Already Fixed, Too Late

What 90 open-source repos reveal about code review blind spots

Guy Vago | *AI Researcher & Software Architect, Qodo*

March 2026

Introduction

A pull request merges into a widely used open-source project. It introduces a subtle bug: a sort comparator function that returns a distance value instead of the negative/zero/positive contract that Array.sort() expects. Elements end up in a nondeterministic order. The code review approves it. The CI pipeline is green. The bug ships to every user who updates.

255 days later, a different developer submits a one-line fix for that exact issue. The fix matched, almost word for word, what Qodo's automated review had flagged on the original pull request.

I wanted to know how often this happens. Not in theory, not as a thought experiment, but measured across real codebases with real commit histories. So I ran Qodo's code review engine against 90 major open-source repositories and analyzed merged pull request pairs: one that introduced a problem, and one that later fixed it.

This pattern repeated 1,258 times across the 90 repositories I studied. The repository's own history confirms each one. A developer wrote a fix, which means the bug was real. This report presents the full findings, methodology, and data from that study.

The study

Selection criteria

I selected 90 widely-used repositories across 11 programming languages, prioritizing projects with active development and a meaningful volume of merged pull requests over the past year. The selection was not random. I chose projects that are well-maintained, actively reviewed, and widely depended upon, because these are the projects where review culture is strongest and the bar is highest.
The languages span Python, TypeScript, JavaScript, Go, Rust, Java, Ruby, PHP, C/C++, and Swift.
The repositories include Django, Angular, VS Code, Terraform, Selenium, Webpack, Supabase, Homebrew, Deno, OpenCV, Prometheus, Mastodon, Ghost, Discourse, Excalidraw, Storybook, n8n, NestJS, Puppeteer, and dozens more. The full list is in Appendix A.

What I measured

For each repository, I analyzed pairs of merged pull requests: one that introduced a defect and one that later addressed it. A finding is only counted as proven if a subsequent PR in the same repository fixes the exact issue that was flagged. Each proven finding is validated against the repository's own commit history. The fix PR, written by a human developer, confirms the bug was real.
I also validated a subset of findings that had not yet been fixed, confirming them as still present in the current codebase. Due to the scale of the study, this validation was limited to a representative sample per repository.
The analysis was performed using Qodo's code review engine.
Several of the repositories in this study already use AI-powered code review tools. The bugs I found were present in pull requests that had been reviewed by both human

reviewers and automated tools. The blind spot is not unique to human review or to any particular tool. It is a property of reviewing changes in isolation, without full codebase context.

What I did not measure

This study does not claim to find all bugs in these repositories. It measures one specific thing: how often a bug introduced in a pull request is later fixed by a separate pull request, and how long that takes. The actual number of bugs across these codebases is certainly higher than what I report here.
I also did not measure the severity of each bug in terms of user impact, incident frequency, or production cost. Those are real factors, but they require access to internal telemetry that I did not have.

