What Can Program Repair Learn From Code Review?

Madeline Endres
University of Michigan
Ann Arbor, Michigan, USA
endremad@umich.edu

Pemma Reiter
Arizona State University
Tempe, Arizona, USA
pdreiter@asu.edu

Stephanie Forrest
Arizona State University
Tempe, Arizona, USA
steph@asu.edu

Westley Weimer
University of Michigan
Ann Arbor, Michigan, USA
weimerw@umich.edu

ABSTRACT

Over the past fifteen years, research on automated program repair has matured, and transitions to industry have begun. However, an impediment to wider adoption is concern over automatically generated patch correctness. A review of 250 program repair research papers suggests that this concern can be addressed by adapting practices from modern code review, such as multiple anonymized reviews and checklists with well-defined terminology, to better evaluate the correctness and acceptability of plausible patches. In this paper, we argue that adopting such practices from modern code review for automated program repair research can increase developer trust, paving the way for wider industrial deployments.

CCS CONCEPTS

• **General and reference** → *Evaluation*; • **Software and its engineering** → *Automatic programming*.

KEYWORDS

Automated program repair, code review, patch correctness, human evaluations

ACM Reference Format:
Madeline Endres, Pemma Reiter, Stephanie Forrest, and Westley Weimer. 2022. What Can Program Repair Learn From Code Review? . In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524459.3527352

1 INTRODUCTION

Automated program repair (APR) is a popular research topic in software engineering [1] with a few initial industrial deployments [2, 3]. Despite recent successes, however, wide acceptance of APR is limited by concerns about machine patch correctness. Often generated through random mutation and validated primarily with test suites, automated patches are susceptible to overfitting [4]. Various methods from held out test-cases to specification-based proofs have been proposed to improve the automated evaluation of patch correctness and quality. Ultimately, however, human expertise remains an essential component; proposed patches are typically evaluated either by human inspection or by comparison to an extant developer-supplied patch. Unfortunately, these inspections and comparisons tend to be unstructured and performed by a single annotator in APR research evaluations. This is an error-prone approach which misses the diverse range of human expectations.

We propose the APR community adopts insights from modern code review practices to improve the trustworthiness of automatically generated repairs. In industry, humans use code review to decide if a patch is correct, i.e., to assess the trustworthiness of developer-written code. By defining organizational expectations, specifying terminology and systematic criteria, and gathering multiple viewpoints, modern code review provides a consistent evaluation framework that increases organizational trust in accepted code. While exact procedures vary, modern code review generally involves both automated and formalized manual checks; because humans have multiple expectations, no single practice suffices. Key code review practices include: (1) continuous integration testing, (2) checklists and policies, (3) multiple reviewers, and (4) anonymized review. We observe, however, that APR researchers have focused almost exclusively on (1) when evaluating proposed patches and tools. In this paper, we consider (2)–(4) in turn and, using evidence from our review of 253 APR papers, highlight best practices from code review that we believe should be incorporated into future APR evaluations.

2 BACKGROUND

We now briefly discuss background relevant to our proposal. Specifically, we outline how trust impacts deployment decisions in industrial software engineering, consider the factors that influence perceived trustworthiness of automated software, and finally discuss deployment risks that are specific to APR.

Deployment Decisions—Risk vs. Reward: Industrial deployment of new technology [5] is generally determined by assessments of risk/reward tradeoffs. The primary potential reward of adopting APR is the efficient and timely production of software bug patches, improving software reliability and saving developer debugging time. Like all software tools, however, APR-generated patches are not risk-free. This tradeoff is captured in psychology, where *trust* is

APR'22, May 19, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527352

often defined as a willingness to accept risk by balancing it against confidence in reward. We claim that understanding how developers gain trust in software tools is critical for understanding and encouraging future APR deployments. A common approach to inspiring trust, for example, is designating a "champion" or advocate who is familiar with the tool and can personally explain risks and rewards and how to avoid mistakes [6]. We argue that human assessments of risk (i.e., trust) dominate deployment decisions for APR.

