

2025 EVALUATION GUIDE

AI code review tools

We built the industry's first controlled evaluation framework to compare leading AI code review tools with real-world code, injected bugs & an objective scoring model.

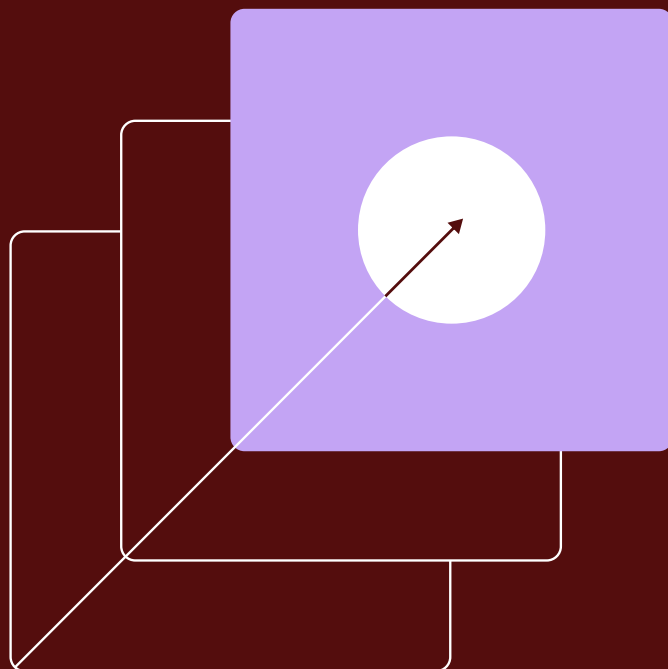
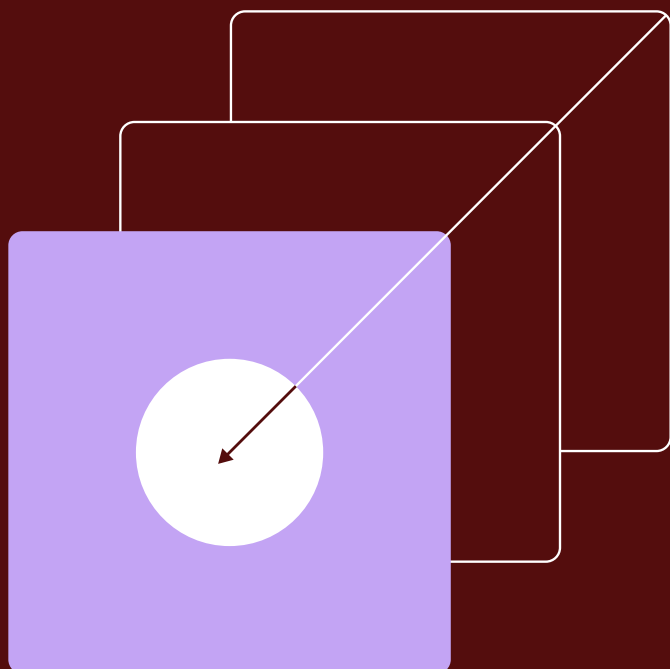


Table of contents

Introduction

3

Aggregate comparison
and tool fit guide

19

Phase 1

8

Don't just take our word for it

21

Phase 2

11

Ready to get started?

23

The dawn of AI code reviews

The AI code review market has experienced explosive growth, with 49% of all engineering teams incorporating some form of AI into their review processes.* This validates a fundamental truth: even as AI transforms code generation, the pull request (PR) process remains central to software development. Code reviews serve as the critical “goalkeeper” function, catching bugs before they reach production. And beyond delivery, code reviews even stand as a system of record for engineering work done and decisions made.

However, the rapid adoption of AI-powered code review tools across the software industry has also outpaced our collective ability to objectively evaluate their effectiveness. While marketing claims and anecdotal user experiences offer fragmented insights, there remains a significant gap in standardized, repeatable methodologies for measuring how well these tools actually perform in real-world development environments.

To bring clarity to the space, we ran a head-to-head benchmark of five leading AI code review tools – **CodeRabbit, LinearB, GitHub Copilot, Qodo Merge, and Graphite Diamond** – using real-world code, seeded bugs, and structured scoring. Our goal was not only to assess their accuracy in detecting and correcting bugs, but also to understand how they behave in the full lifecycle of PR workflows, including noise level, configuration flexibility, and interaction quality.

Our benchmark was built using the following criteria:



Constrained environment

Each tool was tested on the same set of seeded bugs across identical versions of a shared open-source codebase. This eliminated environmental differences and ensured consistency in test inputs.



Multi-factor evaluation

We assessed each tool on both technical performance (i.e., bug detection and fix quality) and experiential factors (clarity, Developer Experience, and configurability). These dimensions were scored independently to provide a multidimensional assessment of value.



