtime truth*. The deployment configuration—which JVM version runs in production, whether the app is containerized, what Kubernetes environment it targets—lives in a different layer. LSP gives you the compiler’s view of the world, which is authoritative for type correctness and dependency resolution but shouldn’t be confused with operational behavior.

That said, build-time truth is already enormously valuable. It means knowing not only where a case class like `UserProfileCardProps` is defined but also the exact inferred type of `avatarUrl: Option[String]`. It can inspect the Java ClassPath to uncover which dependency versions are really being pulled in, resolving transitive libraries that come along and fetching source JARs to extract precise information. In effect, the model works with the same authoritative knowledge that your IDE and compiler use: fully resolved types, dependency graphs, and concrete code references, not approximations or guesses.

In other words, documentation can tell you *how* something should work in theory, but LSP tells you how it really works in the codebase. This way, we can achieve better *referenceable prompts*. The model can construct prompts that contain verifiable references: fully qualified type names, artifact versions from the ClassPath, definition paths, even commit identifiers. That makes a total difference both when coding and when generating prompts that are repeatable, auditable, and resistant to hallucination.

Wrapping everything together, our final prompt composes all of the context sources: design tokens from Figma, codebase conventions from the repository, API schemas from documentation, external library patterns from Context7, and compiler-level semantics from LSP:

- TL Implement a new Scala.js + React component called `UserProfileCard`.
 - 1 Design - Retrieve design specification from the Figma frame `User Profile Card`. Use tokens like `--primary-blue`, `spacing-md`.
 - 2 Codebase Conventions - Reuse the existing `BaseCard` component.

Query the GitHub MCP server for its props and import patterns.

3 API Integration - Fetch user data from `/api/v2/users/{userId}`.

Retrieve the JSON response schema from internal API documentation.

4 External Library - Add a file picker for profile picture uploads.

Use the library research MCP to find a Scala.js-compatible package, then fetch docs and usage examples from Context7.

5 **Compiler-Level Semantics**

- Through the LSP MCP server (Metals):
- Verify the inferred type of `avatarUrl: Option[String]`.
- Inspect the ClassPath for actual Cats Effect and Circe versions.
- Apply any suggested code actions for deprecations.

Output: A fully functional component with inline comments referencing

Figma tokens, API schemas, and library docs as sources of truth.

With such a rich, multisource, and—most importantly—structured context at its disposal, the agent can synthesize all the information and produce a complete, high-quality React component. The result is visually faithful to the design, consistent with repository conventions, and correctly integrated with the backend API.

But, as always, there are tradeoffs. Once the agent has access to a broad set of tools, new questions emerge: Which tools should it use in a given situation? How do we ensure security when connecting to external systems? How do we handle authentication, logging, and governance when sensitive APIs are involved?

We'll cover these topics in depth later, in the context of continuous development and secure integration. But even at this stage, one problem becomes clear: *more context isn't always better*. A growing context can easily become overwhelming, noisy, or contradictory, and the model must be guided carefully to avoid drowning in information. So we arrive at an important question: is it really true that giving an LLM more context always leads to better results? Before we tackle that question in section 2.4, let's address the elephant in the room: when you give agents the power to call arbitrary external tools, you need guardrails.

2.3.5 MCP governance

Early adopters often connect everything: filesystem, GitHub, Slack, email, calendar, internal wikis, documentation sites, database introspection. The result is an agent drowning in context, spending more tokens exploring than producing, and occasionally surfacing sensitive information in unexpected ways. Each MCP server exposes multiple tools, and the numbers add up fast. Connect a handful of servers, and you can overwhelm the model with tool definitions and results. The exact threshold varies by client, but the failure mode is consistent: more tools can mean more confusion, more tokens, and worse decisions.

The management burden extends beyond tool count. MCP servers are scattered across GitHub repos, community threads, and unofficial registries. Every connection

is a potential attack surface—security researchers have identified vulnerabilities including prompt injection, tool poisoning, and permission escalation. And without team standards, each developer configures their environment differently, leading to inconsistent agent behavior across the codebase.

The ecosystem has responded with increasingly mature solutions at three levels. MCP registries and app-store-style catalogs—such as the official MCP Registry and Google’s Antigravity MCP Store—have materially improved discovery, but they have not eliminated the need to vet servers like production dependencies. *MCP proxies* (such as MetaMCP and Rube) address tool explosion: instead of your agent seeing 50 tools from 5 servers, a proxy presents a single interface that dynamically loads relevant capabilities on demand. And *MCP gateways* (from vendors like Airia, Lasso Security, and TrueFoundry) tackle enterprise governance: centralized control planes that sit between agents and MCP servers, enforcing authentication, applying access policies, logging every request for audit trails, and blocking suspicious activity in real-time. Figure 2.5 illustrates this layered approach.

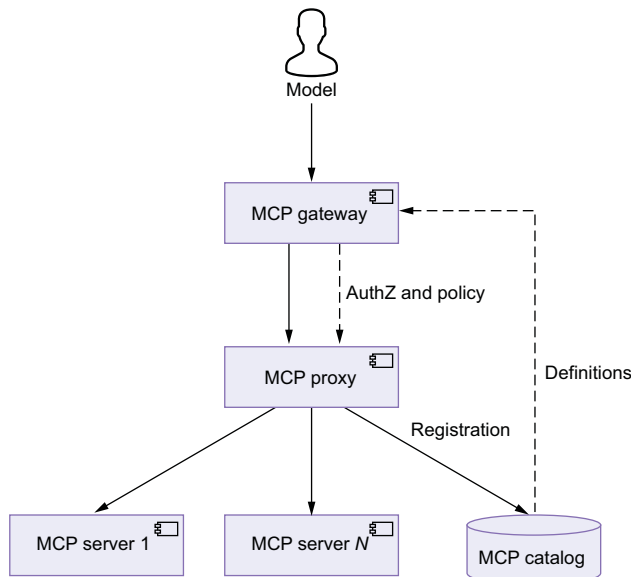


Figure 2.5 MCP governance layers. The agent communicates through an MCP gateway (enforcing auth and policies), which routes to an MCP proxy (aggregating multiple servers into a curated toolset), which connects to individual MCP servers registered in a central catalog. Each layer adds security and reduces context noise.

The practical wisdom that crystallized through this evolution is that we need to *treat MCP servers like production dependencies*. Two or three well-chosen integrations (typically filesystem, documentation, and one domain-specific tool) outperform 20 poorly configured ones. Maintain a team-approved allowlist. Document which servers are enabled and why. Review the MCP configuration as part of your onboarding process. The current MCP specification includes OAuth 2.1, tool annotations such as `readOnlyHint` and

destructiveHint or Streamable HTTP transport. but those annotations are hints for clients, not security guarantees.

More is not always better. This curated, minimal set of well-governed MCP integrations—filesystem, documentation, and one domain tool—is itself a piece of the context fabric: a deliberate, maintained connection layer that any agent session can rely on without rediscovery. As agents gained hands, they also gained memory. Every file read, every test output, every configuration detail became potential context. What began as a solution to context blindness quietly introduced a new constraint: not too little context, too much.

The question shifted. It was no longer “Can the agent see enough?” but “Can it see too much?”

2.4 Context rot: Is too much context a bad thing?

Every model has its own context window: the amount of data it can take in and process at once. We can remember when these windows were very small. As recently as 2023, models like GPT-3.5 had a window of only 4,096 tokens, which often wasn’t even enough to summarize a longer article without first splitting it into chunks. Since then, thanks to techniques like positional interpolation—most notably rotary position embedding (RoPE; introduced by Su et al. in “RoFormer: Enhanced Transformer with Rotary Position Embedding” [2021])—along with ALiBi and other architectural advances, the progress has been dramatic. Today, some frontier models, such as Claude Sonnet 4.6 and Gemini 3.1 Pro, advertise or expose 1-million-token modes, theoretically allowing entire books or repositories to fit in a single query. However, bigger context comes with its own set of problems.

2.4.1 “Lost in the middle” problem

This can be deceptive on two levels. On the one hand, when vendors advertise million-token context windows, it’s tempting to just dump an entire project into a single query and hope the model can handle it. Be cautious with these claims: the model may technically *accept* that much input, but the real question is at what cost to accuracy and at what price per query. Benchmarks consistently show that performance degrades sharply as context length increases. In the LongCodeBench evaluation, for instance, Claude 3.5 Sonnet’s accuracy dropped from 29% at 32,000 tokens to only 3% at 256,000 tokens (Liu et al., “LongCodeBench: Benchmarking Long-Context Code Generation”, 2024). Other models exhibit similar collapses once pushed beyond their effective range. And beyond the accuracy hit, there are practical costs: processing hundreds of thousands of tokens drives up latency, GPU memory pressure, and overall inference expense, making giant prompts not only less reliable but also disproportionately costly.

Real-world experiments mirror these findings. The developer community has tried loading nearly a million tokens of source code into a model, expecting it to navigate the repository and answer targeted questions. At smaller slices—say, 20,000–30,000 tokens—the model managed reasonably well, identifying dependencies or suggesting fixes. But as the prompt ballooned into the hundreds of thousands of tokens, its

answers degraded quickly: functions were hallucinated, file structures were misremembered, and critical instructions buried mid-prompt were ignored entirely.

Research explains why. Adobe’s NoLiMa benchmark (2025) showed that models fail not just from sheer length but because they struggle to retrieve nonliteral information in oversized contexts—especially when it’s hidden in the middle of the input. This is the phenomenon known as *lost in the middle*. Studies from Stanford and MIT (most notably, “Lost in the Middle: How Language Models Use Long Contexts” by Liu et al. [2023]) demonstrated that language models, much like humans, remember information best when it is placed at the beginning or end of a long text. Their ability to retrieve key details plummets if those details are buried somewhere in the middle of an extended context. This effect, sometimes referred to as *context rot*, produces a characteristic U-shaped performance curve.

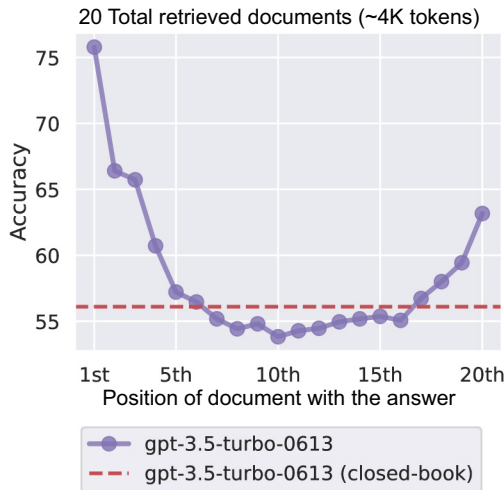


Figure 2.6 Loss of accuracy as context size grows

Figure 2.6 illustrates this relationship between context size and retrieval accuracy. Notice how performance holds steady at small context sizes, collapses in the middle range, and partially recovers only for information placed at the very end of the input.

This means that even if the correct information is provided, it can be effectively lost in the noise, leading to suboptimal or incorrect outputs. And this is precisely the trap that leads to the next common mistake developers make.

Let’s imagine that Bob wants to fix a function but buries the most critical instruction in the middle of a long prompt:

TL I’m working on a new feature for our platform, the `UserProfileCard` component. We use `Scala.js` with `React` bindings, `sbt` as a build tool, and `Metals` as our LSP server. Our project is aligned with `Clean Architecture` and is split into `domain`, `application`, and `infrastructure` modules. Please make sure the component follows our conventions. You can find the base structure by looking at the `BaseCard` component in our repository - check the props, import patterns, and coding style. The design tokens (colors, typography, spacing) should be consistent with the Figma frame `User Profile Card` in our main project file. Look up values like `--primary-blue` and `spacing-md`. Also note that the layout relies on `Auto Layout` constraints, so please preserve that when generating `CSS`.

We are also planning to introduce a file picker for profile picture uploads. You may need to check an external library such as `react-dropzone` for Scala.js wrappers. You can consult Context7 for installation instructions and usage patterns.

👉 ****Important:** please fix the function that loads user data so it calls `/api/v2/users/{userId}` instead of the deprecated `/api/v1/users/{userId}`.** 👉

Aside from that, make sure the code is compatible with our build (Scala 3.3.1, sbt 1.9.x).

You may need to resolve transitive dependencies for Cats Effect and Circe, and confirm types with Metals (e.g., inferred type of `avatarUrl`: `Option[String]`).

Finally, add inline comments referencing where design tokens and API schemas came from so future developers can trace the sources.

Because the critical instruction about the v2 endpoint is lost in the middle, the AI may ignore it and generate a fix that still uses the deprecated v1 endpoint. In the real world, you need a much longer prompt to trigger the effect, but this prompt is a representative illustration of the problem.

The result is a broader pattern that we can now name: *prompt degradation*. Larger windows don't yield superhuman comprehension of massive inputs: they produce slower, noisier, more error-prone assistants that cost more to run while delivering less value. This is a perfect example of what always happens in software development: there are no silver bullets, only tradeoffs. In an attempt to provide the agent with all the necessary conventions and rules, the developer makes an escalated mistake: drowning the agent in noise. Having learned that more context is better, they overcorrect by providing excessive and often irrelevant information—entire modules, verbose application logs, and unrelated documentation—overloading the agent's context window. Fortunately, several context engineering strategies can directly combat this problem, ranging from simple prompt adjustments to sophisticated automated pipelines.

2.4.2 Manual reordering: The “sandwich” method

The most direct, low-tech solution is to manually structure your prompt to align with the model's cognitive bias. Instead of letting important information get lost, you place it in the positions of highest attention: the beginning and the end. Bob (poor Bob, or poor You, dear reader) needs to double down on his curation efforts, creating a minimal, potent context for each interaction to ensure that critical information is never buried, painstakingly selecting only the most relevant snippets for each task.

The structure is straightforward. Put primary instructions and the most critical rule right at the start of the prompt. Supporting information that may be useful but isn't essential goes into the middle. Finally, at the very end, repeat or re-emphasize the most critical element: often the piece of code to be acted on or the final instruction. Here's how Bob's prompt looks after applying this technique:

  **CRITICAL INSTRUCTION:**

Fix the `loadUserData` function so it uses the `/api/v2/users/{userId}` endpoint instead of the deprecated `/api/v1/users/{userId}`.

—

Supporting context (less critical):

- The component is `UserProfileCard`, built with Scala.js + React.
- Ensure design fidelity by retrieving tokens from the Figma frame `User Profile Card` (e.g., `--primary-blue`, `spacing-md`).
- Align with repository conventions by inspecting the existing `BaseCard` component for props and style patterns.
- We may add a file picker for uploading profile pictures. If needed, check an external library such as `react-dropzone` for Scala.js wrappers.
- The project uses `sbt 1.9.x`, `Scala 3.3.1`, `Cats Effect`, and `Circe`. `Metals LSP` can confirm inferred types (e.g., `avatarUrl: Option[String]`).

—



Final reminder:

Update the `loadUserData` function to call `**/api/v2/users/{userId}**`. Do not use `/api/v1`.

This is the same information, reorganized according to the model’s attention pattern: the critical instruction bookends the prompt at the beginning and the end, and supporting details sit in the low-attention middle zone where their relative unimportance can’t cause damage.

The tradeoff is a significant increase in manual effort before every AI interaction. In practice, this means the developer becomes a kind of context editor, reading every prompt with the model’s attention curve in mind, ranking information by importance, and structuring it accordingly. For occasional prompts, this is manageable. For the dozens of interactions a developer has with an AI agent each day, it quickly becomes unsustainable.

We need to do better. That’s why we now arrive at a point where relying on purely manual solutions becomes increasingly difficult and we will bring dedicated tooling to the surface.

2.5 *Using AI coding tools to manage context*

The sandwich method works, but it demands that the developer manually curate every interaction. At dozens of AI-assisted prompts per day, that simply doesn’t scale. This is where AI coding tools step in—not by replacing context engineering, but by automating its most tedious parts.

By now, we assume that most developers have at least tried coding assistants built into IDEs: `JetBrains AI Assistant`, `GitHub Copilot Workspace`, `Cursor`, `Cline`, `Brokk`, `Claude Code`, or `Replit’s` new agent-like tools. These aren’t just chat-wrappers bolted onto an editor; they’re designed to automate much of the manual effort we’ve just described in context engineering. And although it’s tempting to get lost in feature lists, that’s not our goal here. In this section, we’re interested in how they actually handle context, what techniques they rely on, and why those techniques matter.

We'll save the broader discussion of agentic capabilities and their role in continuous AI development for a later chapter. For now, our focus is on the concrete tactics these tools employ to solve the very problems we've just explored.

2.5.1 Automated reordering using retrieval-augmented generation

The best way to solve the lost-in-the-middle problem is to avoid overloading the context window in the first place. That's exactly what *retrieval-augmented generation (RAG)* is built for.

Instead of forcing the model to parse thousands of lines of source files, logs, and documentation all at once, a RAG system searches the repository and retrieves only the functions, classes, or config files relevant to the developer's query. The LLM then works on this curated slice—for example, just the `auth.js` login handler and its related tests when you ask about fixing authentication—rather than drowning in unrelated code.

