Scaling Static Code Analysis Adoption at WhatsApp iOS

Ákos Hajdu
Meta
London, United Kingdom
akoshajdu@meta.com

Jorge Mendez
Meta
Menlo Park, USA
jlmendez@meta.com

Sander van Valkenburg
Meta
Menlo Park, USA
sandervv@meta.com

Artem Kupriianets
Meta
Vancouver, Canada
artemius@meta.com

Matteo Marescotti
Meta
London, United Kingdom
mmatteo@meta.com

Dulma Churchill
Meta
London, United Kingdom
dulmarod@meta.com

Sopot Cela
Meta
London, United Kingdom
scela@meta.com

Abstract

We present an industrial case study on deploying static code analysis to foster developer adoption, which indirectly supports the reliability of the WhatsApp iOS app, with a particular focus on memory management issues. Over the course of a year, we utilized Infer and Clang Static Analyzer to proactively prevent over 1700 regressions. In parallel, we mobilized 50+ engineers and harnessed generative AI to remediate over 1000 long-standing, pre-existing issues within the code base. We share insights on both the technical and people challenges faced, emphasizing the socio-technical strategies required for large-scale static analysis adoption.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Keywords: Static analysis, Infer, Clang Static Analyzer

ACM Reference Format:
Ákos Hajdu, Jorge Mendez, Sander van Valkenburg, Artem Kupriianets, Matteo Marescotti, Dulma Churchill, and Sopot Cela. 2026. Scaling Static Code Analysis Adoption at WhatsApp iOS. In *Proceedings of the 15th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3814987.3814990

1 Introduction

WhatsApp is an instant messaging application serving more than 3 billion people globally. It provides fast and reliable communication across diverse devices and network environments, necessitating a strong emphasis on app health and code quality.

In this paper we report on our experience with deploying two static analysis tools: Infer and Clang Static Analyzer (as part of clang-tidy). In particular, we focus on memory management problems, such as resource leaks, unchecked null pointers and inconsistent nullability annotations. We deployed these tools to run on every code change to prevent regressions. They worked reasonably well using configurations from existing deployments: Infer prevented 709 regressions with an 86% fix rate in a year, and clang-tidy prevented 1005 regressions with a 42% fix rate in 9 months. With improvements to filtering pre-existing issues, we raised clang-tidy's most recent 30-day fix rate to 70%.

Next, we turned our attention to the backlog of pre-existing issues, which is typically more challenging than preventing regressions. While much research focuses on analyzer technicalities [4], human factors pose an equally large challenge at an industrial scale. Engineers are less comfortable touching older code lacking full context, and even small bug fixes require thorough verification. Furthermore, there is a lack of incentive: fixing individual warnings rarely impacts topline business metrics. Because correlating individual fixes with high-level reliability metrics like crash rates is incredibly difficult, we primarily utilize fix rates and developer engagement as proxies for true positives and actionability. Finally, cleanup efforts must scale to industrial code bases with millions of lines and thousands of issues.

To tackle the aforementioned challenges, we applied several methods, including collaboration with power users, being more present around our tools' signal, improving user experience of issue/fix tracking, gamifying cleanup with leaderboards, organizing team events and leveraging generative AI. We successfully mobilized over 50 engineers to fix 664 and 144 pre-existing issues for Infer and clang-tidy, respectively. We have also learned valuable lessons on how to foster effective collaboration between developers and AI, committing 90 (Infer) plus 121 (clang-tidy) code changes, each fixing one or more pre-existing issues.

SOAP '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2709-2/2026/06
https://doi.org/10.1145/3814987.3814990

2 Background

The main goal of this work is to foster static analysis adoption to improve the health and reliability of the WhatsApp iOS app by finding memory management problems. We mainly focus on two static analyzers: Infer and Clang Static Analyzer, but we also briefly touch on clang-tidy. Swift is also an emerging language at WhatsApp, but its analysis is out of scope for this paper.

2.1 Infer

Infer [5, 6] is a static analysis framework that supports various language frontends (including Objective-C; Swift is in progress) and analyzer (checker) backends via a common intermediate language called SIL [3].

In our current work, we focus on six checks related to memory management problems such as null pointers and resource leaks: (1) block parameter not null checked, (2) captured strong self, (3) mixed self-weakself, (4) nil insertion into collection, (5) self in block passed to init, and (6) strong self not checked. We picked these based on their high fix rates at other Infer deployments.

