EXECUTIVE BRIEFING · SOFTWARE DEVELOPMENT ECONOMICS · 2026

# Does AI Actually Save Money on Software Development?

*The truth about AI-assisted software development costs. Written for the people who sign the checks.*

**David Logan**
Your Software CTO · david@yoursoftwarecto.com

THE VERDICT

Yes, AI cuts software development costs. Early on, it cuts them dramatically. **It then comes back to collect. With interest.**

The evidence for this is not anecdotal. Evidence from repository studies and enterprise DevOps research suggests the initial cost advantage of AI-assisted development can erode within one to two years in poorly governed environments. Repository analysis and delivery metrics consistently show rising maintenance pressure, code complexity, and rework rates over time, patterns that can offset early savings if not actively governed. Across multiple independent research streams, these patterns appear with enough regularity to warrant serious planning.

**AI does not solve your biggest software problem. It makes your smallest problem cheaper and your biggest problem more expensive.**

***Consider what triggered this paper:*** *a former software CEO estimated in the New York Times that a data project he completed over a weekend for $200 would have cost a client $350,000 at his old firm. That math is real. The cost of producing functional software has fallen dramatically. Understanding what that actually means for your budget is what the rest of this paper is about.*

KEY FINDINGS AT A GLANCE

**45%**
Security flaws introduced
in controlled testing, 100+ models
*Veracode 2025 GenAI Report*

**19%**
Slower on real tasks
vs. developers' own estimate of 20% faster
*METR 2025 Randomized Controlled Trial*

**10×**
More duplicated code blocks
since AI coding became standard
*GitClear 2025, 211M lines analyzed*

**Significant**
Maintenance burden increase
in ungoverned codebases: several-fold rises
in churn and duplication documented
*GitClear 2025 + DORA 2025*

**What these numbers indicate:** The early productivity gains from AI are real, but they come with measurable trade-offs. Security flaws are more common in AI-generated code. Experienced developers often work slower with AI than without it on complex tasks. And duplicated code accumulates at a rate that compounds future maintenance costs. The gains show up in the first months. The costs show up in the second year.

**WAIT. SLOWER AND CHEAPER?**

Yes. AI writes boilerplate code extremely fast, which is where the cost savings come from early on. But on complex tasks requiring judgment, experienced developers outpace AI significantly. The 19% slowdown is the average across all work. The savings are real at the start. The slowdown compounds over time. AI reduces the cost of producing code, but it does not reduce the cost of understanding and maintaining it.

There is a second reason the productivity math surprises people. The Microsoft Research Time Warp study (2024), which surveyed 484 software engineers across Microsoft to map how they actually allocate their working week, found that developers spend approximately 11% of their time actively writing new code. The rest goes to code comprehension, debugging, review, and coordination. AI primarily accelerates that 11%. The ceiling on productivity gains is inherently bounded by how much of the remaining 89% it can eventually address.

**ONE MORE NUMBER YOU NEED TO KNOW: THE 50-80% MAINTENANCE BASELINE**

It is not an AI number. It is a baseline that has been true across the software industry for decades.

In this context, maintenance means all the ongoing engineering work required after a product launches: fixing defects, updating dependencies as the software ecosystem changes, improving performance, integrating new systems, supporting users, adapting the product as the business changes, and evolving features over time. This is not helpdesk work or cloud bills. It is the continuing cost of keeping skilled engineers working on code that already exists.

For decades, studies have consistently found this work consumes roughly 50 to 80 percent of total lifetime software spending. Building the first version is the cheaper part. Keeping it working and useful is where most of the money goes.

AI does not eliminate this work. In poorly governed environments it increases it, by producing more code faster than any team can fully understand, document, or own.

# Where the Savings Actually Go

The collapse in production costs is real. What has not collapsed is the cost of everything else: deciding what to build, integrating it into a real business, keeping it secure, and maintaining it when the business changes.

### Budgets Move Rather Than Shrink

The result, based on early evidence, appears to be a budget shift rather than a budget cut. Deloitte's 2025 AI and Tech Investment ROI report, which surveyed large enterprises averaging $13.4 billion in revenue, found digital budgets rising from 7.5% of corporate revenue in 2024 to 13.7% in 2025 in those organizations, even as the cost of writing individual lines of code approaches zero.

### Labor Shifts Toward Senior Work

The composition changes significantly. Capital moves away from entry-level coding labor and toward architecture, security infrastructure, quality assurance, and the senior technical judgment required to make AI-generated output production-worthy. Two independent 2025 studies document this shift. A Stanford working paper by Brynjolfsson, Chandar, and Chen analyzing ADP payroll records for millions of workers found a 13% relative decline in employment for workers aged 22-25 in the most AI-exposed roles since late 2022, while employment for workers over 30 in the same fields grew 6-9%. A Harvard working paper by Hosseini and Lichtinger analyzing 62 million résumés across 285,000 firms found that junior employment declined sharply at AI-adopting firms while senior employment continued to rise. Team composition is shifting away from historical ratios of roughly one architect to every eight to ten developers. Some organizations already report much smaller teams directed by senior engineers, with AI agents handling what junior developers once did.

