Towards Better Code Reviews: Using Mutation Testing to Improve Reviewer Attention

Ziya Mukhtarov, Mannan Abdul, Mokhlaroyim Raupova, Javid Baghirov, Osama Tanveer,
Haluk Altunel and Eray Tüzün
Bilkent University
Ankara, Turkey
Email: {ziya.mukhtarov, mannanabdul, mokhlaroyim, javidbaghirov, osamatanveer}@alumni.bilkent.edu.tr,
altunel@bilkent.edu.tr, eraytuzun@cs.bilkent.edu.tr

***Abstract*—Code reviews, while effective, can be crippled by process smells if not performed correctly. A typical process smell that harms the efficacy of code reviews is the 'Looks Good To Me' (LGTM) smell, wherein a reviewer approves a code review task without reviewing the code attentively. Low-quality code reviews can be harmful, as they can cause bugs to slip into a product codebase leading to potentially severe consequences. In this paper, we propose an innovative solution to potentially minimize the occurrence of the LGTM smell commonly found in code reviews. We built a tool that is a proof-of-concept implementation of our solution, which incorporates the concept of mutation testing into code reviews. It provides a platform where pull request authors can apply mutations to the pull request code in GitHub. Reviewer attention and review efficacy are measured based on their mutation score. To the best of our knowledge, our proof of concept implementation is the first-ever code review tool that uses the concept of mutation testing. We validated our proposed solution with eight developers and received promising results.**

***Index Terms*—Code review, code review process smells, mutation testing, LGTM smell, reviewer attention.**

I. Introduction

Codebases are of great importance in the modern computing era since they serve as the backbone for any piece of technology. Thus, it is imperative to reduce the bugs in the codebase before reaching the final product. Code reviews are one of the many methods used to address such an issue. The code review process is an essential and commonly adopted quality assurance method in software development [1].

Although code reviews are widely accepted and applied in the industry, they are often conducted imperfectly due to the human factor involved in the process. Deviating from best practices or providing sub-optimal solutions in the code review process leads to degraded review quality. Such harmful practices are called code review process smells [2]. One such smell is the 'Looks Good to Me' (LGTM) smell [2]. The LGTM smell occurs when the reviewer does not pay the required attention to the code and approves the changeset with a comment such as "Looks good to me." Such a practice could lead to harmful bugs being introduced to the codebase. Other than the inattention of the reviewer, time constraints applied to the reviews are another source of LGTM smells [3]. According to a study conducted with developers on code review smells, LGTM smell appeared to be amongst the most critical smells voted by 31 out of 32 developers [2]. Correspondingly, Microsoft advises its developers not to pressure the reviewers to give LGTM reviews and emphasizes that it harms the codebase in its developer blog [4].

To the best of our knowledge, we propose the first-ever solution to reduce the occurrence of LGTM smell that applies the concept of mutation testing to code reviews. In mutation testing, the quality of test suites is challenged by injecting small syntactic changes into a piece of code [5]. These changes, while small, have the potential to break the code. After the mutations have been applied, the test suite is run on the mutated code. The testers are expected to create robust tests that detect all mutations. The quality of the test suite can be determined by the percentage of mutations it can find [6], [7]. Our solution proposes using a similar approach to the code review process. Authors can apply mutations to their proposed changeset before requesting a code review. When a reviewer completes their code review, the code review quality can be evaluated based on the percentage of mutations they found. For instance, if the reviewer found 50% of the mutations in the changeset, we can estimate that they also managed to find 50% of the real bugs in the changeset. This solution can potentially prevent the LGTM smell, as reviews can be considered faulty if the reviewer does not find the predetermined percentage of bugs in the changeset. Faulty reviews can be rejected, lowering the possibility of bugs finding their way into a product codebase.

The remainder of the paper is structured as follows. Section II introduces our improved code-review process approach. Section III explains the details of the proof-of-concept implementation of our proposed solution. Section IV presents the case study performed on our proof-of-concept implementation. The paper is finalized with a summary and possible future work in Section V.

