Enhancing Aerospace Software Quality with Automated Code Review

Jeremy Ludwig
Stottler Henke Associates, Inc.
San Mateo, CA 94402

Abstract—**High-quality software is indispensable for mission-critical systems like those developed for NASA and the DoD. Reliable, maintainable code with minimal technical debt is fundamental to achieving these goals. Automated code review has become a cornerstone of the software quality pipeline, with SonarQube a leading tool in the field. This paper shares our practical experience integrating SonarQube into two long-standing aerospace software development projects, previously documented in IEEE Aerospace publications. Our primary objective is to disseminate best practices and lessons learned from our use of automated code review to assist others in enhancing their software quality pipelines.**

TABLE OF CONTENTS

1. INTRODUCTION

High-quality software is paramount for mission-critical systems, such as those developed for NASA and the DoD, and for software product lines demanding long-term sustainability and reusability. Reliable, maintainable code with minimal technical debt is fundamental to achieving these goals. Software development teams typically balance code quality with project constraints by employing a variety of design techniques, processes, and tools.

Automated code review has emerged as a cornerstone of the software quality pipeline, with SonarQube a leading tool in the field. This paper shares our practical experience integrating SonarQube into two long-standing, deployed, aerospace software development projects previously documented in IEEE Aerospace publications. In the first project, SonarQube is seamlessly integrated into the DevOps pipeline, analyzing every code change (known as a pull request in git parlance). In the second, it is employed as a periodic gatekeeper, identifying potential security vulnerabilities and critical defects outside the primary development workflow. These distinct approaches yield interesting and instructive outcomes over time.

Our primary objective is to disseminate best practices and lessons learned from our use of automated code review to assist others in enhancing their software quality pipelines.

In the remainder of this paper, we first provide background information on the concept of technical debt and the SonarQube static code analysis tool. Following this, we discuss related work applying SonarQube to software development. We then present two use cases: SonarQube as a continuous integration tool and as a periodic gatekeeper. We conclude with best practices and lessons learned.

2. BACKGROUND

This section provides a brief overview of software quality, technical debt, automated code review, and the SonarQube tool.

Software quality models articulate what is meant by 'software quality.' These models define the desired characteristics and sub-characteristics of software and the relationship between these characteristics and measurable properties of the software. The ISO-IEC 25010: 2011 [1] quality model defines eight desired characteristics of software product quality: Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, and Portability. While all these characteristics are important, this paper focuses specifically on Reliability, Maintainability, and Security.

Software quality models based on static source code analysis generally follow a three-step pattern. They identify specific source code metrics to be calculated, describe how the measurements of these metrics are aggregated, and define how the aggregations are used to assess characteristics of software quality (e.g., Reliability) [2]. The SQALE model [3] assesses software quality by identifying and quantifying potential issues line-by-line, and is the basis for automated code review tools such as SonarQube.

979-8-3503-5597-0/25/$31.00 ©2025 IEEE

Technical debt is a measure of how much work would be needed to move from the current code to higher-quality code [4]. The source of technical debt during development and sustainment stems primarily from making design, implementation, documentation, and testing decisions that focus on short-term value [5]. As technical debt increases, changes to the software become more difficult, error-prone, and time-consuming, and this threatens the reliability, maintainability, and security software characteristics.

This is an especially important take-away for software product lines, where long-lived, reusable modules are intended to be shared by multiple systems. Each module will want to invest in high code quality (low technical debt) initially and maintain this investment in quality over time as it is extended and updated. That is, as part of planned re-usability, each module commits to making a long-term investment to software quality. The likely alternative is that the software quality will gradually degrade until, eventually, the problems become overwhelming [6].

There are several practical tools aimed at improving source code quality, managing technical debt, and improving security such as SonarQube, Codacy, and Fortify. These and similar static code analysis tools use rules to analyze every line of code to identify likely bugs, maintainability issues, and security flaws. SonarQube appears to be one of the most commonly used automated code review tools [7], [8], [9], with their own site listing 400,000 organizations that use SonarQube [10].

