Noname manuscript No.
(will be inserted by the editor)

Studying the Impact of Risk Assessment Analytics on Risk Awareness and Code Review Performance

Xueyao Yu · Filipe R. Cogo · Shane McIntosh · Michael W. Godfrey

Author pre-print copy. The final publication is available at Springer via: `https://doi.org/10.1007/s10664-024-10443-x`

Abstract While code review is a critical component of modern software quality assurance, defects can still slip through the review process undetected. Previous research suggests that the main reason for this is a lack of reviewer awareness about the likelihood of defects in proposed changes; even experienced developers may struggle to evaluate the potential risks. If a change's riskiness is underestimated, it may not receive adequate attention during review, potentially leading to defects being introduced into the codebase. In this paper, we investigate how risk assessment analytics can influence the level of awareness among developers regarding the potential risks associated with code changes; we also study how effective and efficient reviewers are at detecting defects during code review with the use of such analytics. We conduct a controlled experiment using Gherald, a risk assessment prototype tool that analyzes the riskiness of change sets based on historical data. Following a between-subjects experimental design, we assign participants to the treatment (i.e., with access to Gherald) or control group. All participants are asked to perform risk assessment and code review tasks. Through our experiment with 48 participants, we find that the use of Gherald is associated with statistically significant improvements (one-tailed, unpaired Mann-Whitney U test, $\alpha = 0.05$) in developer awareness of riskiness of code changes and code review effectiveness. Moreover, participants in the treatment group tend to identify the known defects more quickly than those in the control group; however, the difference between the two groups is not statistically significant. Our results lead us to conclude that the adoption of a risk assessment tool has a positive impact on code review practices, which provides valuable insights for practitioners seeking to

X. Yu, S. McIntosh, and M. W. Godfrey
University of Waterloo
E-mail: {xueyao.yu, shane.mcintosh, migod}@uwaterloo.ca

F. Cogo
Centre for Software Excellence, Huawei Canada
E-mail: filipe.roseiro.cogo1@huawei.com

enhance their code review process and highlights the importance for further research to explore more effective and practical risk assessment approaches.

Keywords Code Review · Risk Assessment · Controlled Experiment

1 Introduction

Code review has long been a part of quality assurance processes for industrial software development. Its practice has been recognized to offer a number of benefits, including easing onboarding and mentoring of new hires, promoting a shared understanding of the system design, and improving overall code quality, including early detection of defects [1, 2]. There are still a large proportion of changes that, despite being reviewed, introduce bugs into codebases in practice [3]. Prior work [4, 5, 6] suggests that this may be mainly due to the lack of reviewer awareness of the riskiness of proposed code changes. While code review tools typically provide reviewers with a fine-grained view of textual differences that highlight the proposed changes, it can be difficult for even the most seasoned developer to retain the historical context in which changes are performed. As a result, changes to code areas that have been historically prone to defects are likely to be underestimated and insufficiently reviewed, which in turn may allow defects to slip into the codebase.

Despite the existence of various approaches to identifying risky code areas, little research has been conducted to investigate whether and how the code review process can take advantage of change risk assessment techniques. As a preliminary exploration of potential improvements for code review processes, we attempted to incorporate *Just-In-Time (JIT) defect prediction* [7, 8, 9, 10, 11], which mines historical records of defect-fixing and fix-inducing commits to train models that assess the risk of changes based on their characteristics (see Section 3). The motivation to explore those two techniques, in specific, is that they have been well explored by prior research and are readily available as tools that can be integrated into code review. While JIT models can identify risky changes effectively [9, 10, 12], they are not helpful in the code review process due to their inability to provide clear reasoning and actionable messages [13]. Moreover, prior work argues that, to be integrated smoothly into code review, bug detection tools should have a false positive rate of no more than 10% [14, 15]. When we applied JITLine [12] — a state-of-the-art JIT defect prediction approach — on our collected data from Qt and Apache, we obtained a precision of 0.17 and 0.12, respectively, which falls well below acceptable performance levels. Next, we explored if *Automatic Static Analysis Tools (ASATs)* [16, 17, 18, 19, 20] could improve the code review process; ASATs scan codebases for patterns that are associated with concrete defect cases, such as *use-after-free* and *buffer overflows*. We found that while ASATs provide actionable reports, they do not provide a comprehensive assessment of the risk posed by a change. For example, defects that are not associated with known patterns (e.g., logic errors, incomplete implementations or requirements) cannot be detected.