As an example, consider the function below, which has a block parameter (`completion`) getting called without a null check, leading to potential crashes.

```
- (void)doTask:(void (^)(void))completion {
    // Do the task ...
    completion();
}
```

More details on these checks can be found in Infer's documentation.[1]

2.2 Clang Static Analyzer

Clang Static Analyzer[2] is built on top of Clang and LLVM, aiming to find bugs in C, C++ and Objective-C programs. In our current work we mainly focus on the five categories of *inconsistent nullability annotations*:[3] dereferencing, passing, or returning null(able) when non-null is expected. We specifically targeted these five categories because they are a leading cause of production crashes and actively slow down the adoption of Swift within the WhatsApp codebase.

For example, the function below claims to return non-null, but violates this when the length of the input string is 0.

```
- (nonnull NSString *)firstChar:(NSString *)str {
    if (str.length == 0) return nil;
    return [str substringToIndex:1];
}
```

Objective-C does not enforce nullability consistency; calling a method on a null receiver is typically a safe no-op returning a default value. However, when interoperating with Swift code, the Swift compiler treats these annotations as the source of truth and generates non-optional types with forced unwrapping, causing a run time crash if they hold a null value.

We deployed Clang Static Analyzer via clang-tidy,[4] including these five nullability checks alongside over a hundred other linters based on prior experience.

2.3 Deployments

Static analyzers have typically two kinds of deployments: *diff-time* and *continuous*. Diff-time deployments run in the continuous integration (CI) pipeline for every code change (called *diff*) to prevent regressions. Analyzers report newly introduced[5] issues as inline comments during the code review process (along with signal from other automated tools and human feedback). Developers can then submit a new version of the diff, triggering a new analyzer run. We measure *fix rate* by checking what percentage of issues (present in any intermediate version) disappear in the final, committed version. While fix rate is an approximation of the true positive rate, it works exceptionally well in practice because it is automatically measurable and strongly correlates with developer actionability.

Continuous deployments scan the latest revision of the whole code base periodically. Results give an overview of the current state/health of the repository and longer-term trends, but are less visible: engineers need to go to dedicated dashboards or databases that they don't check on a daily basis.

3 Analyzing Code Changes

We deployed both Infer and clang-tidy (including Clang Static Analyzer checks) to run on every code change (diff) and observed fix rates over a period of 12 and 9 months, respectively.

3.1 Infer

Table 1 shows the volume of Infer reported issues and their fix rates over a period of a year. Infer worked well out-of-the-box:[6] 709 issues were fixed out of 821 reported, indicating an 86% overall fix rate. We attribute this success to a variety of reasons. First, these checkers are relatively simple (e.g., often intra-procedural) and targeted for specific bad patterns in Objective-C code. Second, the checkers have been extensively tested and improved at other deployments on a large scale over multiple years. Third, Infer is mainly developed internally at Meta, which means we got dedicated expert support and could easily make modifications to the checkers, e.g., to address false positives. Finally, Infer includes elaborate heuristics to avoid reporting pre-existing issues: it

[1] https://fbinfer.com/docs/all-issue-types
[2] https://clang-analyzer.llvm.org/
[3] https://clang.llvm.org/docs/analyzer/checkers.html#nullability
[4] https://clang.llvm.org/extra/clang-tidy/
[5] We try to avoid reporting pre-existing issues with various heuristics, see details later.
[6] The authors would like to thank Martin Trojer (affiliated with Meta) for his help with the infrastructure of Infer deployment.

Table 1. Diff-time results for Infer.

Issue kind	Reported	Fixed	Fix rate
Block p. not null checked	146	125	86%
Captured strong self	326	254	78%
Mixed self-weakself	223	219	98%
Nil insert. into collection	24	15	63%
Self in block passed to init	65	59	91%
Strong self not checked	37	37	100%
Total	821	709	86%

analyzes both the version before and after the code change, and computes a difference of issues in a way that is agnostic to line shifts or class renamings.

3.2 Clang-tidy

Table 2 shows the volume of clang-tidy reported issues and their fix rates over a period of 9 months. Since there are 100+ checks/lints enabled, we only listed the ones that had at least 20 reported warnings individually. The rest of them are merged into a single "other" entry in Table 2. The fix rates for clang-tidy are lower, 1005 issues fixed out of 2419 reported (42% overall fix rate). We observed two main reasons.