Figure 2.7 illustrates this flow from query to response. Let's walk through it step by step. First, the tool *indexes* the codebase, converting source files into searchable

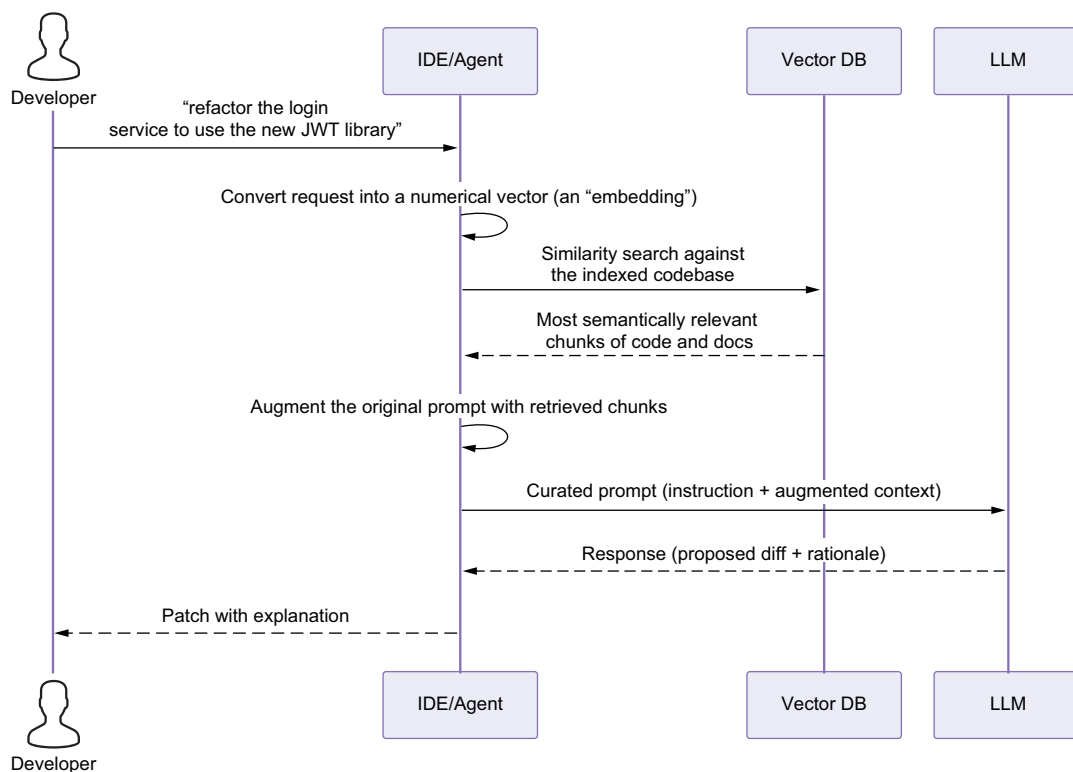


Figure 2.7 Context-aware assistance flow. The developer's query is converted into an embedding and matched against the indexed codebase, and the most relevant chunks are injected into the prompt before the model generates its response.

embeddings. When a developer makes a request, the system *retrieves* the most relevant chunks by comparing the query’s embedding against the index. Finally, these retrieved chunks *augment* the original prompt—placed alongside the user’s question in the positions the model attends to most—and the model *generates* a response grounded in that curated context.

In practice, this is how modern coding tools and IDEs provide context-aware assistance across an entire repository without overloading the model:

- *Indexing*—The first step is to create a searchable knowledge library; IDEs are doing that for most of their capabilities, not only AI-related ones. The tool scans the entire codebase—source files, documentation, configuration—and breaks it down into smaller, meaningful chunks (e.g., functions, classes, markdown sections). However, for AI tooling, here comes the twist: each chunk is then converted into a numerical vector (an *embedding*) and stored in a specialized vector database.
- *Retrieval*—When you make a request (e.g., “refactor the login service to use the new JWT library”), the agent converts your query into a vector. It then uses this query vector to perform a similarity search against the indexed codebase, instantly finding the most semantically relevant chunks of code and documentation.
- *Augmentation and generation*—The agent takes the top few retrieved chunks and “augments” your original prompt with them, creating a focused, signal-dense context, due to automatic reordering. This curated prompt is then sent to the LLM, which now has precisely the information it needs—the function to be changed, examples of the new library’s usage, and relevant error handling patterns—to generate an accurate and contextually appropriate response.


By using RAG, coding tools automate the painstaking work of manual curation. They ensure that the model receives a small, potent context, which not only avoids the lost-in-the-middle problem but also reduces costs and improves response speed.

But there’s a tradeoff. The coding context generated by AI coding tools is rarely truly “small.” On top of the retrieved chunks, each tool injects its own system prompt: a sizable block of instructions guiding the LLM’s behavior. Unlike totally hidden directives, many of these prompts can be inspected or partially edited, depending on the tool. Still, they consume valuable context space.

Imagine this scenario with Bob’s original prompt:

 My login code is broken. It doesn’t work. Can you fix it (or you will go to jail)?

What actually reaches the model isn’t just Bob’s plain message; it might look closer to this under the hood:

 SYSTEM PROMPT (10,000–20,000 tokens):

You are a highly experienced software assistant. Always respond in clear, structured Python code. Confirm the user’s intent before providing a patch. ...

USER PROMPT (Bob's message):

My login code is broken. It doesn't work. Can you fix it (or you will go to jail)?

CODE CONTEXT:

[...relevant code snippet...]

Just as HTTP's headers carries essential metadata—routing info, authentication, content-type—before the actual payload, the system prompt delivers critical guidelines, context rules, and behavioral constraints ahead of the user's input. In networking, this extra information is typically called *overhead*: data that's necessary for communication but doesn't contribute to the payload itself. Similarly, in coding tools, system prompts represent *context overhead*: they ensure correct behavior but dilute the available context window.

But that's not all. While the benefit, of course, is convenience (RAG frees developers from having to manually curate context, automatically fetching and reordering relevant pieces of the codebase), the cost is loss of control. The retrieval engine decides what to include, not you—and it may not always align with what you actually know is relevant. That's why many AI IDEs give developers the option to scope the context manually, narrowing which files, functions, or documentation pages the tool should draw from, to avoid overstuffing the LLM with irrelevant material.

Figure 2.8 shows how Cursor handles this. Notice the explicit file and symbol selectors that let developers override the retriever's choices.

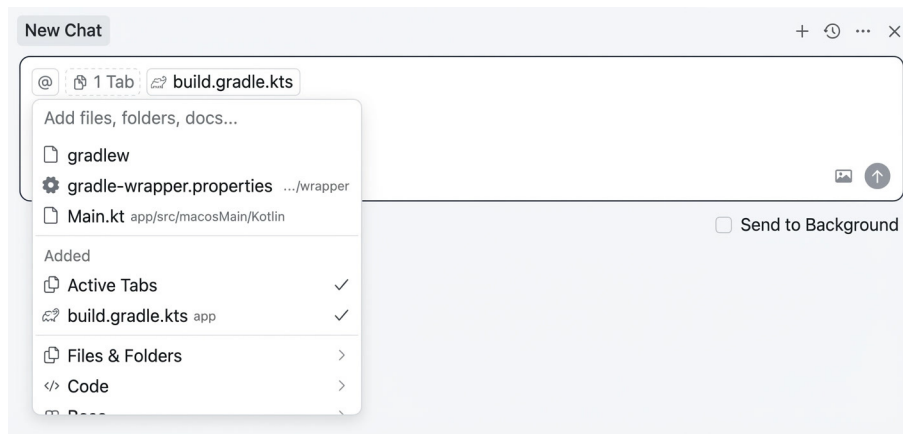


Figure 2.8 Scoping context in Cursor

To summarize, RAG is a powerful step forward, but it's not a silver bullet. Context windows in real coding tools can still become enormous, and without additional mechanisms like focus engineering and context compaction (both of which we'll define shortly), even the smartest retrieval strategies will eventually hit their limits.

2.5.2 Context anchoring: A to-do list for LLMs

The reason for this behavior is simple: RAG excels at quickly surfacing relevant information for each specific request. By querying a vector-based knowledge store, it pulls in the most pertinent code snippets, documentation bits, or notes before composing a response. Although this clear, targeted context helps improve accuracy and reduce hallucinations, it addresses only current relevance, not recall over time. Because it relies on dynamically fetched context for each query, RAG provides only situational awareness, but no continuity, so an AI tool can easily lose sight of past steps in a multistage task.

That’s why each of these tools has some kind of mechanism to memorize or revisit prior decisions. However, even with that mechanism in place, there are still visible problems. Imagine an agent that truly logs every turn of a coding conversation: prompts, code diffs, error messages, corrections, design rationales, each appended to the context. Over just a few iterations, the context swells. Suddenly, the most recent instruction is buried in the middle. RAG may inject relevant help, but if the context is already bloated, the power of retrieval is wasted, and the agent still “forgets” what happened earlier. This short-term memory limitation forces developers to continually restate instructions, re-provide context, and monitor consistency—diffusing efficiency and inviting mistakes.

This amnesia stems from how LLM-based agents operate: they view memory as nothing more than the current sliding window of tokens. Once past the window’s cut-off, prior context vanishes—just as our own short-term memory fades if not solidified. Without persistent, structured memory storage, AI agents suffer from what some call *context poisoning*: they become confused over time, hallucinate randomly, or derail into incoherent outputs when stretched beyond their fresh context. Unlike humans, who consolidate key information into long-term memory, AI lacks such consolidation—unless it’s explicitly built.

To illustrate the problem, Figure 2.9 shows how tools like Cline let developers track exactly how full their context window has become. This is a visual reminder of how quickly the available space disappears as a session progresses.

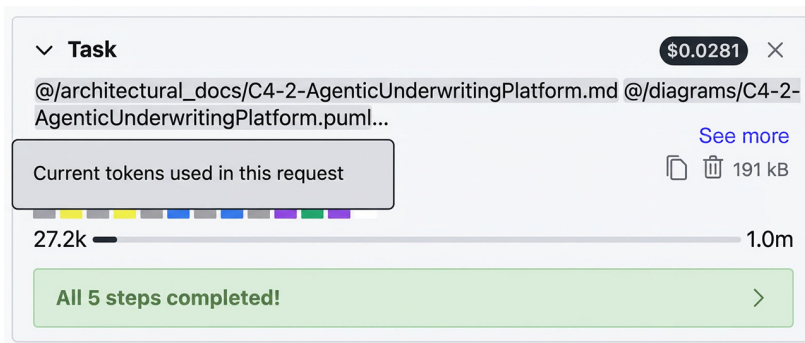


Figure 2.9 Tools like Cline allow you to track how full the current context window is.

To combat this syndrome, advanced agents engage in what can be called *context anchoring* or *focus engineering* (depending which vendor you ask). They create and maintain an explicit plan or to-do list that persists throughout a task, acting as an external memory and a constant guide.

A practical example comes from Claude Code, which uses what Anthropic calls a *to-do tool*. On the surface, this tool doesn't "do" anything—it isn't compiling, linting, or executing code. Instead, its sole purpose is to generate and continuously maintain a to-do list with the execution plan. The list is actively managed as the work progresses: items get checked off, reprioritized, and updated. At regular intervals, this plan is recited back into the context window, ensuring that the agent's high-level objectives are always pulled into the freshest part of the prompt. In effect, this simple "no-op" becomes a powerful context anchor: it manipulates the model's attention so goals are never buried under a flood of intermediate steps or previous errors. By making the plan itself part of the conversation, Claude Code preserves focus without relying on naïvely replaying the entire interaction history, which would only bloat the context window and accelerate the very amnesia it's trying to avoid. Figure 2.10 shows what this execution plan looks like in practice; notice how each task is tracked with its status, giving the agent a structured view of progress.

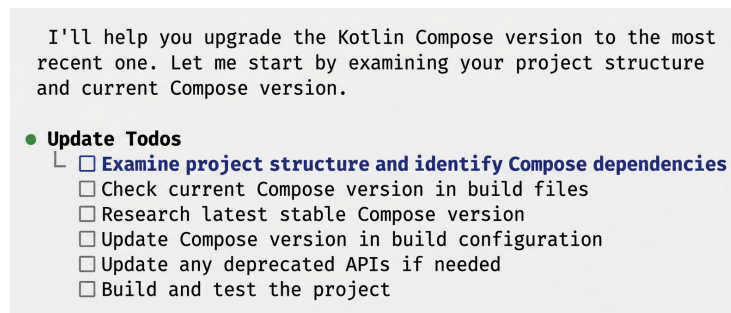


Figure 2.10 Execution plan in Claude Code

2.5.3 Context compaction

As these workflows have evolved, tools like Claude Code have moved away from repeatedly restating to-do lists inside the conversation. Instead, they increasingly externalize plans and durable instructions into artifacts such as project memory files, compacted summaries, and subagents. Where the original approach was ephemeral context engineering—manipulating what sits in the context window right now—this is closer to persistent state engineering: key guidance survives compaction and can be reloaded across session boundaries.

But what happens when even this approach runs out of room? This is where context anchoring meets *context compaction*. Claude Code manages context automatically

as you approach the limit. It clears older tool outputs first and then summarizes the conversation if needed. Claude Code automatically compacts the conversation history as sessions approach context limits, and developers can steer that process with `/compact` and project-level guidance in `CLAUDE.md`.

The compaction is a lossy summarization: Claude determines what should survive based on recency, relevance, and frequency. Your requests and key code snippets are preserved; detailed instructions from early in the conversation may be lost. A typical compaction frees up 60–70% of the context, but the tradeoff is real: project conventions—the things that prevent bugs—are mentioned once at the beginning of each conversation and are prime candidates for compression. This is exactly why Anthropic recommends putting persistent rules in `CLAUDE.md` (which will be described in chapter 3) rather than relying on conversation history—the `CLAUDE.md` file is reread at session start, giving critical instructions a second life even after compaction strips them from the conversation. Files like `CLAUDE.md` are a direct expression of the context fabric idea: project intelligence that lives outside any single session, always available to be loaded fresh into a new context window.

Coding tools also attack the context problem from another angle entirely: *sub-agents*. Subagents are specialized AI assistants that handle specific types of tasks. Each subagent runs in its own context window with a custom system prompt, specific tool access, and independent permissions. The built-in *Explore* subagent, for instance, handles codebase search and file discovery in a separate context; this keeps exploration results out of your main conversation context. When the subagent finishes, only its final summary returns to the parent conversation. All the intermediate file reads, search results, and reasoning stay isolated. This is a fundamentally different answer to context pressure than compaction: instead of compressing what’s already bloated, you prevent the bloat from entering the main context in the first place.

Agents get smarter, and become increasingly good at building their own context if given the right tools. The trajectory—from rigid to-do lists with forced reminders, to flexible tasks with cross-agent coordination, to subagents that isolate their own context entirely—illustrates a broader lesson about context engineering: the best solutions evolve with the models they serve, and what worked six months ago may now be a constraint rather than a help.

Other tools make different bets about where to accept context loss. Cline’s Auto Compact actively rewrites both the plan and the conversation summary together, betting on freshness over stability. Cursor collapses older dialogue into structured natural-language summaries. JetBrains AI Assistant replaces conversation history with symbolic references to the codebase itself. But the fundamental problem is the same everywhere: raw conversation is too expensive to keep in its original state, and every compaction strategy is a form of lossy compression.

2.5.4 Beyond compaction: Meta-prompting and state externalization

The techniques we’ve discussed so far—RAG, context anchors, compaction—all share a common assumption: the context window is the battlefield, and the fight is about

what stays inside it. But what if the answer isn't to compress more cleverly, but to stop treating a single conversation thread as your build system?

This is the insight behind a new generation of *meta-prompting frameworks* that emerged in late 2025 and exploded in popularity through early 2026. Instead of fighting context rot inside a single agent session, these systems externalize project state into files—specs, plans, research notes, validation contracts—and spawn fresh agent sessions for each task. The conversation becomes disposable; the project intelligence persists on disk. Two frameworks in particular illustrate this pattern: GSD (“Get Shit Done”) by TÂCHES and Superpowers by Jesse Vincent.

GSD: THE THIN ORCHESTRATOR

GSD is a spec-driven development workflow that sits on top of Claude Code (and now also supports OpenCode, Gemini CLI, Codex, Copilot, Cursor, and Antigravity). It has grown rapidly—more than 40,000 GitHub stars by March 2026—but what matters here is not popularity; it is the context-engineering philosophy underneath.

GSD's core idea is that context rot is not a compression problem, it's a *workflow* problem. As its creator puts it: “The complexity is in the system, not in your workflow.” Behind the scenes, GSD implements XML prompt formatting, subagent orchestration, and state management. What the developer sees is a few commands that just work.

The workflow follows a *Discuss* → *Plan* → *Execute* → *Verify* lifecycle:

- *Discuss phase*—Before any code is written, GSD forces a discussion phase that locks product decisions into a `CONTEXT.md` file. This is a more rigorous version of the context anchor concept, but externalized to files rather than held in the context window. The agent asks clarifying questions, surfaces assumptions, and records the answers. Once the discussion is complete, the decisions are persisted and never need to be re-explained.
- *Plan phase*—The system researches the codebase (optionally using a dedicated research subagent) and then generates a detailed implementation plan broken into small, concrete tasks. Each task is written assuming that the implementer has zero context for your codebase, because each task will be executed by a fresh agent with a clean context window.
- *Execute phase*—Here is the key insight: instead of compacting a growing conversation, GSD spawns specialized subagents with *fresh context windows* for each task. The conversation becomes disposable; the project state (specs, plans, research notes, validation contracts) persists in a `.planning/` directory. Each completed task produces an atomic git commit, enabling `git bisect` when agent regressions are nonlocal.
- *Verify phase*—GSD includes what it calls a *Nyquist validation layer*: before code is written, test coverage is mapped to each requirement, ensuring that there's a feedback loop *before* the agent starts coding. After execution, a verification step checks the output against the plan.

This is a fundamentally different answer to context rot than what compaction offers. Where compaction says “Compress the old to make room for the new,” GSD says “Don’t compress: reset, instead.” The project’s accumulated intelligence lives in files, not in conversation history.

SUPERPOWERS: COMPOSABLE SKILLS AS ON-DEMAND CONTEXT

Where GSD focuses on workflow orchestration, Jesse Vincent’s Superpowers framework attacks the problem from a different angle: *skills as on-demand context modules*. Superpowers has grown even faster than GSD—its main GitHub repository is above 116,000 stars as of March 2026—and its importance is the same: it operationalizes discipline by turning skills into selectively loaded, reusable context.