Reflecting on the results of our preliminary investigation, we propose Gherald, a prototype that enhances code review interfaces with risk assessment capabilities.

Gherald measures popular risk metrics used by defect prediction techniques and enables developers to gain insight into the riskiness associated with the author, file, and method involved in a code change. We hypothesize that the usage of Gherald during code review impacts developers' *risk awareness* (i.e., the ability to estimate the risk posed by change sets), *effectiveness* (i.e., the ability to identify defects when they are present), and *efficiency* (i.e., the speed at which review tasks are completed). To evaluate our hypothesis, we perform an experimental study to investigate three Research Questions (RQs):

(RQ1) Is the use of Gherald associated with greater awareness of the riskiness of changes?

Motivation: Understanding the risk associated with a proposed change is an important requirement for effective code review [1] and for the optimal allocation of code review effort. Therefore, we conjecture that displaying heuristics during code review can significantly improve reviewers' ability to estimate the risk of a change.

(RQ2) Is the use of Gherald associated with an improvement in code review effectiveness?

Motivation: To avoid defects in customer-facing products, reviewers strive to identify as many as possible after changes are performed in a code base. Identifying defects is one of the key motivators for performing code review [1], particularly for defects that are hard to catch through testing such as those involving concurrency [21]. We hypothesize that Gherald can influence code reviewers' ability to identify existing defects in the set of changes under scrutiny.

(RQ3) Is the use of Gherald associated with an improvement in code review efficiency?

Motivation: A side effect of code review that is often negatively perceived by developers is the potential increase in the time taken for a change to be incorporated into a software release [21]. As a result, developers seek to minimize the duration of the code review cycle, which includes the identification of defects by reviewers. Therefore, we want to investigate whether Gherald can shorten the time taken by code reviewers when identifying defects in a change set.

To investigate our RQs, we evaluated Gherald using a controlled experiment with 48 participants, employing a *between-subjects* experimental design [22] by dividing participants into two groups: one with tool assistance from Gherald (treatment group), and one without (control group). We compared the performance of the two groups on pre-assigned risk assessment and code review tasks. More specifically, we measured how well participants estimate the defect density of a change set (RQ1), how many defects participants are able to detect in a change set (RQ2), and how quickly participants can detect these defects (RQ3).

We found that Gherald was associated with statistically significant improvements in developers' awareness of the riskiness of code changes and their code review effectiveness. However, the difference between the treatment and control groups was not statistically significant with respect to code review efficiency. Overall, we found that the use of a risk assessment tool had a positive effect on code review practices. With the assistance of Gherald, developers had greater awareness of the riskiness of code changes and were able to detect defects more effectively.

1.1 Paper Organization

The remainder of the paper is organized as follows. Section 2 situates this work with respect to the literature and provides a motivating example to illustrate the intended use case. Section 3 introduces our proposed risk assessment prototype. Section 4 outlines the design of our between-subjects experiment, while Section 5 presents the results. Section 6 discusses the findings and practical implications. Section 7 describes the threats to the validity of this study, and Section 8 summarizes our work and presents our conclusions.

2 Related Work and Motivational Example

In this section, we discuss research that relates to code review and risk assessment approaches, and we provide an example that motivates our study.

2.1 Code review

Peer code review, as a manual code inspection process performed by fellow developers, has long been applied to ensure the quality of software projects [23, 24]. The concept of code review dates back to 1976 when Fagan proposed a highly structured process called *code inspection* [25], where the authors and reviewers sit together in an in-person meeting, follow a checklist, and conduct line-by-line group reviews for the code under examination. This process has been proven beneficial in defect finding [26]; however, due to its cumbersome, time-consuming, and synchronous nature [27], the traditional code inspections have been gradually replaced by a more lightweight, tool-based, and asynchronous practice — *modern code review* [1]. Nowadays, modern code review has been increasingly adopted in both industrial (e.g., Google [2], Microsoft [1], Facebook [28]) and open-source projects [29, 30] and inspired numerous related studies [3, 31, 32, 33, 34].