SonarQube comes with a set of default rules that span 30+ programming languages. For example, there are 600+ analysis rules for Java. Rules are bundled into Quality Profiles, where project administrators can select the set of rules that should be applied to a project. The default Java quality profile has about 500 rules. Each rule is associated with a Software Quality (security, reliability, and maintainability), a Severity (high, medium, low), and a Type (bug, vulnerability, code smell). An example rule is shown in Figure 1. Each rule includes an explanation about what the specific problem is and how to fix it. As rules are applied to the code, they generate issues where a line (or lines) of code violates a rule. Customizable Quality Gates define minimum acceptable standards, for example code with only few low severity issues or better can pass the gate. The SonarQube dashboard supports viewing trends in issues over time, with a focus on what has happened in 'new code' (this is configurable, e.g., last 45 days or only changes in the current pull request) vs. the existing codebase. The objective of highlighting new code is to write code at the highest standard going forward, which SonarQube calls "Clean As You Code" [11]. Over time, issues will pile up if nothing is done or slowly disappear if addressed.

It is important to note that Stottler Henke has no financial, personal, or business relationship with Sonar or its tool SonarQube. The views, opinions, and/or findings contained in this paper are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of Sonar. While this paper is based on experience with SonarQube, we do not aim to promote any one tool, and the implications of the presented work apply to other commonly used static code analysis tools.

3. Related Work

This section provides a brief overview of a few related studies that examine how SonarQube issues are triaged by software engineers during development. See [8] for an in depth review.

Yu et al. [7] examined which SonarQube identified issues were actually fixed or closed across 30 long-lived and popular Java open source projects. While it would be expected that issues would be triaged by type and severity, what they found was that whether an issue was fixed mainly related to (i) ease of understanding and fixing, (ii) likely harm caused, (iii) the context surrounding the issue, (iv) the specifics of the triggering rule, and (v) the specific developer. Some of the implications they discussed include the need to customize the project rule base to what will actually be fixed, for developers to continue to learn such that they can understand and fix more complex issues quickly, and to integrate static analysis tools with the development pipeline for continuous analysis. The authors also stress the need for tool builders like SonarQube to continually improve their analysis rules to reduce false positives and to identify the actual severity of issues as accurately as possible.

Alfayez et al. [8] investigated similar research questions about how SonarQube issues were prioritized, recruiting research participants, training them on technical debt, and then giving them a technical debt prioritization activity. The activity involved selecting which issues in a list to select given a fixed amount of time to spend, where each item had an estimated time cost. The majority tried to balance the severity of an issue with costs in selecting which to address, though a solid minority uses a severity-only approach. One individual used a cost-only approach. Across this, a group consisting mostly of highly experienced software developers identified some issues as 'must fix' and did not include these issues in their prioritization. That is, irrespective of costs some issues must be fixed and so were considered to be

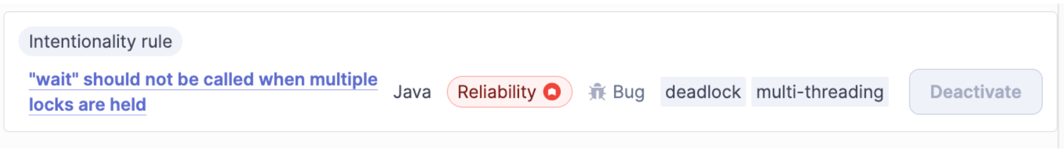

Figure 1. Example SonarQube rule.

outside of prioritization. A primary takeaway from their work is that there is not a one-size-fits-all solution for technical debt prioritization.

Lenarduzzi et al. [12] used SonarQube to analyze 33 Apache open source projects written in Java. They measured fault-proneness of Java classes by analyzing the Git history, and then looked at relationships between classes with more severe bugs, classes with lower severity or no bugs, and fault-proneness. They found that while one bug-finding rule was related to increased fault-proneness, most were not regardless of severity. While there are issues with attempting to determine whether a bug actually caused a fault from reviewing the commit history (or if it might cause a fault in the future), the implications of their work are still worth noting. One implication is the importance of customizing the rule set to the specific project – even though most open-source projects surveyed (98%; 14,732 of 14,957) use the default rule set. A second implication is that it is important for tool builders to continually update their tools based on the actual harmfulness of issues.

We expect that factors influencing technical debt issue identification and resolution in open-source projects and in research studies are likely to differ from those in proprietary development environments. However, the related work reinforces the notion that tools such as SonarQube are a work-in-progress and that better results will be achieved if they are customized to the specific project and development team.