II. Proposed Code Review Process

A. Motivation

Our proposal is directly inspired by the concept of mutation testing. Mutation testing measures the effectiveness of test suites by changing some lines of code (called mutations) and then running the test suite on mutated code (mutant) [8]. The test results are analyzed to see if the test suite could identify the mutations (called killed mutants), and the mutation score

is calculated as the ratio of killed mutants to total mutations [7]. The mutation score is representative of the effectiveness of the test suite.

A mutation testing-inspired approach can be used to test if the reviewer is reviewing the code in detail, and a scoring formula similar to the mutation score can be used as a numerical metric of the reviewer's attention and review effectiveness.

B. Proposed Workflow

Figure 1 shows our proposed new flow for code review-related activities. All code review processes start with a developer (author of the changeset) requesting their new changeset to be merged into the main codebase. After opening the request, the changeset should be mutated and prepared for review. We also refer to the mutation process as bug injection since some bugs are intentionally introduced into the code. There are three ways of applying mutations:

- **Manual Bug Injection** - The author injects bugs into the changeset. The advantage of injecting bugs manually is that the author knows the details of the code and can inject bugs into the lines requiring more reviewer attention. However, this approach is time-consuming for the authors, considering that this is additional work. The authors can inject any type of bug into the code, ranging from simple bugs like flipping an equality check to more complex bugs that may span multiple lines of code. However, this approach places the burden of creating meaningful bugs on the authors.
- **Automatic Bug Injection** - Bug injection tools and algorithms [9], [10], [11] can be used to mutate the proposed changeset automatically. Although this option does take the burden of additional work from the author, there might be some problems with the injected bugs. First, the mutation generation process might take a long time, slowing down the code review process. Second, the types of bugs that can be injected depend on the capabilities of the tools we use. Even though some tools can inject semantically-aware bugs, the generated bugs might be too obvious for the reviewer to find, or they can be equivalent or irrelevant. Finally, such automated tools often limit the programming languages they can work with, making them unusable for some code repositories.
- **Hybrid Method** - The manual and automatic bug injection practices can be combined. Automatically generated mutations can be proposed to the author, who can apply them directly or create new mutations. This option seems to be the best, considering that the author can accept all proposed bugs if they do not want to bother with manual injection, or they can create their mutations if the automatic tools take too long or produce unsatisfactory bugs.

For the proof-of-concept implementation, **manual bug injection** is used to apply mutants to the changeset, after which a reviewer can be assigned to start reviewing it. A review process usually consists of a few iterations of feedback and modification between the reviewer and the author when the initial changeset requires further changes.

During the first iteration of the review process, mutated code is presented to the reviewer. At this stage, it should be ensured that the reviewer cannot access the original code, as it is usually possible for reviewers to check the version control history and identify the injected bugs. Special tool support needs to be developed to prevent this possibility of cheating. The reviewer is then asked to identify as many bugs as possible (both injected and possibly original bugs of the author) and add comments explaining the problem as in regular reviews. Once the reviewer finalizes the first iteration, the effectiveness of the review can be measured using the mutation score, the ratio of found injected bugs to total injected bugs.

The mutation score represents the percentage of injected bugs found. A probabilistic estimate can be drawn from the mutation score to predict the number of real bugs found in the code. This probabilistic estimate may not be an accurate representation, but the goal of preventing LGTM smells can be achieved. Admin users of the code repository can determine a threshold value according to their organizational needs. We refer to the first-iteration reviews with the mutation score less than the preconfigured threshold to be **faulty reviews**, while the mutation score higher than the threshold yields **successful reviews**. Faulty reviews can be estimated to contain the LGTM smell since the reviewer could not find a sufficient amount of injected bugs, showing their lack of attention. Such reviews should not be accepted as proper reviews, and another review should be required. This is not to claim that faulty reviews are useless since they can point out some problems the author should be aware of while missing injected bugs. However, faulty reviews should not be used for changeset approvals.

