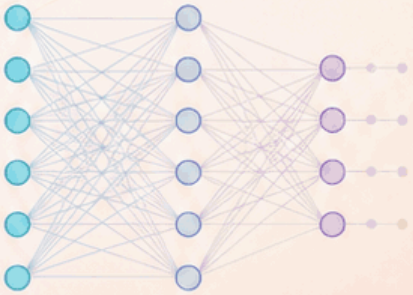
The Maintainability Gap

AI Code Quality in 2026

Usage has never been higher. Neither have the risks.

***623 million** analyzed changes from 2023-2026, tracking eight quality signals as AI authorship reaches record volume.*



William Harding, CEO & AI Researcher
GitClear and Alloy.dev Research
Published June 2026
GitClear AI Code Quality Research v2026.6.6
Sponsored by GitClear and GitKraken Research

GitClear GitKraken

Record AI Adoption Beset by Lingering Quality Concerns

Motivated developers are shipping more code than ever (1.3 times as much as they did before AI R2). But four years of code-change data consistently show maintainability signals sliding backward: refactoring line moves are down **70%,** while long-term legacy maintenance is down **74%** vs 2022 levels. Function invocations (indicative of cross-file connectivity) are down **35%**. Concurrently, we find increases in within-commit copy/paste (**+41%**), code block duplication (**+81%**), error-masking constructs (**+47%**), and near-immediate code churn (**+15%**, matching the rise in industry-standard 2-week churn). Zoom out, and it looks something like a blur of progress, trailed by a wake of sooty debris.

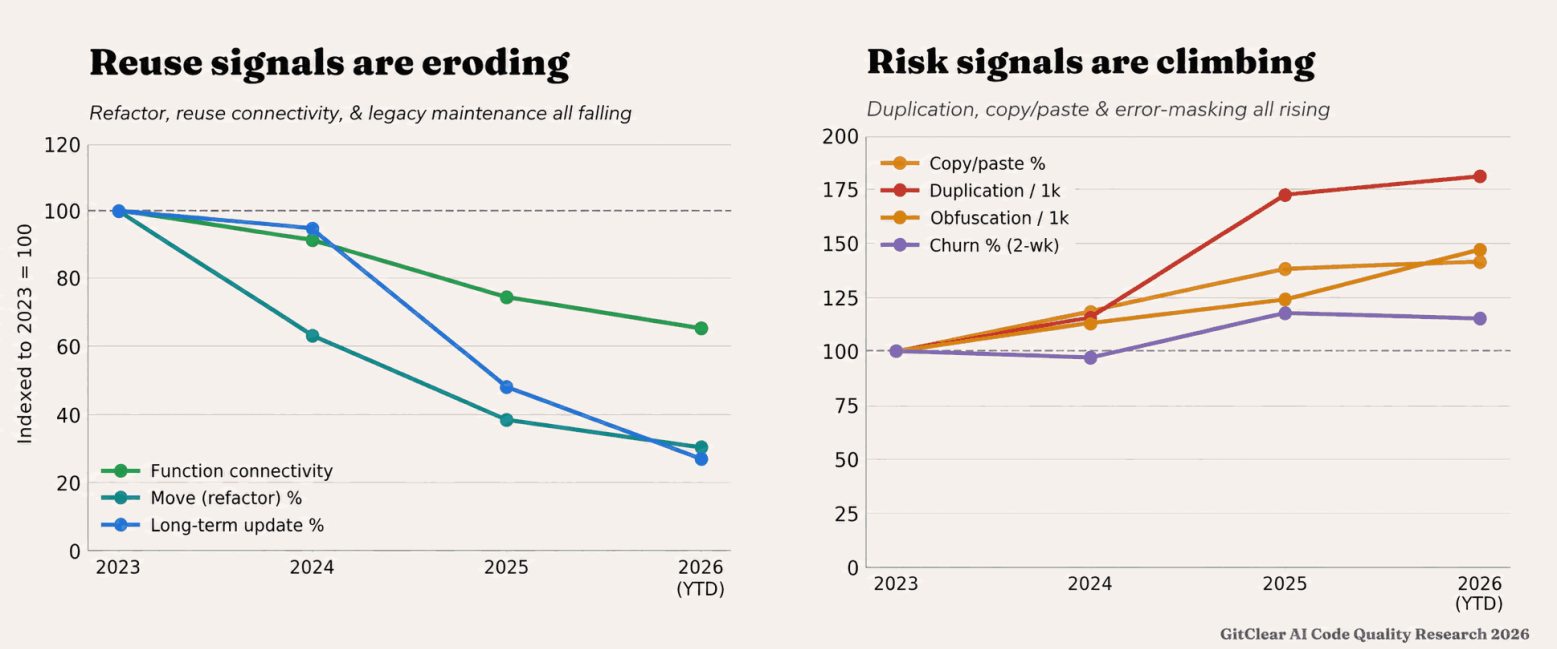
Our prior research, cited by MIT Technology Review and others, has tracked the recent decline of code maintainability. *AI Copilot Code Quality* (Feb 2025) documented that 2024 was the first year on record in which within-commit copy/paste exceeded "moved" (refactored) code. It also showed the share of commits containing a duplicated block had grown about **10x** over just two years. *AI Coding Tools Attract Top Performers* (Jan 2026) showed that heavy AI users out-produce non-users by 4–10x, but most of that gap predated AI. Compared to their past selves, heavy AI users enjoyed a more modest 25% gain.

In this addition to our canon of code quality research, we follow **eight code-quality signals** from 2023–2026, finding that most previous trends kept their momentum… alas, in the wrong direction.

The headline isn't "AI writes bad code." It's that today's coding agents are task-focused to the exclusion of valuable side quests. We request, and receive back, "atomic code": a happy-path feature, a passing unit test, a closed ticket. It looks like "business as usual," except that conscientious developers' "usual business" has been to habitually overdeliver. Prior to 2024, a natural extension of "being assigned a Jira" was to revisit the tangentially-related systems. Those revisits constituted a substantial portion of the total "system architecture" work a repo would receive.

Absent a conscious effort to fill the "thoughtful architecting" gap, modern repos become unwitting participants in the largest-ever experiment of "what if the primary driver of our system architecture is 'random chance'?"

The data in this report seeks to quantify the cost of shipping more while caring less.



Indexed to 2023 = 100. Reuse signals (left) and risk signals (right) diverge as AI authorship scales.

Where This Data Comes From

Every figure in this report is derived from GitClear's database of customers (opted into Industry Stats) & corporate open repos (e.g., Google, Microsoft, Apple). Unlike conventional git tooling, which labels each changed line as a binary "add" or "delete," GitClear resolves changes into a slightly more granular set of: *Added, Deleted, Updated, Moved, Copy/Pasted, Find/Replaced, and No-op*. R8

This finer resolution is what makes signals like "move (refactor) percent" or "copy/paste" possible. Then GitClear weaves back through each changed line's ancestry. This enables metrics like "Churn" and "Legacy update percent." More about our methodology available in A2.

The population size for each of the eight signals is shown in the table below. Counts are not uniform across signals because each metric is defined over the subset of changes where it is meaningful (e.g., function connectivity is computed only within files that contain functions or methods; churn is only calculated for customer repos). **2026 figures are year-to-date and drawn from a smaller, in-progress sample**; they should be read as a leading indicator rather than a closed annual figure.