Repeatable framework

All code changes, injected bugs, review artifacts, and evaluation scripts were documented and preserved in a version-controlled repository. Beyond just inter-tool parity in testing, this allows other teams to replicate, extend, or customize the benchmark for their own evaluation needs.



Phased methodology

We implemented a two-phase testing approach. Phase 1 established a baseline using simple, well-scoped bugs across JavaScript and Python. Phase 2 introduced more complex scenarios, including cross-service interactions and follow-up commits, to assess how tools perform under more realistic and dynamic development conditions.

This evaluation framework helps engineering teams make informed, data-driven decisions when evaluating AI code review tools. Rather than relying solely on demos or feature lists, this benchmark provides a grounded way to measure performance in the contexts that matter most: accuracy, responsiveness, usability, and integration into existing engineering workflows.

Whether you're still studying the offerings in the market or you're already strategizing on how to improve your team's use of AI code reviews, this benchmark is designed to surface the unobvious trade-offs.

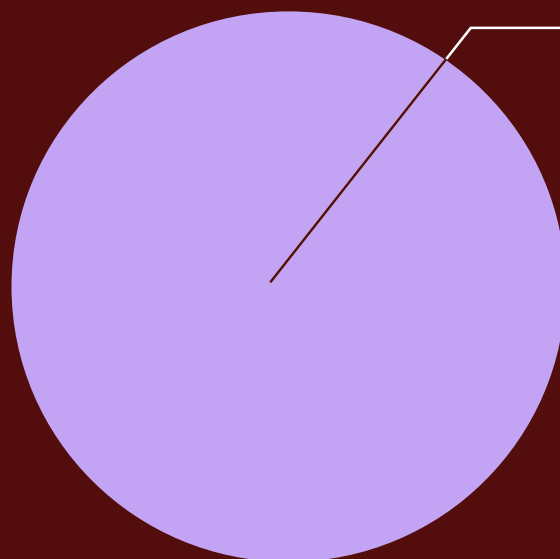
Ultimately, what we found surprised us. Some tools overwhelmed developers with noise. Others silently missed key bugs. A few went the extra mile – resolving comments or suggesting fixes after a follow-up commit. This guide captures those insights and gives you everything you need to run your own comparison.

* Our [2025 AI Data Report](#) found that while 51% of code reviews are still handled solely by humans, 49% now involve AI – either through mixed collaboration (34%) or fully autonomous review (15%).

AI code review breakdown

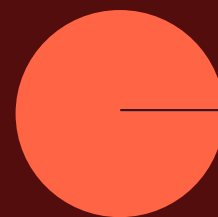
49%

Reviews involving AI



34%

Mixed collaboration



15%

Fully autonomous review

Scoring system

To provide a consistent and objective comparison of the 5 AI code review tools, we developed a structured scoring framework on a scale of 0 to 2 that evaluates each tool across multiple dimensions. Tools were assessed both at the individual PR level (for technical competence) and holistically (for clarity, configurability, and overall DevEx). This rubric ensures that each tool is judged not only on its ability to catch bugs, but also by how effectively it integrates into real-world engineering workflows.

Measurement	Range	Scoring description
PER PR		
Competence	0 - 2	0 – Failed to catch the bug 1 – Caught the bug but did not propose a fix or give adequate context 2 – Caught the bug, proposed a fix, and explained why for learning context
PER TOOL		
Clarity	0 - 2	0 – Extremely noisy, responding where not intended, or responding in excess 1 – Moderately noisy or distracting with extra information 2 – Provides only what is needed in a response
Configurability	0 - 2	0 – Unconfigurable, or lacking scoping or control where the tool is applied 1 – Low configurations, lacking the orchestration needed for sophisticated control 2 – Highly scoped and configurable to fit an enterprise-level infrastructure
UX/DevEx	0 - 2	0 – Friction during setup or adoption, and/or end developer user flow is non-obvious 1 – Has an intuitive user flow 2 – Seamless adoption and DevEx

Tooling overview

 STRONG

 LIMITED

 WEAK

Evaluation criteria	LinearB gitStream	CodeRabbit	GitHub Copilot	Qodo Merge	Graphite Diamond
Git providers	GitHub, GitHub Server, GitLab, GitLab Server, Bitbucket Cloud	GitHub, GitHub Server, GitLab, GitLab Server, Bitbucket Cloud, Azure DevOps	GitHub	GitHub, GitLab, Bitbucket, Azure, Gitea	GitHub Cloud and Server (no OPA)
Runner	GitHub Actions, GitLab Pipeline, Bitbucket Runner	Service OR GitHub Actions	Service	GitHub Action or self-hosted Lambda	Service
Model	Sonnet 4	GPT-4 GPT-3.5 Turbo	Codex-1	All	No public details
Project?	No	GitHub, GitLab, Jira, Linear	No public details	Ticket context (GitHub, Jira, Linear)	No public details
Customization?	Yes, repo files, Trigger, Regex, filtering rules, & PR context-specific configurations.	Config files in project root, AST Grep rules	Limited, in GitHub repo web settings	Yes with settings files in repos, auto approve	No