Trustworthiness of APR: Trustworthiness is a measure of confidence that an interaction will result in a positive outcome, while accepting the consequences of failure. Intuitively, developer confidence can be enhanced by properties like software longevity, reuse, and adaptation. More formally, software's trustworthiness is primarily influenced by: *reputation* (software's perceived quality based on external information); *transparency*, (software's perceived understandability); and *performance* (software's perceived ability to meet project requirements) [7]. As a result, developer software judgements depend on varied individual experiences, and are prone to inconsistency and bias. For example, reviewers may have different expectations for human vs. automated repairs; recent studies find that reviewers trust automated patches more when labeled as human generated [8, 9]. This implicit bias against machine-generated patches is a barrier to trust. By adopting practices that are common in code review such as anonymized evaluations, APR's trustworthiness can be enhanced, potentially leading to wider acceptance.

Risks of APR: We now examine a key risk associated with APR that hampers trust, namely, that it may produce a "repair" that fails to meet human expectations for performance or transparency. This concern extends beyond first generation APR, applying equally to recent proposals [4, 10–12], such as those based on constraints or supervised learning. There are certainly other risks associated with APR (e.g., the process could take too long for real-time releases; it could introduce security vulnerabilities; it could be too expensive). Meeting human expectations, however, is essential for establishing trust and accelerating adoption [6, 7, 13]. Critically, human expectations are context-sensitive. For example, student expectations of tools using patches as educational hints are markedly different from developer expectations of professional debugging tools, security analyst expectations of patches, or open-source pull request merge expectations. In particular, students may be primarily interested in patches that they find easily readable and helpful for their own debugging while security analysts might be primarily concerned with the security implications of a potential patch. This multifaceted complexity suggests that automated metrics alone are insufficient for evaluating APR tools.

3 POSITION: CODE REVIEW FOR APR

To improve acceptability and trustworthiness of APR evaluations we propose the APR research community (1) adopts established practices from modern code review, and (2) incorporates multiple types of assessments, such as structured manual evaluation, to meet multiple human expectations.

Position 1—*expectations of APR should be modeled on code review expectations*: We propose that researchers approach the evaluation of a given APR patch through the lens of modern code review. As discussed earlier, modern code review considers both qualitative and functional factors, and it uses multiple structured techniques to reduce bias and inconsistency such as requiring multiple reviewers, anonymizing the patch, and giving reviewers explicit checklists to consider. In addition, many organizations require that reviewers be familiar with the code base and that the review include at least one person involved with developing the modified module.

Modern code review often augments human capabilities with automated static and dynamic analyses such as style checkers, linters, bug finding tools, and integration tests [14]. We observe that the APR research community has focused predominantly on this last approach (automated testing) and largely overlooked the others; while simple manual inspection is common in APR evaluations, systematic and structured evaluation akin to that of modern code review is rare. Integrating these overlooked code review practices into APR evaluation is a step towards more robust patch correctness evaluations which can transfer to an industrial setting.

Position 2—*APR evaluations should diversify to capture multifaceted human expectations and increase developer trust*: Test-suite validation is the most common APR evaluation methodology. Despite recent advances in detecting overfitting, plausible patches that are deemed acceptable by test cases can still overfit their associated test suite necessitating additional review [4, 10, 15]. In addition, humans generally expect non-functional properties such as readability or maintainability. As a result, assessing a patch solely on automated regression tests is insufficient.

These two issues, overfitting and non-functional properties, can both be addressed through the code-review-inspired systematic manual inspection of patch quality and correctness described in Position 1. However, manual inspection alone (even systematic inspection) is insufficient to thoroughly assess a patch. Some patches address defects that are challenging for humans to reason about, such as race conditions or security vulnerabilities. Further, it is well-known that human reviewers make mistakes [16] and are vulnerable to biases concerning a patch's apparent author [9, 17]. Manual patch assessments that do not mitigate these error sources are insufficient on their own. This points to the need for multiple metrics used collectively, including both automatic and human-based assessments.