Eight signals track facets of code quality

"Code quality" is a broadly-used term that lacks any consensus definition. In this regard, it parallels the largely-failed effort to converge upon a definition of "developer velocity." Both concepts must be defined as "what remains after applying rejection criteria to exclude the opposite."

To attempt to quantify "quality" or "velocity," the only practical option is to accumulate a list of exclusions.

Aside: Why must code quality be the sum of learned pieces?

Or: "Couldn't this be simpler?"

It would be easier for technology executives if they could succinctly define the characteristics of "high quality" or "high velocity" code. Unfortunately, the talent of their developers ensures this won't happen. It's much more practical to define (thus measure) what *isn't* high quality code than what *is*. Why?

If one tries to define what *is* high quality code (e.g., DRY code) or *is* high velocity code (e.g., lots of modified lines), the effort will run headlong into one of the foremost skills possessed by seasoned developers: *articulating edge cases*. There is probably no change that can unilaterally be labeled "high quality" or "high velocity" in every context.

That's what makes it more tractable to define types of change best *excluded* from "high quality" or "high velocity."

For the latter, GitClear defines "velocity" as the progressive exclusion of non-effecting code, ultimately distilling a 3% concentrate from the "changed lines count" a git provider would (and many do) report. For the former, staff and customers of GitClear have helped to identify LLM tendencies that, to the extent they occur, may be considered "anti-patterns."

Code Quality Metrics

Previous rounds of research have reported **Copy/Paste**, **Move**, **Churn** and **Duplication** benchmarks. In this round of research, we introduce three new metrics: **Obfuscations**, **Function Connectivity** and **Throwaway Percent**. For the deep-dive audience, the most detailed version of these metrics is presented in A5:

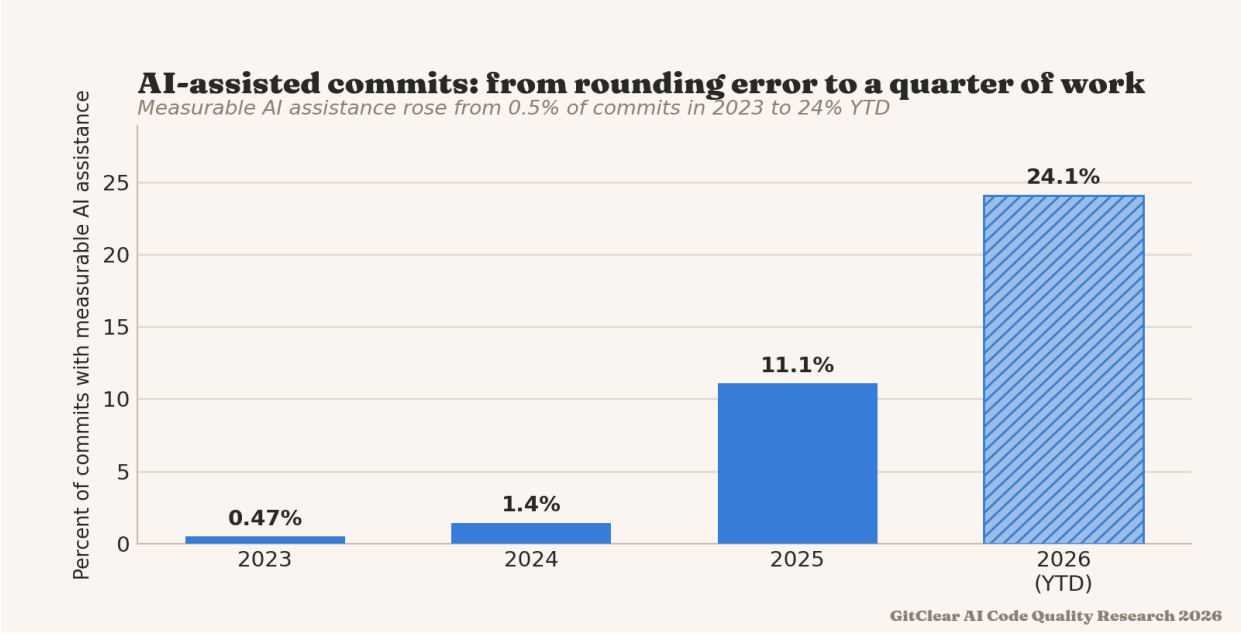
Signal	What it captures	More risk when
Copy/paste percent	Share of line-changes duplicating non-keyword lines within a commit, 3x or more	Higher
Duplication rate	Lines belonging to a 5+ consecutive-line duplicate block, per 1,000 meaningful lines changed	Higher
Obfuscation rate	Density of state-masking constructs, per 1,000 changes (rescue, catch, safe-navigation)	Higher
Churn percent	Share of changes revised or reverted within two weeks of authorship	Higher
Function connectivity rate	Density of connection points to code beyond the core implementation, per 1,000 changes	Lower

Signal	What it captures	More risk when
Move (refactor) percent	Share of changed lines relocating/consolidating existing functionality	Lower
Throwaway percent	Share of changed lines removed in less than a week	Higher
Long-term update percent	Share of changes revising code last touched 12+ months ago	Lower

Function Connectivity evaluates: when a developer writes new code, does it tend to utilize the existing codebase, or is it more of an island? Falling connectivity occurs when code is being written that leans toward "reinvent" over "reuse." **Throwaway Percent** is a special flavor of Churn that identifies the percent of code removed within a week of being pushed (standard "Churn" is two weeks). **Obfuscation** measures the rate of using constructs which hide the state of program execution. Specifically: *rescue*/*catch* blocks, safe-navigation operators (&. in Javascript), and stubbed methods. Each such instance 1) reduces the extent to which maintainers can know that an unexpected state is occurring 2) increases maintainer burden to discern valid/expected responses received from external code 3) creates uncertainty trying to evaluate how an error originates. In moderation these are expected. In bulk, they can become tantamount to sweeping unexpected behavior under the rug.

Adoption: stuck at "exponential"

Before interpreting the AI quality risk signals, it helps to understand the extent to which AI is influencing committed code, as of 2026:



Percent of commits with measurable AI assistance, by year. 2026 is year-to-date as of June.

GitClear's observation of 24.1% commits being AI authored in 2026 is consistent with sources like Handmadecode.dev reporting 25%. It's considerably lower than some sources R9. Since it averages an exponential trend, whatever month this is being read on is either "higher" or "much higher" than 25%.

This matters because it tightens the confidence intervals around every signal below: these are trends measured across millions of changes per year.

We've come a long way since 2024, when our Code Quality report was among the only sources of data on the freefall of refactoring R10. As more new organizations discover they share GitClear's interest in code quality indicators, new ideas to expand what risk signals this report can detect.

Risk Signals to Code Comprehension

The DORA and SPACE era of Developer Analytics has served technologists well over the past 10 years. They have ensconced metrics like "Release frequency" "Cycle time," and "Time to repair" into daily lexicon. Optimizing these metrics correlates with higher build stability, more predictable sprint velocity, and less customer attrition. We are grateful for those teams' work.

