AI Copilot Code Quality

Evaluating 2024's Increased Defect Rate via Code Quality Metrics

211m lines of analyzed code + projections for 2025

William Harding, Lead Researcher & CEO
Alloy.dev Research

Published February, 2025

Abstract

Code assistants diligently accepted a far greater share of code-writing responsibility during 2024 than ever before. According to Stack Overflow's 2024 Developer Survey, 63% of Professional Developers said they currently use AI in their development process. Another 14% say they plan to soon [2].

The 36,894 developers who answered "**most important benefit you're hoping to achieve with AI**" overwhelmingly picked "Increased productivity." Developers seem to view AI as a means to *write more code*, *faster*. Through the lens of "does more code get written?" common sense and research agree: *resounding yes.* AI assistants do beget more lines.

But if you ask a Senior Developer "what would unlock your team's full potential?" their answer won't be "more code lines added." To retain high project velocity over years, research suggests that a DRY (Don't Repeat Yourself), modular approach to building is essential [4] [5]. Canonical systems are documented, well-tested, reused, and periodically upgraded.

This research aims to evaluate how today's profusion of LLM-authored code is influencing the **maintainability** of tomorrow's systems.

To investigate, we will analyze the largest known database of highly structured code change data [A2]. 211 million changed lines of code, authored between January 2020 and December 2024 [A1], will be assessed on quantifiable "code quality" metrics.

The data in this report contains multiple signs of eroding code quality. This is **not** to say that AI isn't incredibly useful. But it *is* to say that the frequency of copy/pasted lines in commits grew 6% faster than our 2024 prediction. Meanwhile, the percent of commits with duplicated blocks grew even faster. Our research suggests a path by which developers can continue to generate distinct value from code assistants into the foreseeable future.

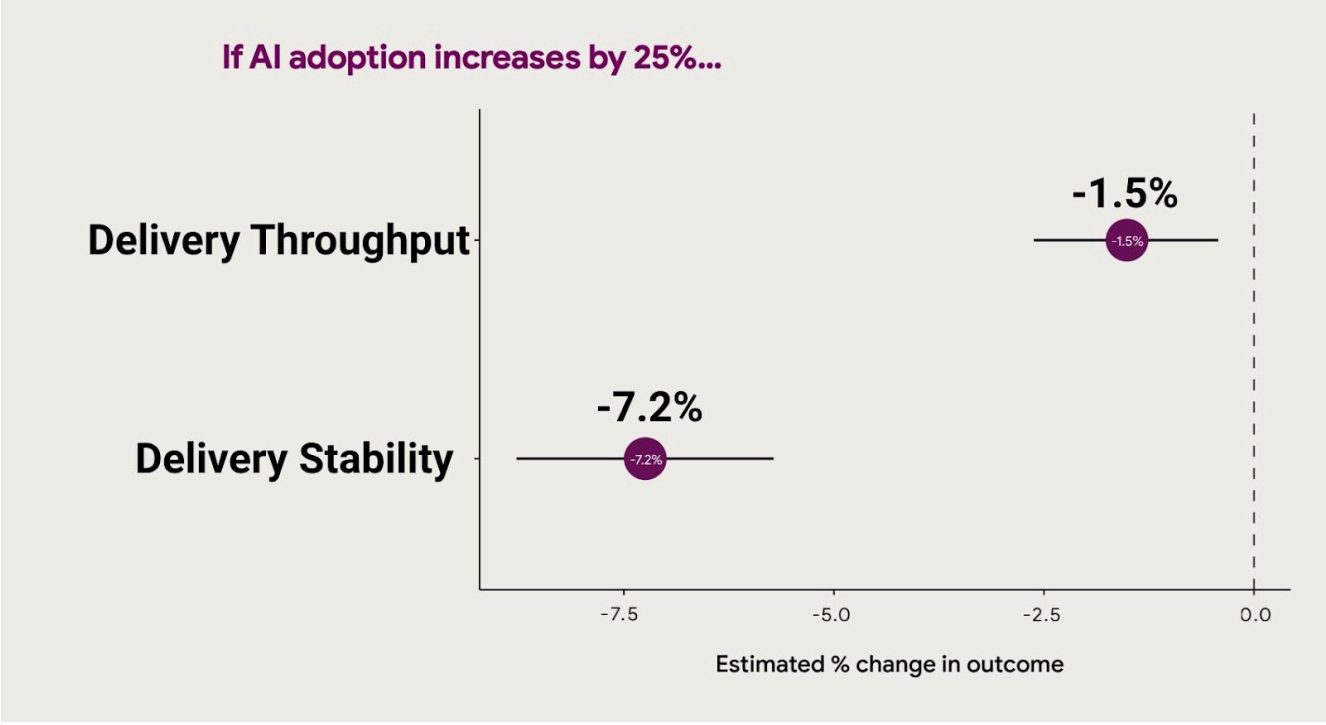
Table of Contents

Growing Evidence that AI-Generated Code Optimizes for the Short-Term

Duplicated code was getting bad in 2023. In 2024, it started getting ugly.

Last year, in our first edition AI Code Quality Analysis, we reviewed three code quality metrics that were trending worse since the debut of AI code assistants. The sharp upward curve of AI adoption seemingly guaranteed that, **if** the identified trends were really correlated with AI use, they would get worse in 2024. That led us to predict, in January 2024, that the annual Google DORA Research (eventually released in October 2024) would show "Defect rate" on the rise. Fortunately for our prediction record, unfortunately for Dev Team Managers, the Google data bore out the notion that a rising defect rate correlates with AI adoption.

But here was something we did *not* predict: how much effort Google would invest into exploring this particular question. They went so far as to attempt to formalize the exact rate at which "greater AI adoption" could be mapped to "higher defect rate":

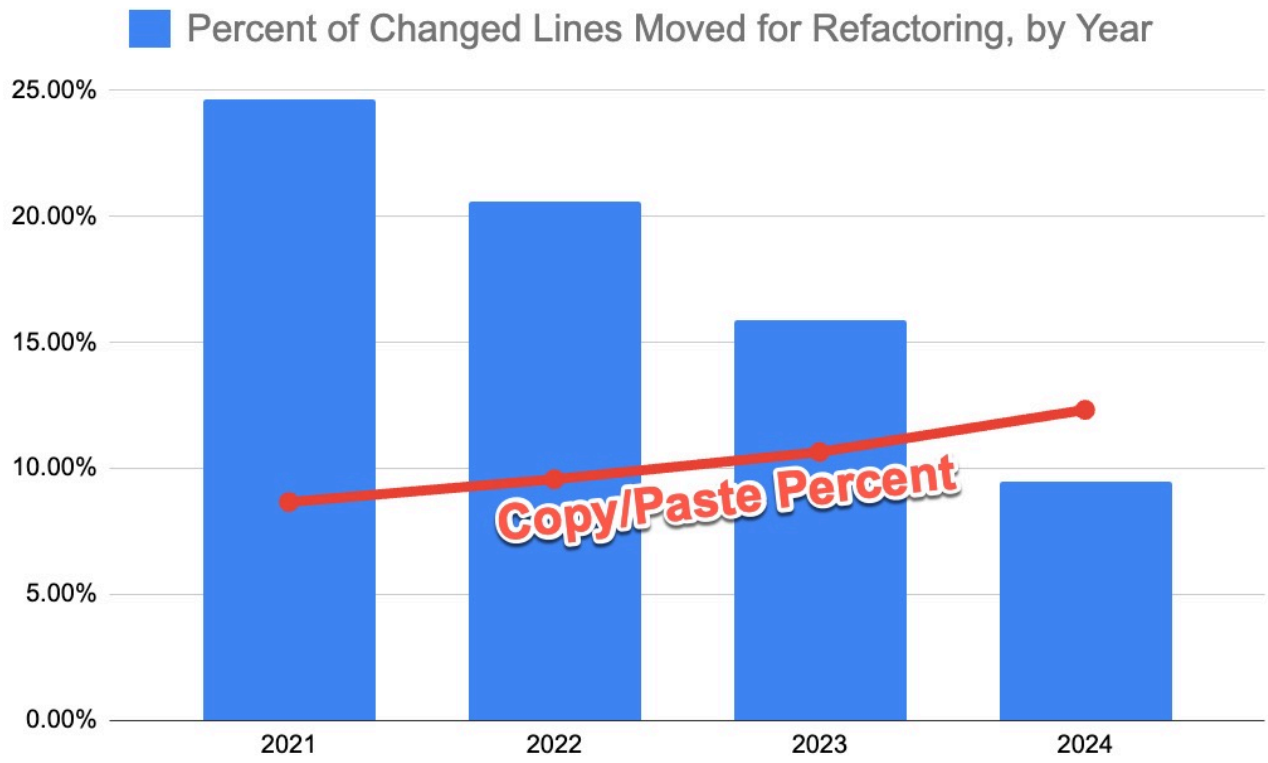


Google DORA 2024's extrapolated change in delivery stability per 25% increase in AI

Our guess that defect rate would increase was based on the 2023 trend line of three metrics: refactoring (down), churn (up), and copy/pasted lines (way up).

In a common theme across 2024 AI projections: we got the general direction right, but underestimated the magnitude of the change.

Bottom line: **2024 marked the first year GitClear has ever measured where the number of "Copy/Pasted" lines exceeded the count of "Moved" lines.** Moved lines strongly suggest refactoring activity. If the current trend continues, we believe it could soon bring about a phase change in how developer energy is spent, especially among long-lived repos. Instead of developer energy being spent principally on developing new features, in coming years we may find "defect remediation" as the leading day-to-day developer responsibility.



2024: The first year on record where within-commit copy/paste exceeded moved lines

Concurrent with this dubious milestone, we recorded an 8-fold increase in the frequency of code blocks with 5+ duplicated lines during 2024. This is a concerning development for executives, Lead Developers, and open source repo leaders.

The Peter Drucker line that “what gets measured gets done” seems to be at play. There is a great hunger for more developer productivity. If “developer productivity” continues being measured by “commit count” or by “lines added,” AI-driven maintainability decay will proliferate.