Despite its wide adoption, developers still encounter challenges in the code review process, especially in understanding the context and the content of changes [1, 35, 36]. As a result, in recent years, considerable research effort has been invested in developing techniques to mitigate this challenge and improve code review performance, which is typically measured by the number of defects found (effectiveness) and the time taken to find them (efficiency) [37]. For example, researchers have explored various visualization techniques to enhance developers' understanding of the code changes [38, 39, 40, 41]. Zhang et al. proposed a tool that summarizes similar changes and detects potential defects based on the change content and context [42]. Tao et al. found that the code review process can be hindered by the presence of large, composite code changes that comprise multiple independent issues [36]. To address this, several approaches were proposed to decompose composite changes into groups of cohesive and self-contained changes [43, 44, 45].

Additionally, given the cognitively demanding nature of code review [1, 30, 46], a number of studies have been conducted to improve the performance of code review by reducing its cognitive load on developers (i.e., the amount of working memory necessary to perform a task) [47]. For example, Gonçalves et al. [48] studied whether explicit review strategies — such as using a checklist — can reduce developers' cognitive load and improve the overall performance of code review. Moreover, there has been research examining the role of displayed file order in code review and exploring its potential to improve code review performance. Baum et al. [49] proposed to reduce the cognitive load of reviewers by presenting the change parts in a more helpful order, such as by grouping related change parts together. Similarly, Fregnan et al. [50] conducted a user study to evaluate the hypothesis that displayed file order has an impact on the code review performance in defect detection. Building upon these ideas, we hypothesize that ranking changes by their associated risk will promote developers' performance (i.e., effectiveness and efficiency) in code review.

2.2 Risk assessment approaches

In the past decades, a plethora of studies have explored how to identify defect-prone components so that quality assurance resources can be allocated effectively. One common approach is to apply *defect prediction models*, which are trained using historical release data to identify modules (i.e., files, classes, or packages) with a higher likelihood of post-release defects [51, 52]. However, traditional defect prediction models tend to provide recommendations at a coarse granularity, which is impractical for real-world application [7, 53, 54]. For example, such models often suggest inspection of large files or packages, which can be too complex for developers to recall and resolve effectively after a release. Moreover, as these files or packages are often contributed by multiple developers, it is unclear who should be responsible for the inspection task. Recently, the concept of Just-In-Time (JIT) defect prediction has emerged [7, 11, 54, 55, 56], which improves the traditional defect prediction methods by making predictions at a finer-grained change-level and assigning predictions to a specific author of the change. More importantly, JIT defect prediction models can monitor code change in real-time as they are created, which allows for prompt defect inspection and correction while design decisions are still fresh in developer's memory. Nowadays, JIT defect prediction has been adopted by many industrial software teams, including Avaya [55], Blackberry [54], and Cisco [57]. Also, an increasing number of recent studies have sought to improve JIT defect prediction models [8, 9, 10, 11, 12].

Automatic static analysis tools (ASATs) are another popular approach to finding potential defects in code changes and to help assure software quality [58]. With the aid of ASATs, some coding standard violations and common defect patterns can be automatically detected, which can significantly reduce reviewers' effort during code review. In recent years, ASATs have been widely adopted into the software development process for defect detection and have supported a variety of programming languages and defect patterns [16, 17, 18, 19, 20, 59].

Our study aims to explore whether the use of risk assessment can enhance a reviewer's risk awareness and improve their code review performance. We have considered using an existing risk assessment approach to evaluate its relationship with code review performance. However, we found that ASATs suffer from a high rate of false alarms so that developers often ignore most of their warnings [14, 60, 61]. Previous research has indicated that a significant percentage, ranging from 35% to 91%, of bug warnings generated by ASATs are routinely disregarded by software developers [62, 63, 64]. Furthermore, ASATs primarily operate at the line level, where they identify defects by matching code against predefined defect patterns. This mechanism limits their ability to offer a comprehensive assessment of the overall risk associated with a code change. We also found that there are still some challenges in integrating JIT defect prediction models into code review, such as the inability to explain the prediction results and provide actionable suggestions [57]. Moreover, existing JIT defect prediction models are unable to achieve the level of performance (i.e., less than 10% false positives) needed for code review integration [14, 15]. Khanan et al. [65] introduced a JIT defect prediction bot, JITBot, providing explainable and actionable feedback in code review; however, it does not take into account historical change characteristics (e.g., prior changes and developer experience), which are essential to the defect-proneness of a change. Recently, Fregnan [15] compiled a set of requirements needed for integrating defect prediction into code review practices. Building upon their findings, we introduce a risk assessment prototype for this study. Further details are described in Section 3.