4 APR AND CODE REVIEW POLICIES

We were interested in the following question: *How do empirical APR studies validate the quality of a proposed patch?* To address this question, we surveyed all of the 253 papers in the Monperrus living APR Bibliography (downloaded, December 2020) that evaluate APR tools or repairs [1]. Each paper in this dataset was read by at least one of the current paper's authors. We identified all papers in the dataset that reported a human assessment of patch correctness or quality (133 papers).

We find that 53% of APR evaluations have some form of human involvement, and these human assessments fall into three general categories: manual patch correctness and/or quality inspection by paper authors, a user study with external reviewers, or developer feedback (e.g., industrial deployment or number of pull requests accepted). Although author-based manual inspection is by far the most common, only 28% reported methodological details such as the number of reviewers or a formalized annotator decision process. Of those that do report such details, even fewer contained high-quality evaluations that resemble today's best code review practices. This lack of rigor increases the likelihood of mistakes, thereby weakening research validity of there empirical studies. Table 1 summarizes the results of our evaluation.[1]

Year	'02-10	'11-13	'14-15	'16	'17	'18	'19	'20	All
Papers	24	29	28	17	21	45	51	38	253
Has Humans	6	11	15	10	12	27	30	22	133
Authors Inspect	4	8	13	10	10	22	22	19	108
Has Methods	0	1	6	3	1	3	7	8	30
User Study	2	4	3	0	2	4	6	4	25
Has Developers	1	1	1	1	1	3	6	3	17

Table 1: Papers with human-based patch correctness or quality evaluations. *Has Humans* is the number of papers with non-automated patch evaluations; *Authors Inspect* is where authors explicitly used manual review for patch correctness or quality; *Has Methods* is papers with manual inspection that explicitly report methodological details such as the number of reviewers per-patch or a systematic decision procedure beyond an unelaborated assertion of checking for semantic equivalence with the developer patch; *User Study* is those that involve non-author evaluation of patches or tools; and *Has Developers* is those with either industrial deployments or open source developer interaction.

We next consider four existing code review practices used in human software engineering, which could strengthen APR empirical evaluations and improve the validity of results: (1) continuous integration testing, (2) checklists, (3) multiple reviewers, and (4) blinding.

Briefly, continuous integration enables scaling of software programs by automating software builds and testing when integrating code changes, thus assuring continued code correctness and a clean environment for code review [18]. Code inspection checklists (i.e. checklists) formalize functional and non-functional software requirements to enable code reviewers to perform consistent and effective evaluations [19, 20]. Because code review quality is associated with an individual reviewer's personal metrics, like workload and experience, completing a code review usually requires feedback from multiple reviewers [21]; similarly, anonymizing the code review process by double-blinding participants addresses personal biases that can impact code review quality and consistency [22].

We now discuss these four code review practices in more detail, highlighting their relevance to APR in the context of our literature review. Because earlier APR research focuses predominantly on the first of these four existing code review practices (continuous integration), in this paper we recommend that the community consider the other three human-driven practices to create a more systematic evaluation process.

Practice 1—*Continuous Integration and Test Suite Overfitting*: Modern code review generally requires that code pass automated tests before being integrated into a project's main code base, a process often done using continuous integration. Continuous integration is one way of enabling the requirement of automated testing at scale by automating software builds and testing [18]. Continuous integration testing refers to the automated testing component of continuous integration.

Automated software tests often feature multiple suites of increasing cost and quality: developer-local unit tests, continuous integration testing (and other static analyses), and more expensive test suites (e.g., full integration/regression tests run overnight or each weekend). APR patches would also likely be required to pass all automated test levels before final integration. Consequently, APR papers commonly use test suites to evaluate potential patches. However, both heuristic and semantics-based APR approaches are prone to overfitting to test suites [4, 10]; potential patches that overfit to tests can negatively affect non-tested functionality.