Even when managers focus on more substantive productivity metrics, like “tickets solved” or “commits without a security vulnerability,” AI can juice these metrics by duplicating large swaths of code in each commit. Unless managers insist on finding metrics that approximate “long-term maintenance cost,” the AI-generated work their team produces will take the path of least resistance: expand the number of lines requiring indefinite maintenance.

Trends in Code Changes: 2020-2024

Among 211 million code lines analyzed [A1], the following table shows how code line change operations have evolved since the widespread adoption of AI Copilots:

	Added	Deleted	Updated	Moved	Copy/pasted	Find/replaced	Churn
2020	39.2%	19.1%	5.2%	24.1%	8.3%	2.9%	3.1%
2021	39.5%	19.3%	5.0%	24.8%	8.4%	3.4%	3.3%
2022	40.9%	19.8%	5.2%	20.5%	9.4%	3.7%	3.3%
2023	42.3%	21.1%	5.6%	15.8%	10.6%	3.6%	4.5%
2024 Projected	43.6%	22.1%	5.8%	13.4%	11.6%	3.6%	7.1%
2024 Actual	46.2%	21.9%	5.9%	**9.5%**	**12.3%**	4.2%	**5.7%**
YoY change	+9.2%	+3.8%	+7.2	**-39.9%**	**+17.1%**	+16.7%	**+26%**

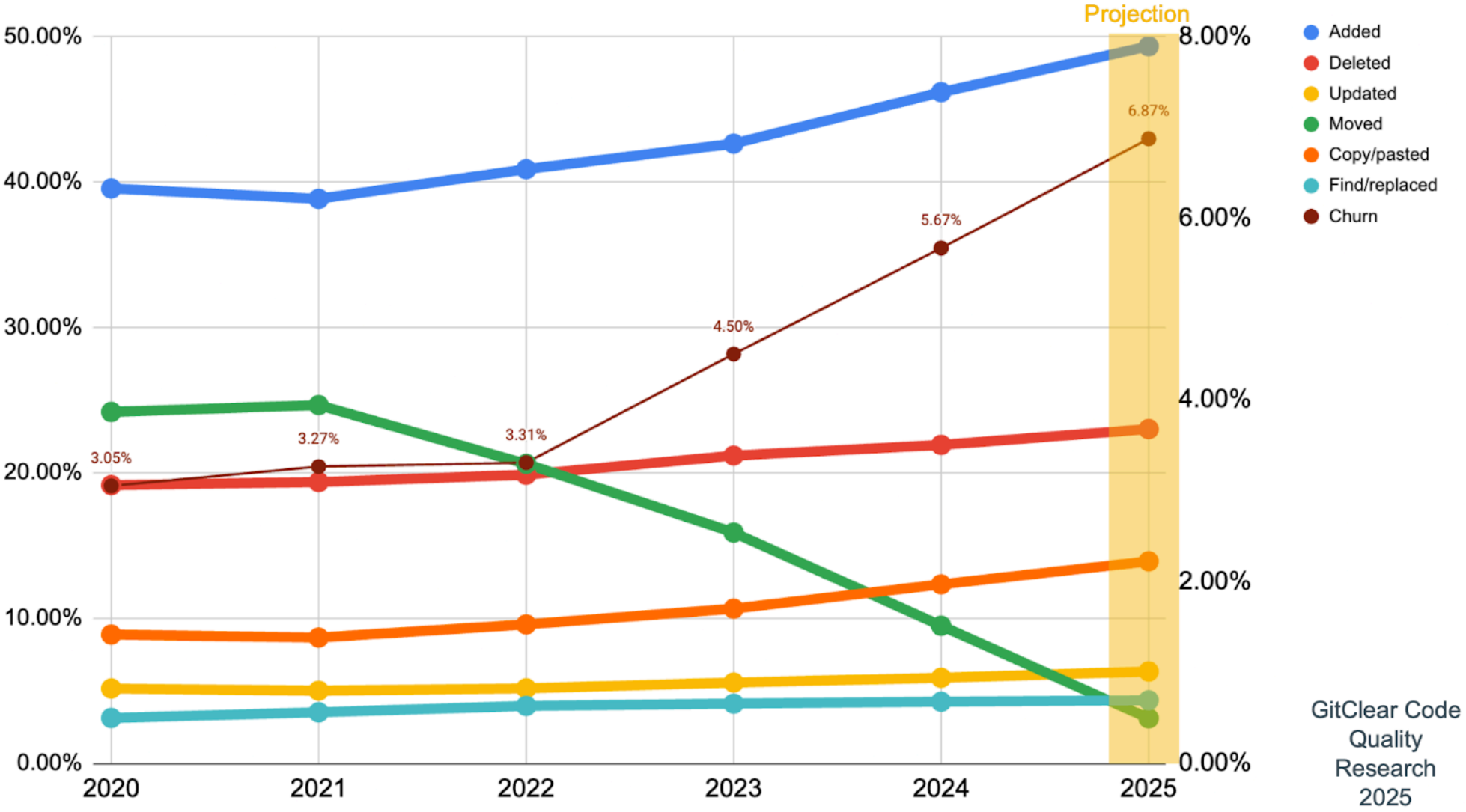
In our 2024 research (analyzing data from 2023), we cautioned that the decline of “Moved” code, toward our projection of 13.4%, coupled with the rise of “Copy/Paste” code, toward a projected 11.6%, would create challenges to code maintainability in 2024.

The actual breakdown of code lines committed during 2024 was substantially worse than our projection.

Prevalence of Code Change By Year

The following visualization shows all 211 million lines analyzed by this research, with the green area depicting the shrinking “moved” lines, while the orange area shows how often code lines are now being duplicated **within a single commit**:

Code Operations and Code Churn by Year



Relative distribution of code operations by year of authorship

Note how the percentage of “Newly added” code has grown from 39% to 46% of code changes since AI has proliferated. Developers have arrived at a point in history where it has never been easier to write half a line, press tab and, and presto! New lines of code to maintain.

Compared to **adding** code, which requires only a single keypress, code refactoring takes *a lot* of work. It requires baseline familiarity with the patterns being used across the repo. It requires judging which functions are similar enough to consolidate. It requires having a general awareness of which core libraries are in use, and which functions are promising enough (e.g., tested in tests, tested in production, documented) to expand upon.

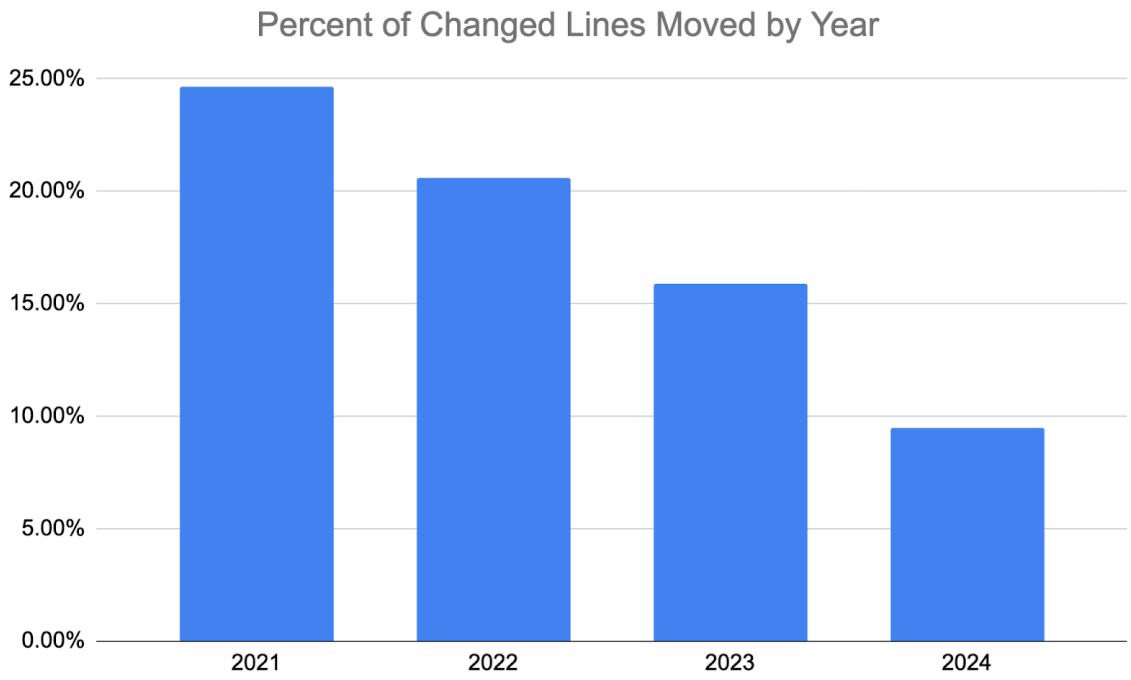
Beyond that, the developer needs to understand the code they are refactoring well enough to be confident they won’t break existing callers that relied on the original behavior of the potentially refactored method.

The good news is that AI is already helping developers with the most trivial types of rewrites. When considering the 17% increase in Find/Replace code, credit is due to AI IDEs like Cursor that will, by default, rewrite code that does not adhere to the project’s linting rules [13]. In practice, it means that projects are, at least, more consistent on a per-line basis. Unfortunately, per-line consistency is not the primary impediment to long-term maintainability.

“Moved” Code Becoming an Endangered Species

Blame the context window size. Blame the UI challenges. Whatever the proximal cause, the essential advantage that human programmers can claim over AI Code Assistants, circa 2024, is the ability to consolidate previous work into reusable modules.

Refactored systems in general, and moved code in particular, are the signature of code reuse. As a product grows in scope, developers traditionally rearrange existing code into new modules and files, to reduce the number of systems that need to be maintained. Code reuse leaves fewer concepts (and “gotchyas”) for new team members to learn. It also increases testing and documentation infrastructure that can accumulate each time the system is invoked by a new team member.