2.3 A Motivational Example

Suppose Alice is a developer who has recently joined a software project. As part of her duties, she is expected to conduct code reviews for changes submitted by the other team members. Recently, she has received three pending review requests, one of which was submitted by a junior developer, Bob, who has made a 10 LOC bug fix to a major feature of the project.

Of course, Alice is also occupied with her own development tasks. In order to effectively manage her workload, Alice decides to focus more of her reviewing efforts on the riskier patches. Based on her intuition and experience, she believes that larger patches are generally more complex and thus have a higher likelihood of containing issues that require explicit consideration. Consequently, she prioritizes the largest patches and takes ample time to review them thoroughly. Bob's bug fixing change, on the other hand, contain only minor modifications to the source code, so Alice quickly reviews and approves it; however, there are underlying risks at play. First, the file that Bob modified is historically bug-prone, which can be inferred from the number of bug fixes it has been associated with. Also, Bob, as a junior developer,

Table 1: The statistics of the studied systems and the performance of JITLine in the preliminary study.

System	Timespan		Changes		Performance					
	Start	End	Total	Defective	Precision	Recall	F1	AUC	FAR	d2h
Qt	08/2009	08/2022	57,784	12,146	0.49	0.17	0.26	0.75	0.03	0.59
Apache	07/2002	07/2022	6,839	590	0.30	0.12	0.17	0.82	0.01	0.62

has little experience with the project and has not modified this particular file before. Consequently, after her cursory inspection, a bug is inadvertently introduced into the codebase as Alice approves the change of an inexperienced developer working on a historically defect-prone file.

Now let us consider the potential benefits of providing Alice with a risk assessment tool that offers contextual information about the changes being made. Such a tool could inform her about the number of changes the author has contributed to the project and the modified files. Also, the tool could provide information about the number of prior changes and prior bugs related to each file and method in the patch.

Such a risk assessment tool would not strive to identify specific coding mistakes vis-a-vis ASATs. Instead, it would provide contextual information that may be overlooked by reviewers. We anticipate that this tool would improve the reviewers' awareness of the risk associated with changes, thereby reducing the likelihood of defective code being introduced into the codebase.

3 Gherald

We introduce a risk assessment prototype — Gherald— to analyze the riskiness of code change based on the historical data of its author, files, and methods.

3.1 Preliminary Study on Existing Solutions

Prior works on JIT defect prediction suggested approaches to measure the risk of a change [7, 8, 9, 10, 11]. We evaluated the potential of incorporating JITLine [12], which is currently the most accurate, cost-effective, and time-efficient approach for predicting JIT defect-introducing changes; however, we decided not to adopt it when we found its performance to be inadequate. As the dataset used for JITLine is outdated, we collected the most recent data from Qt Base (Qt) and Apache Commons Lang (Apache) and replicated the study. Specifically, we started with a dataset of unreviewed changes and extracted two groups of features: a) *token features* i.e., source code tokens of a change, and b) *code features* i.e., specific properties of the code changes, such as the size of the change, the number of modified files, and the number of prior changes that the author produced. We used these features to build a change-level JIT defect prediction model. The model estimates the riskiness of each new change by predicting its defect density. The statistics of the dataset and the results of the study are shown in Table 1. Prior work demonstrated that to be deemed suitable for adoption in code review, bug detection tools need to produce false positives at a

Table 2: The metrics of risk assessment.

	Metrics	Description
Author	Project experience	The relative value of the author's prior changes compared to the other authors; prior changes: the number of prior changes submitted by the author.
	Recent activity	The relative value of the author's recent changes compared to the other authors; recent changes: the number of prior changes submitted by the author weighted by the age of the changes.
	File expertise	The relative value of author's file awareness compared to the other authors; file awareness: the proportion of the prior changes to the modified files that the author has participated in.
File/Method	Prior changes	The number of prior changes to the object[1].
	Recent changes	The number of prior changes to the object[1] weighted by the age of the changes.
	Prior bugs	The number of prior bugs occurred in the object[1].
	Recent bugs	The number of prior bugs occurred in the object[1] weighted by the age of the bugs.