### Prototype Costs Mislead

The gap between prototype cost and production cost is where most organizations get surprised. Deploying an AI system into production requires security hardening, compliance engineering, integration work, reliability infrastructure, and ongoing monitoring, none of which appears in the prototype budget. A proof-of-concept assembled by two or three developers can require a substantially larger team and budget to deploy, secure, and scale. According to a 2025 S&P Global Market Intelligence survey of more than 1,000 enterprises across North America and Europe, 42% of companies abandoned most of their AI initiatives that year, up from 17% the year prior, with the average organization scrapping 46% of proof-of-concepts before production. Escalating costs, data privacy concerns, and missing operational controls were cited as the primary obstacles.

### Ownership Is Where the Value Lives

Organizations that extract durable value from AI are not the ones that eliminated their engineering budget. They are the ones that redirected it, from the people who write code, toward the people who know which code should exist at all.

> **The budget shift in plain terms.** When production costs fall, the constraint does not disappear. It moves. AI has made writing code cheap. It has made the judgment about *what to build, how to connect it, and how to maintain it* more expensive and more consequential than it has ever been. That is not a warning against AI. It is a description of where the value now lives.

# What This Paper Is Based On

The conclusions in this paper are drawn from five independent research streams. No single study proves every claim. The convergence across multiple methodologies is what gives the findings credibility.

**GitClear Repository Study**

211 million lines of changed code analyzed across 2020-2024, drawn from enterprise repositories and 25 major open-source projects. Tracks code churn, duplication rates, refactoring frequency, and maintainability indicators over time. Primary source for code quality trends and duplication data.

**METR Randomized Controlled Trial**

Peer-reviewed RCT published July 2025. 16 experienced developers, 246 real tasks, randomly assigned to AI-allowed or AI-disallowed conditions. Used Cursor Pro with Claude 3.5/3.7 Sonnet. Primary source for the 19% productivity finding and the perception vs. reality gap.

**Veracode GenAI Security Report 2025**

Tested more than 100 large language models across 80 standardized coding tasks with known security-relevant patterns. Measured whether models chose secure or insecure implementations. Primary source for security vulnerability rates by language and model.

**Google DORA State of DevOps 2025**

Annual survey of 4,900+ development and operations professionals measuring software delivery performance, AI adoption, and delivery stability. Provides context for productivity phase patterns and delivery stability data.

**McKinsey Developer Productivity Research 2024**

Enterprise-level analysis of AI tool adoption across large development organizations, measuring output quality and long-term maintainability indicators alongside speed metrics.

**Enterprise Token Case Studies**

Documented deployments including Anthropic and Cloudflare engineering case studies on agent token optimization, and enterprise deployment examples including the multi-agent pipeline projection cited in Section 04.

01

# The Claims You Keep Seeing

On any given day, LinkedIn is full of developers and CEOs claiming they built entire applications with AI in a matter of hours, or entire data platforms over a weekend for $200 that would have cost $350,000 at a traditional firm. The New York Times ran that second story as though it settled the debate. These claims are everywhere, they all sound the same, and they spread because they are partially true. The $200 weekend project was built by an experienced developer who already knew exactly what to build, working on something self-contained with no production users, no compliance requirements, and no integrations. It is a ceiling, presented as a floor. Here is what all of these claims are actually describing.

**A TYPICAL "30,000 LINES IN 7 HOURS" CLAIM: WHAT IT SAYS VS. WHAT IT MEANS** REALITY CHECK

**WHAT GETS POSTED**

- 30,000 lines of code
- 7 hours total
- 80 files
- "Almost no issues"

**WHAT GETS LEFT OUT**

- **88 problems found in review** were listed in that same post. Industry benchmarks put well-written human code at roughly one defect per 1,000 lines. This post came in at one per 341 lines. Nearly three times the defect rate.
- Repository studies consistently show that the large majority of AI-generated volume is structural setup: folder organization, standard connections between systems, configuration files, and test templates. The actual business logic, the part that makes a product work differently from any other product, is a much smaller fraction.
- Token costs, review hours, and the long-term maintenance bill are absent from nearly all of these posts.

AI is genuinely fast at building structure: the skeleton of an application, the repetitive setup that takes a team days, weeks, or sometimes months to assemble. That speed is real and it matters. The business logic that makes a product unique, the security requirements that keep it safe, and the code quality that keeps it maintainable two years from now are a completely different story.

Nearly all of those posts describe the skeleton. Very few ever discuss the body.