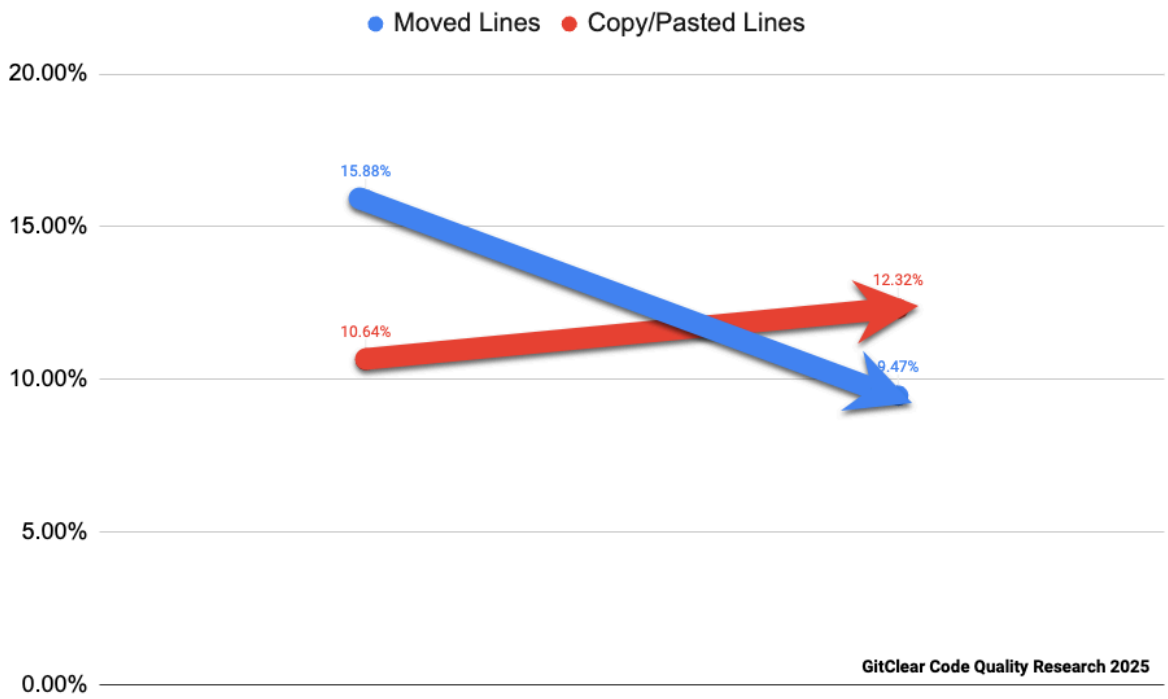


Less refactor, less reuse. Can "moves" still *shrink further?*

In four years, the rate of this code reuse artifact is less than half its 2020 ("fully human") rate. It has sunk **from 25% of changed lines in 2021 to < 10% in 2024.** If this were a matter of "reduced developer communication," the trend line would not have dropped by 44% in 2024 alone (from 16.9% to 9.5%). The growing "back to office" mandates prove that "less reuse" will carry the day, even if developers exist in the same room. Will leaders proactively champion the cause of "code reuse," or will it be discarded on behalf of "*more lines equals more done*"? Difficult to predict; we would welcome opinions from CTOs sent to our Contact Information. We can report on what we hear in next year's report.

The Year that "Copy/Paste" Exceeded "Moved"

2024 was the first year that "Copy/Pasted" frequency beat "Moved" line frequency among the 211 million changed lines we've analyzed over the past five years:



The combination of these trends leaves little room to doubt that the current implementation of AI Assistants makes us more productive at the expense of repeating ourselves (or our teammates), often without knowing it. Instead of refactoring and working to DRY ("Don't Repeat Yourself") code, we're constantly tempted to duplicate.

"Copy/Pasted" code, as measured on a per-commit basis here, quantifies the frequency of non-keyword, non-autogenerated, non-comment lines of code that are committed in a single git commit. Historically, developers have avoided this by creating a single method/function that is invoked by many callers. If necessary, developers parameterize the function so it can be utilized by many call sites [A0]. The more times a function is reused, the more battle-tested it becomes.

Google's latest report suggests that AI has shown great promise in its ability to improve code documentation. But if we keep duplicating snippets within functions, capturing the potential benefit of "well-documented functions" will remain elusive.

The Surge in Duplicated Code Blocks

A limitation of last year's GitClear research was that we could only measure code duplication indirectly, via the count of "Copy/Paste" operations within individual commits [A0]. Thanks to infrastructure upgrades, it recently became possible to detect any region where five or more contiguous lines (ignoring blank/keyword lines) are repeated [16]. "Duplicate Block Detection" is among more than 30 goals that GitClear users can instrument.

Modern code assistants are constantly suggesting multi-line code blocks to be inserted through a single press of the "tab" key. It's readily apparent, from using these tools, that many of the suggested code blocks have their origins in existing code. Since the most popular code assistant of 2024 was limited to roughly 10 files that could fit in its context window [A6], we wondered if it would be possible to measure an uptick in the percent of newly authored code that duplicated adjacent existing code lines [16].

The following table shows the measured frequency of commits that contain a duplicate block, by the year the code was authored:

Year	Commits scanned	Total dupe blocks found	Commits containing dupe block	Duplicate block %	Median dupe block size
2020	19,805	9,227	139	0.70%	10
2021	29,912	9,295	143	0.48%	11
2022	40,010	10,685	182	0.45%	11
2023	41,561	20,448	747	1.80%	10
2024	56,495	63,566	3,764	6.66%	10

According to our duplicate block detection method [A8], 2024 was without precedent in the likelihood that a commit would contain a duplicated code block. The prevalence of duplicate blocks in 2024 was observed to be approximately 10x higher than it had been two years prior.

The Scourge of Duplicated Code Blocks

"So what?" an executive may wonder. So long as the code *works*, does it really matter that it might be repeated?

Both first principles and prior research suggests that it *does* matter.

“First Principle” Drawbacks of Duplicate Blocks

From first principles, it is evident that repeated blocks impose the burden of deciding: should a cloned block change be propagated to its matching siblings? When a conscientious developer deems it prudent to propagate their change, their scope of concern is multiplied by the number of duplicated blocks.

Instead of focusing on the specific Jira they had been assigned, the developer must now tack-on the effort of understanding *every system in the repo that duplicates the code block they are changing*. Likewise for every developer who reviews the pull request that changes code from disparate domains. They, too, must acquaint themselves with every context where a duplicated block was changed, to render opinion on whether the change in context seems prone to defect.

Next, each changed code domain must be tested. Picking the *method* to test these “prerequisite sub-tasks” saps the mental stamina of the developer. And because the impromptu chore of “updating duplicate blocks” is rarely-if-ever budgeted into the team’s original Story Point estimate, duplication is a significant obstacle to keeping on schedule. As a developer falls further behind their target completion time, their morale is prone to drop further. From the developer’s perspective, this task that had seemed *so simple*, has now ballooned into systems where they might have little-or-no-familiarity. All this burden, without any observable benefit to teammates or executives.

Of course, the implicit prerequisite to “deciding whether to propagate a duplicate block change” is to become aware that the code they are changing is duplicated (or nearly so). Tools like GitClear facilitate this awareness (by specifying Code Quality Goals), but many teams do not yet have a consistent method to enumerate where their problem code resides.

Research-Proven Drawbacks of Code Clones

“Exploring the Impact of Code Clones on Deep Learning Software” is a 2023 paper by Ran Mo, Yao Zhang et. al [5] that seeks to quantify the impact of code clones in software that utilizes deep learning. With regard to the prevalence of duplicated code,

the researchers find that “[Deep Learning projects exhibit] about 16.3% of code fragments encounter clones, which is almost twice as large as traditional projects.”

By analyzing 3,113 pairs of co-changed code lines, the researchers observe that “**57.1% of all co-changed clones are involved in bugs**.”

In breaking down the prevalence of the three different types of code clones [A5], the researchers find:

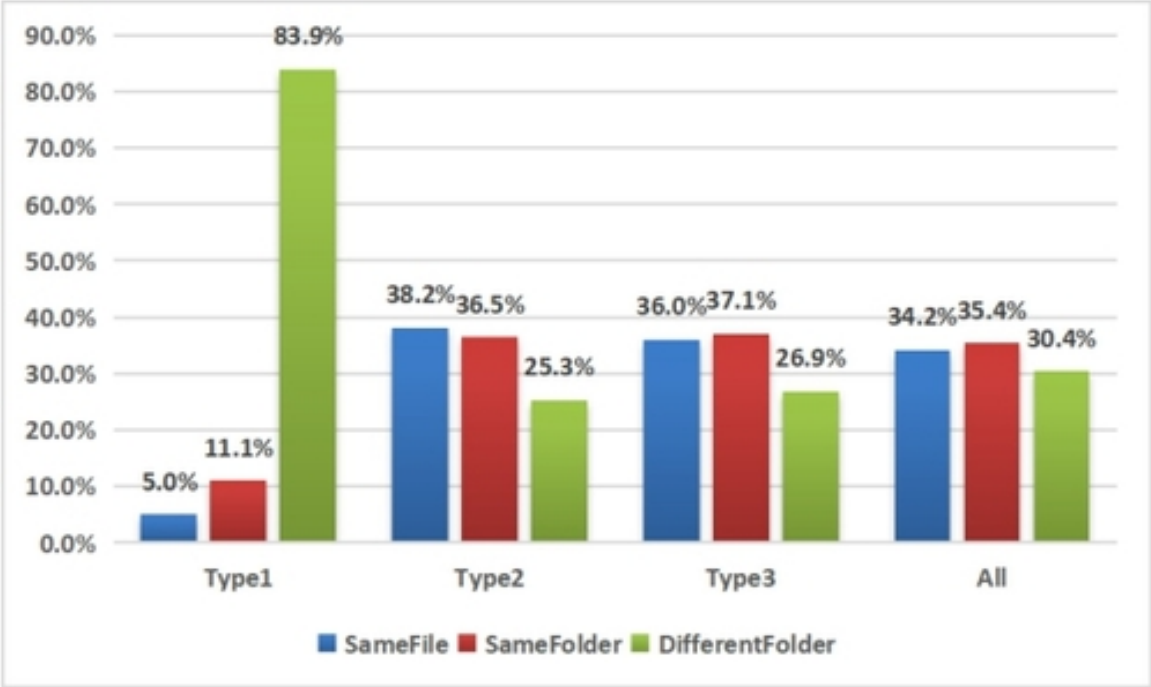


Figure 14 from “Exploring the Impact of Code Clones”

That is, Type 1 clones (clones identical save for spaces and new lines) are often present in different folders, but aside from that, the three types of clones are distributed evenly between “same file,” “same folder,” and “different folder.”