Tooling overview

 STRONG

 LIMITED

 WEAK

Evaluation criteria	LinearB gitStream	CodeRabbit	GitHub Copilot	Qodo Merge	Graphite Diamond
Style guide?	Yes, repo for org level files	Yes, in WebUI from PR comments	Yes, in repo settings, or config files	Yes, in config files	Yes, in UI
Inline code suggestions?	Yes	Yes	Yes	Partial, with checkbox and reaction	Yes
Admin UI?	Yes	Yes	Yes	No	Yes
Orchestration	Trigger, Regex and filtering rules	AST Grep rules	No public details	Auto approve	No public details
Pricing	Free	\$12-\$24/month per user	\$39 per user/month for an Enterprise account	\$30/month per user with 5000 messages limit	\$20 USD per active committer per month

Phase 1

Phase 1 established a baseline using simple, well-scoped bugs in Python and JavaScript to measure basic bug detection capabilities across all 5 tools.



Testing methodology

In Phase 1, all testing for the AI code review benchmark was conducted in a single repository ([linearzig/benchmark-ai-code-review](https://github.com/linearzig/benchmark-ai-code-review)) with multiple branches: a base JavaScript branch and a base Python branch. For each base branch, we tested 4 bugs, for a total of 8. Each of these 8 bugs were tested for the 5 tools, producing 40 test cases of bug branches attempting to merge into its base branch. All bugs were isolated examples with minimal complexity. No clues were included in commits or branches about their relative bugs.

Eight types of bugs (defined below) were defined in structured folders, then manually copied, pushed to a branch, and then used in a PR request where each tool was activated to run a review.

Bugs 1-8

Bug ID	Language	Category	Description	Source/Justification
1	Python	Logic	Off-by-one error in a loop boundary (e.g. using ' <code><=</code> ' instead of ' <code><</code> ')	Common logic error seen in real-world PRs; flagged by Google and Facebook engineering blogs as top bug class; often missed in human code review due to visual similarity
2	Python	API Misuse	Incorrect use of a mutable default argument (e.g., list as default param)	Widely cited in Python best practices (e.g., Fluent Python, Python docs); causes state bleed across calls; hard for AI to detect without understanding call context
3	Python	Maintainability	Nested conditional logic with unclear return paths	Contributes to cognitive complexity; flagged in tools like SonarQube; relevant to platform teams working on observability and testability
4	Python	Security	Unsanitized user input passed to system call (e.g., <code>os.system</code>)	CWE-78 / OWASP Top 10; requires semantic understanding of dataflow from input to sink
5	JavaScript	Security	Unescaped user input injected into HTML without sanitization	OWASP JS DOM-based XSS; common in frontend PRs, especially React if <code>`dangerouslySetInnerHTML`</code> is used
6	JavaScript	Logic	Incorrectly scoped closure inside loop (e.g., <code>var</code> used instead of <code>let</code>)	Classic JS bug from ES5 days; still occurs and breaks async logic; appears in MDN/StackOverflow/JS audits
7	JavaScript	Performance	Inefficient object cloning using <code>JSON.parse/stringify</code> on non-JSON-safe data	JS anti-pattern; flagged in performance audits (e.g. Chrome DevTools, Lighthouse); AI tools may not detect performance traps
8	JavaScript	API Misuse	Forgetting to return a value from an array <code>map()</code> call	Common logic mistake; flagged in dozens of production bug reports; silent failure when working with arrays

Phase 1 established a critical baseline: all five AI code review tools successfully detected every bug in the initial test suite.

Results

- ✓ **CodeRabbit**
Bugs 1, 2, 3, 4, 5, 6, 7, 8
- ✓ **LinearB**
Bugs 1, 2, 3, 4, 5, 6, 7, 8
- ✓ **GitHub Copilot**
Bugs 1, 2, 3, 4, 5, 6, 7, 8
- ✓ **Qodo Merge**
Bugs 1, 2, 3, 4, 5, 6, 7, 8
- ✓ **Graphite Diamond**
Bugs 1, 2, 3, 4, 5, 6, 7, 8

These results demonstrate that the market has reached a has reached a fundamental competency threshold for catching common developer mistakes like off-by-one errors, mutable default arguments, and basic XSS vulnerabilities. However, this universal success revealed that bug detection capability alone was insufficient for differentiation – the real differences emerged in how tools communicated their findings and integrated into developer workflows.

CodeRabbit

Provided the most comprehensive analysis with detailed explanations and fix suggestions, but its by-default verbosity created review friction through excessive collapsed sections, ASCII art, and overwhelming comment volumes that required significant parsing effort.

LinearB

Distinguished itself by providing thorough bug detection with clean, focused reviews enhanced by YAML-based configuration and slash commands that gave teams precise control over review triggers without the noise plaguing other tools.