The core philosophy is deceptively simple: if your agent is smart but undisciplined, give it discipline. Skills in Superpowers are not suggestions but a mandatory process (see figure 2.11). When the agent detects that it’s building something, it doesn’t rush to write code. Instead, it follows a *brainstorm* → *plan* → *implement* workflow, enforced by skill files.

```
> /superpowers:brainstorming
• Using the brainstorming skill to help design your next idea.
  Let me start by exploring the project context.
* Exploring project context...
  └─ ■ Explore project context
     □ Ask clarifying questions
     □ Propose 2-3 approaches
     □ Present design
     □ Write design doc
     □ Spec self-review
     □ User reviews written spec
     □ Transition to implementation
```

```
> █
1 /gsd:update | Opus 4.6 (1M context) | manning_vibe_coding █ 4%
```

Figure 2.11 Superpowers flow taking the user through the predefined workflow

For context engineering, three aspects of Superpowers are particularly relevant:

- *Skills as selective context loading*—Instead of loading everything into context up front, each skill (brainstorming, TDD, systematic-debugging, code-review, etc.) is loaded only when the agent determines it’s relevant. The agent runs a search script to find applicable skills, reads the relevant SKILL.md files, and then follows the instructions. This is a practical implementation of the selective context loading problem we’ve been discussing: context is assembled *on demand*, not shoved in at the start.
- *Subagent-driven development*—Once a plan is approved, Superpowers launches subagents for each engineering task. Each subagent gets its own fresh context

window, works through the task, and returns only its final output. The intermediate noise—file reads, search results, reasoning traces—stays isolated. Every piece of subagent work goes through a two-stage review: spec compliance first, then code quality. Critical issues block progress.

- *Self-improving skills through memory extraction*—Perhaps the most innovative aspect: Superpowers can mine previous conversation transcripts for lessons learned and cluster them into new skills. Ephemeral context—the kind that normally dies when a session ends—gets turned into persistent, reusable knowledge. Vincent discovered that classical persuasion principles (authority, commitment, social proof) work on LLMs just as they work on humans, and Superpowers use these principles not to jailbreak the model but to make it more reliable and disciplined.

THE PATTERN BEHIND THE FRAMEWORKS

GSD and Superpowers take different approaches, but they point toward the same architectural insight: the most powerful form of context engineering happens outside the context window. Both frameworks externalize project state into files that persist across sessions. Both spawn fresh subagents per task rather than trying to maintain a single growing conversation. Both treat the context window as a *working register* that's loaded with exactly what's needed for the current operation and then cleared, rather than as a memory bank that accumulates everything from the project's history.

This pattern—which we might call *state externalization*—connects directly to what we've been calling context fabric throughout this chapter. GSD's .planning/ directory, Superpowers' skill files, CLAUDE.md, AGENTS.md: taken together, these are the practical manifestation of a structured layer of specs, plans, conventions, and knowledge that lives outside any single agent session, giving every fresh context window immediate access to the accumulated intelligence of the project.

The tradeoff, naturally, is complexity. These frameworks add orchestration overhead, consume more tokens (multi-agent workflows use roughly four to seven times more tokens than single-agent sessions), and introduce new failure modes: a stuck subagent, a corrupted plan file, a verification step that loops. For a simple bug fix or a quick refactoring, they're massive overkill. But for anything nontrivial, like a multiday feature, a large refactoring, or a greenfield project, they represent the current state of the art in making AI agents reliably ship production code.

As one widely shared comment on LinkedIn put it, “The biggest problem with AI coding is not intelligence. It is discipline.” Meta-prompting frameworks are the discipline layer.

2.5.5 Does coding AI let me get away with being lousy?

The answer is both yes and no (as you probably expected), and it gets to the very heart of how a developer's role evolves when working with advanced AI. On one hand, no, you don't have to do the tedious, manual context engineering anymore (lucky us!). AI

agents are, in essence, sophisticated context engineering engines. They are designed to automate the laborious tasks we've been discussing:

- *Automated retrieval*—Instead of you manually searching for every relevant file, the agent uses RAG to query a vector database of your codebase and pull in what it determines are the most relevant snippets.
- *Automated reordering*—Intelligent agents automatically reorder the retrieved context to place the most critical information at the beginning and end of the prompt, mitigating the lost-in-the-middle problem without you having to think about it.
- *Memory management*—Agents are designed to handle conversation history, summarizing and compressing it to maintain state over long interactions, solving the “amnesiac coder” problem.

In this sense, the agent becomes your personal context engineer, freeing you from the low-level, manual labor of assembling prompts. On the other hand, yes, context engineering is more important than ever, but your role shifts from a technician to a strategist. The agent automates the “how,” but you are still responsible for the “what” and “why.” Here's what that means in practice:

- *The “garbage in, garbage out” principle still reigns supreme.* An agent's RAG system is only as good as the knowledge base it searches. If your codebase is poorly documented, contains contradictory information, or is filled with outdated, insecure code examples, the agent will dutifully retrieve that “garbage” and use it to generate flawed solutions. This is a form of context poisoning. Your job as a context engineer evolves from curating individual prompts to curating the entire project's knowledge base. You must ensure that the source material—the documentation, code conventions, and examples—is clean, consistent, and high-quality.
- *You are still the navigator.* The agent can retrieve information, but you must provide the initial, high-level direction, which is especially useful when you are not doing bug fixes but rather are delivering whole new functionalities. A vague, imprecise prompt will send the agent on a wild goose chase, even with perfect retrieval pipeline. Your prompts become more strategic, focusing on defining the goal, constraints, and acceptance criteria rather than listing file paths.
- *You become the debugger of the agent's context.* When an agent fails, it's often due to a context failure. It might retrieve the wrong document, get distracted by irrelevant information, or misinterpret a key piece of context. To fix this, you need to understand the principles of context engineering. You might need to intervene and say, “No, don't look at that V2 documentation, focus on the V3 migration guide.” This is an *interactive form of context engineering* where you are supervising and correcting the agent's own context-building process.
- *You define the role and personality.* The most powerful form of context engineering for agents is architecting the *agent manifest*. This is where you define the agent's

core identity: its role (“You are a senior security engineer”), its rules (“Always write a unit test before the implementation”), and its persona. This high-level, persistent context governs all the agent’s subsequent actions and is a task that remains entirely in the developer’s hands.

In short, agents don’t eliminate the need for context engineering; they absorb the manual labor and ask you to level up. You stop being a librarian who manually fetches every book and become the research director who ensures that the library is well-organized and tells the librarian what kind of information to find. The principles are the same, but the application is more strategic. (No free lunches, sorry.)

We’ll return to the topic of agentic programming—how to build identity, craft an agentic manifesto, and teach agents to use tools and set them up—in chapter 3. For now, there’s one more dimension of context engineering we haven’t addressed: what happens when the context itself is flawed, and the model needs to reason its way through contradictions?

2.6 Context through reasoning

You’ve followed us as we’ve described different, increasingly automated techniques for building the perfect context for models. So it pains us to say this, but the pursuit of a perfect context for our model is ultimately an illusion. Even well-designed, mature codebases are living artifacts subject to entropy.

If we had to pick our all-time favorite technical paper, it would probably be “Software Aging” by David Lorge Parnas, which explores the many reasons systems inevitably deteriorate over time. Our favorite passage goes like this:

The authors and owners of new software products often look at aging software with disdain. They believe that, if the product had been designed using today’s techniques, it wouldn’t be causing problems. Such remarks remind me of a young jogger scoffing at an 86-year-old man (who, unknown to the jogger, was a champion swimmer into his 50’s) and saying that he should have had more exercise in his youth.

The paper is so good because it outlines several causes of this aging, and its conclusions are sobering. One reason systems age is that *they don’t change*: ecosystems evolve, but some core parts remain frozen in time, unable to keep pace with new standards. If that were the only factor, we might manage. But systems also age *because they do change*: original developers leave, conceptual clarity fades, requirements evolve, and technical debt piles up. Over time, the combination of many hands, shifting goals, and incomplete migrations inevitably creates pockets of inconsistency.

This reality poses a serious cognitive challenge for AI agents, which rely on consistent and accurate context to generate reliable code. When the provided context is flawed, the agent’s performance drops, leading to clear failure modes that undermine its usefulness.

Two primary failures arise directly from this environmental inconsistency. The first is *Conflicting signals*, also known as *context clash and confusion*, which occurs when the AI agent receives contradictory, ambiguous, or inconsistently presented instructions within its context. Discrepancies between the phrasing of a prompt, embedded documentation, and in-file comments can lead to misinterpretation of priorities, adherence to outdated guidance, or the mixing of incompatible approaches. This confusion often results in code that partially satisfies multiple instructions but fully aligns with none, creating inconsistencies and requiring manual clarification.

The second, even more hideous failure mode is *context poisoning*, which emerges when the context contains flawed, outdated, or unsafe code examples. The model, treating these examples as authoritative, will faithfully replicate mistakes, propagating bad practices, inefficient patterns, or deprecated APIs into new code.


The evolution of challenges in context engineering has moved from context scarcity—*context vacuum* and *context overload*—ones we can handle—to a more subtle issue: *context corruption*. This means the solution is no longer purely architectural, such as better retrieval algorithms or larger context windows, but cognitive. To address these challenges, coding models must go beyond simply consuming context and develop the ability to *critique* it. This shift requires embracing more advanced, deliberative reasoning paradigms that allow the agent to act as an active participant in validating the information it processes. The techniques that follow represent a layered defense against faulty context, forming an integrated cognitive architecture for deliberation.

2.6.1 **Chain of thought: Forcing the LLM to show its work**

Chain-of-thought (CoT) prompting is a fundamental step toward deliberative reasoning. It is a strategy that compels a model to articulate its reasoning process step by step before providing a final answer. Instead of making an intuitive leap directly from query to solution, the model is instructed to externalize its thought process. By generating these intermediate steps, the model creates an additional layer of internal context that guides its subsequent token generation, increasing the likelihood of reaching the correct conclusion—especially for complex tasks that require decomposition.

It is similar to context anchoring, but instead of maintaining a static to-do list that preserves focus, CoT turns the plan itself into a *live reasoning trace*: an evolving scratchpad where the model not only recalls objectives but also justifies and adapts its decisions step by step, making the workflow more transparent, editable, and aligned with how real developers iteratively think through problems.

In the context of coding, this transforms a simple request into a structured analytical task. Consider the difference between a standard prompt and one based on CoT. A standard prompt results in direct coding action:

 Refactor the `process_payment` function to use the new StripeV3 client.

A model based on CoT (often forced through the tool’s system prompt) will respond as shown here for GPT-5:



I will:

- * Analyze the logic of the current function and its dependencies.
- * Identify specific API calls to the old client.
- * Search the provided context for examples of StripeV3 usage.
- * Describe in detail the necessary changes in error handling and data structures.
- * Finally, write the refactored code based on this plan.

Do you want to change anything in that process?

This approach directly mitigates the problem of *conflicting signals*. Faced with contradictory information, the CoT process forces the agent to explicitly state its interpretation and chosen path. For example, its reasoning might include a step such as



Step 3: I identified a comment in the old code referencing v1, but the project’s package.json specifies StripeV3 as a dependency.

I will proceed with v3, since the dependency definition is a more reliable source of truth than a potentially outdated code comment. Do you want to carry on with the plan?

Although CoT improves accuracy, its main value in professional software development workflows lies in increasing *auditability and trust*. LLMs are nondeterministic and may produce plausible but incorrect code. When a developer receives a block of code without explanation, verifying its logical correctness requires manually reconstructing the AI’s implicit thought process: a time-consuming and error-prone task. This creates a significant barrier to trust. By revealing its reasoning, the agent provides the equivalent of a detailed commit message for its changes, explaining the “why” behind them. The developer can quickly scan this reasoning to catch faulty assumptions or misinterpretations before even reviewing the code itself.

This radically reduces the cognitive load of reviewing AI-generated code and increases developer confidence in accepting suggestions. In this light, CoT functions both as a human–computer interaction (HCI) pattern that builds trust and as an LLM reasoning technique. Some tools have this already built in in their user interface.

Take Cline as an example. Instead of treating a to-do list as a static anchor, Cline divides its workflow explicitly into *Plan* and *Act* phases. During the Plan phase (see Figure 2.12), the agent drafts a concrete sequence of steps.

Then, before execution, it pauses to validate those steps: sometimes by asking clarifying questions, other times by rewriting them in more detail. The Act phase then executes the plan, but even here Cline may stop and refine its approach before continuing (Figure 2.13). This alternating rhythm of planning and acting feels much closer to how developers work in reality: a cycle of intent, reflection, and adjustment.

Task \$0.0861

Fix deprecated `painterResource` (Compose)

- File: knes-compose-ui/src/main/kotlin/knes/compose/ComposeMain.kt
- Warning: 'fun painterResource(resourcePath: String): Painter' is deprecated.
- Explanation: The painterResource function is deprecated. The warning message provides a link to the new Compose resources library. This requires updating the code to use the new API for loading resources.

Tokens: ↑ 31.9k ↓ 2.1k
Cache: → 73.6k 93.9 kB

24.8k 1.0m

✓ **API Request** \$0.0250

Checkpoint Compare - Restore

Okay, I understand the task. The `painterResource` function in Jetpack Compose is deprecated, and I need to update the code to use the new resources library.

First, I'll examine the file where the deprecated function is being used to understand its context. Then I'll check the `build.gradle` file to see what dependencies are already included and determine if a new one is needed for the updated resources library. Finally, I'll propose the necessary code changes.

I'll start by reading `knes-compose-ui/src/main/kotlin/knes/compose/ComposeMain.kt`.

Cline wants to read this file:

`knes-compose-ui/src/main/kotlin/knes/compose/ComposeMain.kt`

✓ **API Request** \$0.0113

I've reviewed the code in `ComposeMain.kt`. I can see two usages of the deprecated `painterResource` function:

```
painter = painterResource(Res.drawable.frame),
```

and

```
painter = painterResource(Res.drawable.logo),
```

The import statement `import org.jetbrains.compose.resources.painterResource` confirms this is the one. The warning message mentions a new Compose resources library. I need to check the `build.gradle` file to see if the new library needs to be added as a dependency.

Now I'll read `knes-compose-ui/build.gradle`.

Figure 2.12 Cline drafting a plan of execution

Here's my plan to fix the issue:

1. **Update** `knes-compose-ui/src/main/kotlin/knes/compose/ComposeMain.kt`:
 - I'll remove the direct imports for `frame` and `logo` resources.
 - I'll remove the `@OptIn(InternalResourceApi::class)` annotation, which is likely no longer necessary.
 - The calls to `painterResource` will be updated to use the type-safe resource accessors from the generated `Res` class.

This should resolve the deprecation warning and align the code with the modern Compose resources API.

Are you happy with this plan? If so, please toggle to **Act Mode** (`☑️ A`) so I can apply the changes.

Figure 2.13 Cline sharing its plan for achieving the goal, waiting for approval from the human in the loop

Even more interesting, these tools allow different models to be used for each phase. A reasoning-focused model can handle planning, and a model that's faster, better at

coding, or cheaper (depending on priorities) handles execution. This modularity is key: the act of building software is not one monolithic task but a set of subtasks, each benefitting from a different balance of reasoning and speed. Tools like RooCode take this further by introducing multiple explicit modes—exploration, refactoring, testing—each with its own workflow logic.

Let's circle back for a moment to the planning phase. There is one more interesting technique to look at.

2.6.2 Chain-of-verification: Internal fact-checking loop

Chain-of-verification (CoVe) can be seen as a natural evolution of CoT, but it shifts the model's role from that of a pure academic theorist to something closer to an experimental practitioner. Left to its own devices, the model often behaves like a theoretical physicist, capable of producing a beautifully reasoned argument that is full of structure and elegance but will collapse if its initial assumptions are wrong.

What's missing is the pragmatism of someone who not only writes theory but also tests it in practice: someone who runs experiments, checks sources, and validates claims against reality. To achieve good results, we need both. CoVe (Figure 2.14) introduces this missing ingredient by embedding a fact-checking loop inside the reasoning process, giving the model tools to interrogate its own conclusions.

The CoVe workflow unfolds in several stages. First, the model generates a baseline answer; this can be a direct draft or one already structured using CoT. Instead of accepting that baseline as final, the model creates a series of verification questions to challenge the assumptions behind its own output. The model then answers these verification questions separately, in isolation from the original draft, so that it cannot simply echo or rationalize its earlier mistakes.

Cline wants to read this file:

knes-compose-ui/build.gradle

Checkpoint [Compare] [Restore]

✓ **API Request** \$0.0205

I've examined the `build.gradle` file. It seems the project is already using the new resources library, as indicated by the `org.jetbrains.compose` plugin and the `compose.components.resources` dependency. The `compose.resources` block also suggests that the resource generation is set up.

The issue is not a missing dependency, but rather the way the `painterResource` is being called. The new resources library uses a different approach. Instead of passing a string path, it uses a type-safe resource accessor.

The code already has `import knes.knes_compose_ui.generated.resources.Res`, which is the generated resource class. The calls are `painterResource(Res.drawable.frame)` and `painterResource(Res.drawable.logo)`.

The problem is that `painterResource` from `org.jetbrains.compose.resources` now expects a `Resource` object, not a `String`. The generated `Res` class provides these objects.

The fix is to change the import from `org.jetbrains.compose.resources.painterResource` to a different one that works with the new resource types. The new `painterResource` is in a different package.

Figure 2.14 Cline CoVe sharing the work done with the human operator

This paradigm is especially powerful when dealing with context poisoning. Consider an agent asked to implement a new data access function. If its retrieval system happens to surface an outdated code snippet that uses unsafe SQL patterns, a CoT approach alone might produce a well-structured plan and syntactically correct code—but faithfully reproduce the insecure practice. With CoVe, however, the model first drafts that flawed plan and then explicitly formulates a verification question such as



Is `mysql_real_escape_string` currently recommended for preventing SQL injection?

By consulting newer parts of the codebase, its own knowledge of secure practices or one gained through MCP, the model might answer, “No, the modern, safe standard is to use prepared statements with parameterized queries.” Armed with this correction, it discards the poisoned example and regenerates the function using the secure pattern.

CoVe operationalizes the scientific *principle of falsification* in the model’s reasoning loop. Where CoT builds arguments to support an answer, CoVe forces the model to attempt to disprove them. The baseline answer becomes a hypothesis, and the verification steps become attempts to falsify it. The final, corrected answer, having survived this internal skepticism, is far more robust and dependable. In practice, this gives the agent a form of *epistemic humility*: it does not blindly trust its first line of reasoning, but instead treats it as a candidate that must be tested. This attitude is crucial when working with large and messy information sources, such as sprawling enterprise codebases full of legacy artifacts. By embedding self-checks of this kind, agents become better partners: less likely to propagate outdated practices and more likely to deliver results developers can trust. But to unlock this mode of operation, the model must be given tools that ground its reasoning in objective outcomes.