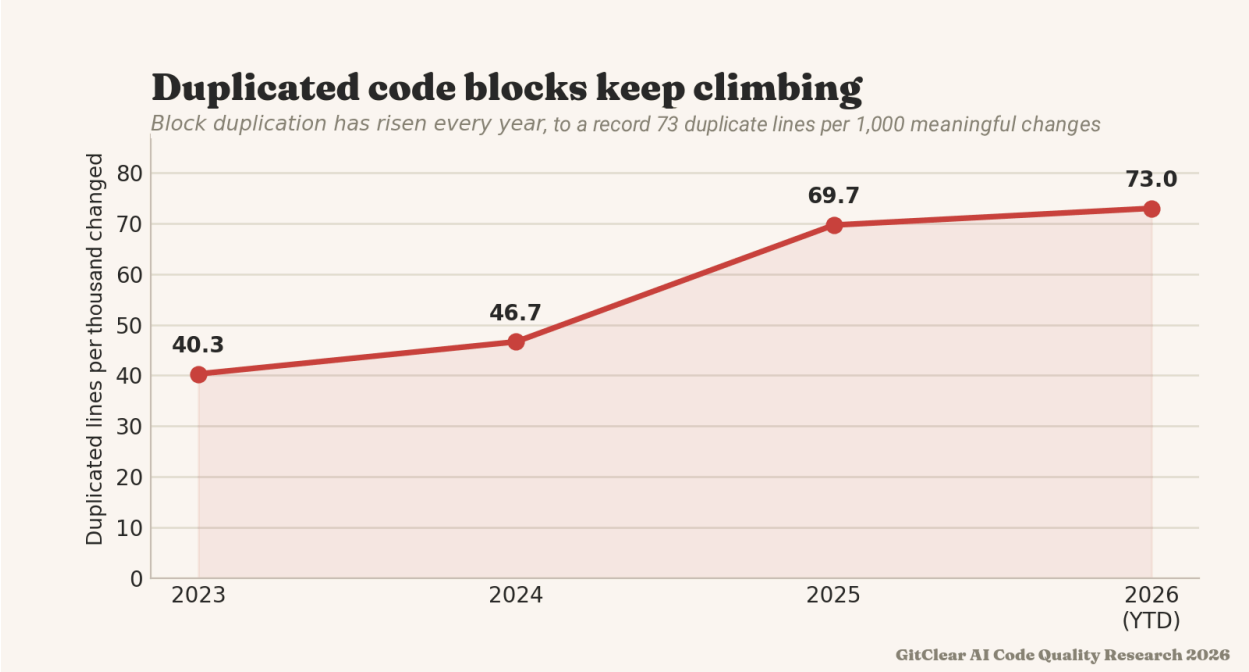
The classic DORA metrics like "Time to recovery" (MTTR) gesture toward the originating source of what can make AI-authored code dangerous. A long time to recover is a common symptom of a critical problem. But a different set of metrics than DORA are needed to detect that problem in advance.

GitClear's theory: "Long recovery time" and "High defect rate" are both downstream of a poorly understood repo. The repo problems often manifest as a mix of "hotspot" files and "V1 implementation" files that give rise to a stream of unpredictable problems. These files breed bugs by thwarting maintainers efforts to predict their behavior.

The rest of this section offers empirical reasons why code authored in recent years has become increasingly difficult to comprehend.

Block duplication climbing

Duplicated code blocks (regions of five or more consecutive repeated meaningful lines A2 A5) have risen every year of the window. Measured per million changed lines, block duplication climbed from 40.3 in 2023 to 46.7 in 2024, 69.7 in 2025, and 73.0 year-to-date in 2026 — an 81% increase over 2023 and the highest level on record.

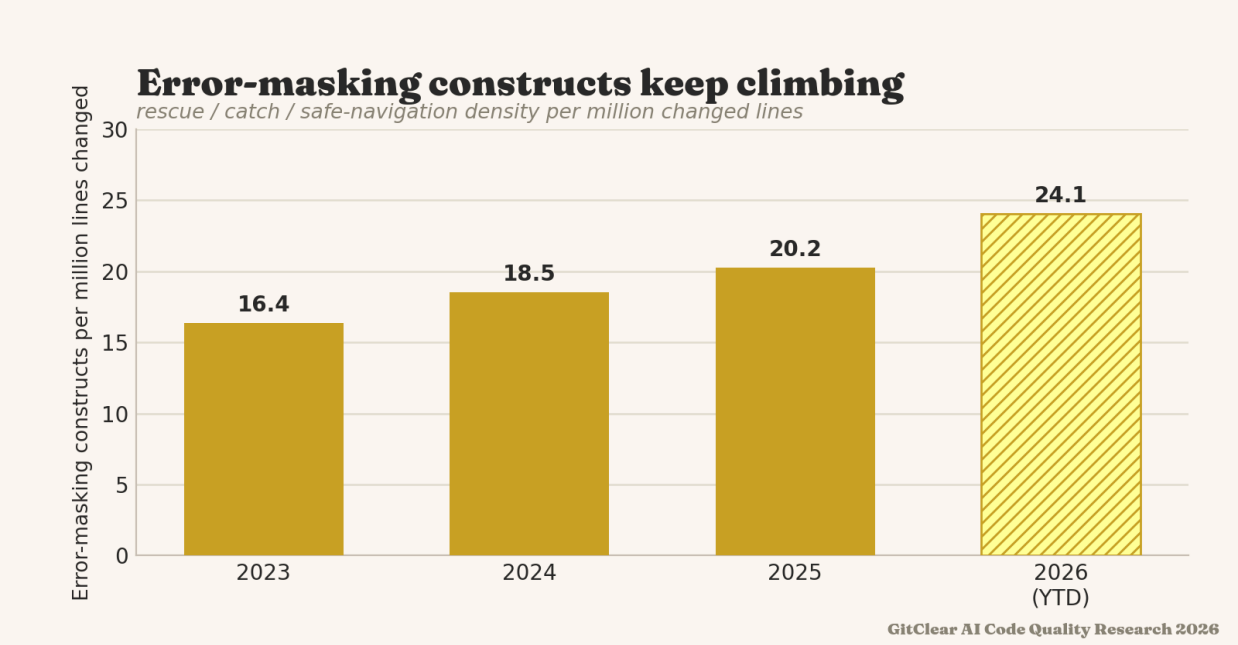


Block duplication has risen every year to a record 73 duplicated lines per thousand changes

Does block duplication matter more than raw copy/paste? The case for "yes": A duplicated block imposes a propagation tax. When a developer changes one copy of a five-line block, they inherit the obligation to find and evaluate **every** sibling — across files and domains they may not know — and decide whether the change must propagate. That obligation is rarely budgeted, frequently missed, and a well-documented source of defects, as discussed in the next section.

Error-masking constructs climbing

The obfuscation signal — density of *rescue*/*catch* blocks, safe-navigation operators, and stubbed methods that squelch unexpected-input signals — has risen steadily since AI began to proliferate. 16.4 per million changed lines in 2023, to 24.1 year-to-date in 2026, a 47% increase relative to base year. The climb can be read as linear or exponential, but either way, it is one of the most consistent trends in the dataset.



Error-masking construct density has risen 47% since 2023, with no annual reversal.

This is a subtle but consequential pattern. A broad *rescue* that swallows any exception, or a chain of safe-navigation operators that quietly returns null rather than surfacing a contract violation, makes code *appear* robust by suppressing signals that would sound alarm bells (e.g., exceptions, thrown errors). Implementing a stub makes an integration test *appear* to prove that an external provider responds as expected, while never proceeding to reconcile how the *actual service* will respond.