Findings

1,258 proven bugs

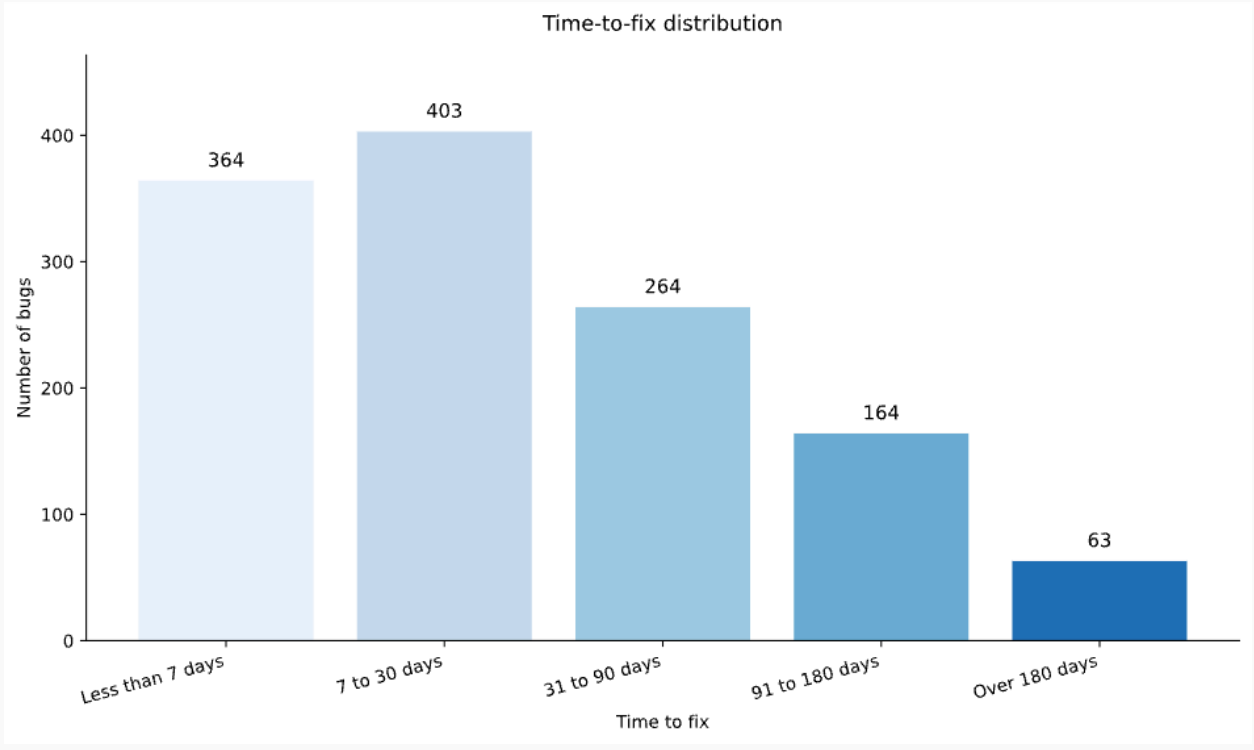
Across 90 repositories, I found 1,258 cases where a bug was introduced in one pull request and proven real by a fix in a later one. The pattern appeared consistently across every language and project type in the dataset.
In addition, I confirmed 449 bugs that are still present in production code across these repositories today.
These numbers represent only the findings I fully validated.
Every finding in this study was flagged by Qodo's review engine on a merged pull request and confirmed to still be present in the repository at the time of analysis. The term "proven" refers to the subset where a separate, later pull request fixed the exact issue, providing independent confirmation from the repository's own history. The remaining findings were validated as present in the current codebase but have not yet

been fixed. "Proven" is a higher bar of evidence, not a higher bar of severity. Both categories represent real bugs in production code.

The time gap

Of the 1,258 bugs that were eventually fixed, I measured the time between the pull request that introduced each bug and the pull request that fixed it.
The median gap was 16 days. The mean, pulled higher by bugs that survived for months, was 46 days.
The distribution:

Time to fix	Percentage	Count
Less than 7 days	29%	364
7 to 30 days	32%	403
31 to 90 days	21%	264
91 to 180 days	13%	164
Over 180 days	5%	63



Consider what each of these fixes actually costs. A developer writes code, submits it for review, gets approval, and merges. Days or weeks pass. Then someone, often a different developer entirely, discovers the bug. That second developer now has to understand code they did not write, reproduce the problem, trace it to its root cause, and submit a fix. Meanwhile, the original author has moved on to other work and other context.

Every one of these 1,258 fixes represents a developer stopping what they were doing to go back and repair something that could have been caught before it merged.

The short-lived bugs

The bugs fixed within a week (29% of the total) were often the most frustrating: a wrong default value, a missing null check, an off-by-one in a loop bound. The kind of thing that takes seconds to fix during a code review but hours to diagnose after it reaches production.