GitHub Copilot

Leveraged its native integration advantage, offering the unique ability to assign reviews directly from draft PRs, though its feedback remained notably shallow in some areas, with limited educational value for developers.

qodo

Delivered clean, well-formatted reviews with actionable suggestions and unique local CLI capabilities.

Diamond

Proved the most disruptive with minimal configuration options and behavior like automatically reviewing all PRs, including already-open ones.

While consolidating all tools into one project was convenient for setup, it introduced significant noise at the PR level. With multiple tools active on the same PRs, their outputs overlapped unpredictably, making it harder to isolate and study each tool's behavior. Tools created extra responses in places where they were not being actively evaluated. That said, because all tool events triggered at roughly the same time, they didn't appear to directly influence each other. From these learnings, a new round of methodology was proposed for the next phase.

Phase 2

Phase 2 introduced more complex scenarios, including cross-service interactions and follow-up commits, to assess how tools perform under more realistic & dynamic development conditions.



Testing methodology

As we previously covered, the original benchmarking process in Phase 1 required a tedious, repetitive workflow: manually creating branches with distinct names, copying in bugged files, and pushing changes to GitHub. It worked, but it wasn't scalable. It was also noisy.

In Phase 2, we re-envisioned that process to be more automated. The benchmarking workflow was rebuilt in Cursor to use bash scripts, git commands, Markdown documents holding prompts, and proposed bugs staged in labeled folders. This effectively turned the Cursor IDE into an NLP testing workbench. It's worth adding that this testing workbench can be operated with any agent, not just Cursor. That's because the local scripts and context from git provide substantial context and determinism to eliminate testing errors. The new process sped up testing dramatically, reduced human errors, and made results reproducible for evaluative purposes.

Our streamlined new process allowed us to split the testing into 5 separate repositories to reduce cross-tool noise without adding unmanageable complexity. It also let us study more complex behaviors to match developer expectations, like submitting follow-up commits. We've open-sourced this process and the [supporting assets here](#). Inside, you'll find the bash scripts, bug planning docs, verification checklists, and more.

In sum, our new process gave rise to the following 5 improvements:

1

Introducing **run.sh**: One command, full workflow

The core of this phase is a bash script – **run.sh** – that abstracts away the manual steps. You can now trigger the full benchmark flow with a single command, like:

```
../benchmarking/run.sh --bug 7 --tool graphite
```

Authored in Cursor, the script is well documented and has CLI help, allowing any new conversation to quickly study and learn how to use it. It takes care of deterministic actions like:

- Pushing the changes
- Activating the AI code review tool
- Collecting the response for scoring
- Inserting a specific bug into a clean repo

This could have been built as an MCP server, but by keeping it a bash script the workflow remained equally executable by both AI and humans, with minimal friction. This not only saves time but also reduces human error and makes results more reproducible.

2

Supporting artifacts for bug creation

To guide repeatable, real-world bug creation, the repo now includes a detailed methodology directory full of context documents:

BUG_VERIFICATION.md

Checklist for validating inserted bugs

BUGS.md

Complete list of bug types and their intended difficulty

BUG_PLAN.md

Mapping of which bugs test which tool behaviors

DEBUGGING.md

Tips for troubleshooting tool performance during testing

BUG_CREATION_GUIDE.md

Step-by-step instructions for introducing bugs

This documentation allows anyone to ideate, insert, & verify new bugs consistently, and test them across all tools (more on this later).

3

Isolated repo setup for fair testing

To ensure clean comparisons, we forked an abandoned and archived open source project – **BioDrop** – into five separate repositories, each pre-configured with only one AI code review tool:

- [BioDrop - Qodo](#)
- [BioDrop - GitHub](#)
- [BioDrop - LinearB](#)
- [BioDrop - Graphite Diamond](#)
- [BioDrop - CodeRabbit](#)

Each repo starts from the same baseline, ensuring tool behavior is isolated and controlled.