When it comes to assessing the effect of these clones, the authors observe:

> *Numerous studies have shown that code clones in traditional software systems could cause bugs [4] [7] [8] [9] [10] [11] [12]*

The Citations offer a multitude of research that has shown clones in traditional software systems prone to cause bugs. In order to evaluate the prevalence of bugs in co-changed code clones, the researchers utilized “a keyword-based method to extract bug-fixing commits (BFCs) by matching a commit’s message with specific keywords,

including *bug*, *fix*, *wrong*, *error*, *fail*, *problem*, and *patch*." This method yielded the finding:

> *Considering all 34 DL projects together, we can observe that 28 of them (82.4%) contain co-changed clones involved in bugs. The ratios of bug-prone co-changed clones varied from 2.1% to 100%, with an average ratio of 57.1% and a median of 48.1%. These findings suggest that developers should pay close attention to the co-changed clones, since they have a high potential of participating in bugs.*

Their research strongly suggests that the elevated rate of bugs in cloned code is contributing to the higher baseline of errors and defects observed since 2022.

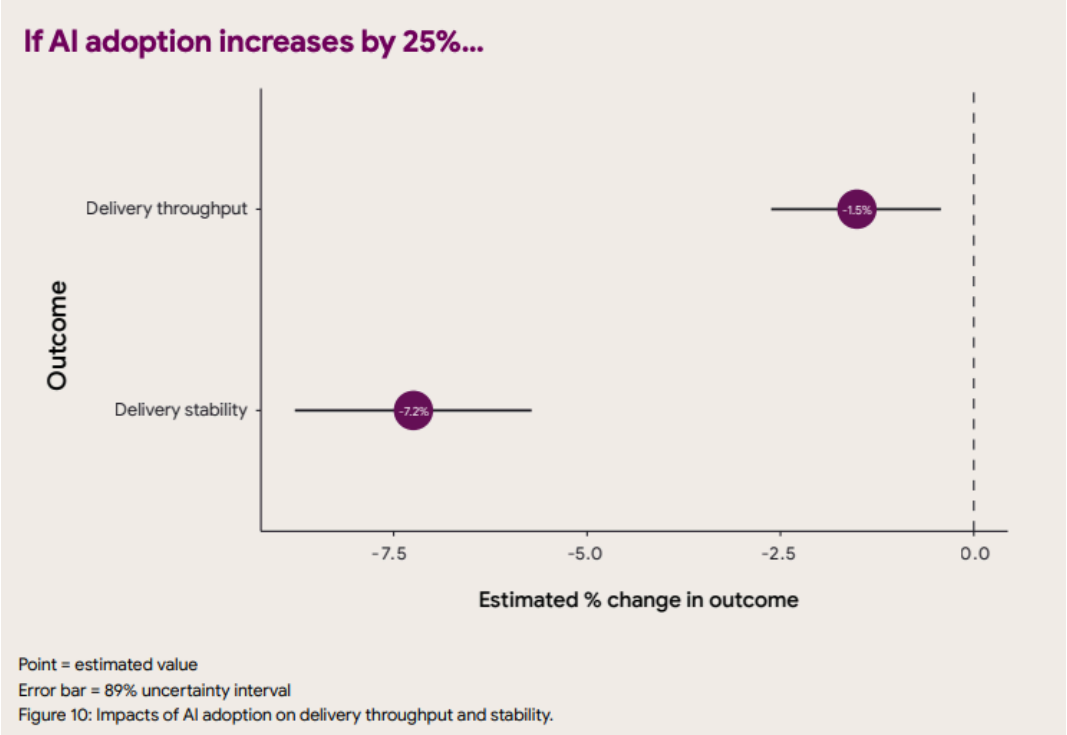
Further research exploring the relationship between "Cloned code" and "Development outcomes" can be found in the appendix [A9]. While there are many different approaches that researchers have used to evaluate the consequences of cloned blocks, all research we reviewed over the past 10 years affirmed that cloned code (in particular, cloned code blocks that do not consistently co-evolve) as a source of observed defects, proportionally to how often they exist within a repo.

Corroboration with 2024 Google DORA Benchmarks

In the “Predictions” section of our 2024 Coding on Copilot research, we theorized that the uptick in Copy/Paste coupled with the continued growth of “recent churn” would manifest as a higher incidence of deployed bugs [A4]. The most thorough, well-funded research on this subject comes courtesy of Google, as part of the annual Google DORA benchmarks for Devops [1].

Google DORA’s 2024 survey included 39,000 respondents–enough sample size to evaluate how the reported AI benefit of “increased developer productivity” mixed with the AI liability of “lowered code quality.” That research has since been released, with Google researchers commenting:

> *AI adoption brings some detrimental effects. We have observed reductions to software delivery performance, and the effect on product performance is uncertain.*



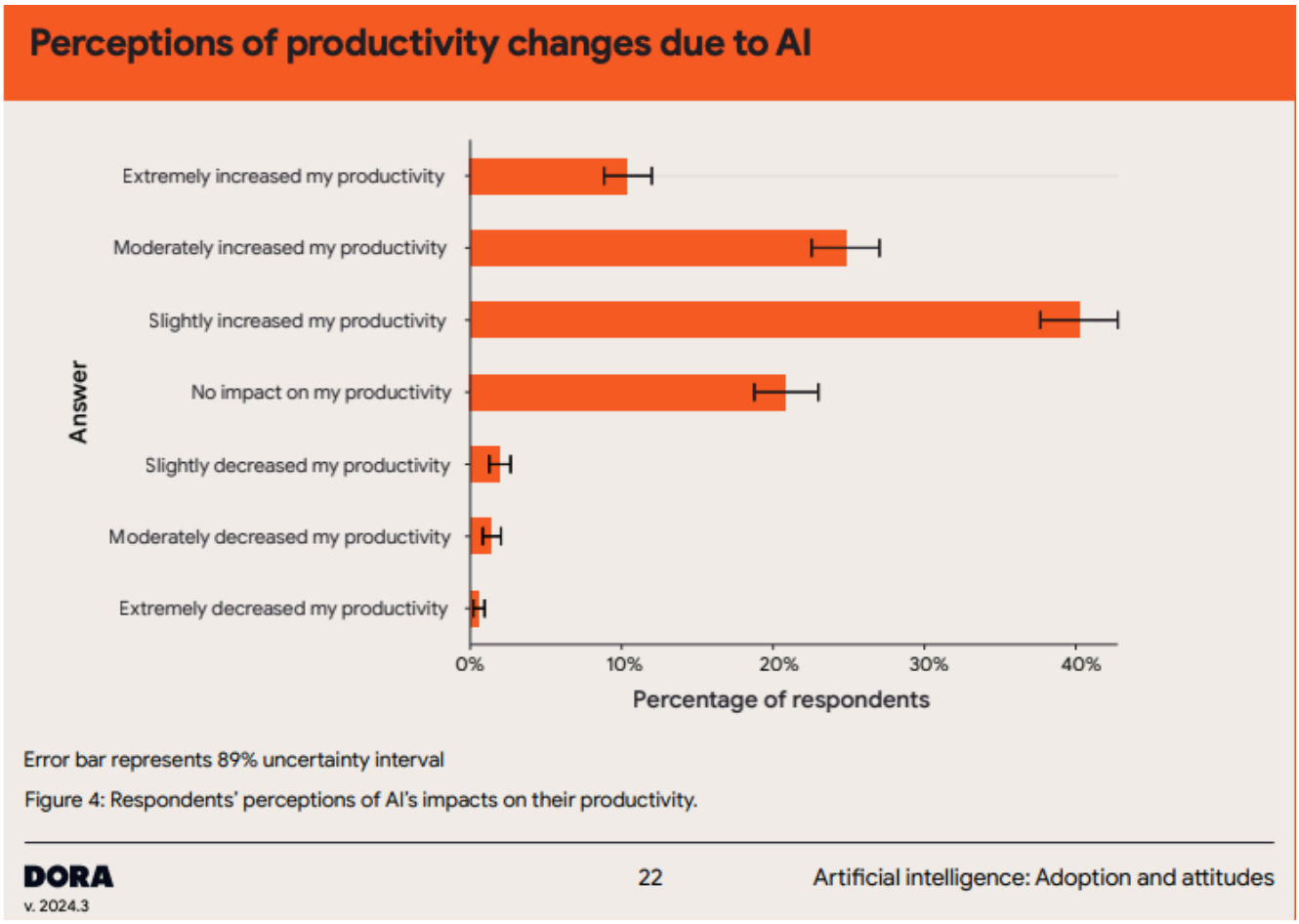
Page 24, Google DORA 2024 Report

This graph shows that for every 25% increase in the adoption of AI, their model projects **a 7.2% decrease in “delivery stability.”** But Google does not have a clear

hypothesis on the cause of ascendant defects. In fact, they describe it as “surprising,” since developers opinion of AI is mostly positive:

> *Given the evidence from the survey that developers are rapidly adopting AI, relying on it, and perceiving it as a positive performance contributor, we found the overall lack of trust in AI surprising.*

To Google researchers, they see AI adoption increasing, and they see developers reporting greater productivity, and so the decrease in quality is interpreted as unexpected.



Page 22, Google DORA 2024 Report

However, the seemingly-contrary findings: “more code authored” with “greater percentage of defects” can be reconciled. When developers swap “refactoring” for “cloning” at the rate we observe, the default outcome is “more code” (“poor man’s productivity”) packaged with “increasing maintenance challenge” that manifests, in part, as a higher defect rate.