First, Clang Analyzer is shallow and conservative about side effects, as demonstrated by the following example.

```
if (obj.prop) { expectsNonNull(obj.prop); }
```

Because getters are functions, Clang Analyzer conservatively assumes subsequent calls might return different values, triggering NullablePassedToNonnull warnings even when checking an object's property for null prior to use. Mitigating this requires defensive programming (e.g., local helper variables) and consistent developer alignment.

Second, up until recently, clang-tidy used a very simple heuristic for filtering pre-existing issues: it analyzed only the new version of the code and skipped issues that were reported on non-changed lines. This easily caused pre-existing issues to be reported when lines moved around, or had unrelated changes. We looked at the top 20 diffs with most unfixed issues (616 in total) and all of them were moving code around (i.e., refactoring efforts such as migrating code to another class or breaking up large functions into smaller pieces), causing pre-existing issues to be reported and to remain unfixed. In particular, NullablePassedToNonnull accounted for nearly half of all reported issues (1178 out of 2419, see Table 2). Given these considerations, we have decided to disable this checker.

Recently, clang-tidy was enhanced to analyze code before and after the change, matching issues via source control.[7] After turning NullablePassedToNonnull off and having improved filtering for pre-existing issues, we have seen the 30-day rolling fix rate of clang-tidy to rise above 70%.

[7]This work was done by Ezgi Çiçek, affiliated with Meta.

At the same time, certain bugs escaped detection that could have been identified by the NullablePassedToNonnull checker. Responding to developer demand, we decided to re-enable this checker, now that pre-existing issues were no longer being flagged. Within a month of safely re-enabling NullablePassedToNonnull, it identified 262 issues with 173 resolved (66% fix rate), highlighting the importance of continuously adjusting configurations as requirements evolve.

4 Pre-Existing Issue Cleanup

Fixing pre-existing issues in a code base is challenging [7, 13, 17]. As outlined in Section 1, these challenges are often rooted in human factors such as context switching and misaligned incentives rather than purely technical limitations. Nevertheless, we managed to mobilize 50+ engineers and leverage generative AI to fix over 1000 pre-existing issues in various ways that we present in this section.

4.1 Human fixes

Power users. Engaging early-stage testers is crucial for fine-tuning analyzers prior to broader rollout. Highly committed engineers are often willing to test prototypes and navigate internal databases; one such power user single-handedly fixed over 60 pre-existing issues from an early-stage Infer deployment. Since engineers are sensitive to false positives, avoiding a poor initial experience is vital to retain trust.

User experience matters. Except for power users, most engineers are unlikely to consult internal databases containing pre-existing issues. To address this, we enhanced the user experience by presenting these issues through a more engaging dashboard featuring a redesigned GUI. The dashboard enables intuitive filtering by category and file path, plus one-click bulk task creation. This approach empowers developers to quickly identify issues relevant to their own modules, and to efficiently delegate, track, and attribute fixes. Before the introduction of this enhanced GUI, only 1-2 power users actively fixed issues; afterward, we observed 40-50 engineers engaging with the system. Although the underlying information remains unchanged from the internal database, improved accessibility and presentation significantly impact usability and engagement.

Being present. Engineers at large organizations are often overwhelmed with signals from a variety of automated tools. This abundance can lead to important alerts being perceived as noise and subsequently ignored. Our observations indicate that maintaining a visible presence around these signals – effectively "putting a face" behind them – significantly improves their credibility. When engineers recognize that an individual is actively developing and maintaining the analyzer, rather than it being a "deployed-and-forgotten" system, engagement and attention increase. This effect is evident not only for pre-existing issues but also for code

Table 2. Diff-time results for Clang Static Analyzer and clang-tidy.