Researchers sometimes address overfitting by using held-out tests to validate repairs after a patch has been proposed [4] or by using heuristics such as test-case execution similarity [12], to guide repair search. More recent work has also proposed systematic metrics and methods for mitigating biases in automated evaluation of APR patches [15]. Although promising, these automated efforts alone do not yet determine patch correctness and quality with sufficient probability to meet developer expectations [16]; in current APR deployments such as SapFix at Facebook, APR patches are integrated into code review processes with manual inspection [2]. We argue that the APR research community should follow this example by integrating formal manual code review processes into evaluations of patch correctness.

Practice 2—*Code Review Checklists and Nebulous 'Repair Quality' Definitions*: Including manual review of APR patches can help better evaluate how candidate patches meet expectations. Human expectations, however, are diverse. For example, a programmer concerned with program efficiency might rate a patch differently from a security analyst. As a result, organizations using modern code review generally train developers with checklists of functional and non-functional properties to specifically look for when reviewing code [23]. At a high level, an extensive checklist ensures adherence to (1) accepted coding standards; (2) a defined architecture; (3) a set of accepted Non-Functional requirements, like maintainability, reusability, reliability, security, and performance; (4) Object-Oriented Analysis and Design (OOAD) principles. In practice, checklists often direct the reviewer through sets of questions, like: "Are names of variable, methods, and classes meaningful?"(coding standards), "Is load balancing appropriately used?"(performance), and "Is the least privilege principle enforced?"(security). Using checklists, developer training, and coding standards reduces annotator variance and helps prevent mistakes.

Annotator variance is prevalent in APR evaluations, potentially affecting not only individual annotators' consistency but also study generalizability (e.g., reproducibility or comparisons to related work). The low level of internal annotator consistency is clear; one recent study found that 35/187 (19%) of patches determined "correct" by informal manual inspection of the authors of three APR papers were actually incorrect when subjected to a more robust evaluation [16]. And, our analysis shows that APR papers rarely report (30/108) explicit rubrics for manual patch correctness and

[1]The data from which Table 1 was generated is available at https://docs.google.com/spreadsheets/d/1JwdM8uxEI5BixVL-my0iRBKbqY5go3l5_5bISDLhzj0/edit?usp=sharing

quality decisions beyond an unelaborated definition such as "semantically equivalent to". We suggest that APR researchers both report and adopt more rigorous rubrics and checklists to increase internal consistency and improve patch classification transparency.

Inconsistency across APR evaluations goes beyond individual variations. We also observe variance with terms and definitions complicating comparative review. One such ambiguity is with patch correctness. When manually reviewing, researchers generally define patch correctness either in relation to a historical developer fix or based on annotator understanding of intended program behavior. However, there is variance within these categories. For instance, several papers assess whether a patch is *semantically equivalent* to the developer patch while others look for *functional equivalence*. While similar, semantic and functional equivalence have different connotations which can lead to different annotator assessments. This complicates the task of comparing patch correctness evaluations across studies, negatively impacting reproducibility.

The lack of terminology standardization in APR evaluations is even more prevalent regarding patch quality. For example, we observe that APR user studies measure a wide range of quality properties through participant survey responses including *usefulness*, *helpfulness*, *trustworthiness*, *maintainability*, *pertinence*, and *relevance*. Other studies simply ask participants to rate *quality* on a Likert scale. Measuring diverse properties is desirable: software quality is nuanced, not one-dimensional. An issue arises when researchers try to compare patch quality evaluations between tools and studies. It is rarely possible to fairly compare human study results between papers. For instance, results rating *helpfulness* cannot be directly equated to results rating *usefulness*. Slight word differences can have significant impact on responses. For example, Scalabrino *et al.* found significant differences in user perceptions of code "understandably" and code "readability" [24]. We encourage the APR research community to not only produce more structured manual evaluations, but also to standardize patch correctness and quality terminology (cf. [25]) to better support reproducible research.