AI assistants are incented to reach for these constructs for several reasons: They reduce *measured* defects in code. They save tokens that would be required to research external callers or systems, and reconcile inconsistencies. They are ready to commit faster. Each is convenient in the moment, corrosive over time.

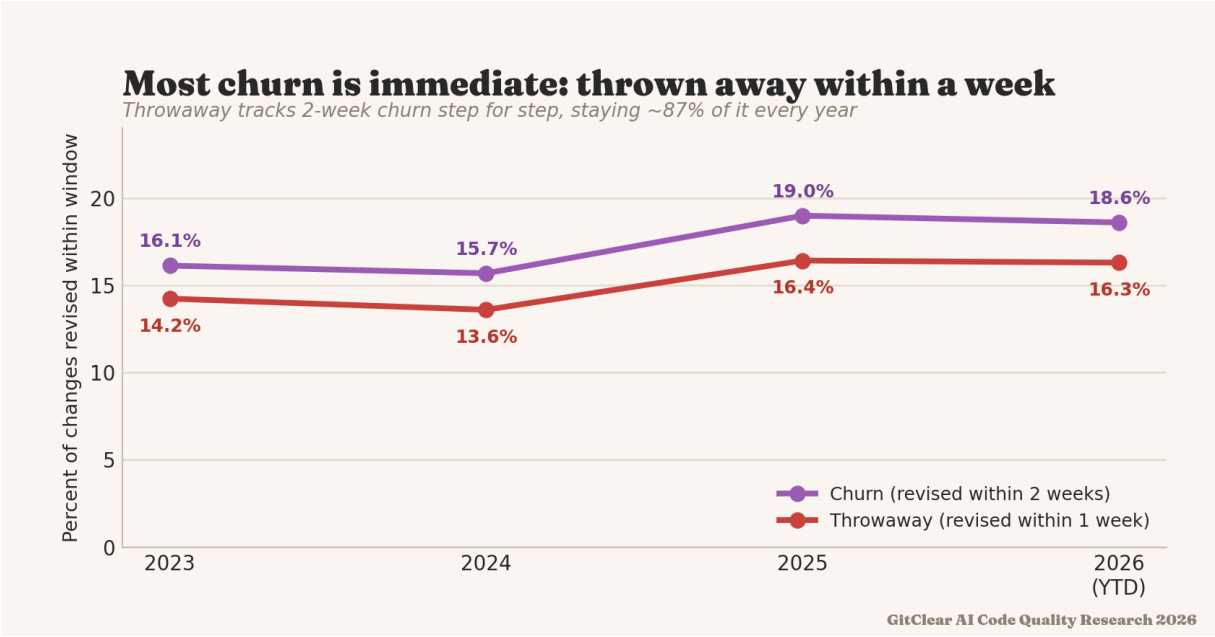
Arguably the most corrosive aspect of unnecessary “catch” and “rescue” statements? How difficult they are to eliminate after inception. Not only will maintainers assume that an error must have occurred to inspire the error-catching construct, but in the effort to justify its paranoid “catch,” LLMs often pair it with a unit test that invents a synthetic error occurring under implausible circumstances. Good luck finding a developer with the confidence to remove both the obfuscation *and* the unit test hallucinated to justify it.

Churn & Throwaway code ascend gradually

Two-week churn, the share of code revised or reverted within two weeks of being written, rose from roughly 16% in 2023–2024 to about 19% in 2025–2026, a 15% relative increase. The implication: a larger fraction of authored code is incomplete or incorrect from the moment it is committed, requiring near-immediate rework. This is consistent with our January 2026 finding

that the heaviest AI users churn code at many times the rate of non-users. Here we see the population-level echo of that behavior.

Two weeks is a deliberate but adjustable boundary. Because our churn measure is computed by linking a line's historical changes, we can tighten our churn boundary to "one week" to isolate code that is discarded almost as soon as it is written. We call this the "throwaway percent." The comparison is revealing.



Throwaway (1-week) churn tracks 2-week churn step for step, staying ~87% of it every year.

Throwaway tracks two-week churn almost perfectly: it rose from 14.2% in 2023 to 16.3% year-to-date in 2026 (a 14.5% relative increase, against churn's 15.3%), and it shares the same sharp 2024–2025 step. Most strikingly, throwaway has held steady at roughly 87% of all two-week churn in every year of the window — 88% in 2023, 88% in 2026.

That stability is the real finding. The growth in churn is not being driven by code that lingers for a week or two before revision; it is overwhelmingly *immediate rework* — code rewritten or reverted within hours or days of being pushed. The bulk of the rising churn signal is, quite literally, work thrown away almost as fast as it is produced

Summary data: Challenges to code comprehension

Finally, we can look at the gross change to how code is being written in the years since LLMs became available for use.

Risk signal	2023	2024	2025	2026 (YTD)	Change
Copy/paste (% of changed lines)	**11.1%**	**13.1%**	**15.3%**	**15.7%**	**+41%**
Duplication (per 1k changes)	**40.3**	**46.7**	**69.7**	**73.0**	**+81%**
Obfuscations (per 1k changes)	**16.4**	**18.5**	**20.2**	**24.1**	**+47%**
Churn (2-week, % of changes)	**16.1%**	**15.7%**	**19.0%**	**18.6%**	**+15%**
Throwaway (1-week, % of change)	**14.2%**	**13.6%**	**16.4%**	**16.3%**	**+15%**

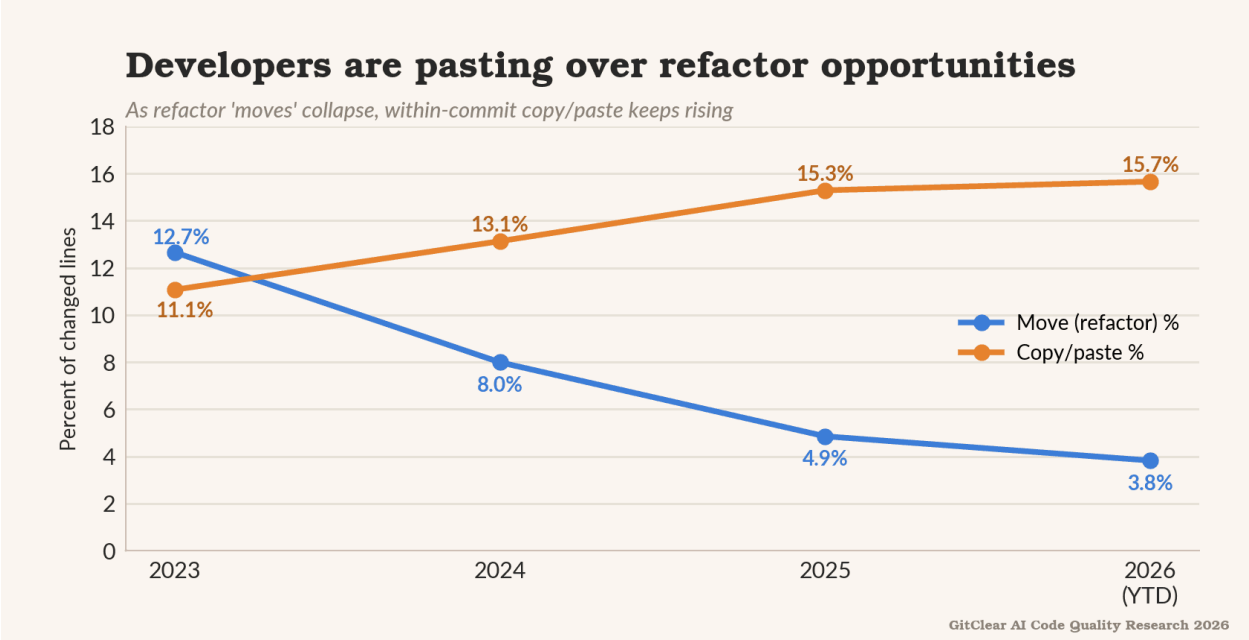
Duplicated blocks have been the fastest overall grower, but Obfuscations have made a bigger year-over-year leap during the first five months of 2026. The cumulative effect is increased difficulty interpreting and predicting the functionality of newly authored code since the debut of effective LLM assistants.