Issue kind	Reported	Fixed	Fix rate
clang-analyzer-nullability.NullablePassedToNonnull	1178	351	30%
clang-analyzer-nullability.NullableReturnedFromNonnull	209	96	46%
clang-analyzer-nullability.NullableDereferenced	186	62	33%
clang-analyzer-nullability.NullPassedToNonnull	118	61	52%
clang-analyzer-nullability.NullReturnedFromNonnull	62	38	61%
clang-analyzer-deadcode.DeadStores	101	58	57%
clang-analyzer-osx.cocoa.RetainCount	94	38	40%
clang-diagnostic-incompatible-pointer-types	37	37	100%
clang-diagnostic-unused-variable	29	29	100%
clang-analyzer-osx.cocoa.NSError	22	6	27%
Others (with <20 individual reported)	383	229	60%
Total	2419	1005	42%

changes (diffs). Concrete examples of this approach include proactively reviewing issues, conducting preliminary assessments and categorization, following up by commenting on tasks and diffs, establishing dedicated feedback channels, and actively responding to feedback.

In addition, we also observed that code speaks louder than words: the most effective way to solicit developers' opinions on a signal (such as determining whether an issue is a true or false positive) is to attempt a fix ourselves and submit it for review. Generative AI can boost this process: while it is typically challenging for analysis tool developers to devise fixes from scratch, leveraging AI-generated solutions for initial assessment and sanity checking lowers the barrier to entry. However, as discussed later, it is crucial to avoid overwhelming engineers with low-quality or insufficiently vetted AI-generated fixes.

Gamification. Fixing pre-existing warnings can be tedious. Counting them on a company-wide leaderboard, however, adds competition and fun. We bulk-filed 500 tasks for Infer issue categories with historically high fix rates and advertised them internally to grab and fix. We created a leaderboard to track completed tasks, strictly requiring a fully reviewed and landed code change to count. Within a month, 50+ engineers fixed over 400 tasks.

Team-up events. We have also organized so-called "team-up" events where engineers got together (both in-person and remotely) and dedicated a day to clear the backlog of issues. The advertisement and community building nature of these events – primarily driven through active internal Workplace[8] posts – helped to bring visibility to the pre-existing issues and to involve more engineers.

Results. Table 3 lists fixes for pre-existing issues reported by Infer. Note that in addition to diff-time checkers (Table 1), it includes *multiple weakself*. This checker did not have a high fix rate in other deployments so we did not enable it on diffs, but surfaced the pre-existing issues nevertheless.

[8]Workplace is Meta's internal collaboration platform.

Table 3. Infer pre-existing issues fixed.

Issue kind	Fixed	Remaining
Block param. not null checked	264	12
Captured strong self	13	240
Mixed self-weakself	222	5
Multiple weakself	27	188
Nil insertion into collection	49	8
Self in block passed to init	2	67
Strong self not checked	87	2
Total	664	522

Table 4. Clang Static Analyzer pre-existing issues fixed.

Issue kind	Fixed	Remaining
NullableReturnedFromNonnull	88	572
NullReturnedFromNonnull	29	8
NullPassedToNonnull	27	15
Total	144	595

There is a noticeable split between the issue kinds: some of them have almost all instances fixed, while others have the majority unfixed. Upon inspection, we found a simple reason: issues with most instances fixed were part of bigger initiatives (e.g., bulk filed tasks with leaderboard, or team-up events), whereas other fixes typically came from individuals proactively looking at the self-service dashboards.

Table 4 presents fixes for pre-existing issues reported by clang-tidy. We observed the same pattern as for Infer: NullReturnedFromNonnull and NullPassedToNonnull were involved in team-up events and the rest came from self-service efforts. Our takeaway is that individuals do look at pre-existing issues and contribute fixes, but large-scale cleanup comes from coordinated efforts.

4.2 Generative AI fixes

We have utilized generative AI tools to address several pre-existing issues within the code base. It is important to note that some of the human-driven fixes described in the previous section may have involved AI assistance (e.g., an engineer picking a task and using an AI coding assistant to draft a solution). However, those efforts were initiated and guided by human engineers. In this section, we focus on *AI-driven* fixes. Here, the static analysis team set up automated jobs (called *codemods*) that generated diffs for other engineers. The iOS experts' main role was to approve, reject, or ask for follow-ups on these AI proposed code changes.

Infer. In our first attempt [15] we targeted unresolved Infer issues that remained after manual remediation (Table 3). In the prompt, we included the issue's location, trace, description, and the corresponding Infer documentation. Altogether, 175 code changes were proposed, with 90 being accepted (51% acceptance rate), fixing 115 issues in total (one code change could fix multiple issues). It is crucial to note that these 90 fixes were reviewed and accepted by domain-expert iOS engineers, not by the static analysis team.