4

Testing more intricate, inter-connected bugs

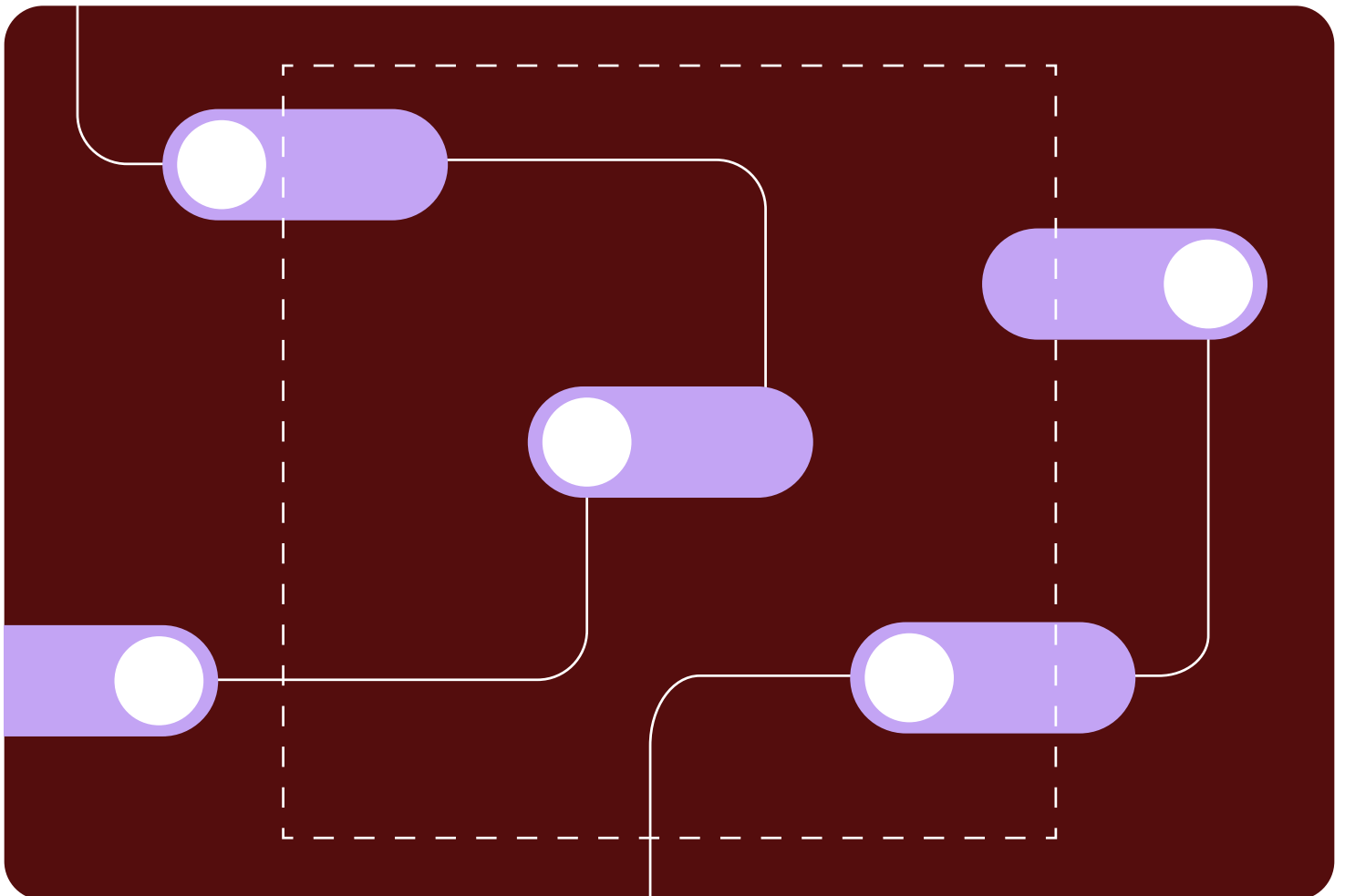
Phase 2 also introduced more complex bugs, designed to push tools beyond simple syntax mistakes and test their ability to reason across systems. These bugs reflect the kinds of issues real engineers face, and reveal which tools can go beyond linting-level suggestions. You'll find a more detailed breakdown of all Phase 2 bugs in the following section.

5

A more flexible, extensible framework

This new setup allows for easy expansion. Tools can be added or removed from the benchmark without disrupting others. New projects can be swapped into test tools across different tech stacks. Bug diffs are preserved in a standalone directory for easy review and validation.

Within Phase 2, benchmarking AI code review tools moved from a series of manual tasks to a scalable, transparent, and extensible framework – one that more precisely mirrors how modern engineering orgs operate. It also bridges the methodology of the first phase into something more automatable, for future benchmarks on the tooling.



Bugs 9-16

Bug ID	Language	Category	Description	Source/Justification
9	JavaScript	Concurrency	Race condition in account-profile association with non-atomic check-then-save operations	Critical concurrency bug in production systems; MongoDB race conditions documented in official docs; requires understanding of atomic operations and distributed systems
10	JavaScript	Security	Fullstack privilege escalation via client-trusted state and session propagation	OWASP Top 10 A01:2021 Broken Access Control; requires understanding of auth flows across frontend/backend. Represents fullstack bugs crossing client/server boundaries.
11	JavaScript	Security	Cross-service data injection via deprecated API parameters and user input injection	CWE-78 OS Command Injection; deprecated API usage creates security holes; requires understanding of API evolution and cross-service data flow
12	Python	Security	Inline JavaScript/HTML injection in Jinja2 templates with unsafe rendering	OWASP XSS; template injection vulnerabilities; requires understanding of template engines and cross-language security. Designed to test embedded JS/HTML inside Python.
13	JavaScript	Concurrency	Microservice race condition with eventual consistency and fire-and-forget notifications	Distributed systems concurrency; eventual consistency problems; requires understanding of microservice architecture and distributed transactions. Also reflects miscommunication between microservices, as flagged in the technical plan.
14	JavaScript	Security	Outdated library functions with weak security parameters (MD5 weak encryption)	CWE-327 Use of a Broken or Risky Cryptographic Algorithm; legacy compatibility creates security risks; requires understanding of cryptographic best practices. Intentionally includes deprecated library usage and outdated function signatures to surface legacy tech debt.
15	JavaScript	Logic	Function calls with wrong argument types causing security bypasses and weak processing	Type safety violations across module boundaries; security bypass through fallback mechanisms; requires understanding of function contracts and type systems. This bug uses functions defined outside the PR scope to simulate boundary-crossing bugs developers frequently miss.
16	JavaScript	Performance	Memory leaks in event-driven architecture with uncleared intervals and growing data structures	Resource management in event-driven systems; memory leaks in production Node.js apps; requires understanding of event loops and garbage collection