[1] Either file or method.

rate of 10% or less [14, 15]. However, in our replicated study, JITLine yields a precision of 0.49 and 0.3 for Qt and Apache projects, respectively, which is substantially below the level of performance needed for code review integration.

Additionally, JIT defect prediction models lack the ability to effectively explain the features that induce the risk of change. As indicated by prior works [66], `Size` properties are the primary contributor for predicting the defect-proneness of a change. Nonetheless, we designed our study so that participants could complete the code review tasks in a reasonable amount of time by selecting small- to medium-sized changes. However, this selection criterion substantially reduces the explanatory power of a defect prediction model. As a result, instead of constructing a defect prediction model that produces an overall score indicating the defect-proneness of a change, we compute and present the metrics that are commonly associated with the riskiness of a change and are widely used in defect prediction techniques.

3.2 Gherald Risk Metrics

We considered the factors that have been shown to be related to the defect-proneness of a change in past work. Previous research has consistently highlighted the importance of developer-related information in assessing the likelihood of defects in a change [55, 67]. For example, Mockus and Weiss found that developers with greater experience have a significantly reduced likelihood of introducing defects [55]. Consequently, features that characterize the experience of the author of a change have been frequently employed in JIT defect prediction models to predict the potential defect-proneness of a change [7, 66].

Historical features of a change have also proven valuable in predicting whether it will induce a defect. Past studies have shown that measures such as the number of prior changes and defect fixes associated with a file are reliable indicators of that file's susceptibility to defects [68]. Moreover, a large proportion of developers opt for the most recently modified file and the most recent buggy file to identify potential buggy files during code reviews [69]. Hence, we believe that historical information, both at the file level and through a more granular method-level analysis, will be beneficial for risk assessment of a change.

While attributes in Size and Diffusion dimensions have been commonly deemed important for assessing change risk, this information is typically evident within the change itself — it is hard for a reviewer to miss an abnormally large change. Furthermore, our experimental design focused mainly on small changes affecting only a small number of files to ensure participants could grasp the context and complete the tasks within the allotted time. Consequently, we did not include these particular features as additional considerations in our study.

Ultimately, we computed a broad range of metrics concerning the three categories: *author*, *file*, and *method*. Table 2 provides an overview of these metrics.

Author metrics measure the author's relative *project experience* (i.e., how many changes the author has committed), their *recent activity* (i.e., how many changes the author has recently committed), and their *file expertise* (i.e., how many changes the author has committed to the files in change) compared to the other authors. For each change, we first measure the author's (a) *prior changes*, (b) *recent changes*, and (c) *file awareness* using past changes. We followed the same approach used in prior studies [55, 66] to assess these attributes. Prior changes refer to the number of past changes the author has submitted to the codebase. Recent changes is measured by weighting the prior changes by their age; specifically, we apply a weight of $\frac{1}{1+age}$ to each change and sum the weighted values, where age represents the time elapsed since the date of change measured in years. File awareness is calculated as the proportion of the prior changes to the modified files that the author has participated in.

Given that the absolute metric values lack interpretability — making it challenging for reviewers to discern whether an author's prior changes are more or less than those of other authors — we transformed these metrics into relative values. To facilitate meaningful comparisons among different authors, we normalized the values by dividing them by the maximum value observed within the six-month period preceding the date of the change. This normalization allowed us to establish the author's relative project experience, recent activity, and file expertise.

File/Method metrics measure the change history and past defect tendencies of the modified files and methods. For each change, we computed *prior changes* and *prior bugs* by measuring the number of prior changes and bugs that are associated with each modified file and method. To account for the recency, we also measured the file and method's *recent changes* and *recent bugs*. Similar to the procedure applied for author's recent changes, we assign a weight to each change or bug, and sum the weighted values.

3.3 Change-Level Risk Score

With the risk metrics measured, we assessed the riskiness at the change level. Instead of presenting an overall score that summarizes all risk metrics and predicts the likelihood of a change being defect prone, we introduced individual risk scores for each category at the change level. This allows reviewers to easily identify the specific risk factors contributing to defect-proneness. Since we did not find a suitable solution in existing literature, we developed our own heuristics to quantify the risk for each category.