4. SonarQube as a Continuous Integration Tool

Stottler Henke is applying SonarQube to the development of critical software for scheduling and deconflicting satellite communications for the US Air Force [13], [14]. It is important to note that this software, like many projects, started as a rapid prototype to demonstrate proof of concept and then began transitioning into a production-level system. This section provides a high-level view of how SonarQube is used to reduce technical debt over time as part of a software quality pipeline.

A software quality pipeline is a combination of a team's or organization's culture, processes, and tools aimed at producing high quality software – sharing much in common with their DevOps pipeline [15]. Just like there is no single DevOps solution that works in all contexts, there is also no single software quality pipeline that works everywhere for all software. In this use case, the culture includes a focus on developing reliable, maintainable, and secure software as a long-term investment, the processes are those common to lean and agile software development, in this case Jira, Bitbucket, Jenkins, and SonarQube all integrated into an automated DevOps pipeline.

Jira, by Atlassian, is used for issue tracking and serves as the primary interface point between project management and development. Bitbucket, also by Atlasssian, is used as the version control system. Jenkins, an open-source project, is used to compile and test the software, and build software releases, to support continuous delivery. Finally, SonarQube is used for automated code review.

Beginning work on an issue involves starting a new source code branch in Bitbucket, which we will call the feature branch. A software developer(s) then makes any changes in this branch. Once the changes are made, the developer issues a pull request that signals the feature branch is ready to merge the change into the main development branch. This triggers several actions. First, Jenkins compiles the feature branch and runs automated tests on this branch. The developer needs to correct any compilation or failed test issues to proceed. Second, the feature branch is analyzed by SonarQube. Any problems that SonarQube finds in the new code are posted to the pull request in Bitbucket. Software developers are expected to correct all identified problems before proceeding to the next step. Third, the feature branch undergoes a manual code review by another software developer. The reviewer will create tasks in Bitbucket to be addressed. Once the reviewer has verified all the tasks have been corrected, then the feature branch is finally merged with the main development branch.

The combination of automated and peer review has three main benefits. First, the resulting code is more reliable, maintainable, and secure. Occasionally, a bug is found, but more often what is addressed are future maintainability issues. Second, these reviews mentor less experienced software developers. Requiring developers to fix all issues, whether from SonarQube or the team lead, before merging in their code generally encourages them to start doing the right thing the first time. Third, the manual reviews spread knowledge of the code out across the development team. While developed independently, this lightweight review system is very similar to that used by Google in terms of tools, process, and motivation [16]. The downside to this approach is that it is difficult to clearly assign credit for improvement, where both manual an automated code review are likely to be responsible for a reduction in technical debt as measured by SonarQube.

As shown in Figure 3, this approach yields steady improvements. This figure shows decreases in the number of potential code smells, vulnerabilities, and bugs over time. Code smells make up the largest percentage of potential technical debt, followed by vulnerabilities and then bugs. All areas improved, with potential bugs and vulnerabilities reaching near zero.

At the same time, Figure 2 illustrates how percent duplicate code decreases over time, a big win for code quality. Percent of comments relative to lines of code also increases, but this is likely due to peer review (and not SonarQube). Note that in general, these fixes do not require dedicated development time because corrections are made only on code that is being created or updated as part of the normal development process.