These quick fixes might seem harmless. They are not. They represent the most efficient

waste in a development process. The knowledge to prevent them existed at review time. It was not applied. The cost of preventing them was close to zero. The cost of fixing them after merge includes a context switch, a debugging session, a new PR, another review cycle, and another deployment.

The long-lived bugs

The bugs that survived for months were different in character: race conditions, incorrect state management, silent error swallowing, edge cases in configuration parsing. These are the bugs that only surface under specific conditions. They accumulate user reports and confused debugging sessions before someone finally connects the symptoms to the cause.

In one case, a project shipped a bug where opening a reply form triggered a sequential API loop that fetched all comments one page at a time, blocking the UI. It took 259 days for someone to notice and fix the performance regression.

In another, a database connection was used before being initialized, but only on fresh connections after an idle timeout. That one survived 229 days.

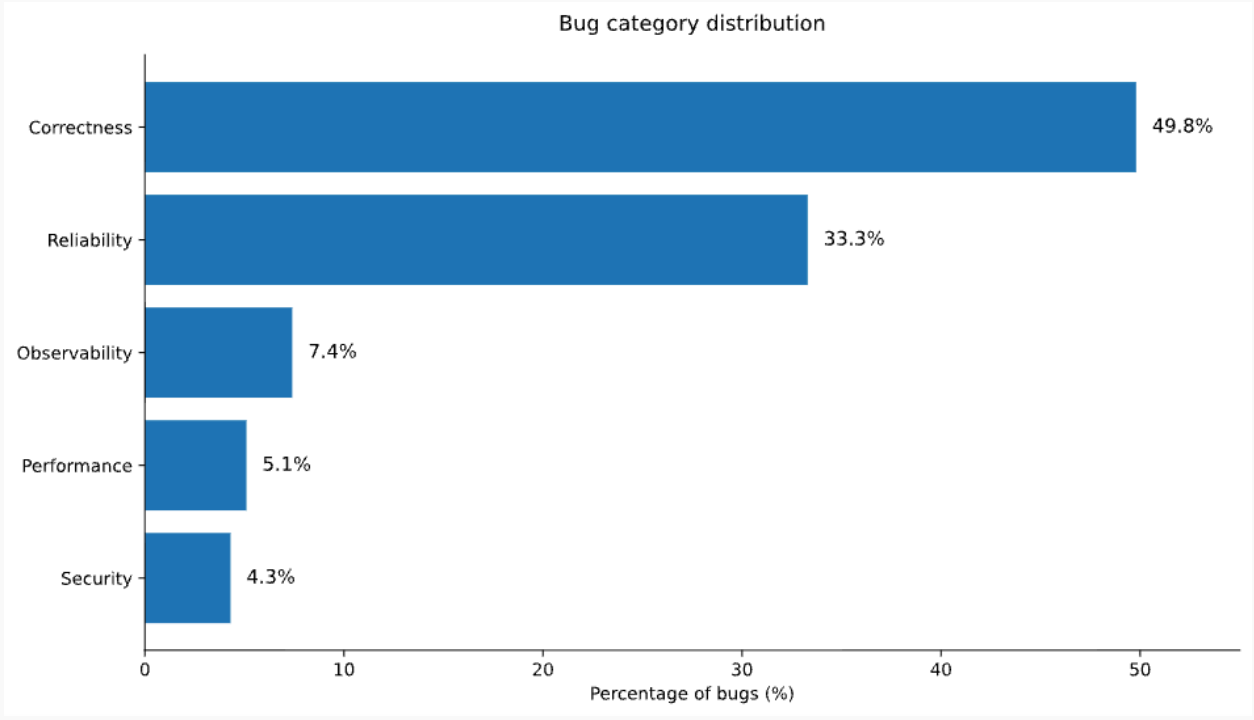
A desktop application committed generated type declaration files that should have been in .gitignore. The unnecessary files lived in the repository for 273 days before someone cleaned them up.

A CRM application hardcoded a search result limit of five items, meaning users could never see more than five results regardless of how many matched their query. That limit stayed in place for 252 days.