**AI Builds the Skeleton**

fast, cheap, impressive at first glance

**You Still Own the Body**

the logic, the security, the maintenance, the evolution

02

# What Actually Happens Over Time

AI projects that run without deliberate structure follow a recurring pattern documented across multiple organizations and repository studies. Well-governed teams can avoid or significantly flatten this curve, but the underlying pressures remain present regardless of intent. The shape of this cycle should concern anyone responsible for a software budget.

**WELL-GOVERNED TEAMS**

Can flatten or avoid this curve entirely with deliberate planning and code ownership practices.

**POORLY GOVERNED TEAMS**

Tend to accelerate through this curve, often without realizing what is happening until they hit The Wall.

MONTHS 1-3

## Euphoria

Features ship fast. Visible output is enormous. Leadership sees the numbers, calls it a success, and often doubles down. This is when those posts get written.

**New feature output: ~150-200% of traditional**

MONTHS 4-9

## Plateau

New features take longer. The codebase is growing faster than anyone understands it. Senior engineers start spending time untangling AI-generated logic instead of building new things.

**New feature output: ~50-70% of traditional**

MONTHS 10-15

## Decline

Routine fixes become investigation projects. Developers spend more time reading and deciphering code than writing it. Ownership of any part of the system becomes unclear.

**New feature output: ~20-40% of traditional**

MONTHS 16-18+

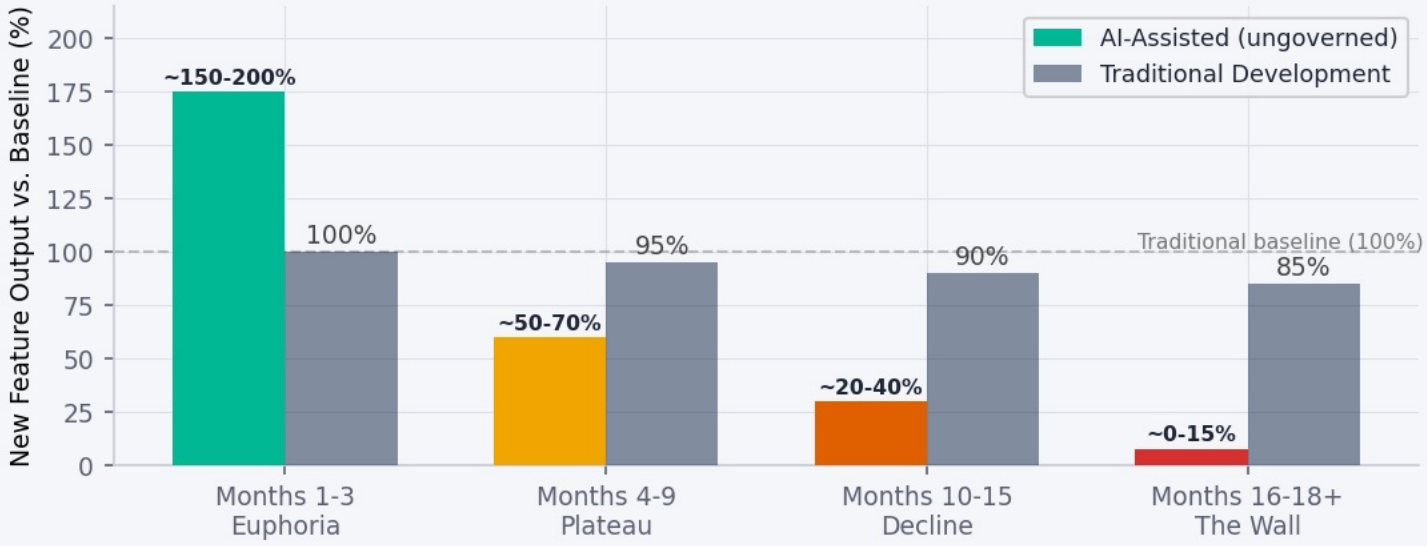
## The Wall

In ungoverned environments, delivery can slow to a near standstill. The team becomes reluctant to modify a codebase they do not fully understand. Each change risks cascading failures. This is where the maintenance cost expansion arrives in full.

**New feature output: ~0-15% of traditional**

**Illustrative Pattern: Feature Output Over Time in Ungoverned AI Projects**

*Conceptual model synthesized from GitClear 2025 repository studies, DORA 2025 State of DevOps, and McKinsey enterprise productivity research. Ranges are indicative, not directly measured. Individual projects vary.*




*Note: these figures measure new feature delivery rate only, not total budget. The separate question of where total budget goes (new work vs. maintenance) is shown in the cost chart in Section 03. Traditional development holds steady because the codebase stays understood and navigable. AI output starts dramatically higher and degrades as the codebase outgrows anyone's ability to reason about it.*

**This problem has acquired a name inside engineering organizations: Comprehension Debt.** Traditional technical debt reflects deliberate shortcuts taken under time pressure. Comprehension debt arises when large volumes of machine-generated code function correctly but are not deeply understood by any engineer responsible for maintaining them. When modification becomes necessary, the cost of rediscovering the machine's logic often exceeds the cost of writing the system deliberately in the first place.