When a faulty review occurs, admin users should be notified and asked to reassign the changeset to a new reviewer. Admin users may give the previous reviewer a second chance, possibly after warning them or assigning an entirely different reviewer. The review directly after a faulty review is still considered the first iteration, so the reviewer will again try to find the injected bugs on the mutated changeset.

The review process for a changeset loops over the previous actions until there is a successful review. To complete the process after a successful review, one minor technicality needs to be handled. Since the original changeset may have been bug-free and the reviewed change set contained injected bugs, all the review comments asking for changes from the author could be related to injected bugs only. After finalizing their review, the reviewer needs to see the details of the injected bugs and verify if all of their comments are related only to injected bugs. If that is the case, the reviewer should be able to update their review decision and approve the changeset. However, if some comments are related to the original changeset, the reviewer needs to be able to confirm that they still want to request changes and notify the author.

In the case of a change request, the author is notified to fix the code. After they fix it, they should request another review from the same reviewer due to the updated code. Since the

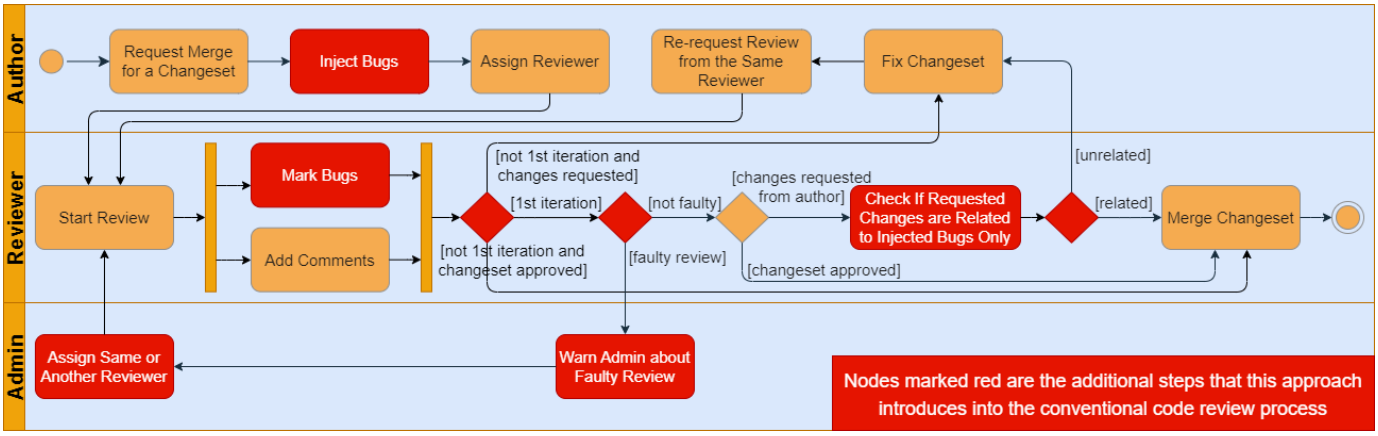

Fig. 1. Proposed code review activity diagram

TABLE I
ADVANTAGES AND DISADVANTAGES OF OUR APPROACH FOR EACH PARTICIPANT

Participant	Advantage	Disadvantage
Author	• Receiving more carefully analyzed code review comments	• Bug injection effort (especially with manual injection) • Longer review process
Reviewer	• Getting performance and improvement measurements	• Wasting time identifying injected bugs • Pressure of being under evaluation
Admin / Team Lead	• Better review results • Identifying problems and fixing them before they reach production • Decreased need for more than one reviewer per changeset	• Overall more complex code review management with the additional burden of dealing with faulty reviews

reviewer's attention was already measured for this changeset, the reviewer is shown the original changeset from the second iteration onward. Although LGTM smell can happen in any cycle in the review process, the same reviewer is unlikely to miss a bug after the first successful review, since the reviewer will need to review smaller changesets related to their initial comments in the following iterations. Hence, the work added for both the author and the reviewer in the following iterations is not worth the resources. Reviewers no longer need to verify their comments and check if they are related to injected bugs since those bugs are not used anymore. The review process loops here between the author and the reviewer until the changeset is satisfactory, which can be approved and merged into the main codebase.