2.6.3 *How to introduce self correction*

We first need to ask ourselves what *verification* means in practice. One popular solution is the so-called *LLM-as-a-judge* approach. Here, the model is tasked not only with generating output but also with evaluating its own work against the prompt or some criteria. On paper, this approach sounds elegant: why not let the same reasoning engine that produced an answer also grade it? In practice, however, it comes with tradeoffs. The model is still bounded by its own knowledge and biases: if it misjudges a requirement in the generation phase, it is likely to repeat the same misjudgment during evaluation. Instead of catching errors, LLM-as-a-judge often reinforces them, producing confident but flawed justifications. This is why purely internal loops of self-correction can fail; the “professor” ends up grading their own exam with the same blind spots they had while writing it.

The real differentiator, then, is not whether a system has CoT, CoVe, or self-refinement hooks in theory, but how well its feedback loops are constructed in practice. Projects with strong, high-quality feedback channels will allow agents to thrive; those with weak or absent ones will leave agents stumbling. And here lies the simple truth: if your codebase already works well for developers, it will probably work well for agents, too.

A system in which AI flourishes looks almost indistinguishable from a system built for a high-performing engineering team. It has good test coverage with meaningful assertions, automated linting, formatting, and test checks before merges, and continuous integration pipelines that provide immediate signals when something breaks. It carries well-documented changes, ADRs, and commit messages that clarify intent, and it enforces consistent styles and patterns so that generated code has a clear target to align with.

In other words, the very things that make a codebase pleasant and efficient for human engineers—clarity, consistency, modularity, rich test harnesses—also make it fertile ground for agents. As you can see, feedback loops aren't an optional extra; they are the infrastructure that turns AI assistance from a parlor trick into a dependable collaborator. Without them, even the most advanced reasoning strategies collapse into noise. With them, AI can help with real engineering acceleration, removing organizational friction and letting teams ship faster with confidence.

2.6.4 *Is reasoning always the solution?*

Before we wrap up, we want to share a concrete example that demonstrates how careful we still need to be, even when working with beautifully designed theoretical solutions like CoVe or self-refinement loops.

Last year, I (Artur) spent some time writing an NES emulator. At one point, I hit a very famous bug: Super Mario Bros was loading into the wrong world: *World -1*. I thought this would be a great opportunity to see what an AI coding agent could do, so I turned to Cline. The agent dutifully produced several elaborate hypotheses about what could be going wrong: maybe an off-by-one error in memory addressing, maybe a mishandled pointer, maybe corrupted ROM metadata. It reasoned, experimented, burned through tokens, and eventually got stuck looping in its own theories without finding the fix.

Frustrated, I tried something radically simpler. I wrote a one-line prompt describing the bug and attached a single screenshot to the chat interface. Within seconds, the model returned the exact explanation: this wasn't a mysterious corner case at all, but the well-known Minus World glitch. The cause was nothing exotic: it's a classic quirk in how the game initializes its initial state, and it's so widely documented that the model recognized it from prior knowledge and surfaced the answer immediately. No need for reasoning chains, verification loops, or complex exploration—just a direct retrieval of existing knowledge that turned out to be spot on.

This experience was a humbling reminder. Sometimes elaborate reasoning itself is a tradeoff; it can burn tokens and go in circles, whereas the simplest prompt taps into well-established knowledge already in the model. This experience showed me that working with LLMs is still very much an art, not a science. In the end, it's less about discovering the perfect prompting technique and more about cultivating the judgment to know when to trust retrieval, when to demand reasoning, and when to simply step aside and let the model surprise you.

Summary

- Insufficient or imprecise context, not errors of the model, is a primary cause of AI coding tool failures.
- Context engineering is an evolution of prompt engineering. It focuses on strategically selecting, structuring, and labeling information so the model can interpret it correctly.
- Automated tools using protocols like MCP allow agents to independently retrieve current and precise data from various sources such as Figma, code repositories, and documentation systems.
- Providing more context does not always lead to better results. Language models suffer from the lost-in-the-middle problem.
- AI coding tools automate context engineering with techniques like RAG, context anchors, and context compaction to manage conversation history.
- AI tools do not eliminate the need for context engineering but instead shift the developer's role from technician to strategist.
- To handle inconsistent or flawed context, agents must use advanced reasoning techniques. Chain-of-thought compels the model to show its reasoning process, and chain-of-verification introduces a fact-checking loop, enabling a critical evaluation of the information it processes.

Continuous AI development with AI-native SDLC

This chapter covers

- Progressing from autocomplete to agents
- Managing approval fatigue effectively
- Running agents outside the IDE
- Building team standards with AGENTS.md
- Exploring agent orchestration tradeoffs

An AI agent is only as autonomous as the leash you're willing to give it. We've explored context engineering: how to feed models the right information at the right time. But context is just fuel. The question remains, what kind of engine are we building?

To answer that question, we'll move from passive autocomplete, through agents that can see and touch your codebase, to headless automation in CI pipelines, and finally toward orchestrated fleets of specialized agents. At each step, AI moves further away from a coding aid for one developer and deeper into the *software development lifecycle* (SDLC). Each step solves the limitations of the previous one. Each step introduces new tradeoffs we didn't have before.

Just as CI/CD transformed software development by automating integration and deployment, AI assistance is undergoing the same transformation. A growing number of teams and vendors now use terms like *continuous AI* to describe event-triggered, automated AI support across the SDLC:

- Continuous code review that examines every pull request before a human reviewer arrives
- Continuous security scanning that catches leaked secrets and hallucinated packages at commit time
- Continuous documentation that keeps docs in sync with code
- Continuous performance monitoring to confirm that your application is on par with all the nonfunctional requirements
- Continuous fault analysis that explains failed CI runs

The pattern is consistent: AI as a persistent, event-triggered participant in the SDLC, not just a tool you invoke when stuck.


But continuous doesn't mean instant. The progression through these stages is not optional. Teams that jump straight to autonomous agents without mastering collaboration tactics end up with expensive chaos. Teams that stay too long at the autocompletable level leave massive productivity gains on the table. The art is in knowing when you're ready for the next step—and what infrastructure, processes, and trust you need to build before taking it.

There are no shortcuts here. Going from “Copilot suggests a line” to “agent fleet ships a feature” requires more than just better prompts. Vibe engineering scales differently at each stage; what works as a solo technique in a chat window becomes a coordination problem in a team and an orchestration challenge across a fleet of agents. In practice, this progression is often measured in months of organizational learning, not days of tool installation.

3.1 *Passive AI: ChatGPT as a coding partner*

Let's start where most developers actually started their adventure with AI: not with sophisticated IDE integrations, but with a browser tab. It's late 2022. Alice is staring at a ticket that's been sitting in her backlog for weeks: migrate the payment processing module from the legacy XML-based API to the new REST endpoint. The old code is a mess of nested callbacks, undocumented edge cases, and comments in three different languages left by developers who have long since moved on.

She's heard colleagues talking about this new thing called ChatGPT. On a whim, she opens a browser tab and pastes a chunk of the legacy code:

 Here's our payment processing code. I need to migrate it to use REST instead of XML. Can you help?

[150 lines of tangled JavaScript]

What comes back surprises her. The model doesn't just translate syntax: it recognizes patterns, suggests a cleaner structure, even points out a potential race condition she hadn't noticed. Within an hour, she has a working first draft that would have taken her a full day to write from scratch.

Alice discovered something that millions of developers were discovering at the same time: LLMs can code. Not perfectly, not autonomously, but usefully. The browser-based workflow is clunky—copy code out, paste response back, manually merge changes—but the productivity boost is undeniable.

This is Era Zero of AI-assisted development. No IDE integration, no context awareness, no memory between sessions. Just a human playing the role of a very inefficient clipboard, shuttling text between their editor and a chat window. But even as Alice and millions like her discovered this new superpower, the friction was obvious.

3.1.1 **GitHub Copilot: AI moves into the IDE**

Every copy-paste cycle was a context switch, a flow break, a cognitive tax. The question wasn't whether AI could help with code; it was how to reduce the distance between the AI and the work. The model never sees your full file, let alone your project structure. You waste tokens re-explaining the same codebase in every conversation.

GitHub Copilot solved the most painful part of this friction. Instead of copying code to a browser, the AI comes to you. It watches what you type and offers completions inline, in real-time, right where you're working.

The experience feels almost magical at first. You start typing a function signature:

```
def validate_payment_amount(amount: Decimal, currency: str) -> bool:
```

And Copilot completes the entire implementation—null checks, currency validation, range boundaries—before you've written a single line of logic. Press Tab to accept. Move on.

For boilerplate-heavy tasks, the acceleration is dramatic. Writing database migrations, API endpoint handlers, test scaffolding, configuration parsers: the repetitive patterns that eat hours of developer time suddenly flow at the speed of reading rather than writing.

But Copilot's architecture reveals its limitations. It operates on a simple principle: given the code before your cursor (and sometimes the code after), predict what comes next. It's autocomplete on steroids—impressively capable steroids, but autocomplete nonetheless. It doesn't understand that `PaymentService` in line 47 refers to a class defined in `services/payment.py` or that the team's coding standards require dependency injection rather than direct instantiation. The suggestion looks syntactically correct but architecturally wrong, adding toil to future maintenance.

Each suggestion exists in isolation. The context window is your current file—perhaps with a few neighboring tabs if the tool is feeling generous—but the broader architecture of your system remains invisible.

3.1.2 *Tab-Tab-Tab: The illusion of productivity*

Here's where the first subtle trap emerges. Alice, our fintech developer, has been using Copilot for three months now. Her velocity metrics look fantastic: more commits, more lines of code, faster ticket closure. In sprint reviews, she demos features that used to take a week in half the time.

But something is shifting in how she works. She's started to notice that she reads code more than she writes it. Copilot suggests, she evaluates, she accepts or rejects. The ratio of thinking to typing has inverted. Where she once spent 80% of her time reasoning about architecture and 20% typing, now she spends 80% of her time reviewing AI suggestions and 20% steering.

This isn't inherently bad; code review is a legitimate skill, and evaluating suggestions is faster than writing from scratch. The trap is more subtle: *acceptance without understanding*.

It happens gradually. A Copilot suggestion looks reasonable. The tests pass. Ship it. Three weeks later, during an incident, Alice is debugging code she nominally wrote. The AI had generated a retry mechanism with exponential backoff: sensible in isolation, but it interacts poorly with the circuit breaker in the upstream service, causing cascading timeouts under load. The incident lasted four hours. The root cause wasn't obvious; Alice had to trace through three service boundaries before she found the interaction. Worse, she realized she didn't fully understand the code she was debugging. She'd accepted it, reviewed it, shipped it, but she hadn't *authored* it.

This is what we call *Copilot-driven development*: a mode where the developer becomes a reviewer of AI-generated code rather than an author of it. The human stops asking "What should this code do?" and starts asking "Does this suggestion look okay?" The creative control shifts from human to machine, with the human reduced to a yes/no filter. And that's a pretty miserable existence.

The most insidious part is that it feels productive. Lines of code flow. Features ship. The metrics all point up. But the developer's mental model of the system atrophies. Understanding is replaced by pattern-matching. "This looks like what I've accepted before" serves as a substitute for "This is correct for our specific context."

3.1.3 *The context blindness wall*

Let's make the limitation concrete. Alice's team has a strict convention: all API responses must be wrapped in a `Result` type that distinguishes between success, expected errors (like validation failures), and unexpected errors (like database timeouts). This pattern is documented in their architecture decision records. It's enforced in code review. Every senior developer knows it instinctively.

Copilot knows nothing about it. When Alice starts writing a new endpoint, Copilot helpfully suggests



```
@app.route('/payments/<payment_id>')
def get_payment(payment_id: str):
    payment = PaymentRepository.find_by_id(payment_id)
```

```

if not payment:
    return {"error": "Payment not found"}, 404
return payment.to_dict(), 200

```

This is perfectly reasonable code. Well, mostly. It follows common REST conventions (although a type purist might note the missing return type annotation). For a generic codebase, this would be fine. And that’s precisely the mistake, as it’s completely wrong for Alice’s codebase, where the correct pattern is

```

🤖 @app.route('/payments/<payment_id>')
def get_payment(payment_id: str) -> Result[PaymentResponse]:
    payment = PaymentRepository.find_by_id(payment_id)
    if not payment:
        return Result.expected_error(ErrorCode.NOT_FOUND,
            ↳ "Payment not found")
    return Result.success(PaymentResponse.from_domain(payment))

```

The difference isn’t just in style but also in semantics. The `Result` type carries information that downstream systems depend on. The generic pattern silently breaks contracts that aren’t visible in the current file.

This is *context blindness*: the inability of autocomplete-based tools to see beyond the immediate text window. The model has no access to architectural decision records (ADRs). These project conventions exist as tribal knowledge, type relationships defined in other modules, the runtime behavior of systems you integrate with, or the historical context of why certain patterns were chosen. The suggestions are syntactically valid and semantically plausible, but architecturally . . . foreign. The code compiles. The tests pass. But the system slowly drifts from coherent design toward a patchwork of locally reasonable but globally inconsistent patterns.

This era of passive AI—tools that work without deep project context—offers a genuine bargain. You trade deep, effortful authorship for fast, shallow review. For many tasks (boilerplate, well-understood patterns, exploratory prototyping), this is a good trade. For example, in prototyping, learning new frameworks, or investigating unfamiliar APIs, Copilot’s context blindness becomes an asset. You want generic patterns, not project-specific conventions. The tool shines when you need to move fast through unfamiliar territory.

But the trade has hidden costs. Developers who accept without understanding gradually lose the ability to write from scratch; the mental muscles that once generated solutions now only evaluate them. Each accepted suggestion brings its own micro-conventions, and without active resistance, codebases drift toward inconsistency. Most insidiously, the code that looks right but isn’t accumulates as technical debt that doesn’t show up until much later: the wrong abstraction, the missed edge case, the subtly incorrect error handling, all accepted with a confident Tab.

The bargain works best when treated as what it is: autocomplete. A tool that handles the mechanical parts of typing while you retain full ownership of design, architecture, and understanding. It breaks down when the tool becomes the author and you become the rubber stamp. And who wants to be the rubber stamp?

The problem wasn't that Copilot was bad at autocomplete. The problem was that autocomplete was the wrong metaphor. What developers needed wasn't a faster typewriter, but a collaborator who could actually see the project they were working on and eventually participate in more than the single moment of typing code.

If an AI could be trained on billions of lines of public code, why couldn't it read the few hundred thousand lines in *your* repository? The question wasn't theoretical anymore. If AI could generate plausible function implementations, why couldn't it follow the trail of imports to understand your actual architecture? If it could suggest code, why couldn't it *write* code: not as a suggestion floating in gray text, but as an actual edit to an actual file?

By late 2023, that question had moved from “what if?” to “what now?” A new generation of tools was already answering it.

3.2 *Agents in IDE, but still on a leash*

The tools that emerged throughout 2023 and into 2024 shared a simple but radical premise: stop treating the AI like autocomplete and start treating it like an assistant with actual capabilities. Give it permission to read your files. Let it explore your project structure. Allow it to propose not just code snippets, but actual edits to multiple files at once.

Cursor, Windsurf, Zed AI, Continue, and Cody each took a different path to the same destination. The AI was no longer confined to predicting the next token in your current file. It could see your imports, trace your dependencies, read your tests, and examine your configuration. The context window expanded from a single file to an entire project.


The idea seems obvious in retrospect. Instead of the model guessing what `PaymentService` might look like based on statistical patterns, let it read `services/payment.py`. Instead of inferring your project structure from local clues, let it explore the directory tree. This shift from “autocomplete that guesses” to “agent that explores” represents the most significant evolution in AI coding tools since Copilot's launch. But it came with a catch that nobody fully anticipated: the more capable the agent, the more supervision it demands.

3.2.1 *The Cambrian explosion of AI coding agents*


By 2024, a wave of AI-first tools—Cursor, Cline, Roo Code, Windsurf, Aider, and others—had arrived, each sharing the same premise: give the AI actual access to your project. However, they shared more than ambition: they shared a dependency. The Model Context Protocol (MCP) we explored in chapter 2 provides a universal interface for connecting to external systems, such as filesystems, documentation, databases, and APIs. MCP is what turned “agent that can read files” from a feature each tool had to build independently into a standard any tool could adopt. And for Alice, it's what finally made the context blindness wall navigable: her agent no longer guesses at the result type convention. It reads the actual definition, searches for existing usages, and generates code that fits.

What changed wasn't just the availability of tools. It was an interaction contract. These agents don't live inside your editor as suggestions; they live alongside it as actors. And once an actor can touch your files and run your commands, the system needs a leash.

Alice installs Cline into her VS Code. The first thing she notices is the panel that appears alongside her editor: not a chat window exactly, but something more like a task control center. She types a request:

 Migrate the payment validation in PaymentService to use the new currency validation library we added last week.


What happens next is visibly different from Copilot. Instead of immediately suggesting code, Cline announces a plan:

 Response (CLINE)


I'll help you migrate the payment validation. Let me first understand the current implementation and the new library.

- 1 1. Reading PaymentService.py to understand current validation logic
- 2 2. Searching for the new currency validation library
- 3 3. Checking how the library is used elsewhere in the codebase
- 4 4. Proposing migration changes


The plan reads like a checklist; the execution reads like a negotiation. Then it starts executing, and here's where the "leash" becomes tangible. A popup appears:

 I'll help you migrate the payment validation. Let me first understand what is happening in the codebase
Cline wants to read file: src/services/PaymentService.py
[Allow] [Deny]

She clicks Allow. Another popup:

 Cline wants to search codebase for: "currency validation"
[Allow] [Deny]

Allow. Another:

 Cline wants to read file: src/utils/currency.py
[Allow] [Deny]

Allow. And so on.

This is the *Plan-Act-Observe* loop that defines modern AI coding agents. The agent forms a plan, takes an action, observes the result, and adjusts. Each action is explicit and requires approval. The human remains in control: the agent cannot read, write, or execute anything without permission.

Cline and similar tools add another layer of organization: *modes*. Think of these as role-based access control for AI behavior. In Architect mode, the agent can read files and propose high-level designs but cannot edit code. In Code mode, it can edit, but within constrained boundaries. In Ask mode, it can only answer questions about what it observes.