Results

Competence

- 0 – Failed to catch the bug
- 1 – Caught the bug but did not propose a fix or give adequate context
- 2 – Caught the bug, proposed a fix, and explained why for learning context

CodeRabbit

0	Bug #9
2	Bug #10
2	Bug #11
2	Bug #12
2	Bug #13
2	Bug #14
2	Bug #15
2	Bug #16

LinearB

0	Bug #9
2	Bug #10
2	Bug #11
2	Bug #12
1	Bug #13
2	Bug #14
1	Bug #15
2	Bug #16

GitHub Copilot

0	Bug #9
1	Bug #10
2	Bug #11
1	Bug #12
2	Bug #13
0	Bug #14
1	Bug #15
2	Bug #16

qodo

0	Bug #9
2	Bug #10
2	Bug #11
2	Bug #12
1	Bug #13
1	Bug #14
1	Bug #15
2	Bug #16

Diamond

0	Bug #9
1	Bug #10
0	Bug #11
2	Bug #12
0	Bug #13
0	Bug #14
1	Bug #15
2	Bug #16

Below you'll find a breakdown of the most important findings from Phase 2, organized by tool.

CodeRabbit

CodeRabbit delivered the most detailed and comprehensive reviews across the board. In any case where it received a top score (2), it consistently surfaced more bugs, better explanations, and multiple code suggestions. However, this level of detail came with a tradeoff:

- Reviews were often overwhelming, with many collapsed sections and dense blocks of commentary.
- In several cases, developers reported that the experience felt like a treasure hunt, where identifying the most critical issues required digging through a large volume of notes.
- While powerful, CodeRabbit introduced high friction into the review process, making it harder to quickly assess and act on feedback.

LinearB

LinearB emerged as a top performer, going head-to-head with CodeRabbit in terms of bug detection and suggested fixes, but doing so with far less noise.

- Its commentary was clear, relevant, and concise, making reviews easier to navigate.
- Developers could use slash commands and `.cm` rules to quickly re-run reviews, adding to its flexibility.
- Its best-performing case was Bug 10, where it provided highly actionable and precise feedback.

One of LinearB's standout features was its ability to resolve its own outdated comments. For example, in Bug 9, when the reviewer committed a fix, LinearB automatically dismissed its earlier review comment, something no other tool in the benchmark did.

This level of contextual awareness not only enhanced review quality but also contributed to a smoother and more intelligent DevEx. It cleared up clutter in the PR and allowed the developer to quickly assess what to tackle next. Re-evaluating the PR after each commit kept the review fresh and up to date.

GitHub Copilot

Copilot was the most convenient tool to use. It was the only one that allowed direct assignment from the PR interface, making it feel native to GitHub. However, its review feedback was minimal and surface-level in some cases:

- Suggestions were often sparse, lacking the depth or specificity needed to help a developer improve.
- In some cases, Copilot failed to offer clear fixes or explanations, delivering just enough information to flag an issue, but not enough to confidently resolve it.
- For beginner developers or quick PRs, it might suffice. But for deeper code quality checks, Copilot came up short.



Qodo Merge stood out for its clean formatting and ease-of-use. It frequently suggested direct code changes that could be applied immediately, making it efficient for developers who want quick, actionable reviews. While its commentary wasn't the most in-depth, its recommendations were clear and low-friction, ideal for straightforward bug scenarios.

Diamond

Graphite was the most inconsistent and temperamental of the tools tested. In some scenarios, it failed to detect bugs entirely, offering no suggested changes.

- One key observation is that Graphite prefers to offload the review process to its external dashboard, which operates as a sort of “reviewed review queue.”
- This approach often clashed with the standard GitHub PR workflow, creating a disjointed experience where developers had to jump between platforms.
- Ultimately, Graphite felt underpowered and disconnected from the typical developer workflow.



A closer look at Bug 9

In real-world development, PRs don't live in a vacuum. Developers often push follow-up commits, either to fix issues raised during code review or to refine work-in-progress code. That's why Bug 9 was designed to test a critical, often-overlooked dimension of AI code reviewers: how they handle iterative feedback cycles. Do tools repeat the same feedback on subsequent commits? If a developer doesn't fix a flagged issue, will the tool surface it again? If a developer does fix it, can the tool recognize and resolve its own past comments?