C. Theoretical Evaluation

Considering that this new approach introduces more overhead, evaluating its theoretical benefits and drawbacks is essential. Table I summarizes this evaluation.

We realize that our approach can introduce some overhead. Authors need to spend extra time injecting bugs, especially if manual bug injection is used, while reviewers need to spend additional time identifying those injected bugs. Moreover, our approach increases the overall review process duration due to added processes, which means that authors might receive initial review comments later than usual, and this longer process might negatively affect the development team's velocity. Most importantly, reviewers might feel pressured being under evaluation, and they may fight this approach in favor of using traditional methods where they are not being monitored. Admins additionally need to track faulty reviews and take actions such as reassigning those changesets to another reviewer or issuing a warning to the original reviewer.

However, we believe that the advantages of our approach outweigh its drawbacks. Although the overall review process might take longer, we believe this overhead would have a time and labor cost significantly lower than the cost of fixing bugs that have otherwise made it into the production codebase. This is a trade-off between the effort spent on the improvement of software quality and the effort spent on fixing bugs in production. Our approach uses mutation score as the indicator of code review quality. This is aligned with the shift-left perspective of software testing, which is catching bugs as early as possible due to the parabolic increase of fixing effort in the later stages of the software development life cycle.

Many development teams assign multiple reviewers for the same changeset to catch defects missed by the first reviewer. With our approach, there is decreased need for multiple reviewers, and hence, reduced review effort since the first reviewer must already be able to catch most of the (injected) bugs to have a successful review, which increases its reliability.

Another benefit of our approach is that the authors get their code analyzed in more detail and receive comments that may have been skipped otherwise. Last but not least, reviewers also receive a numerical measurement of their success for each review process, and they can use it to apply different review methodologies and improve themselves.

There remains a risk that developers may reject the approach altogether in favor of accountability-free code reviews. Nonetheless, the added level of accountability could help

evaluate developers' reviews and prevent possible bugs in production.

III. Proof-of-Concept Implementation

To assess the feasibility of our theoretical approach and get feedback from real developers, we implemented a proof-of-concept code review tool named Tithonus[1], incorporating our proposed process flow. It is integrated with GitHub and uses some of its functionality, such as handling the pull request life cycle and code reviews, with the addition of our mutated code reviews to the rotation.

We provided integration with GitHub, as it is one of the most known code hosting platforms, to increase the usability of our tool. Yet some of our proposed innovative features, such as the injection of bugs to the codebase or tracking reviewers' performance, are neither reachable within GitHub software nor feasible as GitHub's extension. Thus, we built Tithonus as a web application to eliminate LGTM smell during code reviews and assess the feasibility of our innovative approach for applying mutation testing in these reviews.

The users are expected to provide their GitHub account credentials and log in to use Tithonus. After logging in, their roles are automatically obtained from their GitHub repository permissions, where they can enter as one of the three user types: Author, Admin, and Reviewer. Reviewers depend on the authors to complete the described mutation process. Authors can inject as many bugs into the changeset as they want and assign the pull request to the Reviewers. Reviewers, unaware of the number of bugs injected, can then perform a review by marking code lines where they think bugs exist and providing explanatory comments. Figure 2 represents an example of a review completed in our software. In the example, a confirmation screen is shown with a changeset, review comments, and bug tags representing whether a reviewer could mark a mutated code line. The confirmation screen enables reviewers with mutation scores (called review accuracy in the application) higher than the threshold to differentiate between injected bugs and original bugs before requesting changes.