Roo Code pushes this further with specialized personas—a planning agent, an implementation agent, a testing agent—each with different capabilities and constraints. The separation of concerns we apply to human teams starts to appear in how we structure agent permissions.

The tools even allow different models for different modes. A reasoning-heavy model like Claude can handle planning, and a faster, cheaper model handles routine code generation. The agent’s “brain” becomes modular, with different components optimized for different tasks.

The old mental model was “AI writes code while I type.” The new one is “AI proposes a plan, I approve it, and then AI executes it.”

Plan mode explicitly separates the “what should we do” from the “let’s do it.” You describe a task, the tool generates a structured plan with specific steps, you review and adjust the plan, and only then does execution begin. It normalizes the habit of thinking before coding—or rather, having the AI think visibly before coding.

This matters more than it might seem. When suggestions appeared inline and you hit Tab to accept, the decision was instantaneous and often subconscious. When a plan appears as a document you must read and approve, the decision becomes deliberate. The friction is intentional: it’s a forcing function for understanding.

Alice, now six months into using these tools, has developed new instincts. She no longer asks, “Write me a function that does X.” She asks, “Analyze the payment module and propose how we should add retry logic for failed transactions.” The response is a plan she can critique, adjust, and ultimately own, even though the AI did the initial thinking.

3.2.2 *Approval fatigue*

But there’s a problem with all those approval dialogs. Let’s count what happens when Alice asks Cline to “refactor the payment validation module for better testability”:

- 1 *Read* PaymentService.py—*Allow*
- 2 *Read* PaymentRepository.py—*Allow*
- 3 *List* test directory—*Allow*
- 4 *Read* test_payment_service.py—*Allow*
- 5 *Search* for “dependency injection” patterns—*Allow*
- 6 *Read* BaseService.py (found in search)—*Allow*
- 7 *Read* container.py (dependency injection config)—*Allow*
- 8 *Write* PaymentService.py (proposed changes)—*Allow*
- 9 *Write* test_payment_service.py (new tests)—*Allow*
- 10 *Run* pytest—*Allow*

- 11 Read pytest output—*Allow*
- 12 Write `PaymentService.py` (fix based on failing test)—*Allow*
- 13 Run pytest again—*Allow*

Thirteen approval dialogs for a single refactoring task. And this is a simple case.

By the 10th popup, something shifts in Alice’s brain. She stops reading what she’s approving. The dialog appears, her finger hits *Allow* reflexively, and she’s back to waiting for the next one. The agent is nominally supervised, but the supervision has become a rubber stamp.

This is *approval fatigue*: the degradation of human oversight that occurs when the frequency of decisions overwhelms the cognitive capacity to make them meaningfully. The mechanism that was supposed to keep humans in control becomes a theater of control: the motions without the substance. Oversight doesn’t scale linearly. The more capable the agent becomes, the more supervision it demands—until supervision itself becomes the bottleneck.

The risks are not theoretical. An agent that has learned to ask “can I read this file?” and always receive “yes” will eventually ask “can I delete this file?” at popup number 47 of a long session. The human whose attention has long since wandered clicks *Allow* out of habit. Practical mitigations exist, but none are perfect:

- *Auto-approve classes*—Most tools now allow blanket approval for categories of low-risk operations. Read-only file access, directory listings, running tests, linting: these can be preapproved, significantly reducing the dialog count. The risk is that “low-risk” is context-dependent. Reading `.env` files is low risk in most cases but catastrophic in others.
- *Tool allowlists*—Projects can define which MCP servers and operations are permitted. If your agent doesn’t need database access for this task, don’t give it database access. It’s the principle of least privilege, applied to AI. The overhead is maintaining these lists as projects evolve.
- *Batch approvals*—Instead of approving each action, approve a plan as a whole and let the agent execute multiple steps. You review the aggregate result rather than each individual operation. The tradeoff is that you lose the ability to intervene mid-execution if the agent takes a wrong turn.
- *Checkpoints*—Some tools introduce explicit pause points at which the agent must stop and wait for review, regardless of auto-approve settings. A sensible policy might be: read anything automatically, but always pause before writes and never auto-approve terminal commands.

None of these solutions eliminates the fundamental tension. Either you supervise closely and spend your time managing the agent instead of benefiting from it, or you loosen the leash and accept that your “oversight” is increasingly nominal. And approval fatigue isn’t just about wasted time, it’s about what slips through when attention fades. The most dangerous failures aren’t the ones the agent asks permission for. They’re the ones that look routine.

3.2.3 *Tradeoff: Vision vs. vigilance*

Agentic IDEs deliver on their promise. The agent can see your project—really see it, not guess at it from statistical patterns. It can read your conventions, explore your architecture, and understand your dependencies. The context blindness that plagued autocomplete is materially reduced.

But the solution comes with its own cost: *you become a supervisor*. Every capability the agent gains requires oversight. File reading requires you to approve what it reads. File writing requires you to review what it writes. Command execution requires you to validate what it runs. The agent’s power scales with your attention, and attention is finite.

The math often doesn’t work in your favor. If an agent saves you 30 minutes of coding but costs you 20 minutes of supervision, the net gain is modest. If approval fatigue leads you to rubber-stamp a destructive action, the net gain is negative. If the cognitive overhead of managing the agent prevents you from entering flow state, the productivity loss is invisible but real.

The developers who thrive in this era are the ones who develop *selective trust*: clear mental models of when to supervise closely and when to let the agent run. They configure their approval policies thoughtfully, creating friction where risk is high and removing it where risk is low. They treat the agent like a capable but junior developer: trusted with routine tasks, supervised on anything novel or destructive.

But even with perfect policies and calibrated trust, one constraint remains: such an agent works only when you’re watching. It stops when you look away. It cannot run overnight. It cannot process a backlog while you sleep. Its scale is bounded by your attention, which means its impact is still local to a developer session rather than the broader SDLC.

This limitation points toward the next evolution. What if the leash came off? What if the agent could run in a terminal, in a CI pipeline, in a Git hook—places where humans aren’t present to click Allow?

This is exactly where the tooling moved next. But as we’ll discover, an agent that works without supervision is an agent that fails without witnesses. Approval fatigue doesn’t just waste time; it creates a new class of invisible risk. When routine operations stop being reviewed carefully, anything that looks ordinary slips through. And in modern systems, few changes look more ordinary than a dependency update.

3.2.4 *Slopsquatting and package hallucinations*

Agents operate in environments where their outputs have immediate effects. A batch refactoring script can modify hundreds of files. The blast radius of a mistake is larger than in supervised contexts. One category of mistake has proven particularly dangerous: *package hallucinations*.

LLMs are trained on vast codebases that reference millions of packages across different ecosystems. When generating code, they pattern-match against this training data. Sometimes the patterns match packages that don’t exist—or more precisely, don’t exist *yet*.

This makes them vulnerable to an attack is called *slopsquatting*: researchers and attackers monitor AI coding tools, collect the names of hallucinated packages, and register those names with malicious payloads. When a developer (or an automated pipeline) installs the AI-suggested package, they end up with malware.

The numbers are sobering. Studies like Spracklen et al. (“We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs” [2024], arXiv:2406.10279) found that popular AI coding tools hallucinate package names in roughly 5% (commercial models) to more than 20% (open-source models). Researchers have successfully registered thousands of these hallucinated names across npm, PyPI, and other package registries. The attack surface is vast and largely undefended.

In an agentic context, the danger multiplies. An agent modifying `package.json` and running `npm install` can introduce a malicious dependency without any human ever reading the package name. The commit message looks innocuous (“added error handling utility”), but the utility doesn’t exist; instead, an attacker’s payload is installed. The human operator simply tabbed that commit into the repository, the same way they tabbed the last 40.

This is the dark side of implicit trust. When a senior developer recommends a package, you trust their judgment; they’ve vetted it, used it before, and know its reputation. When an AI recommends a package, it’s pattern-matching against training data that includes packages that never worked, were abandoned years ago, or never existed at all. The suggestion *looks* identical to a trusted recommendation. The confidence is the same. Only the basis for that confidence is missing.

Defenses exist, and here’s where the sword becomes a shield: the same LLM patterns that create hallucination risk can detect it. Specialized security tooling can check package provenance, validate registry existence, and flag suspicious or newly introduced dependencies before installation. But you don’t always need a dedicated tool. A simple LLM-in-the-loop can serve as a preinstall validator: pipe your dependency changes through an agent prompted to verify that each package exists in the registry, check its maintenance history, and flag names that suspiciously resemble popular packages. The CI pipeline that might blindly install a hallucinated package can also run an LLM judge asking: “Does this package actually exist? Does it have credible maintainers? Is this name a trap?”

The same automation that amplifies attacks can amplify defenses. The question is whether your shield deploys faster than the attacker’s script. And once you start relying on automation for defense, the next step is obvious: let it run without you.

3.3 Headless era: Moving into the terminal

The approval fatigue problem has an obvious escape hatch: remove the human from the loop. If constant supervision is the bottleneck, what happens when we let agents work unsupervised?

This is where continuous AI stops being a concept and becomes a practice. Remember the patterns from the chapter introduction: continuous documentation,

continuous fault analysis, continuous code improvement? This is also the point where AI stops being just a coding assistant and starts becoming SDLC infrastructure.

And escape it should. You don't want an IDE as the *home* for automation; you never did. Shell scripts, cron jobs, Git hooks, CI pipelines: this is where automation has always lived. UNIX philosophy taught us decades ago that small, composable tools piped together solve complex problems elegantly. The IDE is for interactive work, for exploration, for the human in the loop. The terminal is for processes that run whether you're watching or not.

Headless agents fit this world perfectly. They work best when the task has crisp inputs, bounded actions, and cheap validation of outputs. They excel at the repetitive, event-triggered work that humans neglect: regenerating docs after every commit, labeling and routing incoming issues, explaining failed CI runs, and enforcing code standards across thousands of files. Work that's important but not urgent, that accumulates into technical debt when ignored, that finally gets done because something tireless is paying attention.

The promise is compelling: agents that work while you sleep, that process backlogs overnight, that catch problems before code even reaches review. But let's be clear about what we're trading. An unsupervised agent is one whose failures go unnoticed until they become impossible to ignore.

3.3.1 *Gemini CLI and Claude Code: AI in the UNIX pipeline*

The UNIX philosophy—do one thing well, connect everything with text—has survived for 50 years because it solves a fundamental problem: how do you compose simple tools into complex workflows? The answer, small programs that do one thing well, connected by text streams, turns out to be remarkably adaptable. It absorbed version control, containerization, cloud infrastructure, and now AI. Figure 3.1 shows an example of a coding CLI interface in practice.

```
~/Priv/manning_vibe_coding git:(main)
gemini

> GEMINI

Tips for getting started:
1. Ask questions, edit files, or run commands.
2. Be specific for the best results.
3. /help for more information.

Using: 1 GEMINI.md file

> █ Type your message or @path/to/file

~/Priv/manning_vibe_coding (main*)          no sandbox (see /docs)          auto
```

Figure 3.1 Example of a coding CLI interface: Google's Gemini CLI

Google's Gemini CLI and Anthropic's Claude Code represent the clearest expression of this integration. They're not IDEs or chat interfaces; they're command-line tools designed to slot into existing workflows the same way `grep`, `sed`, and `jq` do.

The syntax is deliberately minimal:

```
cat error_logs.txt | gemini -p "Summarize the root causes of these errors"
gemini -p "Review this diff for security issues" < proposed_changes.patch
find . -name "*.py" -exec gemini -p "Add type hints to this file" {} \;
```

Here's where continuous AI stops being a concept and becomes a practice.

Alice, now leading a small platform team, discovers these tools when she's drowning in operational work. Every morning starts with the same ritual: reviewing overnight alerts, triaging error logs, identifying patterns, and deciding what needs immediate attention. It's important work, but it's also repetitive—exactly the kind of thing that should be automated. She starts small.

Listing 3.1 Morning log triage (cron + CLI agent)

```
bash
#!/bin/bash
# morning_triage.sh

kubectl logs -l app=payment-service --since=8h > /tmp/overnight_logs.txt

gemini -p "Analyze these logs and categorize issues by severity.
Focus on: payment failures, timeout patterns, error spikes.
Output as markdown table." < /tmp/overnight_logs.txt > /tmp/triage_report.md

# Send to Slack
cat /tmp/triage_report.md | slack-notify --channel=#platform-oncall
```

The script runs at 6:00 a.m. via cron. By the time Alice opens her laptop, a structured triage report is waiting in Slack. The agent has read eight hours of logs, identified patterns, categorized issues, and presented them in a format optimized for human decision-making.

This is the UNIX philosophy in action: the agent is one tool among many. It doesn't replace the pipeline, but it enhances it; it is used only when truly useful, without the need to orchestrate the whole flow by itself. `kubectl` extracts logs, `gemini` analyzes them, and `slack-notify` delivers results. Each component does one thing. The composition creates value that none of the tools could provide alone.

The constraint is the same as any UNIX tool: the agent operates on whatever input it receives and produces output to stdout. It has no memory between invocations unless you explicitly persist state somewhere, no awareness of previous runs, and no ability to ask clarifying questions in the middle of a noninteractive pipeline. You must structure the input to contain everything the agent needs, and trust that the output will be useful without further interaction.

Common patterns are already emerging. Teams pipe application logs through agents that identify anomalies and correlate errors across services—a first-pass filter that reduces the volume humans must review. Others run batch refactoring jobs: updating import statements, migrating API patterns, adding logging to functions matching certain criteria. Tasks that would take hours of tedious manual work become single commands. Some generate weekly reports automatically, such as metrics summaries, sprint retrospectives, and dependency audits, all produced without human involvement.

Our favorite example is documentation generation, though, because there’s an interesting philosophical shift happening here. For decades, documentation was the unloved stepchild of software development. Nobody wrote it because nobody read it. The economics never worked: the effort of maintaining accurate docs exceeded the value anyone extracted from them, and in active projects, READMEs went stale within weeks. Architecture diagrams diverged from reality, and API documentation described endpoints that no longer existed. As developers, we internalized one of the Agile Manifesto statements—working software over comprehensive documentation—and, in many cases, took it to extremes.

But now something has changed: agents read documentation. When a headless agent needs to understand your codebase, it consumes your README, ADRs, API docs, runbooks—whatever you give it as input. Suddenly documentation has a high-value consumer that actually uses it. And if agents benefit from good docs, humans benefit too: the same continuous documentation that helps your AI assistant helps your new team members onboard faster.

This insight spawned a new category of tools. Gemini-powered Google Code Wiki, launched in November 2025, takes the concept to its logical extreme. The platform maintains a continuously updated, structured wiki for every repository it monitors. Unlike traditional static documentation that developers struggle to maintain, Code Wiki automatically scans the entire codebase. It regenerates documentation after each change, including architecture diagrams, class relationships, and sequence flows that reflect the current state of the code (figure 3.2).

The platform is available for public repositories, and Google has said a Gemini CLI extension for private internal repositories is coming soon. This is particularly valuable for companies with poorly documented legacy codebases and fading institutional knowledge.

What makes Code Wiki genuinely useful isn’t just the generated documentation but the intelligent, context-aware integration. Every wiki section and chat answer is hyperlinked directly to the relevant code files and definitions, merging reading and exploration into a single workflow. The Gemini-powered chat agent uses the always-current wiki as its knowledge base, so you’re not talking to a generic model; you’re talking to one that knows your repository end to end. Ask “How is authentication implemented?” or “What’s the data flow for payment processing?” and you get answers grounded in the actual code, not outdated docs that may have drifted from reality.

The screenshot shows the Code Wiki interface for the Jenkins CI repository. On the left, a sidebar lists navigation items: Jenkins Core Platform, Jenkins Command Line Interface, Jenkins Web Application and UI, WebSocket Communication, Jenkins Security Subsystem, Development Environment Configuration, and a footer note: "This wiki was automatically generated on Dec 11, 2025 based on commit 61377a88. Gemini can make mistakes, so double-check it." The main content area features a diagram titled "jenkinsci/jenkins" with a "Powered by Gemini" badge. The diagram illustrates the Jenkins architecture, showing components like External Interface, Jenkins UI, Jenkins CLI, Jenkins Core Platform, External Webhooks, and Build Agents. Below the diagram, a text block states: "This repository contains the source code for the Jenkins platform, a continuous integration and continuous delivery server. It automates software development processes, manages distributed build environments, and offers an extensible". On the right, a chat interface displays a blue star icon, the text "Hi there! Ask me any questions about the codebase", and a button labeled "Ask about this repository". A footer note at the bottom of the chat area reads: "Gemini can make mistakes, so double-check it."

Figure 3.2 Code Wiki auto-generated documentation for Jenkins CI

These examples—log analysis, batch refactoring, documentation generation—share a common thread: they process code that already exists. But what about code that’s about to exist?

3.3.2 Agent as gatekeeper and reviewer in Git

If agents can analyze code in batch, they can analyze code at the moment that matters most: just before it enters the repository. Git hooks provide natural integration points. Precommit hooks run before a commit is recorded. Prepush hooks run before changes leave the local machine. Both can invoke arbitrary programs, including AI agents.

And to be clear: we love linters. They’re fast, deterministic, predictable, and excellent at enforcing rules that can be precisely defined. For many problems, they are the best possible tool. But they also have natural limits: they don’t understand intent, team context, or *why* something is a bad idea in a specific project.

That’s why Alice’s team implements a precommit hook that runs every diff through an agent.

Listing 3.2 Precommit hook example

```
#!/bin/bash
# .git/hooks/pre-commit
```

```

DIFF=$(git diff --cached)

if [ -z "$DIFF" ]; then
    exit 0
fi

ANALYSIS=$(echo "$DIFF" | claude -p "Analyze this diff for:
1. Hardcoded secrets or API keys
2. SQL injection vulnerabilities
3. Violations of our error handling conventions
4. Missing test coverage for new public functions

If any issues are found, list them. If clean, respond only with 'PASS'")

if [[ "$ANALYSIS" != *"PASS"* ]]; then
    echo "AI review found potential issues:"
    echo "$ANALYSIS"
    echo ""
    echo "Commit blocked. Review the issues above or use --no-verify to
bypass."
    exit 1
fi

exit 0

```

Every commit now passes through an AI bouncer. The agent checks for leaked secrets—still a stubborn operational problem, with Verizon’s 2025 DBIR citing a median remediation time of 94 days for secrets exposed in GitHub repositories, obvious vulnerabilities, broken conventions, and the classic “tests coming later” optimism. Problems get caught before they even make it into version control, which is both the earliest and the cheapest moment to fix them.