Author risk. The author risk score was calculated by taking the complement of the average score of the author's a) project experience, b) recent activity, and c) file expertise using this formula:

$$\mathrm{CRisk_a} = 1 - \frac{1}{3}(\mathrm{ProjExp} + \mathrm{RecAct} + \mathrm{FExp})$$

File/Method risk. To measure the file and method risk score, we first compute the risk score at the file/method level, which is calculated by taking the odds ratio of recent defect-proneness:

$$\mathrm{FRisk/MRisk} = \log(\mathrm{RecCng} + 2) \times \frac{\mathrm{RecBug} + 1}{\mathrm{RecCng} - \mathrm{RecBug} + 1},$$

To mitigate the occurrence of incomputable values when the denominator equals zero, we increased the denominator by one. We also added one to the numerator to prevent risk scores from being zero when recent bugs are absent, which is quite common in the case of new files/methods with limited recent changes. As suggested by prior research [7], we recognized that modules that have undergone more changes are likely to be more risky, as the reviewers need to recall and track more previous changes. Consequently, a file/method with a substantial number of recent changes is more susceptible to risk than one with only few recent changes, even if they possess the same bug-change odds ratio. To account for such distinction, we incorporated a corrective factor $log(RecCng + 2)$ and multiplied the odds ratio by this factor. We added two to the file recent change to ensure it would not result in negative or zero values.

The overall risk score at the change-level was computed by taking the mean of the risk score of each file/method involved in a change:

$$\mathrm{CRisk_{f/m}} = \frac{1}{n}\sum_{i=1}^{n}(\log(\mathrm{RecCng} + 2) \times \frac{\mathrm{RecBug} + 1}{\mathrm{RecCng} - \mathrm{RecBug} + 1})$$

To improve the explanability and interpretablity of the risk score, we normalized the score by dividing its value by the maximum value in the recent six month prior to the author date of the change. This allows the score to be interpreted as the relative file/method risk compared to the other changes.

4 Experiment Design

We conducted a controlled experiment[1] to examine how the use of Gherald impacts code review practices. We employed a between-subjects experimental design [22] by assigning participants into two groups: the control group, who were not given access to our risk assessment tool Gherald, and the treatment group, who were given access to Gherald. We then asked the participants to perform a set of risk assessments and code review tasks. Afterwards, we compared the groups in terms of their risk awareness and code review performance (i.e., code review effectiveness and efficiency). Figure 1 provides an overview of our study design.

4.1 Pre-Experiment Data Collection

4.1.1 Subject Systems/Communities

We perform our study on Apache Commons Lang,[2] which provides helper utility methods for the manipulation of Java core classes. We chose this system for several reasons: First, Apache[3] is one of the largest open-source organizations, and projects hosted by the Apache Software Foundation follow a standard issue-reporting process and adhere to a common set of code review policies. Also, because it is a library of utility methods, Apache Commons Lang is designed to be easy to use and understand. The files and methods are decoupled and independent. Moreover, each method is named in a clear and descriptive manner and contains a well-written description that not only explains how to use the method, but also provides examples demonstrating its usage. Since our experiment involves participants conducting code reviews, familiarity with Java development is sufficient to understand the changes selected from Apache Commons Lang, without requiring any additional contextual learning.

4.1.2 Data Extraction

We extracted the issue data (e.g., IssueID, CreatedDate, Type) from the Jira issue tracking system (ITS) used by the Apache Commons Lang development team[4] and selected the issues of the type Bug. Next, we extracted the commit data from the git version control system[5] (VCS) and joined it with the issue data using the unique identifier for issues (i.e., IssueID). This allowed us to identify the defect-fixing changes. Then we applied the SZZ algorithm using PyDriller [70] to identify the *fix-inducing changes* — the set of changes that last "touched" the lines that were modified by the defect-fixing changes.