This wasn't just a test of bug detection. It was a test of how tools adapt when the code changes.

What we did

For Bug 9, each AI code review tool was given a PR containing a seeded bug. After observing their initial reviews, we pushed a follow-up commit that either fully addressed the flagged issue, or partially addressed it, to simulate an incomplete fix.

The goal was to see if the AI would:

- Re-run the review process
- Update or refine its feedback
- Dismiss or resolve its own previous comments

What we found

LinearB stood out as the only tool that automatically marked its own inline comment as resolved when the bug was fixed. It not only provided helpful suggestions during the initial review but demonstrated a clear understanding of state change – acknowledging when a developer had addressed its feedback. In one case, it even followed up with a “Looks good to me” after the resolution.

Qodo Merge offered a partial solution by striking through resolved comments. While not as thorough as LinearB, it still indicated change recognition, which helped reduce clutter in the review thread.

CodeRabbit, despite its high competency in detecting bugs, did not resolve its own comments, creating ambiguity in whether the feedback was still valid after the follow-up. Its verbose style (often including prompts and collapsed sections) made this particularly difficult to parse.

Graphite struggled the most. In some cases, it failed to leave follow-up feedback entirely, or did not recognize that the PR had changed. This created a disjointed review experience and left developers guessing whether the original issue had truly been addressed.

GitHub Copilot, though easy to assign and invoke, similarly did not adapt to the follow-up commit, leaving its prior comments untouched and making no acknowledgment of the new code.

Why this matters

In modern development, iterative PR workflows are the norm. Tools that treat reviews as one-and-done events can create confusion, bloat, and friction – especially on fast-moving teams. Bug 9 showed that stateful behavior is a major differentiator.




LinearB's ability to intelligently track resolution and close the loop isn't just a UX improvement; it's a signal that the tool understands context over time. For teams looking to integrate AI into real-world SDLC practices, that makes all the difference.

Aggregate comparison & tool fit guide






After evaluating each tool across both phases of the benchmark – covering 16 seeded bugs and multiple dimensions of DevEx – we identified meaningful patterns in how these tools behave under real-world PR workflows. Below is a holistic comparison to help your team choose the right tool based on your unique priorities:

Aggregate tool scorecard

STRONG LIMITED WEAK

	Bug detection (Phase 1+2)	Clarity	Configurability	DevEx	Final Score (out of 22)
 LinearB	13/16	1/2	2/2	1/2	17/22
 CodeRabbit	15/16	0/2	1/2	1/2	17/22
 qodo	13/16	1/2	2/2	1/2	16/22
 GitHub Copilot	11/16	1/2	0/2	2/2	14/22
 Diamond	10/16	1/2	0/2	1/2	12/22

Tool strengths & trade-offs

	Strengths	Trade-offs	Best fit for
 LinearB	Highly configurable, low-noise, context-aware, resolves comments post-fix, YAML control	No IDE-level integration or conversational feedback	<ul style="list-style-type: none">• Enterprise teams needing precision, control, and seamless team governance.• Choose LinearB if you want a highly customizable tool that respects your developers' time, offers rule-based control, and adapts to follow-up commits.
 CodeRabbit	Deep bug detection, rich explanations, high educational value	Verbose reviews, difficult to parse at scale	<ul style="list-style-type: none">• Teams prioritizing learning and thorough AI feedback.• Choose CodeRabbit if you value rich educational content and detailed bug context – even if it comes with extra noise.
 GitHub Copilot	Fastest to deploy, convenient native GitHub integration	Surface-level feedback, limited clarity on fixes	<ul style="list-style-type: none">• Teams seeking lightweight AI assistance for low-risk PRs.• Copilot is best for teams already inside GitHub workflows who want a light-touch, fast review without needing deep configuration.
 qodo	Clean formatting, decent local CLI support, strikethrough logic on comment resolution	Steep learning curve, limited customization	<ul style="list-style-type: none">• Teams already using Copilot who need basic PR augmentation.• Choose Qodo Merge for CLI-focused teams who want helpful formatting and a solid baseline of review support.
 Diamond	Always-on PR automation, decent default formatting	No config or external dashboard, misses context across commits	<ul style="list-style-type: none">• Small teams experimenting with AI assistance across high PR volume.• Be cautious with Graphite if your team requires nuanced control, contextual understanding, or seamless integration into iterative PR flows.

Don't just take our word for it

Our AI Code Review Benchmark is designed to be fully reproducible, enabling engineering teams to validate these findings and extend the evaluation to their specific use cases. All benchmark code, test cases, and automation scripts are available in the open-source repository (linked below), with detailed documentation for replicating the complete testing methodology.