Weathering the Reuse Recession

The clearest structural finding in this dataset is the steady decline of every reuse signal. Refactoring, cross-file connectivity, and long-term maintenance are all falling over the same window that AI adoption has surged.

Refactoring activity collapses as copy/paste thrives

In our 2025 report, the most ominous milestone was the "refactor vs duplicate" operation count crossover point: 2024 being the first year on record where within-commit copy/paste exceeded "moved" (refactored) lines. Two years on, that crossover has widened into a chasm. After clocking in at 21% in 2022, the percentage of moved code dropped to 13% of changed lines in 2023, before freefalling to 3.8% year-to-date in 2026. Over the same window, copy/paste climbed from 9.4% (in 2022) to 11.1% to 15.7% in the first half of 2026.



Clarity, coding agents and v1 share an enemy: Long-term refactoring. It requires deep pattern recognition & willingness to sacrifice short-term comfort for long-term stability

The balance has coalesced into what's beginning to look like a final form. In the battle of "redundant" vs "refactor," developers now exhibit ~5x greater likelihood to indulge the former. Compare this to the final year of Pre-AI (2022), when Coding on Copilot reported a 2x preference for refactor over redundant (20% moved operations vs 9% copy/paste operations R10).

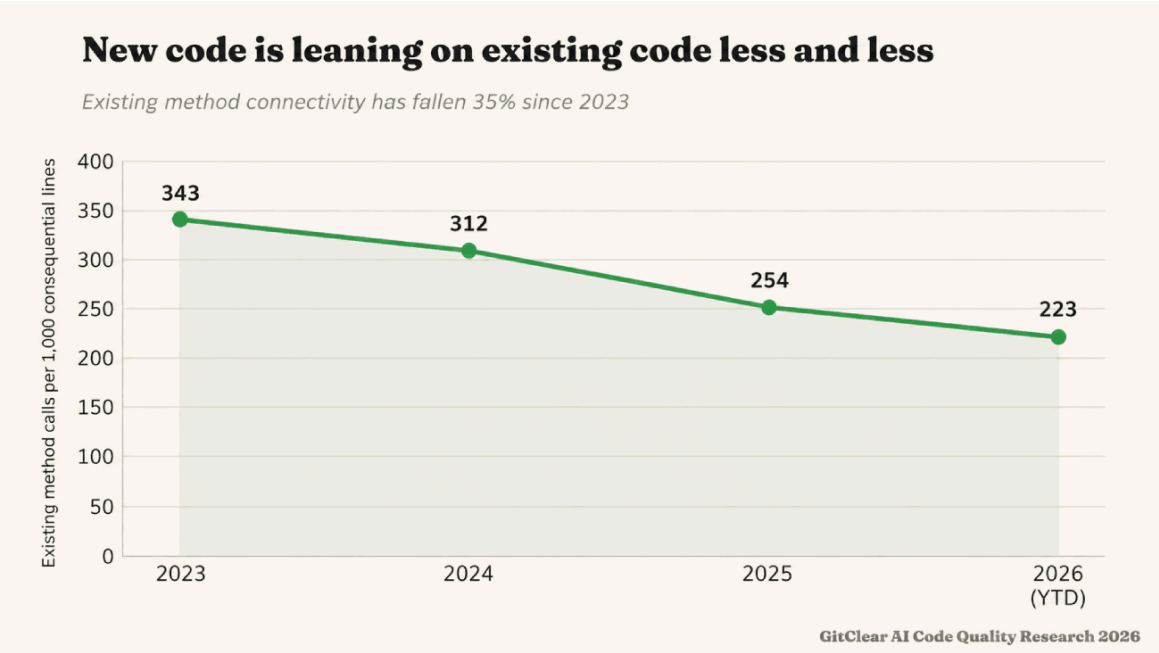
The fundamental problem? Adding code to a new file requires almost zero architectural understanding of the repo. Refactoring requires understanding the surrounding system well

enough to consolidate it safely. As the cost of producing plausible new lines has bent asymptotically toward zero, the path of least resistance shifts ever further toward both *conceptual redundancies* and *outright duplication*.

In spite of the greater effort needed to refactor, most repos would be well-advised to incentivize it. A repo that spends less than 4% of its meaningful changes on consolidating patterns is a repo that developers will struggle to grok as a cohesive whole. Without periodic refactoring, a repo decays into a series of disconnected, "perpetual V1" components. Remember: When it wasn't free to add new lines (pre-2023), **20-25% of all code changes were "moves."** The work that Lead Devs and System Architects undertook to continuously evolve the project's infrastructure provided a reliable foundation to build upon.

New code increasingly stands alone

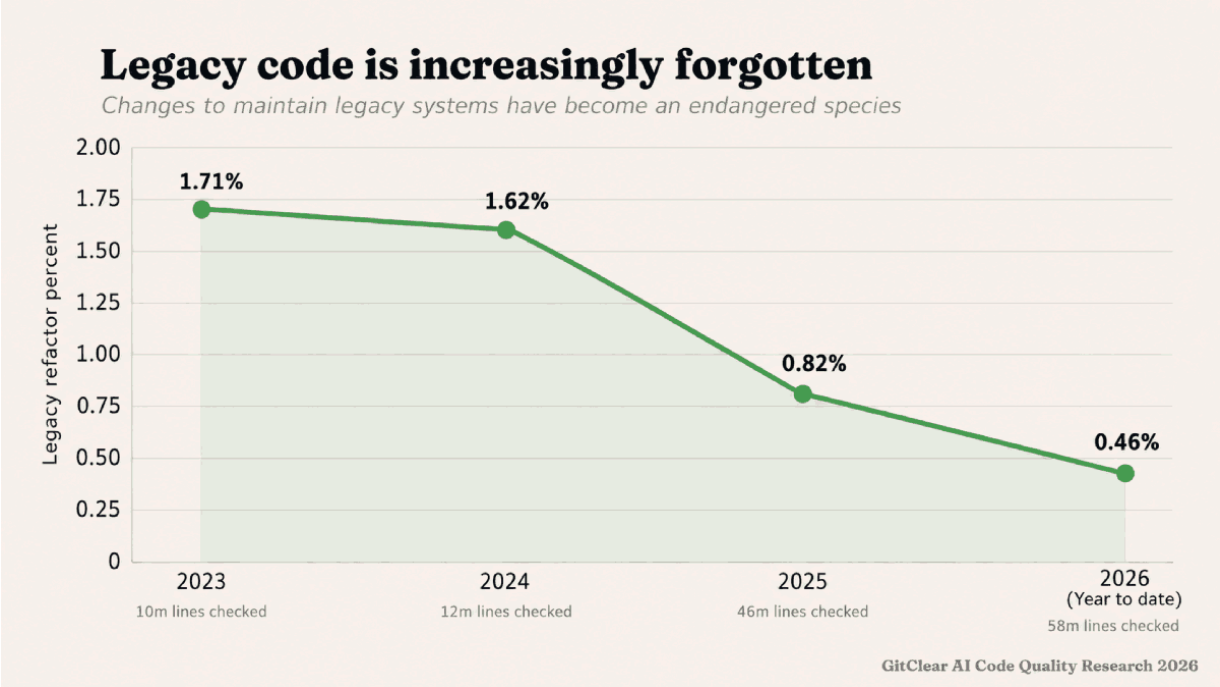
Function connectivity — how often newly authored code connects to a different method or function — tells the same story from another angle. It has fallen 35% since 2023, from 343 method calls per thousand changed lines to 223 year-to-date. New code is less and less woven into the existing codebase. Instead, it is isolated in self-contained files, which is the structural signature of "duplicative reinvention" beating "progressively-improved reuse."