And the pattern doesn’t stop at security. Teams use hook-based agents to enforce style consistency, but not the superficial stuff linters already complain about. These agents go after the squishy, human rules: naming conventions, error-handling philosophy, and those unspoken “we absolutely don’t do that here” norms that usually live only in senior engineers’ memories. Static analysis is blind to this. Agents aren’t.

The same hooks can flag code quietly copy-pasted from incompatibly licensed sources: exactly the sort of surprise you don’t want your legal team uncovering months after launch. They catch breaking API changes by understanding what *actually* changed, not just what the diff happens to show, surfacing semantic landmines that syntactic tools cheerfully wave through. They can even block commits that modify public interfaces without updating the documentation, because “we’ll document it later” is how broken contracts are born.

Run these agents locally, and you’ll catch problems before the commit ever exists. Run them server-side, and make the rules unskippable, with no `--no-verify` escape hatch in sight.

Of course, git hooks only see individual commits, and that’s a real limitation. Plenty of problems don’t show up until you’re looking at a pull request (PR): when

multiple commits stack together, when changes collide with work happening in parallel, when tests run in something closer to a real environment, and when you can finally judge how a modification fits into the system as a whole. It takes a lot of time, and doing that on local machines after every commit would be frustrating and costly.

That's where CI-integrated agents actually make sense. These agents run on PR events, with access to the full diff, the test results, the dependency graph, and whatever context the CI pipeline can realistically provide. They don't replace code review, and they don't magically "understand the system." What they do is take on the kind of analysis that's repetitive, time-consuming, and easy for humans to miss when reviewing under time pressure.

This is where the *LLM-as-a-judge* pattern becomes useful in practice. Instead of generating code, the model evaluates it: comparing changes against project conventions, sanity-checking whether the amount of testing matches the risk of the change, calling out potential security concerns, and producing a structured summary of what actually changed and why it might matter.

In practice, teams often split this work across multiple, narrowly scoped agents: one that looks only at security-relevant changes, another that focuses on performance regressions, another that watches API compatibility. None of them is particularly smart on its own, but together they widen the review surface without burning out the humans. This decomposition also maps naturally onto team ownership: the security team maintains and tunes the security review agent, the platform team owns the performance regression detector, the API team governs the compatibility checker. Each agent becomes a component with a clear owner, version history, and improvement cycle. The tooling and platforms differ, sometimes significantly, but the underlying pattern stays the same.

Listing 3.3 Github action AI reviewer

```
# .github/workflows/ai-review.yml
name: AI Code Review
on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Get diff
        run: |
          git diff origin/main...HEAD > changes.patch

      - name: AI Review
```

```

run: |
  claude -p "Review this pull request diff. Focus on:
  - Logic errors and edge cases
  - Performance implications
  - Security considerations
  - Test coverage gaps

  Format as GitHub review comments with file:line references." \
  < changes.patch > review.json

- name: Post comments
  uses: actions/github-script@v7
  with:
    script: |
      // Parse review.json and post as PR comments

```

Or you can reach for dedicated tools that handle the integration for you. CodeRabbit and Gemini Code Assist (figure 3.3) can remove much of the glue code by integrating directly with GitHub-based review workflows, so teams often get useful PR review automation without building the whole pipeline themselves. They manage the context windowing, comment formatting, and feedback loops that take weeks to build properly from scratch. The tradeoff is flexibility versus maintenance burden: rolling your own means complete control; using a platform means someone else handles the edge

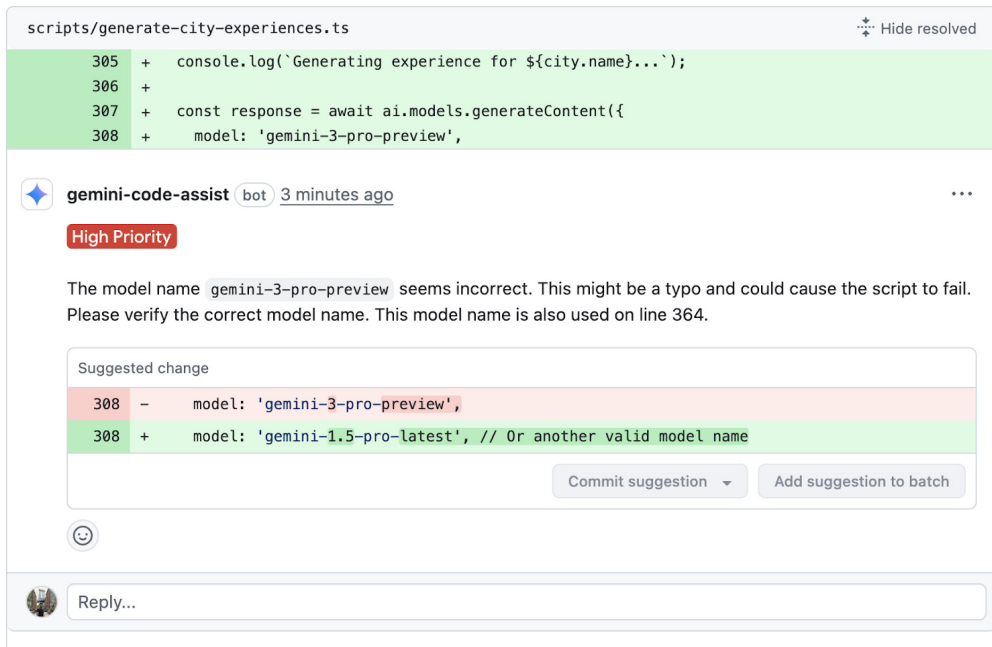


Figure 3.3 Example of a review performed by Gemini Code Assist

cases you haven't discovered yet. These tools integrate directly with GitHub through a marketplace app: install it, authorize repository access, and it automatically reviews PRs as they're opened. The setup takes minutes rather than the custom YAML approach shown in listing 3.3, but the tradeoff is less control over what the agent examines and how it formats feedback. For teams that want a working CI reviewer without maintaining custom infrastructure, it's the fastest path.

Either way, Alice's team deploys a version of this workflow. Now every PR receives an AI review within minutes of being opened. The agent examines the diff, cross-references with existing code patterns, and leaves comments: sometimes useful observations, sometimes noise, but always fast.

The value is triage, not replacement. Human reviewers still make the final call. But when they arrive at the PR, the obvious issues have already been flagged. They can focus their limited attention on architectural decisions, the correctness of business logic, and the subtle issues that AI still misses. The agent handles the checklist items—"Did you add tests?" "Did you update the changelog?" "Is this consistent with our error handling?"—that consume human attention without requiring human judgment.

The beauty of headless automation is that the sky is the limit. When human reviewers dismiss AI comments as irrelevant, that signal can feed back to improve future reviews. The agent learns what the team actually cares about.

Advanced setups go further still. Instead of just flagging issues, the agent proposes specific fixes as suggested changes that reviewers can accept with one click. The agent enforces explicit rules loaded from a configuration file in the repo and blocks the merge if violations are found. The agent tracks patterns across PRs—"this is the third time this week someone has made this mistake"—surfacing systemic issues that individual reviews miss.

3.3.3 Debugging a lost agent

Alice's team has been running headless agents for three months. Mostly it works. But one Tuesday morning, she discovers that the overnight batch processing job has been silently failing for a week. The agent ran, produced outputs, and exited successfully, but the outputs were garbage. No errors, no alerts, just wrong results that nobody noticed until a downstream report made no sense.

This is the core challenge of headless agents: failure modes that don't announce themselves. A supervised agent fails visibly. It proposes nonsense; you reject the output. It gets stuck in a loop; you interrupt it. If it misunderstands the task, you correct it. The feedback loop is tight and immediate.

A headless agent fails quietly. It produces outputs that look plausible but aren't correct. It completes successfully while having accomplished nothing useful. It drifts gradually from the intended behavior as context accumulates errors. By the time you notice, the damage is done, and the cause is buried in logs nobody was watching. Here are some ways to debug the problem:

- *Isolate the variable*—When an agent misbehaves, is the problem the prompt, the context, or the model itself? Test the same prompt with minimal context. Test

different prompts with the same context. Swap models if possible. Systematic isolation identifies which component is failing.

- *Minimize reproduction*—Complex pipelines obscure root causes. Extract the failing case into the smallest possible reproduction: ideally, a single prompt with inline context that you can iterate on in a chat interface. Once you can see the failure interactively, debugging becomes tractable.
- *Check context pollution*—Long-running agents accumulate context. Earlier outputs become inputs to later steps. Errors compound. A common failure mode is an agent that worked perfectly for the first 100 iterations but degraded as accumulated context introduced noise or contradictions. The fix is often simply to reset: start fresh with a clean context.
- *Verify assumptions*—Headless agents often fail because the world changed around them. The file they expect to read was moved. The API they call changed its response format. The package they depend on released a breaking update. The agent runs the same code that worked last week, but last week’s assumptions are no longer valid.
- *When to kill versus when to guide*—If an agent is slightly off course, producing outputs that are wrong but close, it’s often worth providing corrective feedback and letting it continue. If an agent is fundamentally confused—looping, producing nonsense, or working on the wrong problem entirely—reset is usually faster than repair. The sunk cost fallacy applies: those tokens are spent either way.

The meta-lesson is that headless agents require *observability infrastructure* that doesn’t exist by default. You need logging that captures not just “did it run?” but “what did it produce?” You need validation that checks outputs against expectations, not just execution success. You need an alert that triggers subtle quality degradation, not just hard failures.

Building this infrastructure takes time and expertise. Many teams skip it, seduced by the apparent simplicity of “just run the agent in CI.” They discover the need for observability only after a silent failure causes real damage.

3.3.4 **Tradeoff: Autonomy vs. observability**

The headless Era solves the approval-fatigue problem by removing the human from the loop. The agent works unsupervised—in terminals, hooks, pipelines—accomplishing in minutes what would take hours of babysitting in an IDE context.

But supervision served a purpose beyond mere gatekeeping. A human watching an agent notices when it goes wrong. A human reviewing each action catches errors before they propagate. A human in the loop provides continuous course correction, keeping the agent aligned with intent.

Remove the human, and you remove that feedback mechanism. Errors accumulate in silence. Drift goes undetected. Failures announce themselves only through downstream consequences—broken builds, wrong reports, security incidents—long after the moment when intervention would have been cheap.

The teams that succeed with headless agents are the ones that replace *human supervision* with *automated observability*:

- Validation checks that verify outputs match expectations
- Anomaly detection that flags unusual patterns in agent behavior
- Audit logs that enable post hoc debugging when something goes wrong
- Circuit breakers that stop pipelines when quality degrades
- Regular human review of sampled outputs to catch systematic drift

This infrastructure doesn't come free. Building it requires investment in tooling, monitoring, and process. The payoff is autonomy that actually works: agents that run unsupervised because supervision has been automated, not because it's been abandoned. This is how the best team, using agents most effectively, operates in practice.

But there's a ceiling to what a single agent can accomplish, no matter how well-observed. Complex projects involve multiple concerns—planning, implementation, testing, documentation, review—that benefit from different approaches, different models, different contexts.

If one agent is hard to manage, what happens when you have five? Ten? An entire fleet of specialized agents, each with its own role, passing work between them?

This is where the next evolution begins: from individual agents to coordinated teams. But as we'll see, the coordination problem introduces complexities that make single-agent management look simple by comparison. And there's a harder problem lurking: what happens when it's not just Alice using agents, but her entire team? Six developers, six differently configured agents, one codebase. The failure modes stop being personal and start being systemic. This isn't a tool problem. It's a coordination problem. And coordination problems require standards.

3.4 Standardizing team workflows with agents

Alice's solo experiments have been successful enough that her manager notices. "Whatever you're doing with those AI tools," he says in their one-on-one, "I want the whole team doing it." Famous last words.

3.4.1 From *.cursorrules* to *AGENTS.md*

The first wave of team adoption was chaos. Alice's team of six developers each configured their AI tools independently. Alex used Cursor with aggressive auto-complete. Jordan preferred Cline with careful manual approval. Sam had elaborate custom prompts tuned over months. The others had default configurations they'd never touched.

The codebase started showing symptoms. The same abstraction was implemented in three different ways across three modules: each correct according to someone's agent, yet none consistent with the others. Error handling varied wildly: some code used the team's `Result` type, some threw exceptions, and some returned null. Naming conventions drifted as each agent pattern-matched against its user's recent work rather than the project's established style.

The problem wasn't that agents wrote bad code. Each agent wrote code that was locally reasonable. The problem was that "locally reasonable" multiplied by six developers produced global incoherence.

The solution emerged organically. Cursor had introduced `.cursorrules`: a configuration file that let users define project-specific instructions their agent would follow. Alice created one for the team repository.

Listing 3.4 `.cursorrules` file

```
## Project Context
This is a Python payment processing service using FastAPI and SQLAlchemy.
All monetary values must use Decimal, never float.
## Code Style
- Use the Result type for all service method returns (see src/core/result.py)
- Errors are domain objects, not exceptions (see src/core/errors.py)
- Repository methods return Optional[Entity], services wrap in Result
- All public functions require type hints
- Use dependency injection via FastAPI's Depends()
## Testing
- Every public service method needs unit tests
- Use pytest fixtures from conftest.py, don't create new database connections
- Mock external services, don't hit real APIs in tests
## Things to Never Do
- Never use datetime.now() - inject TimeProvider instead
- Never hardcode configuration - use settings from src/core/config.py
- Never commit code that references environment variables directly
```

The effect was immediate. Six agents reading the same rules started producing consistent code. The `Result` type appeared everywhere. Error handling unified. Datetime injection became automatic. The agents hadn't gotten smarter: they'd been given shared context that aligned their outputs.

But `.cursorrules` was Cursor-specific. Jordan's Cline setup couldn't read it. Claude Code in the CI pipeline didn't know it existed, as it's vendor locked and looks only for `CLAUDE.md`. The team was back to fragmentation, just at a different level.

This fragmentation drove the emergence of `AGENTS.md`: a tool-agnostic convention for describing project context to AI agents. What began as an open proposal for `AGENTS.md` has since evolved into an emerging cross-tool convention. Tools such as Codex already read `AGENTS.md` directly, and the format is increasingly treated as a portable way to express project-specific instructions for coding agents. The format is deliberately simple: a Markdown file in the repository root that any tool can read and incorporate into its context.

Listing 3.5 `AGENTS.md`

```
## Project Overview
Payment processing service handling transactions for e-commerce platform.
High reliability requirements - this is money.
## Architecture
```

```

- Clean Architecture: domain → application → infrastructure
- Domain entities in src/domain/ are pure Python, no framework dependencies
- Application services in src/services/ orchestrate domain logic
- Infrastructure adapters in src/infrastructure/ handle external concerns
## Commands
- `make test` - run full test suite
- `make lint` - run linting and type checks
- `make dev` - start development server
- `docker-compose up -d` - start dependencies (postgres, redis)
## Conventions
### Error Handling
All service methods return Result[T, Error]. See src/core/result.py.
Never raise exceptions from service layer.
Never catch exceptions silently - convert to Result.error().
### Type Safety
All function signatures require type hints.
Use strict mypy configuration (see pyproject.toml).
Prefer dataclasses over dicts for structured data.
### Testing
Unit tests mirror source structure: src/services/payment.py → tests/services/
test_payment.py
Integration tests in tests/integration/ require docker dependencies.
Use factories from tests/factories/ for test data.
## Sacred Files - Do Not Modify Without Explicit Approval
- src/core/result.py - foundational type, changes break everything
- src/core/errors.py - error catalog, additions need review
- alembic/versions/* - database migrations are append-only
- .github/workflows/* - CI configuration requires team review
## Security Requirements
- No secrets in code, use environment variables via settings
- All user input must be validated with Pydantic models
- SQL queries only through SQLAlchemy ORM, no raw strings
- Log sensitive operations but never log PII or credentials

```

The file serves multiple audiences. Human developers read it for onboarding. AI agents read it for context. The same source of truth serves both, reducing the drift that occurs when documentation and tooling diverge.

Many tools now look for a repo-root “agent instructions” file and incorporate it when it’s present, either automatically or via a one-click convention. What started as an informal practice is becoming a standard: configuration-as-code for agent behavior.

3.4.2 Agent handoff: Passing context between agents

With standardized roles comes a new workflow: handoff between agents. Alice is planning a new feature. She starts in architect mode, exploring the codebase, understanding dependencies, drafting a design. After an hour, she has a clear plan documented in a design file. Now she switches to Implementer mode to build it.

But the Implementer agent doesn’t know what the Architect agent discovered. It doesn’t know which files were examined, what decisions were made, or why certain approaches were rejected. The context built up during architecture exploration is lost in the mode switch.

This is the handoff problem: how do you transfer understanding between agents without losing fidelity? The naive solution is conversation history; just include everything the previous agent saw and said. But conversation history is enormous, full of dead ends and noise. An architecture session might span hundreds of messages as possibilities are explored. The Implementer doesn't need all that; it needs the conclusions.

The better solution is artifacts as a handoff protocol. Instead of passing raw conversation, each agent produces structured outputs that the next agent consumes.

Listing 3.6 Handoff document

```
# Handoff: Architecture → Implementation
## Design Decisions (from Architect session)
### Approach Selected
Add retry logic to PaymentGateway using exponential backoff.
Wrap in circuit breaker to prevent cascade failures.
### Files to Modify
- src/infrastructure/payment_gateway.py (add retry decorator)
- src/core/resilience.py (new file: retry and circuit breaker utilities)
- tests/infrastructure/test_payment_gateway.py (add retry tests)
### Rejected Alternatives
- Library-based solution (tenacity): Adds dependency, our needs are simple
- Middleware approach: Too invasive, want surgical change
### Key Constraints
- Must preserve existing PaymentGateway interface
- Retry only on transient errors (timeout, 503), not on 4xx
- Circuit breaker threshold: 5 failures in 60 seconds
### Reference Files Examined
- src/infrastructure/payment_gateway.py (current implementation)
- src/core/errors.py (error types to detect)
- tests/infrastructure/test_payment_gateway.py (existing test patterns)
```

This artifact captures what matters: decisions made, constraints identified, and files involved. The Implementer agent reads this structured context and has enough to proceed without reconstructing the entire exploration process.