The downstream effects show up in code review. Usage data from Faros AI, analyzing throughput across enterprise engineering organizations, found that AI usage increases pull request sizes by an average of 154%. Because human reviewers lose cognitive effectiveness after roughly 80 to 100 lines of code, these oversized AI-generated submissions cause peer review times to spike by 91%. Bug rates climb 9% as quality gates struggle to process the sheer volume of machine-generated diffs. Salesforce Engineering observed the same pattern: as code volume grew 30%, review times for the largest submissions actually began to decline, a signal that engineers had stopped meaningfully engaging with the code at all. The code was being merged. It was not being understood.

***"Why not just feed the broken code back into AI and have it fix itself?"***

It is the obvious question and it sounds completely logical. The problem is that AI produced a bug because it did not fully understand the requirement the first time. Feeding the broken code back gives it the same incomplete picture and asks it to fix something it could not get right on a clean slate. Each pass adds more code that was not written deliberately and that no engineer fully owns. The codebase gets larger, the understanding gets thinner.

At some point a human developer has to go in, understand what the AI was trying to do, and fix it correctly. At that point the question answers itself: would it have been faster to have the developer write it in the first place?

03

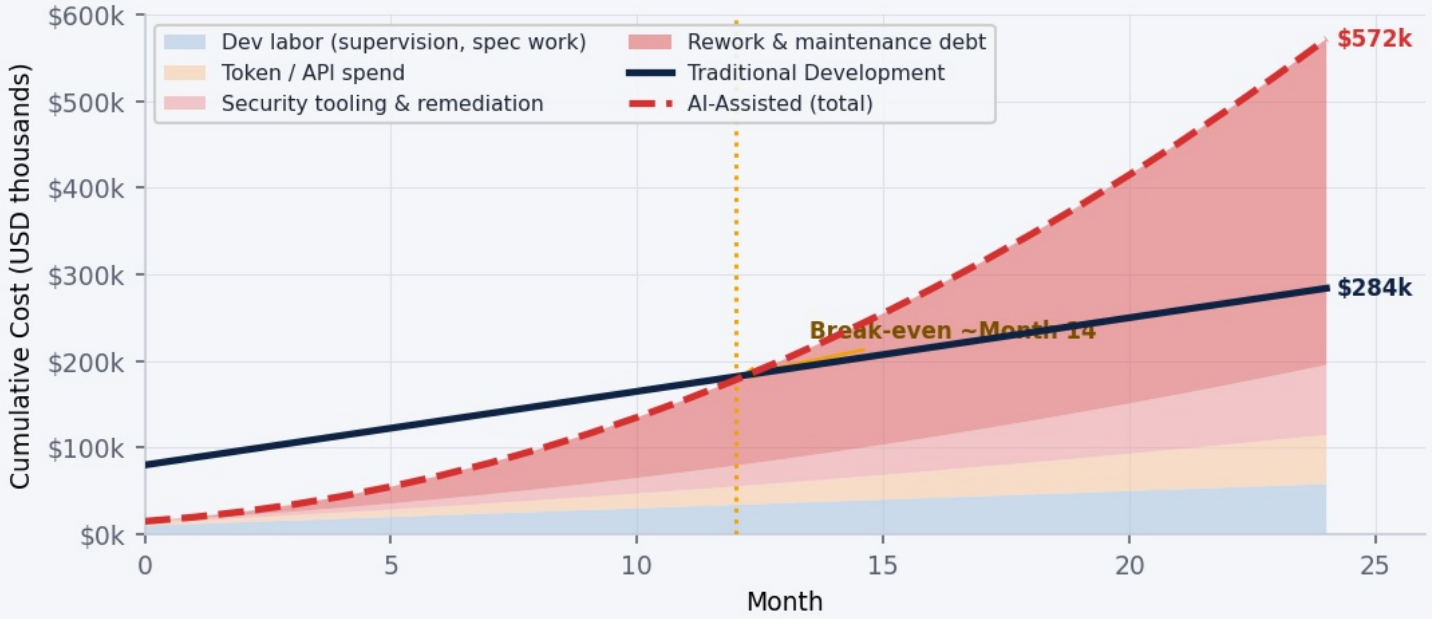
# The True Cost Picture

Software development has three distinct cost phases. AI changes the math on all three, but not in the same direction.