[1] This experiment was reviewed by and received ethics clearance from the University of Waterloo Research Ethics Committee (ORE #44022).
[2] `https://commons.apache.org/proper/commons-lang/`
[3] `https://www.apache.org`
[4] `https://issues.apache.org/jira/projects/LANG/issues/`
[5] `https://github.com/apache/commons-lang/`

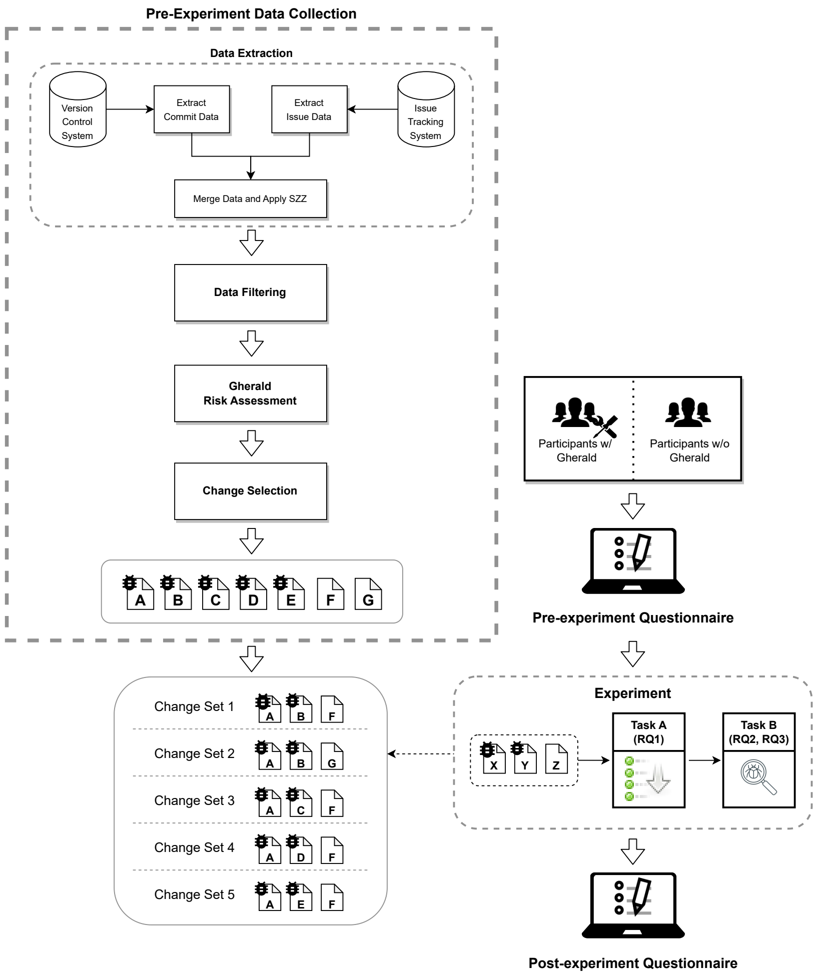


Fig. 1: An overview of our study.

4.1.3 Data Filtering

Since the SZZ algorithm may result in false positives in identifying fix-inducing changes, we applied a list of filter steps to remove the suspicious data and to mitigate noise [66]. We closely followed McIntosh and Kamei's data filtering approach that refined and produced a widely-adopted dataset in JIT defect prediction studies. Table 3 shows the number of fix-inducing changes after applying each filter sequentially. First, we filtered out changes that updated only code comments and whitespace (F_1) [71], as such changes are typically irrelevant to fix-inducing events. Next, we excluded the fix-inducing changes that were committed after the date that the defect was initially reported (F_2) [72]. We also removed the outliers that change more than 10,000 lines (F_{3a}) or more than 100 files (F_{3b}). Furthermore, due to limitations of the SZZ algorithm, which identifies only commits that introduce new lines as possible fix-inducing changes [73], we eliminated changes that did not introduce new lines (F_{3c}). Then, we stratified the data into time periods and analyzed the rate of fix-inducing changes. We excluded time periods with unstable fix-inducing rates (F_4), particularly the last year of data collection with a diminishing fix-inducing rate approaching zero. Finally, we filtered out the suspicious fixes (F_5) and suspicious fix-inducing changes (F_6). Specifically, we calculated the upper Median Absolute Deviation (MAD) of the number of bugs fixed for each change and the number of fixes induced by each change. Then, we ignored changes that fix or induce more than the respective MAD value.

Table 3: The number of changes after each filtering step.