Practice 3—*Multiple Reviewers and Annotator Counts*: Beyond checklists and structured reviews, another way that modern code review (and even traditional code inspection) increases confidence in error-prone human annotators is through multiple reviewers, which reduces the likelihood that all reviewers make the same mistake, especially if reviewers include diverse expertise, familiarity with the code base, or are respected senior developers. This effect applies in many domains [26]. Organizations that employ such guidelines would not merge a pull request with only one reviewer.

The importance of avoiding reviewing bias and mitigating reviewer error rates requires that APR patch correctness and quality evaluations be determined by more than one annotator. Unfortunately, such redundancy is not currently standard for manual inspection in APR papers. We observe that very few (14/108) papers with manual review by the paper authors report using multiple annotators. An additional 11 papers collected multiple annotations through a user study for a total of 25/133 papers with human participation. Due to limited methodological detail in many papers, it is possible that more studies used multiple reviewers than those which were explicitly stated in the paper text. However, we note that of the 18 papers who specified the number of annotators, 4 explicitly used a single reviewer, indicating that using multiple reviewers is not universally adopted. This analysis points to the need for the APR research community to adopt more rigorous standards for author-driven patch correctness evaluations using the methods outlined above. In addition, relevant statistics such as inter-annotator agreement should be reported.

Practice 4—*Blinded Code Review and Bias and Trust*: A final element of code review that is relevant to APR is the impact of human biases on patch acceptance. Modern code review acknowledges that biases influence code quality assessments in nuanced ways. For example, Terrell *et al.* found that while women are more likely to have open source pull requests accepted than men, pull requests from women who were project outsiders were less likely to be accepted than those of male outsiders. [17]. Avoiding such biases is required for accurate assessment of risks and rewards.

A few recent APR studies have also identified bias surrounding automated patches, particularly bias connected to labeling a repair as generated by a developer or a machine. For instance, Huang *et al.* found that participants were significantly less likely to accept a patch labeled as machine generated, even if the patch was actually generated by a human developer [9]. This is important because non-functional properties are often more important the functional ones in subjective human tool [6, Sec. 4] and patch or code assessments [7]. Such issues are independent of test suites and overfitting, but because different humans have different biases and different expectations, one practice (such as using multiple annotators) is not enough to mitigate this concern. We argue that the lessons learned about bias in code review can also inform correctness and quality evaluations for automatically produced patches. In author inspection and user studies of APR patches, for example, study instruments could avoid indicating if a given patch was machine or human generated as much as possible.

5 DISCUSSION AND CONCLUSION

While we acknowledge that formalized code review can be expensive, especially in academic settings, we argue that the benefits of more reproducible research, evaluation error mitigation, and increased trust are worth the cost. Additionally, although the time-consuming nature of more rigorous patch evaluations by humans may reduce the feasible size of some empirical studies, we argue that quantity over quality is not helpful in this circumstance and that smaller studies that are more carefully conducted will ultimately prove more useful to the APR community in evaluating the relative merits of different methods and research prototypes.

This is especially true because APR patches tend to fix simple bugs: the adoption cost for researchers will relatively low. We therefore conclude with actionable suggestions for incorporating checklists, multiple reviewers, and anonymized review into future manual APR evaluations:

Incorporating Checklists and Definitions:

- Encourage annotators to adopt a standard rubric and publish the rubric along with the results.
- Liu *et al.* [11, Sec. 3.3] provides an example set of rules for systematically determining similarity of a plausible patch to a human's.

Incorporating Multiple Reviewers:

- Use 2–3 independent reviewers to evaluate each patch. When appropriate, report inter-annotator agreement and/or have a second evaluation pass to resolve annotator disagreements.
- Consider including reviewers with varying levels of expertise, and if possible, including one reviewer who is a code-base expert.