| Phase | When | Traditional Development | AI-Assisted Development | Verdict |
|---|---|---|---|---|
| BUILD **Getting to a working product** | Months 1-6 | High upfront cost. Skilled engineers work in full sprints. The code that comes out is understood, structured, and maintainable. | Dramatically lower. Repetitive setup generates in hours. A working prototype is genuinely faster to reach and cheaper to produce. | **AI wins** |
| RUN **Keeping it operational** | Ongoing | Predictable flat-rate tooling costs. No consumption-based surprises. | Usage-based billing that scales with every interaction. Usage costs can scale rapidly if not carefully governed, and some organizations report unexpectedly large consumption spikes. | **Unpredictable** |
| OWN **Maintaining and evolving it** | Year 1 onward | Maintenance accounts for 50-80% of total lifetime cost, a figure that has been true for decades. But engineers understand the code and can work in it confidently. | In ungoverned AI codebases, GitClear's longitudinal repository analysis and DORA's 2025 deployment data indicate maintenance overhead commonly grows to several times traditional levels by the second year, though the degree varies considerably by organization and codebase. The code grew fast but few engineers fully understand it. Finding one defect means locating every copy across a codebase that was never systematically mapped. | **AI loses** |

### Cumulative Development Cost Over 24 Months

*Where the AI cost actually goes. Shaded areas show the components driving AI spend past the traditional line. Derived from GitClear 2025 (211M lines), METR 2025 RCT, and Veracode 2025 GenAI Security Report.*




*Derived from GitClear 211M-line study, METR 2025 randomized controlled trial, and Veracode 2025 security report. The AI line starts dramatically lower. In poorly governed environments the AI cost curve can cross the traditional curve within one to two years and may continue rising thereafter.*

04

## The Hidden Tax: Tokens

Most software tools cost a flat monthly fee per seat. AI tools do not work that way. Every action an AI agent takes is billed: reading a file, answering a question, attempting a fix, retrying after a failure. These are called tokens, and they are the unit of consumption for every AI interaction. Token costs are variable, non-intuitive, and easy to dramatically underestimate before you are committed to production scale.

**TRADITIONAL TOOLS**
Fixed monthly cost per seat. Predictable. Budgetable. No consumption surprises.

**AI SYSTEMS**
Costs scale with every action taken. One poorly configured session can exceed a month of prior spend.

In one enterprise deployment of a multi-agent automation pipeline, a proof-of-concept that cost approximately $50 in API usage during testing was projected to exceed **$2.5 million per month** at full production scale, due to unoptimized token consumption and retry loops. The system was functioning as designed. The cost model had simply never been projected at scale before the organization committed to the architecture.

Most organizations running AI agents today are nowhere near optimal configuration, and there is currently no industry playbook for

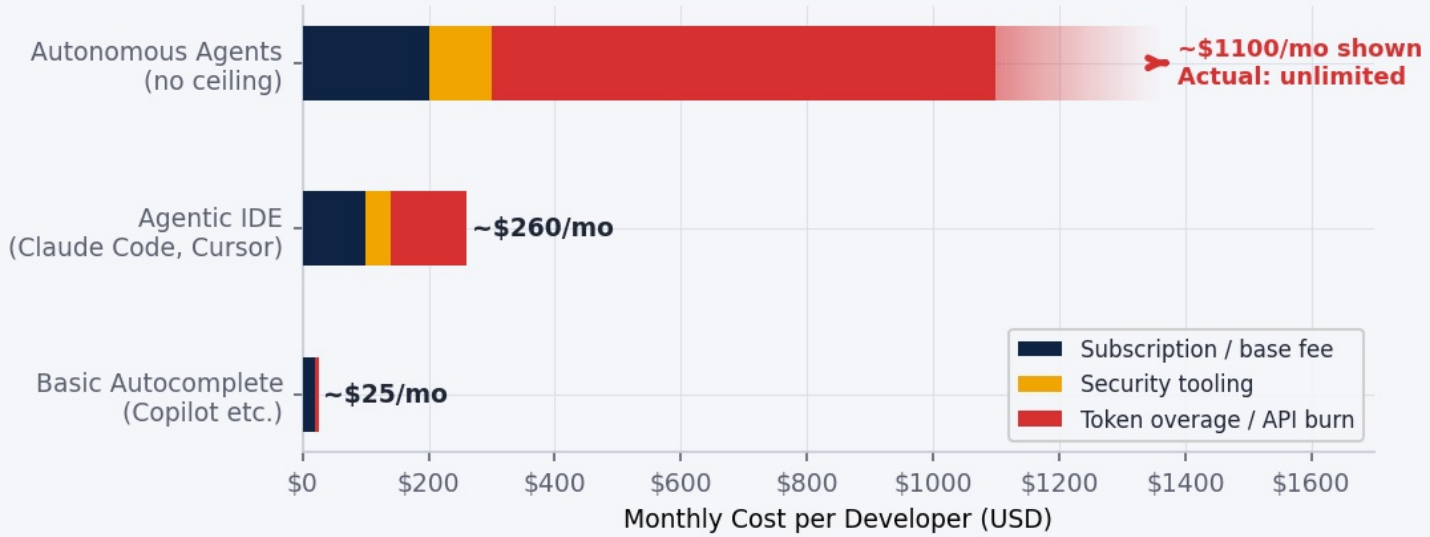
getting there. It is largely discovered by trial and error, on your budget, on your timeline, by developers who are already under pressure to ship and are now also expected to become token optimization experts on the side. There is no course for this. There is no playbook. There is no one to call.