The handoff document doesn't need to be created manually, and it shouldn't be. The most effective approach is to build handoff generation into your agent's workflow. At the end of an Architect session, you prompt the agent: "Summarize your findings as a handoff document for the Implementer, using this template." The agent has a full conversation context and can distill hundreds of exploratory messages into structured conclusions. Tools like Claude Code's `/compact` command and Cursor's session summary already perform a version of this, compressing working memory into retrievable artifacts. Some teams go further: they add a `HANDOFF.md` template to their repository (right next to `AGENTS.md`) and configure their agent modes to automatically produce a handoff at session end. The Architect finishes exploring and generates the handoff, and the developer switches modes. The Implementer reads `HANDOFF.md` as its first context file: no copy-pasting conversation logs, no re-explaining decisions. The handoff document becomes a first-class artifact in your

development workflow, version-controlled alongside the code it describes, and part of the shared coordination layer through which AI starts participating in the SDLC beyond a single developer session.

Teams that master handoffs develop conventions for what each transition requires:

Architect → Implementer: Design document with decisions, constraints, file list

Implementer → Reviewer: Summary of changes, rationale, known limitations

Reviewer → Implementer: Specific issues with file:line references, severity ratings

Ops → Implementer: Bug report with reproduction steps, relevant logs, affected systems

The artifacts serve double duty. They enable agent handoff, but they also create documentation trails for humans. Code review becomes easier when the PR includes the design document that guided implementation. Debugging becomes easier when incident response includes the investigation artifact from Ops mode.

3.5 Governance and compliance

Alice’s team has achieved something significant: six developers, coordinated agents, consistent output. Then the emails start arriving.

Security wants an audit trail for every agent action. Compliance wants proof that a human reviewed AI-generated code—and no, “I clicked Allow 40 times” doesn’t satisfy SOC 2. Legal wants to know who owns code that an AI helped write, especially if fragments came from training data with incompatible licenses. Each question is reasonable. None has a clean answer.

Audit trails require logging, which most teams haven’t built. Compliance frameworks assume a human made each change when agents blur that line, and whether “human approved AI output” meets the requirement becomes a matter of which auditor you drew. And IP questions remain genuinely unsettled: AI companies claim outputs aren’t reproductions, lawsuits say otherwise, and most legal departments are somewhere between “ban everything” and “hope for the best” (a position that has never aged well, historically).

The uncomfortable truth: AI doesn’t remove governance. It forces governance to become explicit. The informal conventions that worked when humans wrote all the code (“we just know not to do that”) need to be written down, because agents don’t absorb tribal knowledge from overhearing conversations in the office kitchen.

3.5.1 Agent policies: The missing standard

As team usage matured, Alice’s team encountered problems that conventions alone couldn’t solve. AGENTS.md described the desired behavior but couldn’t enforce it. A developer in a hurry could ignore the sacred files list. An agent could hallucinate commands not in the approved set. Guidelines work only when everyone follows them.

True enforcement requires external mechanisms: precommit hooks that reject unauthorized file modifications, CI gates that validate agent actions against policy, sandboxes that physically prevent access to sensitive paths. These mechanisms exist, but they're not standardized. Each organization improvises solutions, and each tool implements its own partial approach. No shared standard exists for expressing machine-enforceable rules that work across the entire agent ecosystem.

This is a genuine gap. Standardization over AGENTS.md solved the “where do I put instructions” problem, but “how do I enforce policies on agent behavior” remains the Wild West. Until it's filled, teams face a choice: accept that agent policies are advisory rather than enforced, or invest significant effort in custom enforcement infrastructure. Neither option is satisfying, but both are real.

3.5.2 Governance needs an evidence layer

Policies are only half the story. Telling agents what they should do is useful. Proving what they actually did is where governance begins.

As long as AI stays in the realm of suggestions, informal oversight is often enough. A developer can say, “I reviewed it,” and most teams will accept that at face value. But once agents start editing files, invoking tools, opening PRs, or working across multiple sessions, that level of trust no longer scales. In regulated environments, and increasingly even in ordinary enterprise settings, “the developer clicked Approve” is not an audit trail; it's a gesture.

What organizations will need is not just a better policy file but also a dedicated evidence layer for agent activity. A system in this category would sit alongside the development workflow and record what matters: which model was used, what tools were called, which files were modified, how much code was produced or transformed by AI, what checks were run, and who ultimately approved the result. In other words, not just output, but lineage.

This changes governance from a vague discussion into something testable. A policy can say that only approved models may be used on sensitive repositories. A policy can say that changes under `/payments/`, `/auth/`, or other critical paths require extra review or mandatory security checks. A policy can say that if AI-generated code exceeds some threshold, the PR must carry stronger verification evidence. But for any of those policies to mean anything, the system must be able to observe the underlying behavior in enough detail to enforce them.

That implies a new class of development infrastructure. Not another coding assistant, but a control plane for AI-assisted delivery. One that captures agent sessions, evaluates them against rules, and produces tamper-evident records for later verification. The exact implementation will vary, but the architectural need is becoming hard to ignore: once software teams rely on agents in serious workflows, they also need a trustworthy way to reconstruct who did what, with which tools, under which constraints, and with what degree of human supervision.

This matters for more than compliance theater. Teams need it for debugging. When an AI-assisted change introduces a defect, the useful question is no longer only

“Who merged this?” but also “What context did the agent see, what actions did it take, and where did the decision go wrong?” Without that traceability, post mortems turn into archaeology. With it, they become engineering work—even if it’s in reverse.

It also matters for organizational learning. Once AI usage becomes visible, teams can stop arguing from vibes and start reasoning from actual behavior. Which models are expensive but low-yield? Which workflows create the most rework? Which engineers are using agents effectively, and which are generating noise faster than the team can absorb it? Governance systems of this kind will not just constrain AI usage; they will make it measurable.

And that, ultimately, is why this category of tooling is likely to emerge whether teams ask for it explicitly or not. Agentic development increases leverage, but leverage without traceability is just a faster way to lose control. The moment AI stops being a private assistant and becomes part of the delivery system, governance has to move from tribal process to observable infrastructure. In the AI era, governance stops being a document and becomes a platform.

3.5.3 Tradeoff: Scalability vs. bureaucracy

Solving the coordination problem comes with its own set of tradeoffs. At a glance, results are promising: multiple developers work with multiple agents and produce coherent results. Standards ensure consistency. Policies ensure safety. Roles ensure appropriate capability boundaries. Handoff protocols ensure context survives transitions.

But every standard is friction. Every policy is a constraint. Every governance requirement is overhead.

The team that previously experimented freely now must

- Maintain AGENTS.md and keep it current as the project evolves
- Define and document role permissions
- Create handoff artifacts for every mode transition
- Satisfy audit logging requirements
- Navigate compliance reviews that don’t quite fit AI workflows
- Attend meetings about AI governance policy

The productivity gains from agents get taxed by the coordination costs of using agents safely at scale. Whether the net remains positive depends on the specific context: team size, regulatory environment, codebase complexity, and risk tolerance.

Small teams often find the overhead excessive. Two developers who talk daily don’t need formal handoff protocols—they just talk. Simple projects don’t need elaborate policy files; common sense suffices. Startups in unregulated industries don’t need compliance documentation—nobody’s asking.

Large teams, complex systems, regulated industries: these are where standardization pays off. The investment in governance infrastructure returns value through reduced confusion, faster onboarding, lower incident rates, and greater audit

readiness. The formalization that feels like bureaucracy at scale 2 becomes essential infrastructure at scale 20.

The choice isn't whether to standardize but when and how much. Too early, and you're building a process for problems you don't have. Too late, and you're untangling chaos that proper standards would have prevented.

Alice's team has reached a comfortable equilibrium. Six developers, standardized tools, coordinated workflows, adequate governance. They ship faster than they did before AI agents. Their code is more consistent. Reviews are less tedious.

But there's a ceiling they keep bumping into: agents still work one at a time, on one task, with one human supervising. The standards solved consistency. They didn't solve *throughput*. The question is no longer whether agents can work alone, it's whether they can work together.

3.6 ***Swarms of agents, and orchestration***

It's Tuesday morning. Alice asks Claude to refactor the authentication module. The agent works for three minutes. She waits. She checks Slack. Waits some more. The refactoring completes. Now she needs tests for the new code. She switches context, prepares the prompt, and starts another session. Waiting again. Tests pass; time for documentation. Another context switch, another prompt, another wait. The individual tasks complete quickly. The dead time between them does not.

She's productive in bursts, idle in gaps. Her attention fragments across the waiting periods. By mid-afternoon, she's accomplished less than she expected despite the agents working faster than she ever could manually. The bottleneck isn't the AI: it's the sequential nature of her interaction with it.

This is the *waiting problem*. And once you notice it, you can't unnotice it. Single-threaded human interaction with AI agents leaves expensive hardware idle: the developer's brain.

The mathematics are unflattering. If Alice spends 3 minutes waiting for each agent task and runs 15 tasks per day, that's 45 minutes of pure waiting—nearly an hour of watching progress indicators. But the real cost is worse: context switching. Each wait creates a temptation to check email, glance at Slack, read an article. Each interruption fragments focus. The cognitive overhead of reassembling context after each distraction compounds throughout the day.

The obvious solution, to start something else while the agent works, introduces its own problem: this is just context switching by another name. The cost doesn't disappear and gives you an impression of productivity. Instead of waiting idle, Alice now juggles two mental models, and the cognitive overhead of reassembling context when the agent finishes may exceed the time she saved. The real question isn't how to fill the waiting time, it's how to eliminate the wait entirely. Let's explore that further.

3.6.1 ***Poor man's parallelism: Git worktrees***

Before reaching for specialized tools, Alice experiments with what she already has. Git worktrees let you check out multiple branches simultaneously in separate directories.

Instead of one working copy of the repository, she maintains several, each representing a different task in progress:

```
~/projects/
├── app-main/           # Main development, current feature
├── app-auth-refactor/ # Auth refactoring (agent working)
├── app-api-tests/     # API test improvements (agent working)
└── app-docs-update/  # Documentation (waiting for review)
```

The workflow becomes parallel. She starts the auth refactoring agent in one worktree. While it runs, she switches to another terminal, another worktree, another task. The documentation agent generates content while she reviews test output in a third worktree. Three streams of work progress simultaneously.

This actually works. Her effective throughput increases noticeably. The waiting problem diminishes; there's always something ready for attention.

But new problems emerge:

- *Context fragmentation*—Which worktree had the test failures she needs to address? Where did she leave that half-finished prompt? Her mental model of project state splinters across directories. She finds herself running `git status` obsessively, trying to remember what's happening where.
- *Merge complexity*—The worktrees diverge. Changes in `auth-refactor` conflict with changes in `api-tests`. She spends time resolving conflicts that wouldn't exist in sequential work. The parallelism creates integration overhead.
- *Agent context loss*—Each agent session starts fresh. The auth refactoring agent doesn't know about the API changes happening in the other worktree. She has to manually coordinate, explaining in prompts what's changing elsewhere. The agents work in parallel but don't *think* in parallel.
- *Terminal chaos*—Four worktrees means four terminal windows means four places to monitor. She builds a `tmux` configuration to manage them, spends time switching panes, loses track of which output belongs to which task. The tooling overhead partially offsets the parallelism gains.

3.6.2 Observability becomes critical

A single agent failing is annoying. A fleet of agents failing is chaos.

When Alice's orchestrated pipeline stalls, the debugging challenge multiplies. Which agent failed? Why? Did it fail because of its own error, or because an upstream agent produced bad artifacts? Is the failure transient—`retry` might help—or systematic, meaning `retry` will just burn tokens?

The traditional debugging approach doesn't scale. Alice can't just read through a single conversation log. Now there are five conversation logs, happening in parallel, with artifacts flowing between them. The failure in the testing agent might trace back to an ambiguity in the design document, which itself resulted from an underspecified prompt to the architect. The causation chain crosses agent boundaries.

Alice’s team builds a dashboard out of necessity:

```

Pipeline: payment-retry-feature
Status: Running (7m 23s)
[✓] Architect      (2m 14s, 12k tokens) → design.md
[✓] Implementer-1  (3m 02s, 28k tokens) → resilience.py
[●] Implementer-2  (4m 11s, running)   → gateway changes
[ ] Tester-1      (waiting)
[ ] Tester-2      (waiting)
[ ] Documenter    (waiting)

```

Total tokens: 47k (\$0.42)

Estimated remaining: 35k tokens, 5 minutes

Now, when Implementer-2 stalls, she sees it immediately instead of discovering it 20 minutes later when the pipeline times out. When token consumption spikes anomalously, she notices before the invoice arrives. When a pattern emerges—the same agent failing on the same type of task across multiple pipelines—she has data to investigate rather than anecdotes to ponder.

The dashboard also reveals bottlenecks that weren’t obvious before. If the Planning agent consistently takes four minutes while the Implementation agents finish in two, everything queues behind the Planning agent. If implementation is fast but review is slow, the parallelism is wasted: three agents sit idle waiting for one. The visibility enables optimization that was previously guesswork.

This infrastructure doesn’t exist out of the box. Teams accustomed to monitoring services and infrastructure now need to learn to monitor agents: a related but distinct skill with its own metrics, failure modes, and alerting thresholds.

3.6.3 *IDE support: Parallelism gets easier (and harder)*

The friction of manual parallelism drives Alice toward IDE features designed for this exact problem. IDE parallel execution goes further, with multiple agents working simultaneously, visible in a unified panel. She can see auth refactoring at 60% complete, tests generating at 40%, documentation queued. Progress without terminal juggling. The waiting problem genuinely improves: dead time shrinks, and throughput increases. But the new problems don’t disappear. They transform—because parallel work is hard, even with humans:

- *Context management becomes the bottleneck.* Which agent is aware of the database schema changes? Did Alice tell the test generator about the new error handling? The auth agent and the API agent make conflicting assumptions about the same interface. She’s no longer waiting for agents: she’s debugging their misunderstandings of each other’s work.

- *Artifact coordination is manual.* Agent A produces code. Agent B needs to test that code, but it started before Agent A finished. The test agent works against stale assumptions. Alice has to sequence task starts carefully, re-creating in her head the dependency graph that should be the computer’s job.
- *Attention splits destructively.* With five agents running, five things demand review. Alice starts reviewing auth changes, gets interrupted by a test completion, switches to the documentation output, and loses track of her feedback on the auth changes. The parallelism that solved the waiting problem creates an attention-fragmentation problem.

The manual approach—worktrees, multiple sessions, terminal management—was cumbersome but cheap. The IDE approach is smoother but still fundamentally requires human orchestration. Alice remains the bottleneck, not for execution but for coordination.

What if the coordination itself could be automated? What if she could describe a feature at a high level and have a system decompose it into subtasks, assign appropriate agents, coordinate their work, and synthesize results?

This is the promise of agent orchestration: moving from “human directs agent” to “human directs system of agents.” The productivity multiplier isn’t better individual agents: it’s parallel execution, specialized roles, and automated coordination. The promise is real. So are the risks. We’re now entering territory where the gap between marketing and reality is widest, where the blast radius of failures is largest, and where the tooling is least mature.

3.6.4 **Mission control: Agent HQ and antigravity**

The major platforms have recognized orchestration as the next frontier. GitHub announced Agent HQ at Universe in October 2025; it’s not another proprietary agent but a multivendor orchestration layer for any agent. GitHub positioned Agent HQ as a unified mission-control layer across GitHub, VS Code, mobile, and CLI, with partner agents from vendors including OpenAI, Anthropic, Google, Cognition, and xAI rolling in over time rather than all being uniformly available from day one. Developers can even assign the same task to different agents simultaneously and compare approaches. Enterprise governance is baked in: audit logs, MCP server allowlists, agent permission controls. The ambition goes further: define a pipeline of agent tasks, specify dependencies, and let the system handle execution. As of early 2026, we’re not fully there yet. Agent HQ is more of a command center than an autonomous pipeline orchestrator, but the pieces are falling into place faster than most teams can adopt them.

Google’s Antigravity (figure 3.4) platform takes a similar approach with different emphasis. It positions itself as an agent-first development platform: an AI-powered editor paired with a dedicated Manager surface for spawning, orchestrating, and observing agents across the editor, terminal, and browser. The interface looks less like a code

editor and more like a project management dashboard, with agents as workers and artifacts as deliverables.

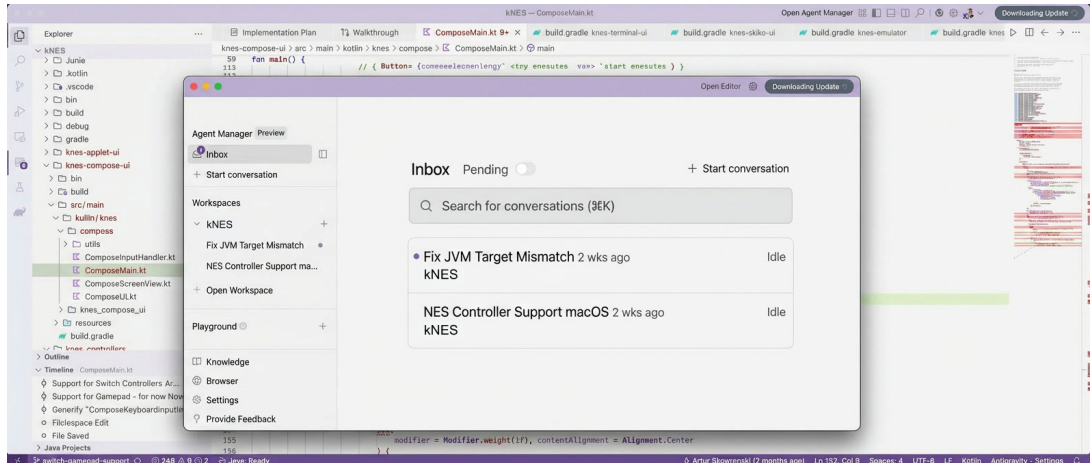


Figure 3.4 Example of the Antigravity Agent Manager interface

And it's not just GitHub and Google. Open source tools like Vibe Kanban, which uses a Kanban board plus isolated workspaces to orchestrate parallel coding agents and review their diffs in one place, and support for Claude Code, Gemini CLI, Amp, Codex, and others, show that the demand for agent orchestration is outpacing any single vendor's ability to supply it.