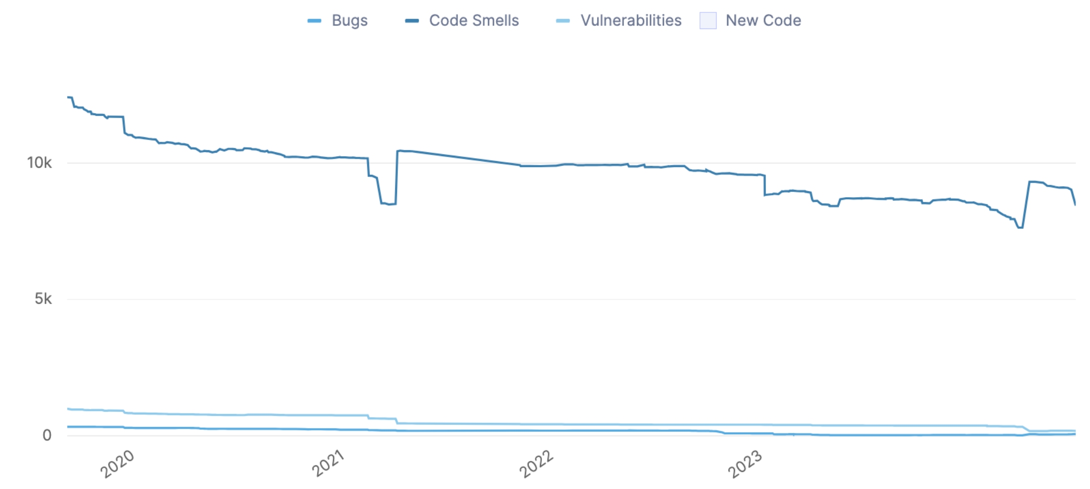

Figure 3. Bugs, code smells, and vulnerabilities as identified by SonarQube.

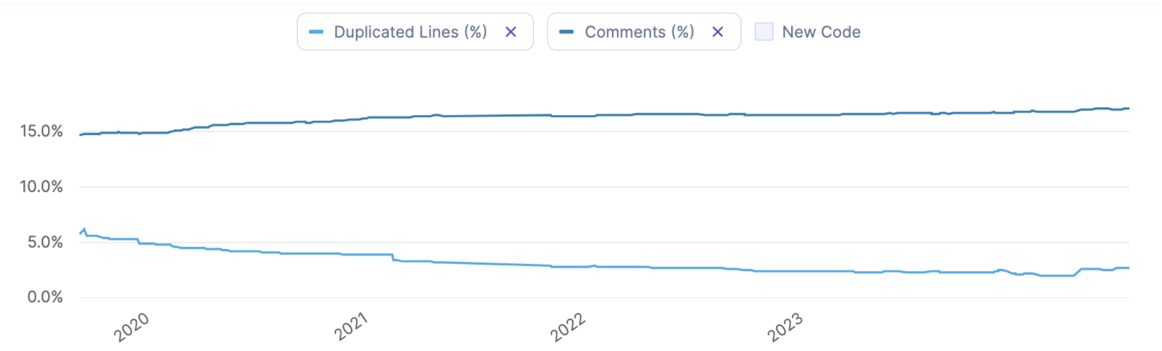

Figure 2. Percent duplicated lines and percent comments relative to code over time.

Slow and steady gains improve the reliability, maintainability, security of the code with essentially zero cost in terms of additional development time – even while the overall lines of code continues to grow.

There are two interesting things to note. First, we say potential issues because there are a significant number of false positives across all categories as well as suggestions that are simply not worthwhile to correct. Second, there are several noticeable spikes in the graph. These are generally caused by changes to the specific set of rules SonarQube is applying, which are often updated with each major and minor release. While the project started with highly customized rules, we reverted to mostly the default rule sets for each language, disabling rules that were found to occur often and always be labelled 'won't fix'. The upgrade spikes are visible across all three issue types.

This approach provides several benefits. First, it enables early detection of potential quality issues, preventing them from being merged into the codebase. Second, it fosters a culture of quality among developers, encouraging them to write clean, maintainable code following best practices the first time around. Additionally, developers get immediate feedback that encourages them to not keeping making the same mistake. Finally, addressing issues as they arose in each pull request helps to reduce the overall cost of software development by preventing bugs and vulnerabilities from being discovered later in the development lifecycle. It is faster and easier for a developer to fix a problem right away than it is to go back and fix the same problem later.

While integrating SonarQube into the DevOps pipeline generates benefits, there are some challenges to overcome. One issue is the need to configure Bitbucket, Jenkins, and SonarQube to correctly analyze each pull request, and then to keep the configuration working in the face of updates to each of these tools. A second issue is that the rules applied to each language do need to be customized for the code base. Specifically, rules that are false positive prone or are always marked as 'won't fix' need to be turned off to maintain developer engagement. As the people most impacted by automated code review, and in the best position to see how it could be done better, developers need to be empowered to suggest improvements such as changes to rules and quality gates. Additionally, SonarQube constantly updates the default rules for each language which requires integration with the existing rule updates. Overall, these DevOps costs are fairly limited, leading to a good cost/benefit ratio in favor of SonarQube for this project.