Finding categories

The bugs I found fall into five categories:

Category	Percentage	Description
Correctness	49.8%	Wrong behavior: incorrect logic, bad return values, broken contracts
Reliability	33.3%	Fragile behavior: crashes, unhandled errors, null dereferences
Observability	7.4%	Silent failures: swallowed exceptions, missing logs, lost errors
Performance	5.1%	Unnecessary work: redundant calls, blocking operations, memory waste
Security	4.3%	Unsafe behavior: stale auth state, missing input validation



Half the bugs are straightforward correctness issues: the code does the wrong thing. A third are reliability problems: code that works under normal conditions but fails on edge cases, unexpected input, or unusual state. The remaining 17% are issues that may not break functionality immediately but degrade the system over time: lost errors that make debugging harder, unnecessary work that slows things down, security gaps that go unnoticed until exploited.

Finding severity

Beyond categories, each finding was classified by severity: how urgently it needs attention.

Severity	Percentage	Description
Critical	61%	Requires immediate action: bugs, data loss risks, security gaps

Severity	Percentage	Description
Important	36%	Should be addressed: reliability issues, error handling gaps
Minor	3%	Worth noting: style-adjacent issues with functional impact

Critical	 61%
Important	 36%
Minor	 3%

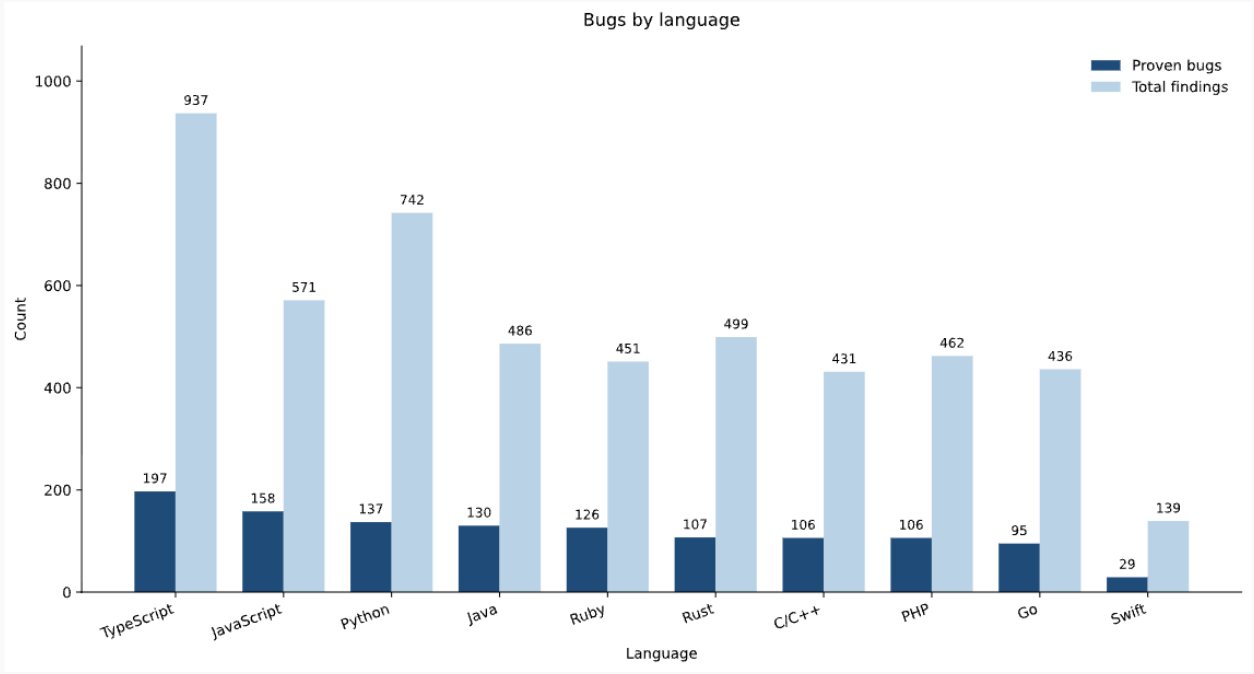
Nearly two-thirds of all findings are critical. These are not style suggestions or optional improvements. They are bugs that will affect users.