**WHAT THIS ACTUALLY COSTS A MID-SIZE TEAM**

| | |
|---|---|
| 12 developers on agentic IDE tools (mid-range estimate per developer) | **$240/mo each = $2,880/mo** |
| Token overage and API burn (conservative estimate) | **$120/mo each = $1,440/mo** |
| Security tooling (mandatory, not optional) | **$40/mo each = $480/mo** |
| **Total annual AI infrastructure cost added on top of existing headcount** | **$57,600/year. Before a single autonomous agent runs.** |

**Monthly Token Cost per Developer, by Tool Type**

*What you are paying on top of your existing development costs, before any ROI calculation. Cost ranges from Anthropic, Cursor, and GitHub Copilot published pricing; agent overage from documented enterprise deployments.*




*These numbers are lower than the monthly cost of a single developer. That is not the point. Token costs are additive. They sit on top of your existing headcount costs. And for autonomous agents, there is no natural ceiling and no built-in shutoff.*

| Tool Type | Monthly Cost / Developer | What Drives the Spend | Budget Risk |
|---|---|---|---|
| **Basic autocomplete** *GitHub Copilot etc.* | $10-$30 | Flat subscription. Single-file suggestions only. Predictable. | **Low** |
| **Agentic IDEs** *Claude Code, Cursor, Windsurf* | $100-$300+ | The AI reads entire codebases for context and runs retry loops when something fails. One session configured poorly can cost as much as a full month of prior usage. | **Medium-High** |
| **Autonomous agents** *AutoGPT-style pipelines* | **$1,000-$50,000+/mo at scale** | Goal-chasing loops with no human oversight. If an agent gets stuck attempting the same wrong fix repeatedly, it keeps billing until something stops it. Nothing stops it by default. | **Extreme** |

05

## Where It Actually Works

The organizations extracting real, durable value from AI are not the ones using it most aggressively. They are the ones who drew a clear line around what AI is and is not responsible for, and held it.

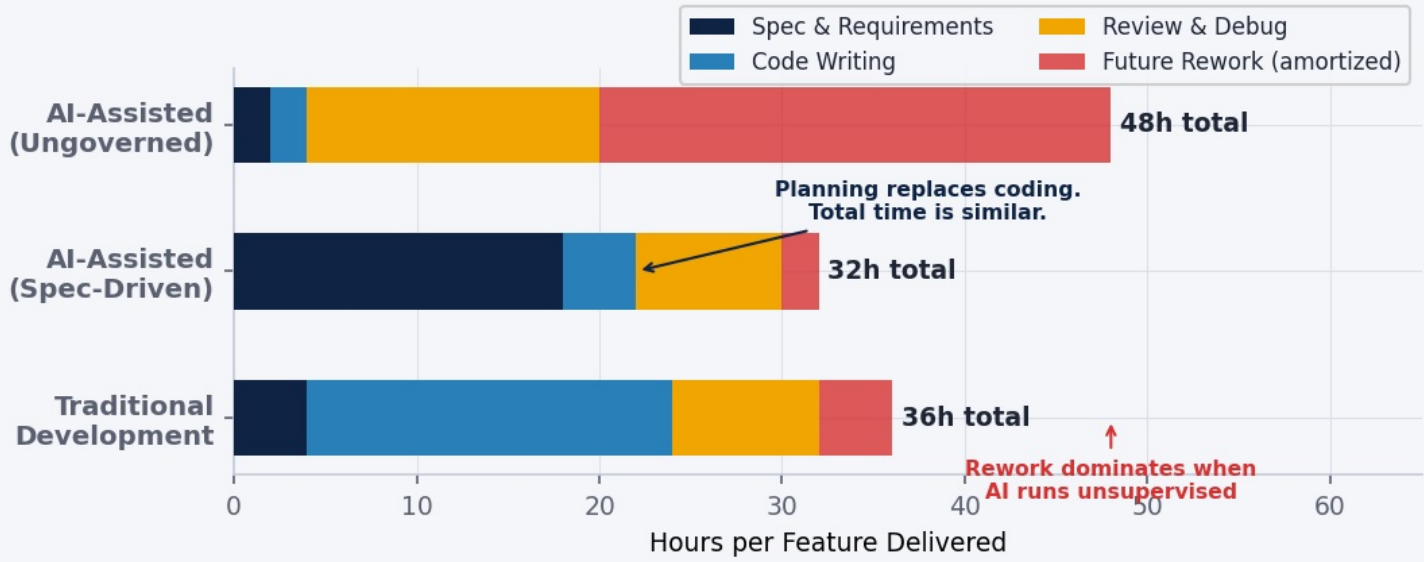
| Company | How They Used AI | Measurable Result |
|---|---|---|
| **Uber** | Built an internal AI tool to review code, not write it. It now flags problems in 90% of 65,000 weekly code submissions automatically, before a human reviewer sees them. A separate AI tool generates unit tests, the repetitive verification work that developers find tedious and often skip. | **21,000 developer hours saved. Test coverage up 10%. AI never touches core business logic.** |
| **Shopify** | Runs controlled experiments: one group uses AI tools, a control group does not. Measures the difference in actual revenue generated, not code volume. Refuses to count lines of code as a success metric. | **Results were positive enough that Shopify expanded the program. They are one of the only organizations measuring AI in dollars, not output volume. The discipline of how they measure is the real lesson.** |