Initially, these code changes were assigned for review in an "unsolicited" manner by identifying module owners and recent contributors. In conventional workflows, reviewers expect the author has thoroughly validated a code change. With AI, proposing modifications outside one's expertise became easier, misaligning expectations. As tool developers, we expected code owners to rigorously evaluate our AI suggestions. Conversely, reviewers assumed AI-generated changes were already vetted to the standard of manually authored code, consistent with incentives rewarding authorship over review. Additionally, confusion arose from repurposing a subsystem historically used for deterministic, large-scale modifications (like code reformatting).

The above factors contributed to a few instances of less comprehensive reviews, and a few changes inadvertently introduced new bugs rather than resolving existing ones. Based on these experiences, we refined our process by (1) clearly communicating the AI-generated nature of proposed changes and emphasizing the need for thorough review, (2) placing code changes in a queue from which proactively contacted owners could select items for review, (3) and publishing AI-generated fixes only for issues with the highest observed fix rates, thereby minimizing the risk of hallucinated "fixes" for false positives.

Clang Static Analyzer nullability. A couple of months later, we made another attempt, this time focusing on Clang Static Analyzer inconsistent nullability warnings. Meanwhile, a dedicated AI refactoring platform was built, making non-determinism obvious to reviewers and providing follow-up tools (e.g., blocklisting AI on specific failed changes).

This time we started slower: just publishing one change per day, giving us time to follow along and do an initial assessment – often before the code owner took a look. We observed that proactive communication, and having a "face" behind an AI diff helped with engagement. In addition, we focused on the two categories of NullableReturnedFromNonnull and NullPassedToNonnull, which are almost always true positives.

So far, in roughly 3 months, 121 changes have been committed (each fixing one or more warnings), and 45 rejected. Many of the rejected ones were actually duplicates, because at the same time there was a "better engineering" event where engineers fixed some of these warnings in parallel. The overall feedback is highly positive, and we are now more confident in the quality of the AI-generated suggestions. While this feedback process was heavily based on unstructured developer comments, direct conversations, and positive reactions to diffs and tasks (rather than a systematic user study), it accurately reflected our lived industrial experience. A way to scale up would be to partner with teams to generate fixes for the modules they own, with upfront alignment to better decentralize the reviewing effort and avoid the bottleneck of us having to do an initial assessment.

Discussion and open questions. We learned that generative AI fixes need a good baseline true positive rate to avoid hallucinating "fixes" that instead introduce new bugs. In addition, reviews assigned in an unsolicited way can lead to less thorough review due to misaligned expectations and incentives. It is important to either align with code owners proactively, or do a sanity check before assigning the review. We can also see that the bottleneck shifted to the review phase (instead of coding the fix) and reducing it is currently an open question. A potential solution could be to use automated end-to-end verification (including UI), raising confidence in the safety of the code change, or – where applicable – use generative AI to implement a deterministic fix (e.g., based on AST transformations). Furthermore, our experience revealed that human-AI interaction is more mature when human is driving it (e.g., delegate task to coding assistant), but we need to improve processes when AI is the driver (e.g., cleaning up pre-existing issues).

5 Related Work

Infer had success at other deployments at WhatsApp. It has been analyzing privacy properties of server Erlang code [11, 14], and performance of Android code [10].

There are multiple studies [7, 13, 17] on deploying static analyzers, reaching the same conclusion that getting pre-existing issues fixed is challenging. Our paper presents some concrete approaches that helped us in solving these challenges at scale.

While previous studies have highlighted the challenges of fixing pre-existing static analysis issues, recent work has

increasingly focused on human-centered and AI-assisted remediation. Washizaki et al. [1] demonstrated that gamifying static analysis by awarding points for bug resolution significantly increases developer engagement and warning remediation. Nguyen Quang Do [16] similarly argued that integrating game design elements into static analyzers is critical for preventing tool abandonment and overcoming notoriously poor developer UX. Our industrial deployment validates these academic findings at scale, showing that dashboards and leaderboards are essential for mobilizing engineers.