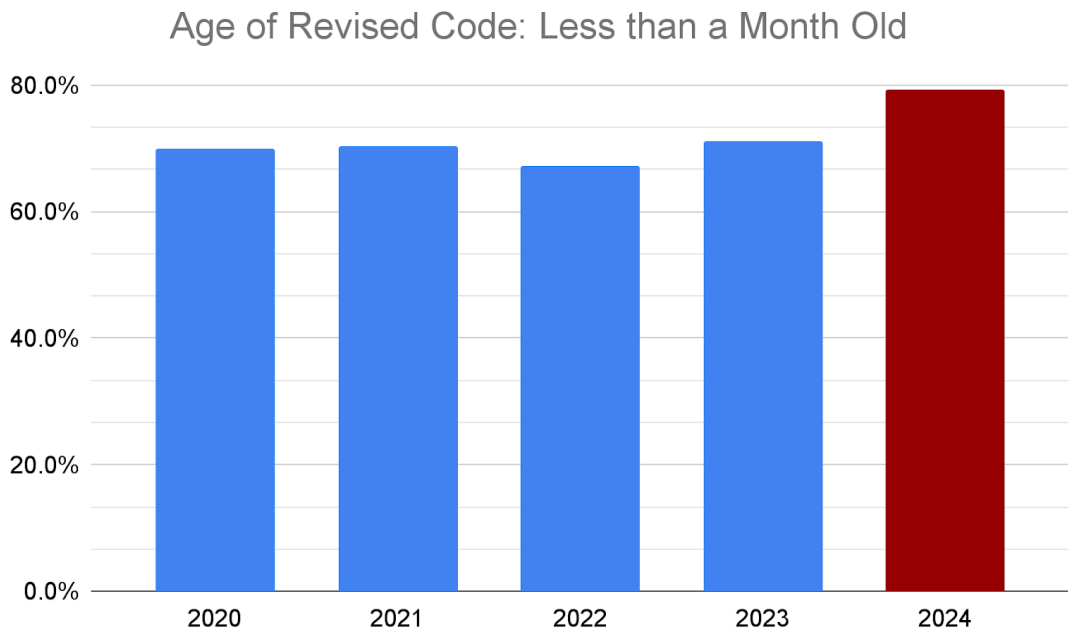
Changes to Age of Revised Code

Another means to evaluate how code authorship has evolved in the age of AI Assistants is to inspect “how much time is passing between when code was authored, and when it receives its next meaningful change?”

Here is the age of about eight million changed lines across our ~10,000 repo sample set over the past five years [A0]:

	Less than 2 weeks	2-4 weeks old	Less than one month old	Less than one year	1-2 years	2+ years
2020	60.4%	9.6%	70.0%	24.7%	4.2%	1.1%
2021	61.1%	9.3%	70.4%	24.2%	4.7%	0.7%
2022	57.5%	9.8%	67.3%	25.4%	6.5%	0.8%
2023	60.6%	10.6%	71.2%	21.9%	6.2%	0.8%
2024	69.7%	9.5%	79.2%	16.9%	2.6%	1.3%

Note that this data does not include newly added lines – we analyzed those in the “Trends in Code Changes” section. Here’s how the percentage of code being changed within a month of authorship looks since 2020:



Interpreting the Trend Toward Revising Newer Code

Performed in moderation, revising recent code is often a *good* thing. It suggests a developer is conscientious about submitting code that has been tested & polished [6].

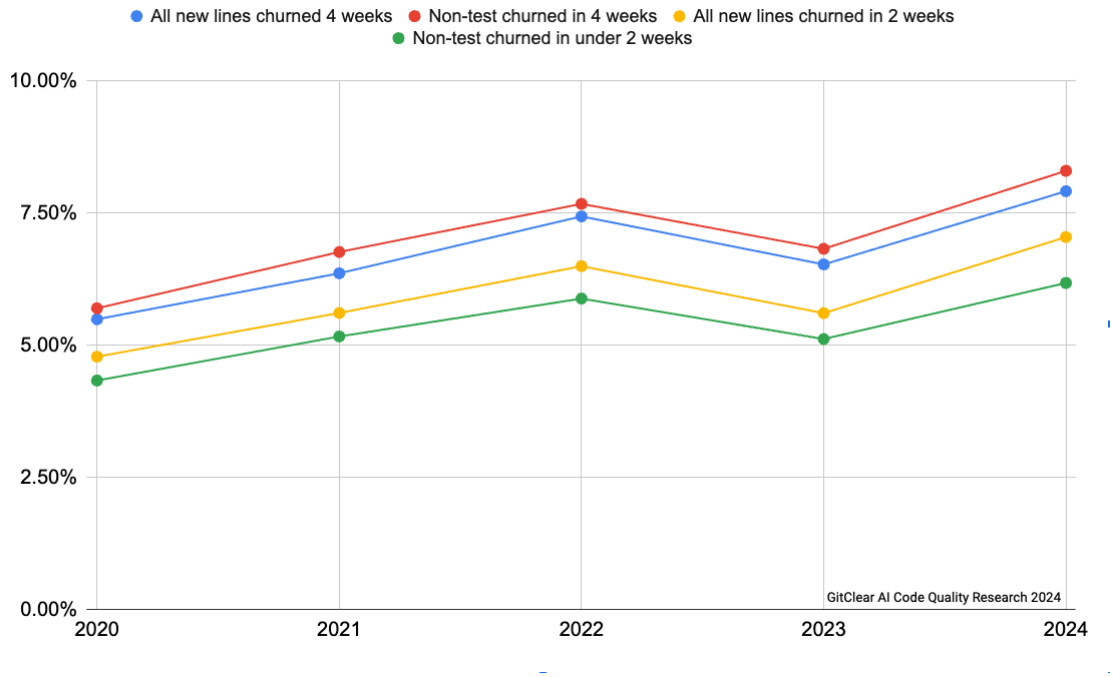
But the 2024 ratios for "what type of code is being revised" do not paint an encouraging picture. During the past year, only 20% of all modified lines were changing code that was authored more than a month earlier. Whereas, in 2020, 30% of modified lines were in service of refactoring existing code.

In git repos that maintain their change velocity over the course of years, developers must divide their attention between implementing new features and updating/removing legacy code. If every new feature is implemented via **added** code (without seeking out **update**, **move**, and **delete** opportunities), the repo grows crowded with *cloned* and *near-cloned* blocks. Adding new developers becomes increasingly expensive, as they struggle to determine which function, among a multitude of similar choices, should be considered the "canonical" version that their code should call.

Churn Percent of New Code

While the shift away from changing legacy code seems perilous to maintainability, perhaps this trend has a benign explanation. If developer teams are shifting toward writing code that implement AI features, it would be reasonable to expect that developers would have less occasion to reuse code that had been authored in the dark ages of AI, like… 2021.

To gauge whether the shift toward modifying newer code was driven by "changing team focus" vs "changing author tendencies," here's another set of yearly data points:



Percent of newly added code modified within 2 or 4 weeks, by year

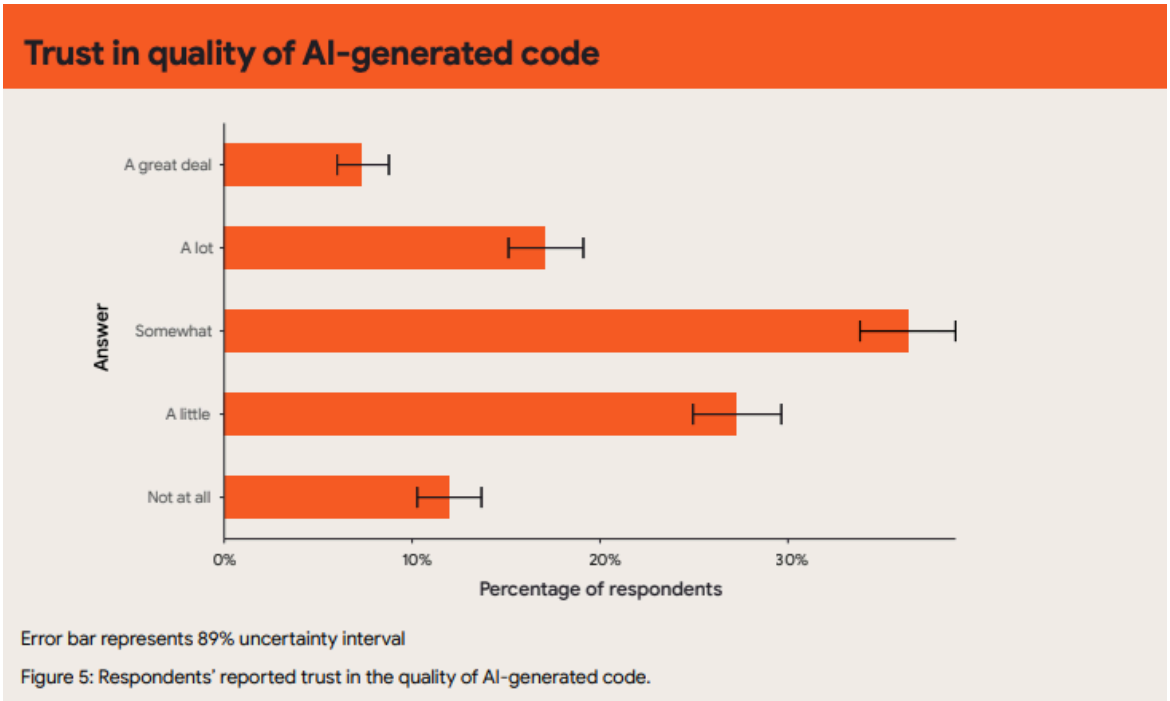
In the data table from [A7], we analyzed only lines that were new to the repo. For the 45 million lines that met this criteria, we measured the percentage of newly added lines that went on to be revised within 2 or 4 weeks.

The trend line here is a little cagey, with 2023 faking a return toward pre-AI levels. But if we consider 2021 as the “pre-AI” baseline, this data tells us that, during 2024, there was a 20-25% increase in the percent of new lines that get revised within a month.

Conclusion: Human Refactoring Edge

As assistants continue improving, humans still have distinct advantages

Google DORA's 2024 report proves that developers trust the current generation of AI assistants about as much as we trusted the previous generation, i.e., not much:



The median developer trusts AI-generated code "Somewhat," with good reason

There is a lot of utility that AI provides, but the data from this year affirms why long-term-oriented devs might eye their "tab" key with a faint sense of foreboding.