The benchmark also includes an automated bash script that can deploy any combination of bugs across multiple AI tools, making it straightforward to test new tools or create additional vulnerability scenarios. Whether you want to verify our results, benchmark additional tools, or adapt the methodology for your organization's specific technology stack, below you'll find everything you need to run comprehensive AI code review evaluations in your own environment.

Here's how to get started:

1 Use our open benchmarking repo

We've packaged everything into a public GitHub repo: github.com/linearzig/benchmark-ai-code-review. Inside, you'll find:

- Scripts and docs for staging and validating bugs
- Evaluation rubrics, scoring templates, and example PRs
- A collection of seeded bugs for both JavaScript and Python

2 Choose your tools

You can test any combination of tools – simply fork a clean project into separate repos, each

with one tool enabled (as we did with [BioDrop](#)). This isolation ensures a clean, unbiased evaluation.

3

Generate bugs with the agentic workflow

Use our `run.sh` script to generate and insert bugs with a single command:

```
./run.sh --bug 7 --tool graphite
```

This workflow allows for on-demand bug generation via natural language prompts, staging of bugs into isolated branches or repos, plus automated PR creation and review triggering

4

Score each tool

We recommend that you evaluate tool performance at two levels:

- Per PR: Did the tool detect the bug? Did it suggest a fix?
- Per tool: How does it handle clarity, configurability, and Developer Experience?

A full scoring rubric is included in the repo along with a worksheet to track your results.

5

Test follow-up commits

In advanced tests (like Bug 9), push follow-up commits to see how tools respond. Do they resolve old comments? Re-review updated code? Or do they repeat old feedback? LinearB stood out here – it was the only tool to mark its own comment as resolved after a bug was fixed.

6

Improve it

The power of this framework is its adaptability. Here are two high-leverage ways to take it further:



Add an LLM judge to automate scoring

Manual scoring is insightful, but time-consuming and subjective. By integrating an LLM-based judge into your workflow, you can automate evaluation at scale.

You can start by prompting an LLM with the provided rubric and sample reviews, or go further and build a scoring pipeline that feeds reviews directly into an LLM-based evaluator. Some teams have even fine-tuned lightweight models to emulate expert review heuristics.



Automate it in your pipelines

Once you've benchmarked once, it's easy to turn it into a regression suite that runs continuously:

- **CI/CD integration:** Drop the benchmark into a staging pipeline. Every time your AI tool updates, rerun key bugs and evaluate changes in performance.
- **Tool drift monitoring:** Just like unit tests catch regressions in code, your benchmark suite can detect when a tool starts missing previously detected bugs or introduces noise.
- **Pre-adoption testing:** Integrate this process into tool evaluations when selecting a vendor. It's a powerful way to validate marketing claims with reproducible tests.

Think of this benchmark as your own test harness. With LLM scoring and CI automation, you'll move from one-time assessment to a living, evolving system for AI DevEx evaluation.

Ready to get started?

AI code review tools have officially moved from the hype phase to a necessity. Our benchmark revealed that all five tools tested are now capable of detecting common bugs with a reasonable degree of accuracy. That alone signals a maturing market: one where AI is no longer just a novelty in the code review process, but a practical asset in day-to-day engineering work.

As the AI code review market continues to evolve rapidly, this benchmark provides a foundation for ongoing evaluation and informed decision-making that will become increasingly valuable as new tools emerge and existing solutions mature. The methodology established here is just the beginning. Future iterations will expand language coverage, incorporate more sophisticated vulnerability patterns, and track tool performance improvements over time.

Now that you've seen our assessment, try testing the tools out for yourself:

- [LinearB](#)
Precise, low-noise reviews with config control
- [CodeRabbit](#)
Highly detailed reviews with strong detection and explanation
- [GitHub Copilot](#)
Quick setup, native GitHub integration

- [Qodo Merge](#)
Clean, developer-friendly CLI-powered reviews
- [Graphite Diamond](#)
Opinionated PR automation, external dashboard view

The tools above will keep evolving – models will improve, interfaces will change, and new capabilities will emerge. Looking ahead, we expect major progress in three areas over the next 12 to 18 months. First, tools will become more “stateful,” meaning they’ll develop the ability to track PR evolution across commits and dismiss outdated feedback. Second, configuration will become more powerful and accessible: allowing teams to tailor reviews not just by file or repo, but by intent and impact. Finally, we’ll see these tools expand to cover more programming languages, deeper vulnerability patterns, and broader integration into the SDLC.

With a consistent, extensible evaluation strategy in place, your organization can evolve alongside them. This framework gives you more than a snapshot; it’s a foundation for continuous assessment, informed adoption, and long-term alignment between AI tooling and your engineering standards. In a space defined by rapid innovation, the ability to measure what matters has never been so important.



LinearB is an engineering productivity platform that helps enterprises define exactly how code is brought to production and maximize the efficiency of their engineering organizations. With full visibility and control over your team's operations, you can finally improve your developer efficiency, effectiveness, and experience.

[Book a demo](#)