A third issue we found with SonarQube is that it is important to establish clear guidelines for handling SonarQube violations detected in pull requests. Specifically, educating the development team that every issue flagged by SonarQube should be addressed or senior developers should sign off on a 'won't fix'. Initially, peers approved pull requests with unaddressed SonarQube issues or reviewed and merged pull requests that were never even analyzed (e.g., when a security token expires SonarQube stopped running). Making the best possible use of SonarQube requires training and commitment from the development team.

5. SonarQube as a Periodic Gatekeeper

The second aerospace software development project, a crew trainer and experimentation tool for the MH-60R and MH-60S helicopters [17], required a more flexible approach to code review. Due to the project's DevOps constraints, it is not feasible to integrate SonarQube into the DevOps pipeline for every pull request. Instead, SonarQube is used as a periodic gatekeeper, analyzing the codebase at regular intervals through the Gradle build tool.

The DevOps pipeline for this project involves Jira for issue tracking, a stand-alone Git repository for source version control, and Gradle as a build tool. As in the previous project, the development workflow involves creating a feature branch, writing code, and then generating a request for peer review before merging the new code into the main branch. The main difference between this project and the previous project is that SonarQube is not integrated into the DevOps pipeline and instead requires a developer to manually run a Gradle task to perform the analysis and upload the results to the SonarQube web application. This approach allows the development team to identify potential code quality issues that may have been missed during the primary development workflow.

The primary challenge with this method, not present in the other project, is that the issues found by SonarQube must be prioritized. As previously mentioned, developers greatly prefer to make forward progress on feature requests rather than address potential SonarQube issues from the past. It also takes time to go back and fix issues once developers have committed changes and moved onto something else. Because of this, it isn't reasonable to address all the new issues that have been added to the code.

Prioritization is done using the SonarQube functionality for filtering issues by type (vulnerability, bug, code smell) and by severity (high, medium, low). First, all vulnerabilities are addressed. Second, all potential bugs marked high are addressed. These two filters catch the issues most likely to affect maintainability, reliability, and security. All other issues are ignored. This Java code is an example of the types of things that are ignored:

```
if (!isVisible())
     return;
```

These statements are flagged as a code smell with severity high for not putting the return call into curly braces (or alternately putting all the code on a single line). This code is technically correct, but if a future developer adds more statements before the return, they will need to remember to add curly braces or it will not execute properly.

Our prioritized approach to code review strikes a balance between the clean as you code philosophy and no automated code review. By focusing on the most critical issues, we significantly enhance software reliability, maintainability, and security without incurring excessive costs. While it is ideal to address all potential issues, this strategy offers a practical solution that delivers substantial benefits in the case where SonarQube could not be more closely integrated into the development pipeline.

While this approach successfully improves software quality, there were some drawbacks to using SonarQube as a periodic gatekeeper. First, the long delay between making a code mistake and the correction reduces the developer's ability to learn from their mistake. This is really a lost opportunity for the development team to continue to learn and grow. Second, the manual process of running SonarQube, filtering issues, and then addressing or creating Jira tasks for most critical items is tedious, adding a bit of friction into the DevOps pipeline. In a busy production environment, tasks with friction do not happen as often as you would like.

6. Best Practices and Lessons Learned

Based on our experiences with these two projects and related work, we identify several best practices for integrating SonarQube into software development pipelines:

- **Early adoption:** Integrate SonarQube as early as possible in the development lifecycle to catch issues before they become more difficult and costly to fix.
- **Customization:** Configure SonarQube to meet the specific needs of your project by customizing the Quality Profile (rules) and Quality Gates.
- **Education**: Provide training and support to developers to help them understand how to use analysis results from SonarQube effectively.
- **Integration**: Integrate SonarQube into your existing development tools and processes to streamline the automated code review process. Consider putting a merge check in place that prevents merging new code that has unaddressed SonarQube issues.
- **Prioritization:** Establish a process for prioritizing and addressing SonarQube issues in existing code based on their severity and potential impact on the software's reliability and security.