The never-ending rollout of more powerful AI systems will continue to transform the developer ecosystem in 2025. In an environment where change will be constant, we would suggest that developers emphasize their still-uniquely human ability to "simplify" and "consolidate" code they understand. There is art, skill and experience that gets channeled into creating well-named, well-documented modules. Executives that want to maximize their throughput in the "Age of AI" will discover new ways to incentivize reuse. Devs proficient in this endeavor stand to reap the benefits.

Citations

1. Google: Accelerate State of DevOps [Google, 2024]
2. Stack Overflow: 2024 State of Development Survey [Stack Overflow, 2024]
3. An empirical study on bug propagation through code cloning [ScienceDirect, Mondal et. al, 2019]
4. On the Relationship of Inconsistent Software Clones and Faults [Arxiv, Wagner et. al, 2016]
5. Exploring the Impact of Code Clones on Deep Learning Software [Association for Computing Machinery, 2023]
6. Four Worst Software Metrics Agitating Developers [GitClear, 2021]
7. Bug Replication in Code Clones: An Empirical Study [IEEE, Islam, Mondal, 2016]
8. Context-based detection of clone-related bugs [FSE Conference, Jiang, Su, 2007]
9. Assessing the effect of clones on changeability [IEEE, Lozano, Wermelinger, 2008]
10. Tracking clones' imprint [IWSC Proceedings of the 4th International Workshop on Software Clones, Lozano, Wermelinger, 2010]
11. Comparative stability of cloned and non-cloned code: an empirical study [ACM Symposium, Mondal et. al, 2012]
12. A Change-Type Based Empirical Study on the Stability of Cloned Code [IEEE, Rahman, Roy, 2014]
13. Cursor IDE automatically rewrites linter violations [Cursor, 2025]
14. CCFinder: a multilinguistic token-based code clone detection system for large scale source code [IEEE, Kamiya et al, 2002]
15. Oreo: detection of clones in the twilight zone [ACM, Saini, 2018]
16. Duplicate Code Block Detection [GitClear, 2025]

Appendix

Data used to build this research is included below.

A0: Code Change Definitions

To analyze how code quality is changing, we reviewed the differences in types of code changes observed in 2024 vs. the years prior. GitClear classifies code changes (operations) into seven categories. The first six operations are analyzed in this research:

1. **Added code**. Newly committed lines of code that are distinct, excluding lines that incrementally change an existing line (labeled "Updates"). "Added code" also does not include lines that are added, removed, and then re-added (these lines are labeled as "Updated" and "Churned")
2. **Deleted code**. Lines of code that are removed, committed, and not subsequently re-added for at least the next two weeks.
3. **Moved code**. A line of code that is cut and pasted to a new file, or a new function within the same file. By definition, the content of a "Moved" operation doesn't change within a commit, except for (potentially) the white space that precedes the content.
4. **Updated code**. A committed line of code based off an existing line of code, that modifies the existing line of code by approximately three words or less.
5. **Find/Replaced code**. A pattern of code change where the same string is removed from 3+ locations and substituted with consistent replacement content.
6. **Copy/Pasted code**. Identical line contents, excluding programming language keywords (e.g., `end`, `})`, `[`), that are committed to multiple files or functions within a commit.
7. **No-op code**. Trivial code changes, such as changes to white space, or changes in line number within the same code block. No-op code is excluded from this research.

Specific examples of GitClear's code operations can be found in the Diff Delta documentation. GitClear has been classifying git repos by these operations since 2020. As of January 2025, GitClear has analyzed and classified around a billion lines of code over five years, from a mixture of commercial customers (e.g., NextGen Health, Bank of Georgia) and popular open source repos (e.g., Facebook React, Google Chrome). 211 million lines of code were meaningful (not No-op) line changes, used for this research.

Along with the evolution of code change operations, we are also exploring the change in "**Churned code**." This is not treated as a code operation, because a churned line can be paired with many operations, including "Added," "Deleted," or "Updated" code. For a line to qualify as "churned," it must have been authored, pushed to the git repo, and then reverted or substantially revised within the subsequent two weeks. Churn is best understood as "changes

that were either incomplete or erroneous when the author initially wrote, committed, and pushed them to the company's git repo."

A1: Raw Data for Changed Line Counts

This data evolves by percentage points from year-to-year, as customers evolve, so our line counts this year were not identical to the numbers that our database produced at the beginning of 2024.

Year	Added	Deleted	Updated	Moved	Copy/pasted	Find/replaced	Lines changed	All-line Churn*
2020	**10,847,860**	**5,250,911**	**1,414,590**	**6,631,652**	**2,431,654**	**857,244**	27,433,911	837,906
2021	**14,614,537**	**7,281,627**	**1,883,202**	**9,275,559**	**3,257,592**	**1,322,834**	37,635,351	1,229,154
2022	**15,577,512**	**7,566,948**	**1,969,547**	**7,854,120**	**3,644,844**	**1,507,249**	38,120,220	1,261,756
2023	**20,945,576**	**10,405,205**	**2,735,862**	**7,804,193**	**5,229,684**	**2,021,600**	49,142,120	2,213,791
2024	**27,123,900**	**12,873,051**	**3,462,781**	**5,564,610**	**7,236,304**	**2,495,057**	58,755,703	3,331,468
2025	**34,101,000**	**15,899,000**	**4,376,000**	**2,146,000**	**9,613,000**	**3,005,000**	69,140,000	4,752,000

* This is any line that was authored, and then subsequently changed within 2 weeks. Since this includes lines that might get re-re-revised, we opted to focus this year more on the percent of churned lines that had been newly authored (in the Churn Percent section)

Here is how the data translates to percentages:

Year	Added	Deleted	Updated	Moved	Copy/pasted	Find/replaced	Churn
2020	39.54%	19.14%	5.16%	24.17%	8.86%	3.12%	3.05%
2021	38.83%	19.35%	5.00%	24.65%	8.66%	3.51%	3.27%
2022	40.86%	19.85%	5.17%	20.60%	9.56%	3.95%	3.31%
2023	42.62%	21.17%	5.57%	15.88%	10.64%	4.11%	4.50%
2024	46.16%	21.91%	5.89%	9.47%	12.32%	4.25%	5.67%
2025	49.32%	23.00%	6.33%	3.10%	13.90%	4.35%	6.87%

Open source repos that were analyzed as part of the data set included:

1. Chromium (chromium/chromium)
2. Facebook React (facebook/react)

3. Microsoft Visual Studio Code (microsoft/vscode)
4. React Native (facebook/react-native)
5. Postgres (postgres/postgres)
6. Kubernetes (kubernetes/kubernetes)
7. Signal (signalapp/signal-desktop)
8. Jax (google/jax)
9. Tensorflow (tensorflow/tensorflow)
10. Swift (swiftlang/swift)
11. Ruby (ruby/ruby)
12. Go (golang/go)
13. Clickhouse (clickhouse/clickhouse)
14. Babel (babel/babel)
15. Standard Notes (standardnotes/desktop, mobile & web)
16. Electron (electron/electron)
17. Shotcut (mltframework/shotcut)
18. Timescaledb (timescale/timescaledb)
19. Bluesky (bluesky-social/atproto)
20. Elasticsearch (elastic/elasticsearch)
21. Microsoft Powertoys (microsoft/powertoys)
22. Ruby on Rails (rails/rails)
23. Homebrew (homebrew/brew)
24. Python (python/cpython)
25. Ansible (ansible/ansible)

Any of these repos, plus about 80 others, can be visited at https://www.gitclear.com/open_repos to gather more data and stats on the code being authored in popular open source repos.

Queries used to produce data

The data was stored in a Postgres database and was queried via Ruby on Rails' ActiveRecord.

```
# Operation by year
2020.upto(2024).map { |year| CodeLine.where(authored_at: Time.new(year, 1,
1)..Time.new(year + 1, 1, 1), commit_impacting:
true).group(:operation_em).count }
# Commits by year
Commit.impacting.where(authored_at:
Time.local(2020,1,1)..Time.local(2025,1,1)).group("EXTRACT (year from
authored_at)").count
# Committers committing by year
2020.upto(2024).map { |year|
Committer.joins(:commits).merge(Commit.impacting).where(commits: {
```

```
authored_at: Time.local(year,1,1)..Time.local(year+1,1,1)
}).group(:id).count.size }
# Repos changed by year
2020.upto(2024).map { |year|
Repo.joins(:commits).merge(Commit.impacting).where(commits: { authored_at:
Time.local(year,1,1)..Time.local(year+1,1,1) }).group(:id).count.size }
# Files by year
2020.upto(2024).map { |year|
CommitCodeFile.impacting.joins(:commit).where(commits: { authored_at:
Time.local(year,1,1)..Time.local(year+1,1,1) }).group(:id).count.size }
```

A2: Largest Database of Structured Code Data

As of February 2025, there is no publicly available dataset that indexes code changes. Published data on code stats originates only from the handful of companies that hold or evaluate code data. Among these companies, including GitHub, GitLab, Bitbucket, Azure Devops, GitKraken, and SonarQube, all classify code changes as a binary “add” or “delete” [example]. GitClear is the only company thus far that recognizes a broader set of operations:

1. Added code
2. Updated code
3. Deleted code
4. Copy/pasted code
5. Find/replaced code
6. Moved code
7. No-op code

GitClear’s data is split about two-thirds private corporations that have opted in to anonymized data sharing, and one-third open source projects (mostly those run by Google, Facebook, and Microsoft). In addition to the code operation data, GitClear’s data set also segments and excludes lines if they exist within auto-generated files, subrepo commits, and other exclusionary criteria enumerated in this documentation. As of February 2025, that documentation suggests that a little less than half of the “lines changed” by a conventional git stats aggregator (e.g., GitHub) would qualify for analysis among the 211m lines in this study. The study does include commented lines – future research could compare comment vs. non-comment lines. It could also compare “test code” vs “other types of code,” which probably influences the levels of copy/paste.