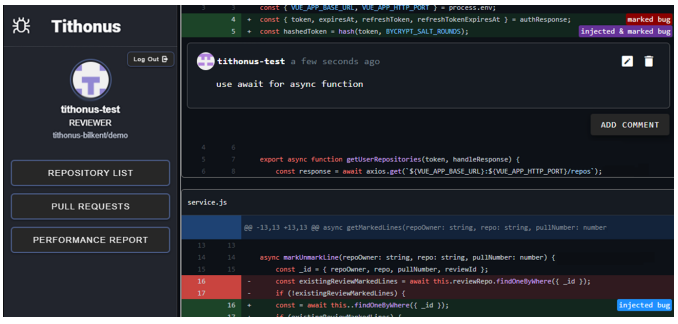

Fig. 2. Example of results of a review in Tithonus

The admin users can see the reviews submitted for the repository and check review details. A reviewer's review and performance history are only available for admin users. Still, they can allow the reviewers to see their results by enabling it through the configuration settings. Figure 3 shows an example of a reviewer performance report. We consider this feature essential since the performance of the reviewers can be tracked to see whether they have improved themselves over time. The admins can also apply time and data interval filters and interact with the graphs to see additional details. In addition to accessing review details, admins can also handle faulty reviews by assigning the pull request to a different reviewer.

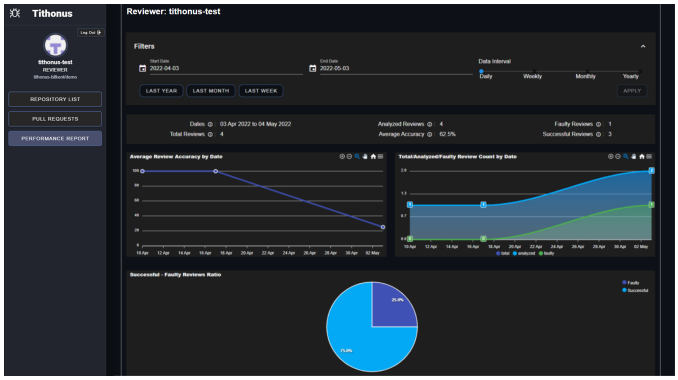

Fig. 3. Example of a performance report in Tithonus

One important thing to emphasize is that in our tool, the code mutations do not affect the original codebase; the bug-injected versions of code lines are only available for Tithonus users and are only visual. We used manual bug injection for the mutation process, which means the authors can manually edit changeset lines. The main reason behind this decision was that most bug injection algorithms have limitations regarding the programming languages used for the code. This would also limit our purpose of reaching out to the software development community. Other than that, our trials with these automated tools showed that the mutated code pieces were usually too obvious for reviewers to find, which would defy our aim of assessing the code review quality. Instead, Tithonus allows injecting manual bugs at lines considered worthy of attention.

IV. Case Study

We conducted a pilot user study to gauge the usability of our tool. The study consisted of four sessions with eight developers from diverse backgrounds with an average experience of eight years. The survey results and interviews from the experiments are presented in the following sections.

A. Case Study Setup

The four experiments consisted of a pair of developers each. A GitHub repository[2] was created along with two GitHub accounts for the users to avert potential privacy concerns. The experiment comprised two scenarios. Scenario 1 involved assigning a developer the Author and Admin roles, and the second developer was given the Reviewer role, while Scenario 2 involved swapping the roles mentioned. In each scenario, the

[1]https://tithonus-bilkent.github.io/

[2]https://github.com/tithonus-bilkent/tithonus-test-environment

author was responsible for adding three bugs for the reviewer to find, whose count was determined based on the length of the sample code and the overhead caused by the injection operation. Since the primary goal of the experience was to get feedback about the usability of our tool, placing exactly three bugs were considered sufficient during the case study. The changeset of the branches included the implementation of well-known algorithms, such as merge sort, in JavaScript language. We expected the developers to be familiar with the workings of these algorithms. Upon completing the experiment, participants filled out a survey[3] and took part in a short interview conducted for application feedback.