Furthermore, the integration of Generative AI into static analysis is an emerging area of interest. Academic tools like CodeCureAgent [12] and Sorald [9] have proposed automated, LLM-driven, and template-based approaches to classify and repair static analysis warnings. However, as our experience at WhatsApp shows, deploying such AI-driven codemods in an industrial setting introduces unique socio-technical challenges – such as misaligned review incentives and the risk of hallucinated fixes – that require careful, human-in-the-loop mitigation strategies.

Finally, Autili et al. [2] survey 261 static analysis studies for mobile apps, noting that iOS is heavily understudied compared to Android (with rare exceptions like PiOS [8]). While our socio-technical findings on adoption generalize beyond any single operating system, this industrial case study provides valuable empirical insights into the underrepresented iOS ecosystem.

6 Conclusions

We reported on scaling static code analysis adoption to improve reliability of the WhatsApp iOS app, detailing our approach to both technical and socio-technical challenges. In roughly a year, Infer and Clang Static Analyzer prevented over 1700 regressions, while 50+ engineers and generative AI remediated over 1000 pre-existing issues.

References

[1] Satoshi Arai, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2014. A Gamified Tool for Motivating Developers to Remove Warnings of Bug Pattern Tools. In *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 37–42. doi:10.1109/IWESEP.2014.17

[2] Marco Autili, Ivano Malavolta, Alexander Perucci, Gian Luca Scoccia, and Roberto Verdecchia. 2021. Software engineering techniques for statically analyzing mobile apps: research trends, characteristics, and potential for industrial adoption. *Journal of Internet Services and Applications* 12, 3 (2021). doi:10.1186/s13174-021-00134-x

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 115–137. doi:10.1007/11804192_6

[4] D. Beyer and J. Strejček. 2025. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *Proc. TACAS (3) (LNCS 15698)*. Springer, 151–186. doi:10.1007/978-3-031-90660-2_9

[5] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. doi:10.1007/978-3-642-20398-5_33

[6] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 3–11. doi:10.1007/978-3-319-17524-9_1

[7] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. doi:10.1145/3338112

[8] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting privacy leaks in iOS applications.. In *NDSS Symposium*.

[9] Khashayar Etemadi, Nicolas Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. 2022. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2022), 3733–3745. doi:10.1109/TDSC.2022.3167316

[10] Ákos Hajdu, Roman Lee, Gavin Weng, Nilesh Agrawal, and Jérémy Dubreil. 2025. Compositional Static Callgraph Reachability Analysis for WhatsApp Android App Health. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. ACM, 15–21. doi:10.1145/3735544.3735583

[11] Ákos Hajdu, Matteo Marescotti, Thibault Suzanne, Ke Mao, Radu Grigore, Per Gustafsson, and Dino Distefano. 2022. InfERL: Scalable and Extensible Erlang Static Analysis. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*. ACM, 33–39. doi:10.1145/3546186.3549929

[12] Pascal Joos, Islem Bouzenia, and Michael Pradel. 2025. CodeCureAgent: Automatic Classification and Repair of Static Analysis Warnings. *arXiv preprint arXiv:2509.11787* (2025). https://arxiv.org/abs/2509.11787

[13] Junjie Li and Jinqiu Yang. 2024. Tracking the Evolution of Static Code Warnings: The State-of-the-Art and a Better Approach. *IEEE Trans. Softw. Eng.* 50, 3 (2024), 534–550. doi:10.1109/TSE.2024.3358283

[14] Ke Mao, Cons T Åhs, Sopot Cela, Dino Distefano, Nick Gardner, Radu Grigore, Per Gustafsson, Ákos Hajdu, Timotej Kapus, Matteo Marescotti, Gabriela Cunha Sampaio, and Thibault Suzanne. 2024. PrivacyCAT: Privacy-Aware Code Analysis at Scale. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 106–117. doi:10.1145/3639477.3639742

[15] Ke Mao, Timotej Kapus, Cons T Åhs, Matteo Marescotti, Daniel Ip, Ákos Hajdu, Sopot Cela, and Aparup Banerjee. 2026. WhatsCode: Large-Scale GenAI Deployment for Developer Efficiency at WhatsApp. In *Proceedings of the 48th International Conference on Software Engineering: Software Engineering in Practice*. ACM.

[16] Lisa Nguyen Quang Do and Eric Bodden. 2018. Gamifying Static Analysis. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 904–907. doi:10.1145/3236024.3264830

[17] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. doi:10.1145/3188720

Received 2026-02-27; accepted 2026-04-15