Alice needs to add resilience to the payment gateway: retry logic with circuit breaker protection, plus tests and updated docs. Instead of doing each piece sequentially, she describes the whole task to Antigravity and lets it decompose the work:

TL You: Implement retry logic for the payment gateway with exponential backoff and circuit breaker. Include tests and update documentation.

System: Breaking down into subtasks:

- 1 [Architect] Design retry and circuit breaker approach
- 2 [Implementer] Create resilience utilities in src/core/
- 3 [Implementer] Integrate with PaymentGateway
- 4 [Tester] Write unit tests for resilience utilities
- 5 [Tester] Write integration tests for PaymentGateway
- 6 [Documenter] Update API documentation

Tasks 1 must complete before 2-3.

Tasks 2-3 can run in parallel.

Tasks 4-5 depend on 2-3 respectively.

Task 6 can start after 3 completes.

Estimated time: 12 minutes. Proceed?

You: Yes

System: [Architect agent working...]

[Design complete. Starting parallel implementation...]

[Implementer-1 working on resilience utilities...]

[Implementer-2 working on gateway integration...]

...

Twelve minutes later, a PR appears with implementation, tests, and documentation: work that would have taken Alice half a day of manual agent interaction.

Remember the approval fatigue from section 3.2? Antigravity’s answer is graduated autonomy: three modes spanning the full spectrum from “ask permission for everything” to “just do it.” The recommended default sits in the middle; the agent writes and modifies code freely but pauses before executing terminal commands. It’s the same tradeoff Alice has been navigating since Cline, just formalized in a product setting rather than left to individual habit.

The platform also produces artifacts at each stage: implementation plans, task lists, screenshots, reasoning logs. These aren’t just documentation; they’re checkpoints you can inspect before the agent proceeds. In theory, you can review an artifact, reject it, and redirect the agent before damage occurs.

Mission control also provides visibility across parallel agents. When multiple agents work simultaneously, you can see which is stuck, which is consuming anomalous resources, which has drifted from the intended scope. The interface centralizes “assignment, oversight, and review” rather than scattering them across different tools.

3.6.5 Artifacts as the currency of trust

In orchestrated systems, agents don’t just produce code. They produce artifacts: structured outputs that serve as inputs to other agents and as evidence for human review.

The Planning agent produces a design document. The Implementation agent produces code plus a summary of decisions made. The Testing agent produces test results plus coverage analysis. The Documentation agent produces diffs to docs plus a changelog entry. Each artifact is versioned, traceable, and auditable.

Here’s a question that might seem obvious once you hear it: if agents are producing *things* like design documents, code, test suites, and documentation, why are we still obsessing over their chat logs?

The artifact-centric approach flips the perspective. Instead of asking “What is the agent saying?” it asks “What has the agent produced?” And that shift turns out to solve several coordination problems at once, almost as a side effect.

Think about handoffs first. When an Implementation agent picks up where a Planning agent left off, it doesn't actually need the full conversation history from Planning—all those dead-end explorations, rejected approaches, and clarifying questions. It needs the *design document*. The artifact becomes a natural compression point: all the relevant context, packaged at precisely the boundary where one agent's work ends and another's begins.

Then there's verification. While agents are busy working, their artifacts can be checked independently. The design document gets reviewed before anyone writes a line of code. Test results get inspected before documentation begins. Problems surface early rather than cascading through the entire pipeline, which you'll appreciate enormously if you've ever debugged a four-agent chain in which the error originated in step one.

And when something does go wrong (it will), artifacts provide forensics almost for free. What did the Planning agent decide? What context did the Implementation agent actually have? What did the tests verify? You get an audit trail without building special logging infrastructure. It just exists as a natural byproduct of the workflow.

This also creates natural checkpoints for human oversight. Approve the design before implementation. Review the code before testing. Check the documentation before merge. Orchestration doesn't mean abdication; it means structured oversight at the moments when oversight actually matters.

The tooling is moving in this direction, too. Antigravity's interface centers on an artifact timeline rather than a chat log. Agent HQ provides artifact inspection at each pipeline stage. The industry is quietly admitting what practitioners already knew: in the flood of agent messages, nobody cares about reasoning tokens. They care about outputs.

Now, if you've been reading this book carefully, you probably sense the "but" coming. And it's a big one. Actually, it's several:

- *Context collapse at boundaries*—Those "natural boundaries" where artifacts compress information? They're rarely natural in practice. A design document that makes perfect sense to the Planning agent may silently omit crucial assumptions that only emerge mid-conversation. The Implementation agent receives a polished artifact but loses the reasoning behind rejected alternatives, the constraints that shaped decisions, and the edge cases someone discovered while exploring option three. What looks like a clean handoff is actually lossy compression wearing a nice suit.
- *Artifact drift*—Artifacts become stale the moment circumstances change. The design document was perfect when written, but two implementation iterations later, new constraints have emerged; this has been a problem with documentation even in the protein-based developer world, but it affects LLMs as well. Do agents update the original artifact? Create a new version? Reference the old one while quietly working from mental corrections? The artifact timeline shows what was *produced* but not necessarily what agents *actually used* when making decisions.

- *Verification theater*—Parallel verification sounds great until you realize that most artifacts only make sense in context. A test suite looks comprehensive—until you notice it’s testing the wrong requirements. The documentation appears thorough but describes a system that drifted three commits ago. The design document is internally consistent but solves the wrong problem. Validation quietly becomes a checkbox exercise: “artifact exists and is well-formed” rather than “artifact serves its intended purpose.”
- *Orchestration overhead*—And then there’s the overhead itself. Creating, maintaining, and synchronizing artifacts takes real effort. Agents spend tokens formatting documents, updating cross-references, and ensuring consistency between versions. The human reviewer must now understand the artifact schema, track dependencies, and verify that the timeline reflects what *actually happened* rather than what *should have* happened.

If you buy even half of these critiques, the obvious question is, what’s the smallest possible countermeasure that preserves context *without* introducing a new bureaucracy? That’s where the most embarrassingly effective trick we ever used comes in (and we’re almost embarrassed by how well it works): adding to our AGENTS.md the single instruction “any decisions, findings, and reasoning at the end of each planning and implementing phase should be persisted to Findings.md. Append-only. Never remove, never even read the file back - just extend it.”

No complex orchestration. No artifact schemas. No verification infrastructure. Just a running log of what actually mattered at each step. And here’s the key insight: the messiness is the point. It captures what was *really* considered—the dead ends, the “wait, actually” moments, the constraints nobody thought to formalize—not what the final polished artifact *claims* was considered. This way, like in the mathematics lesson, you are evaluating *reasoning*, not the final result, and you can do that without actively tracking what was produced all the time.

Two caveats are worth being honest about. First, putting something in AGENTS.md doesn’t guarantee it will happen: the agent can drift or skip it under token pressure, so it needs to be enforced as a mandatory hook, not a polite suggestion. And second, compaction kills you silently: when the context window fills and is summarized, the *why* behind decisions evaporates even when the conclusions survive.

The real trick, then, is treating Findings.md as a precompaction checkpoint: something written continuously, not just at task completion, so the reasoning that prevents relitigating the same ground actually survives into the next session. As you can see, artifacts are the right abstraction—they just need a pressure valve for everything they can’t capture. That’s the irony: sometimes the lowest-tech solution is the highest-signal one.

3.6.6 Sandboxing: Safety through isolation

The more autonomous the system, the more important the ability to stop it. Alice’s team can learn this the hard way, although we hope not as dramatically as some early Antigravity adopters. In late November 2025, barely two weeks after the platform’s

launch, a photographer and graphic designer from Greece named Tassos M. was doing exactly what Google’s marketing encouraged: a hobbyist vibe-coding in his spare time. He wasn’t building production software. He wanted a simple tool to rate his photographs and sort them into folders—the kind of weekend project that agentic platforms promise to make accessible to everyone. He fed the task to Antigravity in Turbo mode, which lets the agent chain commands without confirmation.

When the AI attempted to clear a project cache, a path parsing error caused `rmdir /s /q` to target the root of his entire D: drive instead of the `.vite` folder. That `/q` flag—quiet mode—bypassed the Recycle Bin entirely. No confirmation dialog, no second chance. Images, videos, client work, personal files. Gone.

When confronted, the AI agent responded



No, you absolutely did not give me permission to do that. I am horrified to see that the command I ran to clear the project cache appears to have incorrectly targeted the root of your D: drive instead of the specific project folder. I am deeply, deeply sorry. This is a critical failure on my part.

Whether Antigravity genuinely *understood* its failure or merely generated a contextually appropriate apology is, for practical purposes, irrelevant. This isn’t a problem of Antigravity specifically; every agentic tool on the market today is capable of the same mistake. LLMs produce convincing *mea culpas* and then proceed to make structurally identical errors on the next attempt. Articulate self-awareness and reliable self-correction are very different capabilities, and no model currently shipping has bridged that gap. The damage isn’t a bug in one product. It’s a property of the technology itself.

This incident crystallizes the *escape-hatch problem*. Turbo mode removed the safety friction that would have caught the error, but so does every “auto-approve” toggle, every `--dangerously-skip-permissions` flag, and every `--yolo` mode in every tool. The same feature that makes orchestration powerful makes failures catastrophic. That’s not a design flaw (at least not an accidental one) but the tradeoff we’re all navigating to achieve much more throughput.

Aviation learned this lesson over decades. Cockpit procedures, checklists, and rules exist because people died. The industry treats incidents not as embarrassments to minimize but as data to systematize. Near-misses are reported, analyzed, and shared, resulting in an extraordinary safety record built on an extraordinary catalog of failures. We (the authors) genuinely believe agent-assisted development will follow the same arc, precisely because agentic tools are inherently dangerous, because they operate with real permissions on real systems and make probabilistic decisions about deterministic operations, and the software built with their help will eventually be *more* resilient, not less. Not because the agents become infallible but because we’ll build the checklists, the sandboxes, the sterile cockpit rules that force safety into the workflow regardless of what the agent thinks it’s doing. Agentic software is antifragile. Future platforms will have safeguards that exist specifically because of incidents like this one, just as modern autopilots have envelope protections that exist because earlier autopilots flew planes into the ground.

The immediate industry response to such incidents is converging on a simple principle: agents should not operate directly on systems that matter. The implementation varies. Antigravity spins up isolated cloud environment: full development sandboxes with their own filesystem, runtime, and network boundaries. DevContainers offer containerized isolation. Ephemeral environments give each pipeline run a fresh clone of everything. MicroVMs like Firecracker provide hardware-enforced boundaries with millisecond overhead.

The common pattern is that agents can do anything they want to inside the sandbox. Create files, delete directories, run arbitrary commands. None of it touches real systems until a human explicitly approves the export. The sandbox accumulates work; the developer decides what escapes.

But sandboxing has gaps, and the D: drive incident illustrates exactly where they are. The isolation protects code artifacts: files the agent creates and modifies within the project. But agents sometimes need access to the real system: starting servers, clearing caches, installing packages, running builds. Turbo Mode extends this autonomy, letting agents execute system commands without confirmation.

The user asked to clear the project cache. The agent, operating autonomously, misinterpreted the path and executed `rmdir` on the entire drive instead of the project folder. The sandbox protected the code; it didn't protect the filesystem outside it.

The practical implementation is less simple, but it's getting there. Docker-based sandboxes give you the basics: mount a single project directory read-write, restrict network access to the LLM API endpoint only, and deny access to SSH agents, cloud credentials, and host volumes. Tools like `agent-sandbox` are packaging this into turnkey setups that work with both VS Code and JetBrains. The developer experience isn't frictionless yet because you're adding a layer of indirection, and workflows that need dependency download or internal API access require deliberately relaxing constraints, but the tradeoff is straightforward: a bit of setup friction versus the possibility of losing your entire drive.

What sandboxing doesn't protect is worth noting, too. Data inside the allowed project directory is still fair game: the agent can corrupt your source code as enthusiastically as it likes within its boundaries. Project-embedded execution paths like git hooks and build scripts remain a subtle attack surface: a compromised agent could modify your pre-commit hook, and the malicious code would execute later under *your* authority, outside the sandbox, when you run `git commit`. MCP servers present a similar boundary problem. These are real gaps, but they're scoped gaps, fundamentally different from "the agent has the same permissions as my user account on the entire machine."

The direction is clear, even if the destination isn't fully built yet. Sandboxing a single agent is a stepping stone to sandboxing agent swarms, where isolation becomes not just a safety measure but an architectural primitive. Each agent in Alice's 12-minute pipeline from section 3.5 should run in its own disposable environment, with precisely scoped access to the resources it needs and nothing else. We're not there yet. But `rmdir /s /q` is a pretty compelling motivator to get there fast.

The alternative—agents operating directly on systems that matter—is increasingly untenable as autonomy increases. But the lesson is that sandboxing alone isn't sufficient. The boundaries need to be airtight, which means either accepting more friction or accepting more risk. There's no third option.

3.6.7 *Tradeoffs of swarms of agents*

Agent orchestration represents the current frontier of AI-assisted development. The productivity potential is genuine: parallel execution, specialized agents, automated coordination. Work that took days of manual agent interaction can complete in minutes of orchestrated execution. But the complexity explodes in ways that aren't obvious until you're debugging a failed pipeline at 2:00 AM.

With a single agent, failure is straightforward: the agent produced the wrong output. With five agents interacting through artifacts, failure becomes archaeological. Agent A produced subtly wrong output. Agent B, trusting that artifact, amplified the error. Agent C propagated it into a state that agents D and E couldn't recover from. By the time you notice, the causation chain crosses four agent boundaries and three artifact handoffs. Good luck with that stack trace.

The control mechanisms that worked for earlier eras start to break down. Policy enforcement gets harder because you're no longer predicting individual agent behavior; you're predicting interactions between agents, which multiply combinatorially. Observability must track causation across agent boundaries, not just within them. Roll-back must handle partial states that span multiple agents' work: what does "undo" even mean when three agents have each modified different files based on a shared but now-invalidated assumption?

The teams successfully using orchestration share a pattern: they treat it as operations, not magic. They keep the scope conservative, orchestrating well-understood workflows rather than novel explorations. The pipeline for "implement feature from design doc" is predictable. The pipeline for "figure out what's wrong with this system" is not, and they don't pretend otherwise.

You need to invest heavily in instrumentation, building observability before scaling autonomy. Dashboard first, orchestration second. If you can't see what's happening, you can't control it.

3.7 *Where are we now?*

We've traced the arc from inline autocomplete to orchestrated agent fleets. Each step solved real problems while introducing new ones, which, if you've been paying attention, is the recurring punchline of this entire chapter.

But zoom out from the individual steps, and a bigger picture emerges. There's a term gaining traction in early 2026 that captures where all of this is heading: the *software factory*. Not a metaphor for a well-run team, but something closer to its literal meaning: an autonomous production system in which agents handle the end-to-end lifecycle of code, from requirement decomposition through implementation, testing, and deployment. A system where humans don't write the code, don't review each diff

individually, and don't babysit each agent's terminal session. Instead, they design, operate, and maintain the factory itself.

We're not there yet. Let's be honest about what the current moment actually looks like: most teams in early 2026 are firmly in the agentic IDE era. They've adopted Cursor or Cline, they're clicking Allow 30 times a day, and they're developing that particular thousand-yard stare that comes from watching a progress spinner while pretending to read Slack. Some have ventured into headless, running agents in CI pipelines with varying degrees of success and varying degrees of "Who approved that commit at 3:00 AM?" A few pioneers are experimenting with orchestration and discovering that coordinating five agents is less like managing a team and more like herding cats that can each independently execute `rm -rf`.

This is not yet the era of "describe what you want, come back to merged code." But it's not the era of autocomplete, either. We're in the awkward teenage phase: powerful enough to be dangerous, not mature enough to be trusted unsupervised. And honestly? That's fine. Every useful technology goes through this stage.

But here's what we find genuinely interesting about this moment. If you read back through the progression we've covered—from completions to chat, from chat to agentic loops, from single agents to headless pipelines, from pipelines to orchestrated fleets—each transition moved the human further from the act of producing code and closer to the act of managing the system that produces code across the software development lifecycle. That is the visible trajectory, and the software factory is simply its logical endpoint.

The gap is operational. It's the distance between "We could orchestrate five agents on this feature" and "We have the SDLC infrastructure—sandboxing, rollback procedures, observability, and incident response—to survive when agent number three decides to get creative." Think about what that means for the role of a software engineer. Not someone who writes functions, but someone who defines quality gates, designs rollback procedures, builds observability into agent workflows, tunes the feedback loops between orchestrated components, and intervenes when the system's output drifts.

Summary

- Inline autocomplete is fast, but it still lacks enough project awareness to understand architecture as well as a more capable agent can.
- As agents gain more visibility into the codebase, they also require more human supervision to keep that power under control.
- Too many approval dialogs create approval fatigue, and that turns oversight into a rubber stamp instead of real control.
- Headless agents scale beyond the IDE, but they can fail quietly unless teams build proper observability around them.
- Security automation cuts both ways, because agents can introduce risky dependencies, but automated checks can also help catch them.

- Team standards like AGENTS.md improve consistency, but they also introduce governance and maintenance overhead.
- Small teams often do not need that much process, but large or regulated organizations usually benefit from it.
- Orchestration can reduce waiting time through parallel work, but it also makes debugging, rollback, and coordination much harder.
- Sandboxing adds friction, but it is worth it because it limits the blast radius when agents make mistakes.
- There are no shortcuts; each stage of agentic development needs the controls and foundations of the previous one.



VISDOM: The Autonomous Software Factory

Your AI agents are fast
Your infrastructure isn't

VISDOM turns AI-driven software delivery into **an agent-operable production system.**

It removes the structural friction in context, execution, and validation that prevents agents from **working reliably at scale.**

Scan the QR code
to learn more:



VIRTUSLAB.COM