Connectivity of new lines to existing has fallen 35% since 2023.

Legacy code: not gone, just forgotten

The third reuse signal is the most dramatic in percentage terms. "Long-term update percent," the share of changes that remove or update code last touched more than twelve months ago, has fallen 74%: from 1.7% in 2023 to 0.46% year-to-date in 2026.



Healthy, long-lived repositories require developers to periodically return to old code: consolidate it, document it, retire it. That maintenance work is increasingly not happening. The codebase grows outward with new v1 features, while its older strata are left frozen. These neglected sections gradually calcify until something breaks.

Reuse signal	2023	2024	2025	2026 (YTD)	Change
Function connectivity (per 1k fns)	343	312	254	223	−35%
Move / refactor (% of changed lines)	12.7%	8.0%	4.9%	3.8%	−70%
Long-term update (% of changes)	1.7%	1.6%	0.82%	0.46%	−74%

Do These Signals Predict Real Cost?

Skeptics will reasonably ask: do these metrics imply substantive problems, or just stylistic preferences? To assess the salience of the metrics gathered, we turn to existing research literature. Each of the trends above maps to an outcome. Defects, maintenance burden, and delivery instability have been independently quantified by recent research efforts.

More duplication means more bugs

The link between cloned code and defects is among the better-established findings in software engineering. Mo, Zhang et al. (2023), analyzing deep-learning software, found that 57% of co-changed clones were involved in bugs, and that 82% of the projects they studied contained bug-implicated co-changed clones. Independent work converges on the same range: Mondal, Roy et al. (2017) found that up to 33% of clone fragments experiencing bug-fix changes contained propagated bugs, while Wagner et al. (2016), studying inconsistent type-3 clones in an industrial codebase, found 17% contained faults. That two methodologically distinct teams landed within roughly one percentage point of each other (17–18% bug rates) lends robustness.

The mechanism is the propagation tax described earlier: a defect discovered in one copy of a block must be hunted down and fixed in every sibling, and the ones that are missed become latent bugs. An 81% rise in block duplication is, by this literature, a direct upward pressure on defect rate, except to the extent that greater Obfuscation Rate hides defects.

Less refactoring exacts higher maintenance costs

The reuse signals matter for the inverse reason. Refactored, consolidated code — a single well-named function, invoked from many call sites — is documented once, tested once, and hardened by repeated use. Every duplication of that logic reduces reuse. Duplication multiplies the surface area that future developers must learn, test, mull consolidation cost, and keep in sync. As refactoring moves fall 70% and code connectivity falls 35%, codebases are accumulating sprawl that makes onboarding slower and changes riskier. This is the long-recognized rationale behind the DRY principle, and the cost it warns of is precisely what untended, duplicated, weakly-connected code imposes.

Corroboration from Google DORA

The most heavily-funded external check on the value of these metrics remains Google's DORA program. Its 2024 report, drawing on roughly 39,000 respondents, modeled that each 25% increase in AI adoption was associated with a 7.2% decrease in delivery stability and a 1.5% decrease in delivery throughput. The researchers described the quality decline as "surprising" given how positively developers rated AI's effect on their own productivity. Our data offers a mechanism for that surprise. When teams swap refactoring for cloning, and surface-level robustness (broad *rescue* blocks) for genuine error handling, the default outcome is exactly

DORA's pairing: more code authored, alongside more (initially masked) defects and less stable delivery.

> *"More code" and "more defects" are expected to correlate when optimizing for the visible diff while taxing the invisible structure.*

Recent examples where lack of shared code harms company

Some readers may accept that, yes, AI is driving less reuse, more duplication, and less shared code than ever – but does it matter? If all the trends from this report are percolating in real companies, we should expect to find a growing list of examples where the lack of connectivity between "new AI features" and "existing systems" causes headaches. Fortunately, Autonoma has been collecting recent examples, which are provided in its Vibe Coding Failures post R11.

In a representative example, Autonoma describes

> *Wiz Research discovered a critical authentication flaw in Base44, an AI vibe coding platform that builds applications from natural language prompts. The flaw was architectural. Two API endpoints, registration and OTP verification, required no authentication at all. An attacker only needed an app_id, which was publicly accessible in app URLs and manifest files, to register and access private applications. SSO enforcement was bypassed entirely. From there, the attack surface widened: an open redirect that leaked access tokens, stored cross-site scripting, insecure authentication design, sensitive data leakage, and client-side-only enforcement of premium features.*

In a traditionally architected (pre-AI) system, a reuse-oriented infrastructure layer would be built to collect and evaluate the set of "customer ID," "endpoint URL," and "authentication credentials" before permitting access. Authentication systems are a classic example of shared code for most projects. In a world without code reuse, the project leader is left hoping that each developer who implements an auth endpoint will precisely implement the complex interplay required to execute a secure Oauth2 login.

Five Ideas for Teams to Resist Quality Decline

The positive implication from this data is that the risks are measurable, and as the adage goes, "what is measurable can be managed."

Teams that broaden their measurement circumference constrain their debt with negligible impact to velocity. Five specific suggestions to consider:

1. Budget for refactoring and legacy maintenance

Make consolidation and legacy revisiting first-class work. Long-term update percent doesn't fall 74% because Managers decided "maintenance is now irrelevant."

It drops to this rate because nobody is assigned to pare and refine existing code, and *AI doesn't share the human developer's benevolent instinct to recognize and consolidate patterns* as they emerge.

In an AI-driven environment, if reuse and consolidation don't make the sprint cut as first-class objectives, they just won't happen. While AI removes the friction of adding code, it never pauses to consolidate and fortify the emergent patterns. It is maximally risk-averse, and consolidation regularly trades short-term risk for long-term simplicity. The judgment to recognize what should be reused remains a distinctly human contribution as of 2026; empower developers to extract patterns, and your AI features can rely on this infrastructure as a backbone. If nothing else, consider iteratively building tools (potentially tools that can measure duplication or detect `catch/rescue` prevalence?) to improve processes over the long-term.

2. Put a tripwire on duplicate blocks

Alert when a cloned block is changed in one place but not its siblings. Block duplication is the single signal most directly tied to defects in the literature, and it is up 81% over the last few years. A notification that fires when one copy of a duplicated block diverges from the others catches the propagation-tax defect at the moment it is introduced, rather than in production. GitClear or GitKraken can help. A4

3. Review for error-masking explicitly

Treat broad rescue/catch blocks and reflexive safe-navigation as review smells. The 47% rise in error-masking density is one of the most consistent trends in the data and among the easiest to catch in review once a team is looking for it. Ask of each: is this handling a real, anticipated condition, or is it suppressing a signal we would rather have seen?