## The Development Model That Produces Durable Results

Every organization extracting sustainable value from AI coding tools shares one practice: they write a thorough plan before the AI generates a single line of code. An experienced architect will find AI genuinely useful as a collaborator in that planning process: as a thinking partner, a stress-tester of assumptions, a fast generator of options to evaluate. But the human has to be driving. The AI does not own the decisions. It accelerates the person who does.

### Where the Hours Actually Go: Three Approaches Compared

*Hours per feature delivered, broken down by how time is actually spent. Derived from GitClear 2025 repository study and DORA 2025 State of DevOps.*




*Spec-driven AI development shifts time from writing code to writing the plan. Total hours per feature are similar to traditional development. Ungoverned AI looks fastest at the start, until the accumulated rework from code that was never fully understood starts dominating the budget.*

**THE ONLY FLOW THAT SCALES**

**Write the Plan** → **Make Architecture Decisions** → **Define What Each Part Does** → **AI Generates the Code** → **Human Reviews and Ships**

*Time saved writing code is reallocated to planning. Total calendar time to ship is similar. The codebase is actually maintainable.*

A concrete illustration: I am currently building a home-assistant-driven thermostat as a personal side project. One person, no users, no production requirements. Before writing a single line of code, I spent two full days doing nothing but planning work with AI, producing a 1,000-line requirements document and a task tracker with hundreds of items. That upfront investment is what allows AI to cycle through the implementation reliably.

To put the speed gain in perspective: a senior developer completing one test case on that project (writing the test, writing the code, compiling, debugging, and updating the documentation) takes 30 to 60 minutes. AI completes the same test case in 6 to 7 minutes. That speed gain is real. But it is the middle of the equation, not the whole thing. The LinkedIn posts show you the fast middle. Your AI vendor's ROI calculator shows you the fast middle. What neither of them shows you is the two days of planning work that made it possible, or the maintenance bill that will arrive at the back end. Every calculation you have seen about AI savings is built on the middle section only. The full equation looks different.

If a thermostat requires two days of planning before AI can be trusted to build it, consider what a business-critical application requires. Or a system with real users. Or anything that has to keep working in eighteen months. The planning work is not optional, it is not free, and it will not appear anywhere on the ROI calculator your AI vendor sent you.

01

### Aim AI at Low-Risk, Repetitive Work

Test generation, code review, and structural setup have proven, measurable ROI. Core business logic, security-sensitive code, and anything connecting to production systems without human sign-off should stay in human hands.

02

### Measure Output, Not Volume

Lines of code are the wrong metric. Track rework rate: how much AI output gets rewritten within 30 days. Frameworks like DORA measure what actually matters: how often you ship, how often it breaks, and how fast you recover.

03

### Treat Tokens Like a Budget Line

Know your actual token spend per developer per month. It is an additional cost on top of your existing headcount. It grows as usage scales. Most organizations are not managing this effectively yet. Most do not know their own number.

**FOR THE TECHNICALLY MINDED**

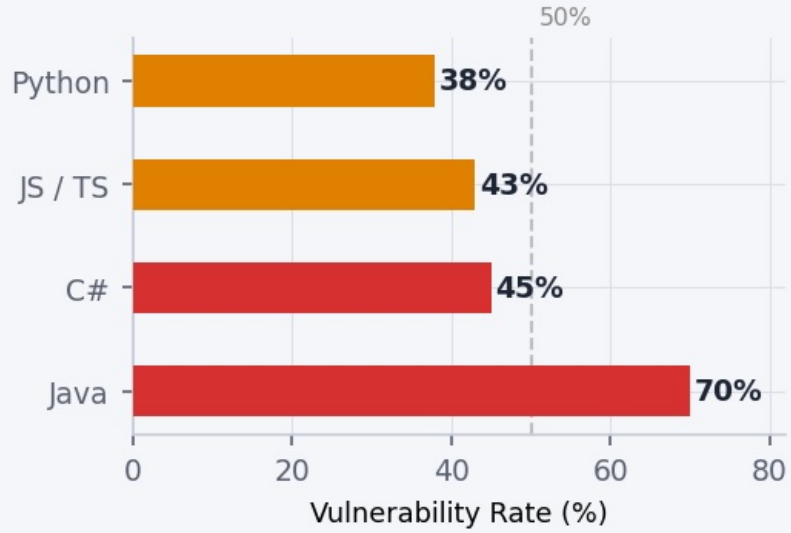
*The following section goes deeper on code quality and security. The business-level summary is above.*