In addition to these best practices, we have also learned several important lessons from our experiences with SonarQube:

- **Continuous improvement:** It is essential to use SonarQube consistently throughout the development lifecycle to ensure that code quality is maintained and improved over time. It is much more agreeable to the development team to address issues as they go rather than spending time fixing issues from the past.
- **Collaboration:** Effective use of SonarQube requires collaboration across the entire development team. Everyone needs to be on board with the goal of improving the reliability, maintainability, and security of the code base and empowered to help with this process.

By following these best practices and lessons learned, other teams can effectively leverage SonarQube to enhance their software quality pipelines and deliver high-quality software products.

REFERENCES

[1] Organización Internacional de Normalización, ISO-IEC 25010: 2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Geneva: ISO, 2011.

[2] R. Ferenc, P. Hegedűs, and T. Gyimóthy, "Software Product Quality Models," in Evolving Software Systems, T. Mens, A. Serebrenik, and A. Cleve, Eds., Springer Berlin Heidelberg, 2014, pp. 65–100. doi: 10.1007/978-3-642-45398-4_3.

[3] Squale Consortium, "Visualization of Practices and Metrics," Mar. 2010. Accessed: Feb. 07, 2017. [Online]. Available: http://www.squale.org/quality-models-site/research-deliverables/WP1.2_Visualization-of-Practices-and-Metrics_v1.1.pdf

[4] W. Cunningham, "The WyCash portfolio management system," SIGPLAN OOPS Mess, vol. 4, no. 2, pp. 29–30, Dec. 1992, doi: 10.1145/157710.157715.

[5] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, in ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 50–60. doi: 10.1145/2786805.2786848.

[6] E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," IEEE Softw., vol. 29, no. 6, pp. 22–27, Nov. 2012, doi: 10.1109/MS.2012.130.

[7] P. Yu, Y. Wu, J. Peng, J. Zhang, and P. Xie, "Towards Understanding Fixes of SonarQube Static Analysis Violations: A Large-Scale Empirical Study," in 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2023, pp. 569–580. doi: 10.1109/SANER56733.2023.00059.

[8] R. Alfayez, R. Winn, W. Alwehaibi, E. Venson, and B. Boehm, "How SonarQube-identified technical debt is prioritized: An exploratory case study," Inf. Softw. Technol., vol. 156, p. 107147, Apr. 2023, doi: 10.1016/j.infsof.2023.107147.

[9] "The 2019 State of Code Review Report," SmartBear Software. Accessed: Oct. 05, 2020. [Online]. Available: https://smartbear.com/resources/ebooks/the-state-of-code-review-2019/

[10] "Customers & Organizations Using Sonar." Accessed: Sep. 27, 2024. [Online]. Available: https://www.sonarsource.com/company/customers/

[11] "Clean Code: The Essential Approach." Accessed: Sep. 25, 2024. [Online]. Available: https://www.sonarsource.com/solutions/our-unique-approach/

[12] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study," J. Syst. Softw., vol. 170, p. 110750, Dec. 2020, doi: 10.1016/j.jss.2020.110750.

[13] J. Ludwig, D. Cline, and A. Novstrup, "A Case Study Using CBR-Insight to Visualize Source Code Quality," in Proceedings of IEEE Aerospace Conference 2020, Big Sky, MT, Mar. 2020.

[14] R. Stottler and R. Richards, "Managed intelligent deconfliction and scheduling for satellite communication," in 2018 IEEE Aerospace Conference, Mar. 2018, pp. 1–7. doi: 10.1109/AERO.2018.8396654.

[15] A. Wiedemann, N. Forsgren, M. Wiesche, H. Gewald, and H. Krcmar, "Research for practice: the DevOps phenomenon," Commun ACM, vol. 62, pp. 44–49, 2019, doi: 10.1145/3331138.

[16] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern Code Review: A Case Study at Google," in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, in ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 181–190. doi: 10.1145/3183519.3183525.

[17] R. Richards and B. Presnell, "OMIA: MH-60R Helicopter Desktop Crew Trainer & Software Change Experimentation Tool," in 2022 IEEE Aerospace Conference (AERO), IEEE, 2022, pp. 1–7. Accessed: Sep. 26, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9843720/

BIOGRAPHY

Jeremy Ludwig, *Ph.D., is a group manager at Stottler Henke Associates, Inc. Dr. Ludwig leads AI research teams, applying reasoning, knowledge representation, and machine learning to develop innovative solutions for complex challenges.*