If you know of other companies that report code operations of comparable granularity, please contact hello@gitclear.com and this section will be updated, and a new PDF document will be uploaded with credit given to the contributor (if desired).

A3: GitClear Solutions

All of the data analyzed in this report is available to GitClear customers with a basic subscription. Many of the stats are even available to developers on GitClear's free "Starter" subscription. Any curious developer or manager using GitHub, BitBucket, GitLab or Azure Devops can sign up for a trial of GitClear. **No credit card is required to start a trial**. Data is analyzed from "Most recent' backward, so within hours, you will be able to assess how your team's "Moved," "Copy/Pasted," and "Duplicated Blocks" prevalence compares to the industry benchmarks we have reported.

Here's a rundown of the pertinent documentation for the reports that were referenced in this research:

1. Viewing "Copy/Paste" and "Moved" percent compared to industry benchmarks
2. Setting up team goals to track where cloned code blocks are being introduced
3. "Code Operation" and "Diff Delta velocity" reports
4. Code Line Provenance graph: How old is the code that is being revised by team, by repo, or by time range?
5. Tech Debt browser helps to identify directories with atypical velocity or bug characteristics
6. Pull Request Review tool: provably reduces lines to review by roughly 30%

For the metrics that matter most, many teams opt to set up Slack notifications. For example, notifications can be set up to trigger if:

- A pull request has been awaiting review longer than one business day
- More than **n** business days (n = user configurable) have passed since the last activity (comment or commit) on a pull request
- A pull request has experienced more than **n** rounds of feedback
- Multiple cloned code blocks are found, but one was modified while the other(s) weren't
- The team's overall percentage of "Bug work" exceeds some pre-determined threshold

GitClear also offers Automated Changelogs, which can be used within a GitHub Profile [example], a repo readme [same invocation as on profile], or embedded within a product's website [example].

A4: Prediction on Google DORA results

The language of the specific prediction made by the 2024 Coding on Copilot research:

If the current pattern continues into 2024, more than 7% of all code changes will be reverted within two weeks, double the rate of 2021. Based on this data, we expect to see an increase in Google DORA's "Change Failure Rate" when the “2024 State of Devops” report is released later in the year, contingent on that research using data from AI-assisted developers in 2023.

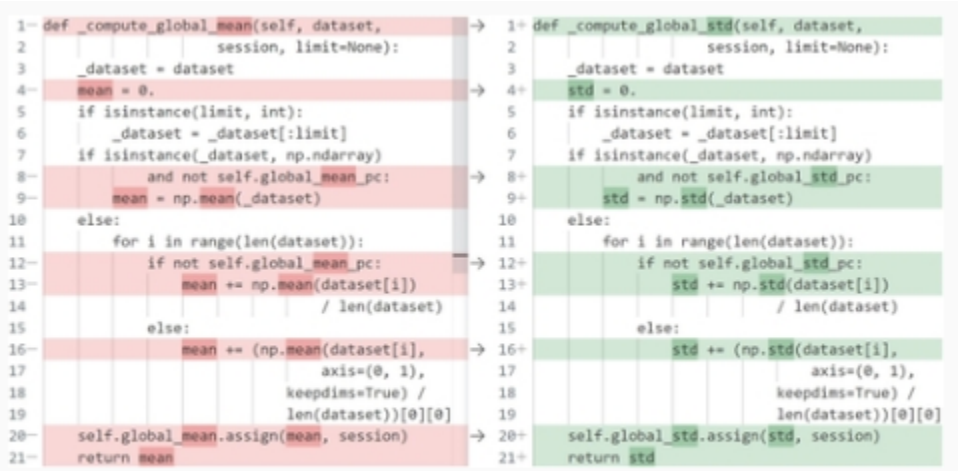
Thankfully, short-term churn has not yet reached 7%, but the overall shift toward increasingly recent code being revised was sufficient to drive the increase in defects noted by the 2024 Google DORA report.

A5: Delineating Between “Code Clone” Types 1, 2 and 3

Most of the existing research on code clones differentiates between what are called “Type 1,” “Type 2,” and “Type 3” clones. In Exploring the Impact of Clones on Deep Learning Software, these clone types are described as:

> A code clone means a pair of code fragments that are identical or nearly similar. It commonly happens when programmers copy and paste code over different locations or reuse framework or design patterns [2]. In this article, we leveraged the tool NiCad [56] to detect code clones in DL software. NiCad has been widely used for detecting code clones and with high accuracy [10, 24, 42, 48, 54]. It categorizes code clones into three types according to the similarity between involved code fragments:
>
> - **Type 1**, indicates two code fragments are identical except for blank spaces or comments. It is also called an exact clone.
> - **Type 2**, indicates two code fragments have similar syntax, but contain different variable names, constant names, function names, and so on. It is also called a renamed clone. Figure 1 shows an example of a Type 2 clone.
> - **Type 3**, indicates two code fragments that add, delete, or modify statements on the top of a Type 2 clone. It is also called a gap clone. Figure 2 shows an example of a Type 3 clone.

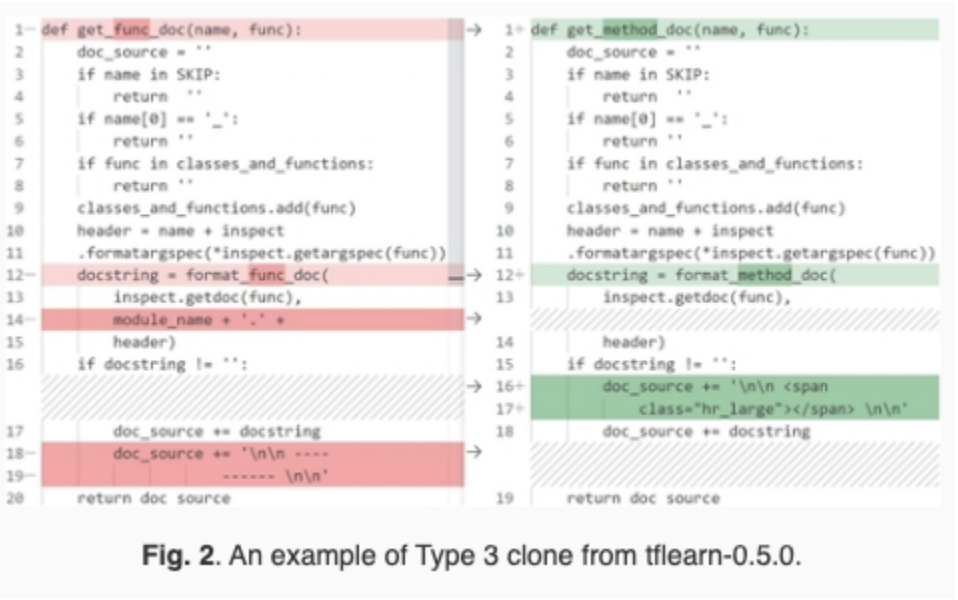
They offer the following visual examples:



```
def _compute_global_mean(self, dataset,
                         session, limit=None):
    _dataset = dataset
    mean = 0.
    if isinstance(limit, int):
        _dataset = _dataset[:limit]
    if isinstance(_dataset, np.ndarray)
            and not self.global_mean_pc:
        mean = np.mean(_dataset)
    else:
        for i in range(len(dataset)):
            if not self.global_mean_pc:
                mean += np.mean(dataset[i])
                                / len(dataset)
            else:
                mean += (np.mean(dataset[i],
                                 axis=(0, 1),
                               keepdims=True) /
                               len(dataset))[0][0]
    self.global_mean.assign(mean, session)
    return mean
```

```
def _compute_global_std(self, dataset,
                        session, limit=None):
    _dataset = dataset
    std = 0.
    if isinstance(limit, int):
        _dataset = _dataset[:limit]
    if isinstance(_dataset, np.ndarray)
            and not self.global_std_pc:
        std = np.std(_dataset)
    else:
        for i in range(len(dataset)):
            if not self.global_std_pc:
                std += np.std(dataset[i])
                               / len(dataset)
            else:
                std += (np.std(dataset[i],
                               axis=(0, 1),
                             keepdims=True) /
                             len(dataset))[0][0]
    self.global_std.assign(std, session)
    return std
```

Fig. 1. An example of Type 2 clone from tflearn-0.5.0.