4. Direct coaching where judgment is thinnest

Pair AI adoption with explicit instruction on its failure modes. The throughput gains concentrate among developers who already had strong structural instincts; the risks concentrate among those who don't yet recognize when an AI suggestion is duplicating, masking, or reinventing. The highest-leverage intervention is not restricting the tools — it is teaching less-experienced developers to see the patterns this report measures, so their rising AI use compounds into durable code rather than durable debt. Finding ways to incorporate these failure modes into the AGENTS.md file and related instruction files can significantly reduce AI propensity to default-hide unexpected state.

5. Measure structure, not (just) volume

Rewarding high line count is anathema to high quality. As long as productivity is proxied by output volume, AI will reliably inflate that proxy by duplicating until your repo

becomes a series of poorly understood V1 modules. If metrics like move/refactor percent, function connectivity, or duplication-per-thousand are unavailable, seek to create equivalents that introduce a "costs" side of the "new code" ledger, visible to the people setting incentives. GitClear can help. A4

Conclusion: Slop Is the Opp

Over the past four years, we have measured & documented the evolution of commit norms. Everywhere we look, we find repos lacking the accoutrements (e.g., concisely documented base classes) that once sprang from the habit of experienced authors to improve systems tangential to their task. Developers, once prized for their ability to spot latent patterns, have become a new type of creature. If we want to preserve the values that refactored and extended "v1" systems into v2s, we would do well to prove that these methods have measurable, long-term value.

For better or worse, the distinct value imparted by an effective Senior Developer is often the *absence* of code, aka the code *removal* and *refactoring* born of deep comprehension. The best do it with less. To build an agent that approximates what a great Senior Developer can bring, it would need to be reinforced on a function akin to "features implemented/sum(lines last changed by me)." The more reusable modules a Senior dev contributes, the more of their informed precedents are absorbed by less experienced team members.

We believe that our 2026 research is distinguished by its breadth of metrics (eight), all pointing toward the possibility that modern commit practices are substantively different from what was historically considered desirable.

In the past, *recognizing* and *consolidating patterns* was the essence of what imbued a product and programmer with value. These were the "fundamental building blocks." You didn't guess the implementation correctly during v1, because you couldn't know how they'd ultimately be used. When we began our annual AI code quality analyses in 2024, we could still see back to a year when "refactored code" outweighed "copy/pasted code" by 5x. It's getting more distant now.

It's tempting to simplify reality and round to "AI is slop" and "all this crap is AI." But, no. Even if all this crap is AI, the habits that keep code maintainable can still be promoted in places like AGENTS.md. Code reuse, duplication, hotfixes from one-too-many v1s: all of these events are sitting in your commit history, holding lessons for whoever cares to come hither for a listen.

> *The biggest risk isn't that AI writes code your team can't maintain. It's that the bill for its completely unmeasured tech debt comes due at the worst possible time.*

Appendix

A1. Full data tables

All figures derive from GitClear's code-change-operation database. 2026 values are year-to-date, minus the interval needed to evaluate duration-adjacent stats. The YTD values are a smaller, in-progress sample; not closed annual figures.

Signal values by year

Signal	2023	2024	2025	2026 (YTD)
Copy/paste percent	11.1%	13.1%	15.3%	15.7%
Duplication rate	40.3	46.7	69.7	73.0
Obfuscation rate	16.4	18.5	20.2	24.1
Churn percent (2-week)	16.1%	15.7%	18.9%	18.6%
Throwaway percent (1-week)	14.3%	13.6%	16.4%	16.3%
Function connectivity rate	342.7	312.3	254.1	223.2
Move (refactor) percent	12.7%	8.0%	4.8%	3.8%
Long-term update percent	1.71%	1.62%	0.82%	0.45%

Sample size (record count) by year

Signal	2023	2024	2025	2026 (YTD)
Copy/paste percent	14.2M	16.7M	60.7M	74.7M
Duplication rate	3.07M	4.08M	15.5M	13.5M
Obfuscation rate	1.06M	1.19M	3.97M	13.2M
Churn percent (2-week)	8.53M	10.2M	35.3M	32.3M
Function connectivity rate	1.05M	1.28M	4.58M	15.7M
Move (refactor) percent	14.2M	16.7M	60.7M	74.7M
Long-term update percent	10.0M	12.2M	46.1M	58.5M

A2. Methodology notes

Operation classification

GitClear resolves each changed line into one of seven operations (Added, Deleted, Updated, Moved, Copy/Pasted, Find/Replaced, No-op). No-op changes are excluded. This is the same classification framework used in prior GitClear AI research.

Per-signal populations

Each signal is computed over the subset of changes where it is defined, so record counts differ across signals. Function connectivity is restricted to files containing functions/methods; duplication and obfuscation are normalized per million changed lines.

2026 YTD

2026 figures are year-to-date as of publication and reflect a smaller sample than the completed years. They are presented as leading indicators and visually distinguished (hatched bars, “YTD” labels) throughout.

LLM fingerprinting heuristics

To connect “LLM” to “commit,” a variety of heuristics are combined. The first-tier commit assignment uses direct telemetry data, e.g., from the Claude Telemetry agent. After exhausting telemetry-based assignment, the system retrieves API-reported “Lines added” and “Lines deleted,” applying a scalar to these lines that ranges from 0.5-0.75 to estimate the percent of AI Provider lines that were ultimately committed to the repo. GitClear associates such lines by examining which git commits include work that was authored on a date that an AI provider asserts had X lines added and Y lines deleted for a particular contributor. Large added blocks are preferentially associated with LLM work, but any type of change can be associated with an LLM when the AI provider reports lines committed to a repo that the committer authored code changes in on the day reported.

Note that, in this research, we did not examine differences in specific LLMs, since the data size is insufficient for our desired confidence interval. We expect to utilize line assignment in future research to understand differences in how specific models approach code authorship.

The dataset's repo composition

The dataset comprises a mix of roughly 80% commercial (private) repos and 20% open source repos, including Facebook React, Microsoft VS Code, Chromium, Google Jax, Ruby on Rails, Homebrew and Babel, among others. For the commercial repos, the median entity size was 10 active developers. The largest private organizations included host 500-1,000 active contributors.

A3. References

R1. AI Copilot Code Quality: Evaluating 2024’s Increased Defect Rate via Code Quality Metrics. GitClear / Alloy.dev Research, 2025.

R2. AI Coding Tools Attract Top Performers — But Do They Create Them? GitClear Research, in partnership with GitKraken Insights, 2026.

R3. Mo, R., Zhang, Y., et al. Exploring the Impact of Code Clones on Deep Learning Software. Association for Computing Machinery, 2023.

R4. Mondal, M., Roy, B., Roy, C., Schneider, K. An Empirical Study on Bug Propagation through Code Cloning. ScienceDirect, 2017.

R5. Wagner, S., et al. On the Relationship of Inconsistent Software Clones and Faults. arXiv, 2016.

R6. Janes, A., et al. Managing Changes in Requirements: An Empirical Investigation, 2013.