Bugs by language

The bugs appeared across every language in the study. No language was immune.

Language	Repos	Proven (fix exists)	All confirmed findings
TypeScript	9	197	937
JavaScript	11	158	571
Python	11	137	742

Language	Repos	Proven (fix exists)	All confirmed findings
Java	9	130	486
Ruby	9	126	451
Rust	8	107	499
C/C++	8	106	431
PHP	8	106	462
Go	9	95	436
Swift	4	29	139



TypeScript and JavaScript repositories had the highest absolute count of proven bugs, which likely reflects the size and activity level of those projects (VS Code, Angular, Supabase, Ghost, webpack) rather than anything inherent to the language. The distribution across all other languages is remarkably even.

Code health scores

I scored each repository on a 0-100 scale. The score is based on the number and severity of findings per pull request, weighted so that critical issues count three times more than minor ones. The result is compared against a benchmark of 47 repositories to produce a percentile rank, then adjusted for the project's annual volume of pull requests. Standard letter grades apply: A is 90 or above, F is below 60.

Grade	Repos	Percentage
A	12	13%

Grade	Repos	Percentage
B	3	3%
C	11	12%
D	18	20%
F	46	51%

Over half the repositories in this study scored an F. 71% scored D or F. These are projects with thousands of GitHub stars, active contributor communities, and established review cultures. The scores reflect a structural gap in code review, not a lack of effort.

One per language. To show that the pattern is not specific to any ecosystem, here is the lowest-scoring repository for each of the ten primary languages in the study:

Repository	Stars	Language	Score	Findings/PR	Critical
spacedrive	38k	TypeScript	3	4.5	58
Maccy	19k	Swift	17	3.0	7
alist	49k	Go	18	2.2	26
htmx	48k	JavaScript	19	2.3	34
supertokens-core	15k	Java	20	2.5	20
bat	58k	Rust	22	2.1	14
anything-llm	57k	Python	22	2.0	74
koel	17k	PHP	22	1.9	70
vagrant	27k	Ruby	23	2.0	11
whisper.cpp	48k	C/C++	25	2.1	35

These ten projects have a combined 360,000+ GitHub stars. The pattern is not about any one language or team. It is about what code review can and cannot see.

The bugs still in production

Not all the bugs I found have been fixed.

Hundreds of confirmed issues remain in production code across these repositories today. These are not theoretical risks surfaced by a static analysis scanner. They are logic errors, race conditions, and incorrect error handling in code that is running right now.

Details about these bugs are not included in this report. See the Responsible Disclosure section below.

What this tells us

Code review is effective. It catches style issues, enforces conventions, verifies naming, and flags obvious logic errors. Teams that practice code review ship better software than teams that do not.

But the data from this study points to a consistent blind spot. Code review struggles with context: how a change interacts with the broader codebase, edge cases that emerge from specific state combinations, regressions that appear perfectly correct when viewed as an isolated diff.

This is not about blaming reviewers. The developers who reviewed these 1,258 pull requests were doing their jobs, often on large and complex codebases, under real deadlines and competing priorities. Many of these repositories have rigorous review cultures with multiple required approvers. The bugs still got through.

The gap is structural. A human reviewer looking at a diff sees the lines that changed. What they cannot easily see is the full set of assumptions those lines depend on, the state that flows through them at runtime, or the interactions with code in other files that the diff does not show. No amount of diligence fully closes that gap, because it requires holding more context than a review interface presents.

The question is not whether your team is good at code review. It is whether code review alone is enough.

Methodology notes

Analysis scope

Each repository was analyzed over approximately one year of merged pull request history. The analysis focused on pull requests that introduced changes to source code (documentation-only, test-only, dependency-update, and version-bump PRs were filtered out before analysis).