B. Case Study Results

The experiment results showed that developers were likely to use the application in practice. Additionally, developers reported the tool to be a means to improve the code review process and evaluate reviewer performance on code reviews. Most of the developers voted on using an integration of the application and GitHub in favor of GitHub standalone. Even though developers rated the application to have a slightly steep learning curve, they agreed on its practicality and potential to improve code quality. Despite their first time using the application, the average time it took the developers to inject three bugs was four minutes. We asked the developers to rate our system with a rating scale ranging from 1 to 5, with one being the least in favor of the tool and five being the most in favor of the tool. The average likeliness of usage and code quality improvement was scored at 3.6 and 3.5, respectively, while the average practicality score stood at 4.4. A response phrased the application as a good step towards improving the code review process and tracking reviewer performances. Another participant mentioned that the application enhanced the reviewer's attention toward code reviews.

As per the survey results, additional improvements in user experience and automated bug injections are necessary. Developers also suggested expanding the tool to integrate other version control software after implementing high-priority features such as automated bug injection. Although this is a new approach to the code review process, the initial results are promising.

V. Conclusion and Future Work

We introduced the idea of integrating mutation testing principles into code reviews to solve the LGTM smell and improve the code review quality. A tool is implemented to prototype our solution through a platform where authors of the changeset can inject manual mutations to the changeset to test the reviewer's attention and the quality of the review. Manual bug injection was preferred to inject meaningful bugs into the sections of the changeset that require the reviewer's attention. The percentage of injected mutations detected during the code review determines the review's quality. Case studies with developers resulted in primarily positive feedback indicating a promising future for the application and its usage in the industry.

In our future work, we plan to handle the bug injection step automatically. According to the feedback from developers, another requested feature was the integration of Tithonus with other platforms such as GitLab[4] or BitBucket[5].

In addition to the technical improvements, we plan to investigate the potential usage of reviewers' mutation scores in the reviewer recommendation tools. For instance, the analytical data gathered by the tool can be used to form a reviewer list where reviewers with high average mutation scores can be prioritized. Moreover, we plan to conduct a test with a large user base to validate the results further. We hope to expand this idea in the future and see its successful applications in real-world code review processes.

[3]https://figshare.com/articles/online_resource/Product_Testing_Survey_for_Tithonus/21904038

[4]Gitlab: https://about.gitlab.com/

[5]BitBucket: https://bitbucket.org/product/

References

[1] J. Klünder, R. Hebig, P. Tell, M. Kuhrmann, J. Nakatumba-Nabende, R. Heldal, S. Krusche, M. Fazal-Baqaie, M. Felderer, M. F. Genero Bocco, S. Küpper, S. A. Licorish, G. Lopez, F. McCaffery, O. Özcan Top, C. R. Prause, R. Prikladnicki, E. Tüzün, D. Pfahl, K. Schneider, and S. G. MacDonell, "Catching up with method and process practice: An industry-informed baseline for researchers," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 255–264.

[2] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, p. 106737, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921001877

[3] Google, "Google engineering practices documentation - speed of code reviews," https://google.github.io/eng-practices/review/reviewer/speed.html, 2020, [Accessed: 2022-05-25].

[4] L. Spieß, "How we do code review," https://devblogs.microsoft.com/appcenter/how-the-visual-studio-mobile-center-team-does-code-review/, Sep 2017, [Accessed: 2022-05-25].

[5] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[6] P. R. Mateo and M. P. Usaola, "Mutant execution cost reduction: Through music (mutant schema improved with extra code)," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 664–672.

[7] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275–378. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245818300305

[8] M. Young and M. Pezze, *Software testing and analysis*. Nashville, TN: John Wiley & Sons, May 2007.

[9] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 511–522. [Online]. Available: https://doi.org/10.1145/3106237.3106280

[10] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Ibir: Bug report driven fault injection," *ACM Trans. Softw. Eng. Methodol.*, jun 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3542946

[11] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 906–918. [Online]. Available: https://doi.org/10.1145/3468264.3468623