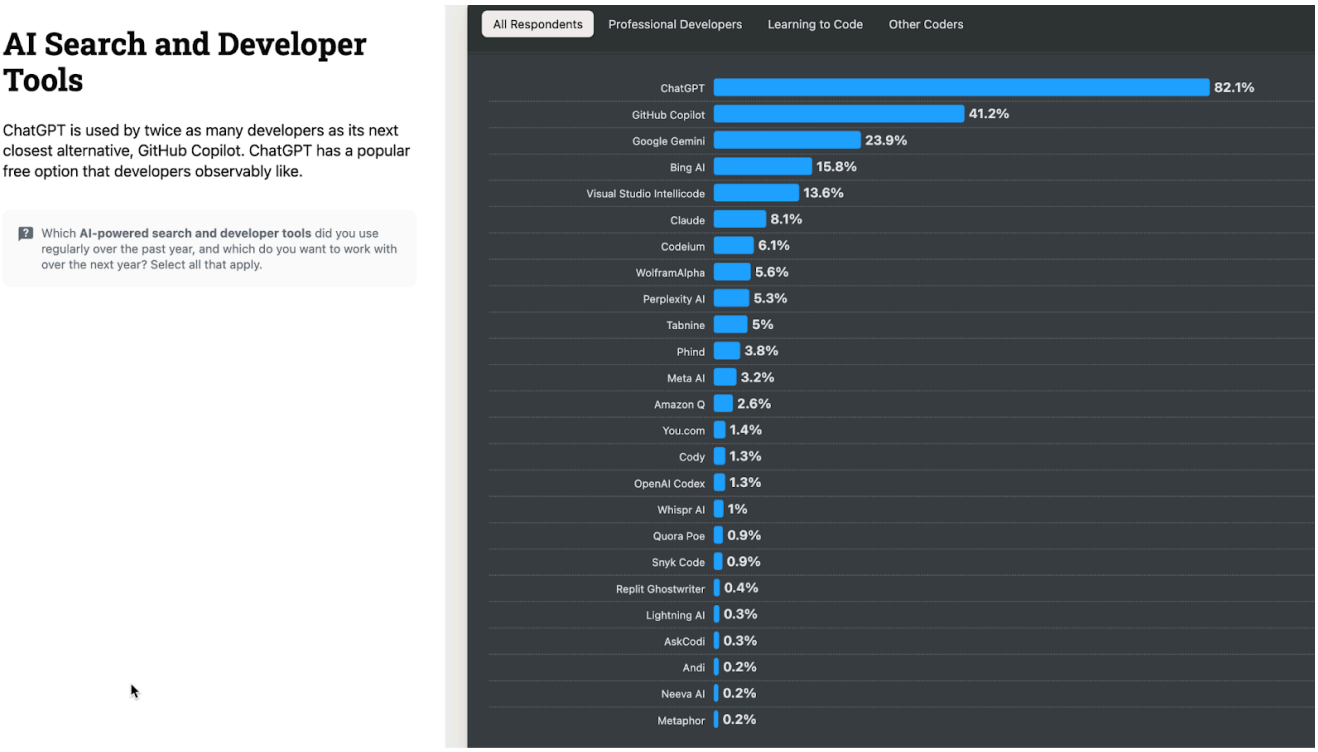
```
def get_func_doc(name, func):
    doc_source = ''
    if name in SKIP:
        return  ''
    if name[0] == '_':
        return ''
    if func in classes_and_functions:
        return ''
    classes_and_functions.add(func)
    header = name + inspect
    .formatargspec(*inspect.getargspec(func))
    docstring = format_func_doc(
        inspect.getdoc(func),
        module_name + '.' +
        header)
    if docstring != '':

        doc_source += docstring
        doc_source += '\n\n ----
                      ------ \n\n'
    return doc source
```

```
def get_method_doc(name, func):
    doc_source = ''
    if name in SKIP:
        return  ''
    if name[0] == '_':
        return ''
    if func in classes_and_functions:
        return ''
    classes_and_functions.add(func)
    header = name + inspect
    .formatargspec(*inspect.getargspec(func))
    docstring = format_method_doc(
        inspect.getdoc(func),

        header)
    if docstring != '':
        doc_source += '\n\n <span
            class="hr_large"></span> \n\n'
        doc_source += docstring


    return doc source
```

Fig. 2. An example of Type 3 clone from tflearn-0.5.0.

A6: Estimating the size of coding assistant context window in 2024

According to the 2024 Stack Overflow Developer Survey [2], among code assistants that integrate with IDEs (where files can be analyzed), GitHub Copilot was by far the most utilized:



GitHub Copilot, with its 41.2% usage, has about 3.5x greater adoption than the next-highest IDE tool, Visual Studio Intellicode.

How much context window did GitHub Copilot offer in 2024? GitHub does not offer a documented number for this, but it's not especially difficult to piece together an estimation:

- Reddit from post August 2024 asking why context window is limited to 4-8k depending on backend chosen
- GitHub blog post from December 2024 announcing the new availability of 64k context for queries made to ChatGPT-4o, implies that context window couldn't have been more than 32k previously in 2024
- GitHub issue from 2023 asking why copilot is limited to 8192 tokens

Taken together, it seems likely that GitHub Copilot offered context in the 4-8k range, as the Reddit post describes. If a project's files average 300 lines of 50 character-long lines, that implies that GitHub Copilot would be able to send at most (300 lines * 50 characters) / (4 characters per token) = 3,750 tokens per file, which would imply that around three full files of this magnitude could fit into an 8k context window.

While this extrapolation relies on a heavy dose of "back-of-napkin math," it is consistent with all published information to estimate that the most popular code assistant could probably fit no more than 10 full files of context during 2024. The rest of the most popular code-writing

and inquiry tools don't have direct access to a project's files, so moved code opportunities aren't a possibility.

A7: Code Provenance Report

"Code Provenance" is the label given to the graph on GitClear that shows the age of changed code lines – per team and/or per repo, over a user-choosable duration.

Code Provenance derivation is described within the GitClear documentation. Code provenance data that was used to produce the chart:

	New lines added	**Revised in under month**	**Churned in month**	**Revised under 2 weeks**	**Churned in 2 week**	**New exl test**	**Under month**	**Percent churned**	Revised under 2 weeks	Churned under 2 weeks
2020	5,434,044	298,369	5.49%	259,921	4.78%	4,759,520	271,202	5.70%	235464	4.33%
2021	6,374,719	405,372	6.36%	357,602	5.61%	5,499,090	371,847	6.76%	329250	5.16%
2022	7,118,977	529,287	7.43%	462,304	6.49%	6,261,462	480,336	7.67%	418699	5.88%
2023	13,805,181	901,014	6.53%	773,878	5.61%	12,079,904	824,071	6.82%	706422	5.12%
2024	12,762,612	1,009,552	7.91%	898,991	7.04%	10,700,097	887,648	8.30%	788268	6.18%

A8: Duplicate block detection method

The method by which GitClear detects duplicate blocks is described within the GitClear documentation.

To backfill years prior to 2024, we queued up the largest (by Diff Delta) 1,000 commits made to each repo between 2020 and 2023. Each of these 1,000 commits were evaluated for the presence of Type 1 cloned code blocks [A5].

Since our backfill method gave precedence to analyzing large commits, and "more lines" implies "more opportunity to contain a duplicate block," our method should bias toward reporting a greater percentage of "commits with a duplicate block" for the years prior to 2024.

That the analyzed commits showed negligible code cloning, relative to 2024, further supports the notion that cloned code has been on a steep upward trajectory.

A9: Additional research on estimating the detriment of duplicated code blocks

In the effort to keep the main body of this research maximally concise, we limited our research discussion primarily to the work of Ran Mo, Yao Zhang, et al [5], but there is a wide corpus of research on this subject. A few other notable contributions to this field of research include the following.

An Empirical Study on Bug Propagation through Code Cloning

A 2017 paper by Mondal, Roy, Roy, Schneider [4] investigates the prevalence and impact of cloned code lines on development. The "Highlights" section cites "severe bugs can sometimes get propagated through code cloning," and "around 18.42% of the buggy code clones are involved with bug propagation."

The authors further elaborate: "According to our study on thousands of commits of four open-source subject systems written in Java, up to 33% of the clone fragments that experience bug-fix changes can contain propagated bugs. Around 28.57% of the bug-fixes experienced by the code clones can occur for fixing propagated bugs."

This paper's finding is mirrored by a similar paper, On the Relationship of Inconsistent Software Clones and Faults, by Wagner et. al, which studied code duplication at TWT GmbH in 2016 [4]. These researchers focused on "type-3 clones" which are "inconsistent clones" where minor differences existed between otherwise-identical systems [A5]. Wagner found that, among these type-3 clones, 17% contained faults.

While their methods (and repos analyzed) were disparate, the papers agreed within 1.4% in their assessment of bug prevalence within code clones. The fact that both research teams detected a 17-18% rate of propagated bugs lurking in cloned code suggests a level of robustness to their collective estimation.

Compendium of research: OpenAI Deep Research

This document contains a synopsis of several industry papers and academic studies on the impact of "code cloning" on various measurable outcomes. It was generated by OpenAI's "Deep Research" functionality that debuted in February 2025.

OpenAI Assistant Data Projection

code_interpreter

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import re

# Load the data into a pandas DataFrame
data = {
    "Year": [2020, 2021, 2022, 2023, 2024],
    "Added": ["10,847,860", "14,614,537", "15,577,512", "20,945,576", "27,123,900"],
    "Deleted": ["5,250,911", "7,281,627", "7,566,948", "10,405,205", "12,873,051"],
    "Updated": ["1,414,590", "1,883,202", "1,969,547", "2,735,862", "3,462,781"],
    "Moved": ["6,631,652", "9,275,559", "7,854,120", "7,804,193", "5,564,610"],
    "Copy/pasted": ["2,431,654", "3,257,592", "3,644,844", "5,229,684", "7,236,304"],
    "Find/replaced": ["857,244", "1,322,834", "1,507,249", "2,021,600", "2,495,057"],
    "Lines changed": ["27,433,911", "37,635,351", "38,120,220", "49,142,120", "58,755,
    "Churn": ["837,906", "1,229,154", "1,261,756", "2,213,791", "3,331,468"]
}

# Convert the numbers in string format with commas to integers
for key in data.keys():
    if key != "Year":
        data[key] = [int(re.sub(",", "", value)) for value in data[key]]

df = pd.DataFrame(data)

# Train a quadratic model
X = df['Year'].values.reshape(-1, 1)
y = df.drop('Year', axis=1)

# Polynomial of degree 2
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

# Fit a model for each column
models = {}
for column in y.columns:
    model = LinearRegression()
    model.fit(X_poly, y[column])
    models[column] = model

# Predict for the year 2025
X_predict = poly.transform(np.array([[2025]]))
predictions = {}
for column in models:
    predictions[column] = models[column].predict(X_predict)[0]

predictions
```

Provenance interpreter

Using a quadratic projection model based on the provided data, the estimated values for each column in 2025 are as follows:

- Added: 34,101,004
- Deleted: 15,899,504
- Updated: 4,376,201
- Moved: 2,146,658
- Copy/pasted: 9,613,909
- Find/replaced: 3,005,949
- Lines changed: 69,143,225
- Churn: 4,752,489

Updates

- No updates have been made since publication on February 4, 2025. We will list substantive revisions to the research here, to the extent they transpire.

Contact information

It's tempting to predict that the momentum of "Added," "Churned" and "Copy/Paste" code lines are fueling a rise in the number of lower-quality pull requests being fielded by dev teams. If you have any experience with, or interest in, this question, drop us a line and we might investigate it in 2025. If we do investigate it, we pledge to write up and make our results publicly available, whether they demonstrate a positive, negative, or null result.

If you would like to discuss this research, or have ideas on how to improve it, please contact hello@gitclear.com or bill@gitclear.com directly.

We are happy to consider improvements to the clarity of this writing, or to explain how GitClear can help teams measure the metrics explored by this research.