Incorporating Blind Review and Mitigating Bias:

- Include one reviewer unfamiliar with the tool's implementation.
- Anonymize who or what wrote the patch. If possible, do not let participants know they are rating patches from multiple sources.
- Huang *et al.* [9, Sec. 3] give an example of a relevant "deceptive" experimental setup.

Evaluating whether or not APR patches meet human expectations is critical to understanding programmer trust, a key prerequisite for future deployments. We propose applying insights from modern code review such as multiple reviews, checklists, and blinding, to better evaluate if these expectations are met. We further advocate for increased research trust through transparency and reproducibility and call on researchers to open source their source code, patches, and patch correctness results for public review, further mitigating the risk of manual patch assessment — we thank our anonymous reviewer for this important point.

As human expectations for patches are varied and context dependant, a single evaluation approach generally does not suffice. Unfortunately, the bulk of published APR evaluations contain only automated approaches and/or unstructured error-prone manual evaluation. We argue that future APR evaluations employ multiple automated and systematic manual review metrics in tandem for a more robust understanding of patch trust and acceptability.

ACKNOWLEDGMENTS

We acknowledge the partial support of the NSF (CCF 1908633, CCF 1763674) and a Google Faculty Research Award.

REFERENCES

[1] M. Monperrus, "The Living Review on Automated Program Repair," Dec. 2020, working paper. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01956501

[2] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[3] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: How genetic improvement became an overnight success," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, p. 1513–1520.

[4] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[5] T. Gorschek, P. Garre, S. Larsson, and C. Wohlin, "A model for technology transfer in practice," *IEEE Software*, vol. 23, no. 6, pp. 88–95, 2006.

[6] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[7] G. M. Alarcon, L. G. Militello, P. Ryan, S. A. Jessup, C. S. Calhoun, and J. B. Lyons, "A descriptive model of computer code trustworthiness," *Journal of Cognitive Engineering and Decision Making*, vol. 11, no. 2, pp. 107–121, 2017.

[8] I. Bertram, J. Hong, Y. Huang, W. Weimer, and Z. Sharafi, "Trustworthiness perceptions in code review: An eye-tracking study," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–6.

[9] Y. Huang, K. Leach, Z. Sharafi, N. McKay, T. Santander, and W. Weimer, "Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 456–468.

[10] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

[11] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs," in *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020.

[12] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 789–799.

[13] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, no. 7, p. 68–76, Jul. 2011.

[14] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 931–940.

[15] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021.

[16] D. X. B. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.

[17] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. Murphy-Hill, C. Parnin, and J. Stallings, "Gender differences and bias in open source: Pull request acceptance of women versus men," *PeerJ Computer Science*, vol. 3, p. e111, 2017.

[18] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[19] J. R. de Almeida, J. B. Camargo, B. A. Basseto, and S. M. Paz, "Best practices in code inspection for safety-critical software," *IEEE software*, vol. 20, no. 3, pp. 56–63, 2003.

[20] M. C. Code and G. Carullo, "Implementing effective code reviews."

[21] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 111–120.

[22] E. R. Murphy-Hill, J. Dicker, M. Hodges, C. D. Egelman, C. N. C. Jaspan, L. Cheng, L. Kammer, B. Holtz, M. A. Jorde, A. M. K. Dolan *et al.*, "Engineering impacts of anonymous author code review: A field experiment," 2021.

[23] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, 2015.

[24] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?" in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 417–427.

[25] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Springer Science & Business Media, 2012, vol. 5.

[26] S. Nowak and S. Rüger, "How reliable are annotations via crowdsourcing: A study about inter-annotator agreement for multi-label image annotation," in *Proceedings of the International Conference on Multimedia Information Retrieval*, ser. MIR '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 557–566.