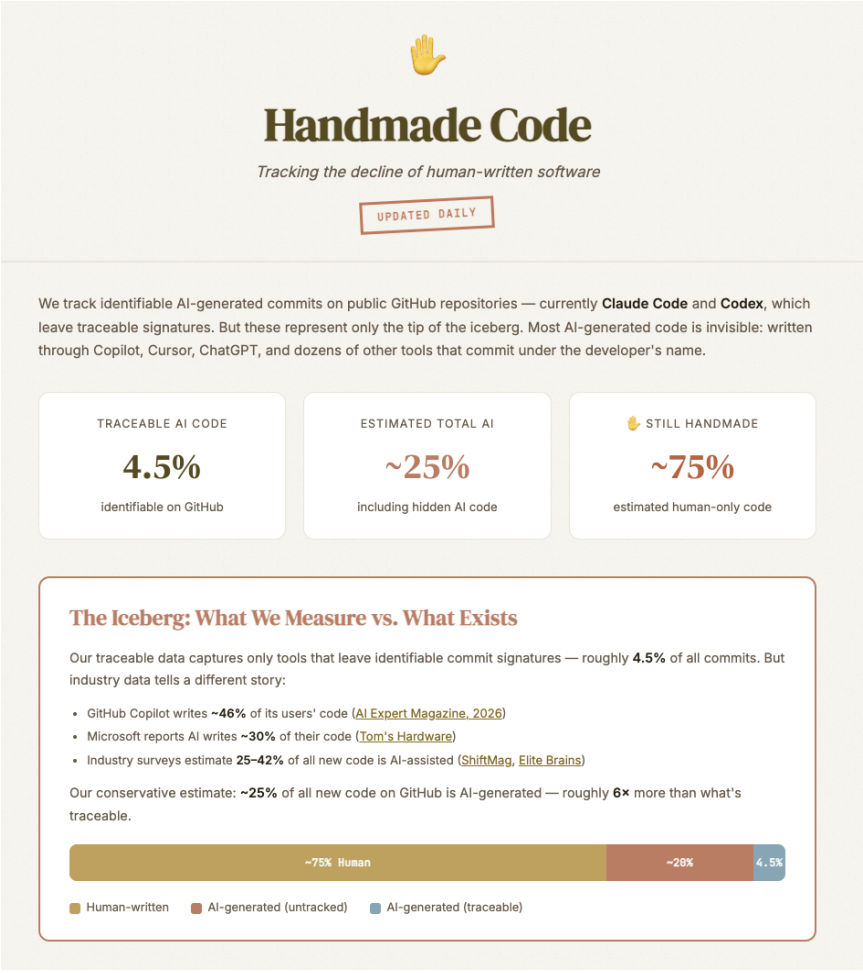
R7. Google. Accelerate State of DevOps Report (DORA), 2024.

R8. Calculating Diff Delta operations. GitClear, 2026.

R9. Handmadecode.dev, as of June 2026, estimates 25% of code is AI authored. Meanwhile, they report AI Experts Magazine offering a "46% of users code written by Copilot," and Tom's Hardware stating that Microsoft claims to author 30% of code via AI. In Q2 2026, 25% is effectively the lowest defensible number you will find reported as the percentage of code that is AI-authored.

R10. Coding on Copilot: 2023 Data shows downward pressure on code quality.GitClear, Alloy.dev Research, 2024

R11. Vibe Coding Failures. getautonoma.com



A4. Contact & participation

GitClear publishes research on a roughly quarterly cadence. Teams interested in generating any of these signals for their own repositories — or in contributing model-attributed data toward the

per-model comparison planned for the next report — can reach us at **hello@gitclear.com** or **bill@gitclear.com**. As with prior reports, we pledge to publish results whether they are positive, negative, or null.

GitClear and GitKraken Insights both offer measurement for every quantity mentioned in this report:

- Duplication rate and duplication change alerts
- Rate of introducing the three types of obfuscating code, plus tools to help distinguish between healthy background use and AI FUD-related overuse.
- Understand percent of Move and Copy/Paste on a per-team basis
- Review percentage work spent on of post-PR merge bugfixes
- Evaluate which teams succeed at making cross-file connections, signaling higher likelihood of successful code reuse
- Understand how metrics are changing from Sprint-to-Sprint, quarter-to-quarter, or over a multi-year time horizon
- Watch how Churn and Throwaway code correlate with Defect Rate, when the necessary iterations do not take place prior to a product release
- More AI code research where this came from
- Technical documentation on maximizing the health of a dev team

Email hello@gitclear.com while mentioning this research to receive a subscription discount code.

A5. Encyclopedia of developer analytics terms

For the fully illustrated Encyclopedia of Developer Analytics, see GitClear's quixotic effort to capture every labeled metric that developer teams ask about in the Online Encyclopedia of Developer Analytics. This particular instantiation will focus specifically on fleshing out the finer details of how GitClear counts the metrics it refers to in this research.

- **Duplicated blocks / Duplicated lines**. Among non-test (where duplication can be subjectively OK), non-auto-generated (e.g., compiled), non-key/value languages (e.g., not CSS, XML, etc., where duplicated lines are expected), and non-ignored (e.g., templates) files, how many sequences of five or more substantive lines (ignoring blank lines) can be found? A "substantive line" is one that is not a keyword (keywords can be included in a duplicate block, but they cannot constitute one of the core five lines that must be repeated to qualify as a duplicate block) and not purely symbolic (e.g., it contains text and/or numbers). The block of 5+ lines must be present in more than one location, either within a single file or across multiple files.
- **Copy/paste code**. Defined in GitClear's Diff Delta documentation: a non-keyword, non-repo-idiom line that is repeated multiple times in a commit.
- **Moved/refactored code**. Defined in GitClear's Diff Delta documentation: any non-keyword, non-repo-idom line that is cut from one location and pasted to another (within or outside the original file)

- **Obfuscations per 1,000 changed lines**. The count of newly introduced obfuscating lines per 1,000 changed lines in non-key/value files. These lines reduce a maintainer's ability to reason about interactions between the method and the outside world, and include catch/rescue blocks, safe navigators (characters that obfuscate whether an argument is null-ish), and stub/mock methods in tests. All of these constructs have value when used sparingly, after weighing their pros and cons. The problem emerges when their historical prevalence is driven continuously upward by agents seeking to reduce token use, by skipping the work that would be necessary to evaluate which local<>external interactions are plausible. That work is delegated to maintainers.

A6. A NOTE ON PER-MODEL COMPARISONS

A natural next question is whether the code quality trends differ by model? Are there substantive differences in whether code attributed to one assistant duplicates, masks errors, or churns more than another, or more than "unattributed" human-written code?

We are not yet prepared to make per-model claims with the confidence this audience deserves. The model-attributed portion of our dataset, while growing quickly, is not yet large or evenly distributed enough to support head-to-head conclusions that would survive the kind of scrutiny we apply to the aggregate trends in this paper.

We expect that to change. Our next report should carry enough model-attributed data to compare assistants, and to better compare AI-attributed versus unattributed code, with meaningful confidence. In the meantime, readers who want an early, directional sense of how models are faring relative to one another can explore the live, continuously-updated segment graphs at gitclear.com/industry_stats/ai_code_quality_signal_graphs. This page will segment each of these signals by model and let you eyeball per-LLM patterns against the unattributed baseline. We would treat any pattern visible there today as a hypothesis to be confirmed: not yet a robust finding.