06

## The Security and Code Quality Problem

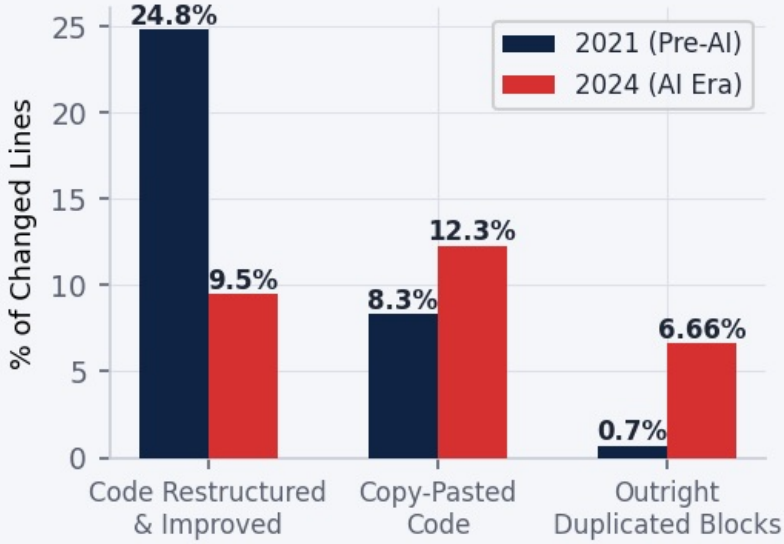
**Security Failure Rate by Language**

*% of AI-generated code with critical vulnerabilities (Veracode, 100+ LLMs tested)*




**Code Quality Before and After AI**

*GitClear study, 211 million lines of code analyzed, 2021 vs. 2024*




According to Veracode's 2025 GenAI Code Security Report, which tested more than 100 large language models across 80 standardized coding tasks, AI systems introduced OWASP Top 10 vulnerabilities in 45% of test cases when given the choice between a secure and insecure implementation. Critically, this was not a study of deployed production code: it tested whether models, when prompted to complete code with known security-relevant patterns, chose the secure path. Nearly half the time they did not. This rate held consistent regardless of model size or training sophistication. Larger and more expensive models performed no better on security than their predecessors.

> **The cost of fixing a security flaw scales dramatically with when it is found.** This is a long-established principle in software engineering, predating AI entirely. Caught during design, a flaw costs hours to address. Found post-deployment, it costs weeks of engineering time plus potential legal exposure and breach notification obligations. AI accelerates the creation of vulnerable code while largely eliminating the human review stages that traditionally catch these issues early.

The code quality picture is equally stark. Analysis of 211 million lines of changed code shows that since AI coding became standard, the amount of code being thoughtfully restructured and improved has dropped by more than half, while copying the same logic into multiple places has increased by 48% and outright duplication of code blocks has grown nearly ten times.

Both problems are addressable with the right tooling and discipline: strict test-driven development enforces that code has to work before it is accepted, and automated code analyzers catch duplication before it accumulates. These are not exotic solutions. They are standard practices that most AI-assisted teams are simply not applying.

**When a bug is found in duplicated code, every copy must be located and fixed separately.** Either a developer hunts it down manually across a codebase that grew faster than anyone mapped it, or you spend tokens having AI find and patch each copy. Neither option is free, and neither gets cheaper as the codebase grows.

*

**BOTTOM LINE**

MIT Sloan research evaluating AI adoption across industries identified a J-curve pattern: organizations frequently experience a measurable, temporary decline in performance as legacy workflows are disrupted and the complexity of integration sets in. The short-term friction is real and should be planned for. But organizations that endure it and invest in capability-building consistently show stronger long-term output and revenue growth than non-adopters. The expectation of immediate, frictionless return on investment is not aligned with the historical reality of any general-purpose technology adoption.

Whether AI saves money on software development depends entirely on your time horizon. If you are building a short-lived prototype or a project with a defined end date, the economics are genuinely favorable. **If you are building something you intend to maintain, grow, and depend on for the lifetime of your business, the data is clear: AI development without deliberate structure costs more. Not eventually. By year two.**

The organizations doing this well are not the ones spending the most on AI. They are the ones who decided, deliberately, what AI is and is not responsible for. They treat it like a power tool: fast, capable, and prone to making the same mistake at tremendous scale if nobody is watching.

**David Logan** · **Your Software CTO** ·
**david@yoursoftwarecto.com**

GitClear 2024-25 (211M lines of code) · METR 2025 RCT (246 tasks, 16 developers) · Veracode GenAI Security 2025 (100+ LLMs)
DORA State of DevOps 2025 · McKinsey Developer Productivity Research 2024 · Anthropic and Cloudflare engineering case studies