What "proven" means

A bug is counted as proven if and only if a separate, later pull request in the same repository fixes the exact issue. The matching is performed by Qodo's analysis engine and validated against the commit history. The fix PR must address the same code location and the same logical defect. Coincidental fixes to the same file do not count.

Validation limits

For each repository, I validated up to five findings as still present in the current codebase. This limit was a practical constraint of the study's scale, not a reflection of the actual number of unfixed bugs. The 449 confirmed-in-production bugs are therefore a lower bound.

What this study does not claim

This study does not claim that these repositories are poorly maintained. The opposite is true: I chose well-maintained, actively reviewed projects specifically to test code review effectiveness under favorable conditions.
This study does not claim that Qodo would have prevented every bug. It claims that the

bugs were detectable at the time of the introducing PR, and that the fix PR later confirmed they were real.
This study was conducted using a subset of Qodo's analysis capabilities.

Responsible disclosure

Findings that represent confirmed, unfixed bugs are being handled through a coordinated disclosure process. No specific bug details, code snippets, or repository-specific findings will be made public before the affected maintainers have been notified and have had reasonable time to respond.
This report contains only aggregate statistics about unfixed bugs. No information in this report can be used to identify or exploit a specific vulnerability.
Maintainers can contact Dana Fine (dana.f@qodo.ai) or me (vago@qodo.ai) to request a review of findings related to their project.

Appendix A: Repositories

---

90 repositories across 11 languages.
C/C++ (8): dragonflydb/dragonfly, duckdb/duckdb, ggml-org/whisper.cpp, nlohmann/json, opencv/opencv, qbittorrent/qBittorrent, shadps4-emu/shadPS4, telegramdesktop/tdesktop
Go (9): AlistGo/alist, caddyserver/caddy, fatedier/frp, go-gitea/gitea, gofiber/fiber, hashicorp/terraform, jesseduffield/lazygit, rclone/rclone, traefik/traefik
Java (9): OpenAPITools/openapi-generator, SeleniumHQ/selenium, bazelbuild/bazel, dbeaver/dbeaver, google/guava, halo-dev/halo, redisson/redisson, skylot/jadx, supertokens/supertokens-core

JavaScript (11): axios/axios, bigskysoftware/htmx, excalidraw/excalidraw, google/zx, jquery/jquery, mermaid-js/mermaid, mozilla/pdf.js, mrdoob/three.js, puppeteer/puppeteer, storybookjs/storybook, webpack/webpack
PHP (8): bagisto/bagisto, filamentphp/filament, firefly-iii/firefly-iii, fruitcake/laravel-debugbar, koel/koel, krayin/laravel-crm, livewire/livewire, symfony/symfony
Python (11): Mintplex-Labs/anything-llm, astropy/astropy, django/django, langgenius/dify, matplotlib/matplotlib, pydata/xarray, pylint-dev/pylint, pytest-dev/pytest, scikit-learn/scikit-learn, sphinx-doc/sphinx, sympy/sympy
Ruby (9): Homebrew/brew, basecamp/kamal, discourse/discourse, faker-ruby/faker, fluent/fluentd, forem/forem, hashicorp/vagrant, mastodon/mastodon, spree/spree
Rust (8): astral-sh/ruff, denoland/deno, meilisearch/meilisearch, nushell/nushell, pola-rs/polars, rustdesk/rustdesk, sharkdp/bat, sharkdp/fd
Swift (4): MrKai77/Loop, p0deje/Maccy, pointfreeco/swift-composable-architecture, swiftlang/swift-package-manager
TypeScript (9): TryGhost/Ghost, angular/angular, microsoft/vscode, n8n-io/n8n, nestjs/nest, spacedriveapp/spacedrive, supabase/supabase, usebruno/bruno, zed-industries/zed
Other (4): LizardByte/Sunshine, matomo-org/matomo, ml-explore/mlx, prometheus/prometheus

---

Guy Vago is an AI Researcher and Software Architect at Qodo, which builds AI-powered code quality tools for engineering teams.