#	**Filter**	**Total**	**# Fix-inducing**	**% Fix-inducing**
F_0	No filters	6839	590	9%
F_1	Code comments and whitespace	6361	590	9%
F_2	Issue report date	6250	479	8%
F_{3a}	Too much churn	6245	477	8%
F_{3b}	Too many files	6231	474	8%
F_{3c}	No lines added	5922	474	8%
F_4	Period	5468	473	9%
F_5	Suspicious fixes	5281	286	5%
F_6	Suspicious inducing changes	5256	261	5%

4.1.4 Gherald Risk Assessment

After cleaning the dataset through the filtering phase, we proceeded to compute the risk metrics and calculate the change-level risk scores for changes in the dataset, as detailed in Section 3.2 and Section 3.3.

4.1.5 Change Selection

We required a set of code changes to seed the reviewing tasks that participants were asked to complete. To select the code changes for our experiment, we applied a list of inclusion/exclusion criteria to the filtered changes in the project. To avoid outdated changes that may be refactored or deprecated by recent changes, we excluded changes submitted ten years prior to our change selection. Due to the limited time that participants had to perform the experiment, we ignored large changes — those that modify more than 200 lines or more than ten files — since they require a considerable investment of time to review. Then, we inspected the remaining changes and selected a sample set that, in our opinion, clearly state a well-scoped problem, are conceptually self-contained, and are straightforward to understand without requiring reading other source files or documentation.

Table 4: Code changes selected for the experiment.

Change	Class & Method	Detail	Defect
A	`StringUtils`: unwraps a string from a string/char	Add condition checking for invalid string length	Incomplete condition checking
B	`StringUtils`: checks if any one of the `CharSequences` are not empty/blank	Add new methods	Return incorrect boolean for certain case
C	`NumberUtils`: checks whether the String is a valid Java number	Handles octal notations	Incorrect conditional branching for octal numbers without considering decimal fractions
D	`NumberUtils`: turns a string value into a `java.lang.Number`	Deal with all possible prefixes for hex numbers; handle large hex numbers	(1) Incorrect conditional branching for `Long` and `BigInteger`; (2) missing a hex number prefix type
E	`NumberUtils`: converts a `String` into a `BigInteger`	Handles hex and octal notations	(1) Hex number prefix typo; (2) missing a hex number prefix type
F	`StringUtils`: wraps a string with a string/char	Add new methods	N/A (has not been found yet)
G	`StringUtils`: gets the substring before the first occurrence of a separator	Add a new method	N/A (has not been found yet)

Seven code changes were ultimately selected for the experiment. Of these, five changes (i.e., change A-E) were labelled as *Buggy changes*, as they necessitated further fixes. Two changes that did not induce any fixes by the date of data collection were labelled as *Neutral changes*. The selected changes were diverse in terms of class types, modified methods, as well as associated defects. For example, changes A, B, F, and G related to character string handling, while changes C, D, and E pertained to number conversion and parsing. A more detailed description of each change, along with its associated defect, is provided in Table 4.

4.2 Experiment Platform

We developed a web application for participants to complete their experimental tasks. The participants were able to directly access the application by opening the URL provided in their invitation emails. After obtaining the consent of the participant, the application automatically logged their answers and timed their tasks.

4.3 Experimental Artifacts

4.3.1 Change Sets

We created five change sets, each consisting of three experiment code changes with varying Gherald risk scores and defect density. The assignment of code changes to

Table 5: Five change sets selected for the experiment.

Change set	Change	Defect Count	Defect Density
	A	1	0.17
1	B	1	0.05
	F	0	0.00
	A	1	0.17
2	B	1	0.05
	G	0	0.00
	A	1	0.17
3	C	1	0.06
	F	0	0.00
	A	1	0.17
4	D	2	0.13
	F	0	0.00
	A	1	0.17
5	E	2	0.10
	F	0	0.00

their respective change sets is presented in Table 5. Each participant was assigned one of the five change sets for their experiment. We conjectured that changes C, D, and E were more challenging as they are related to hex and octal numbers which, compared to basic character string utilities, required a higher level of Java knowledge and specialized familiarity with Java number types. This conjecture was validated in a pilot study involving 20 graduate students in computer science, where none of them identified the defects in changes C, D, and E. Despite this, to improve the generalizability of the results, we still included these changes to explore the effect of Gherald on different types of defects at different levels of difficulty. However, we avoided assigning change sets containing these changes to participants with less than one year of development experience or Java experience.

4.3.2 Gherald

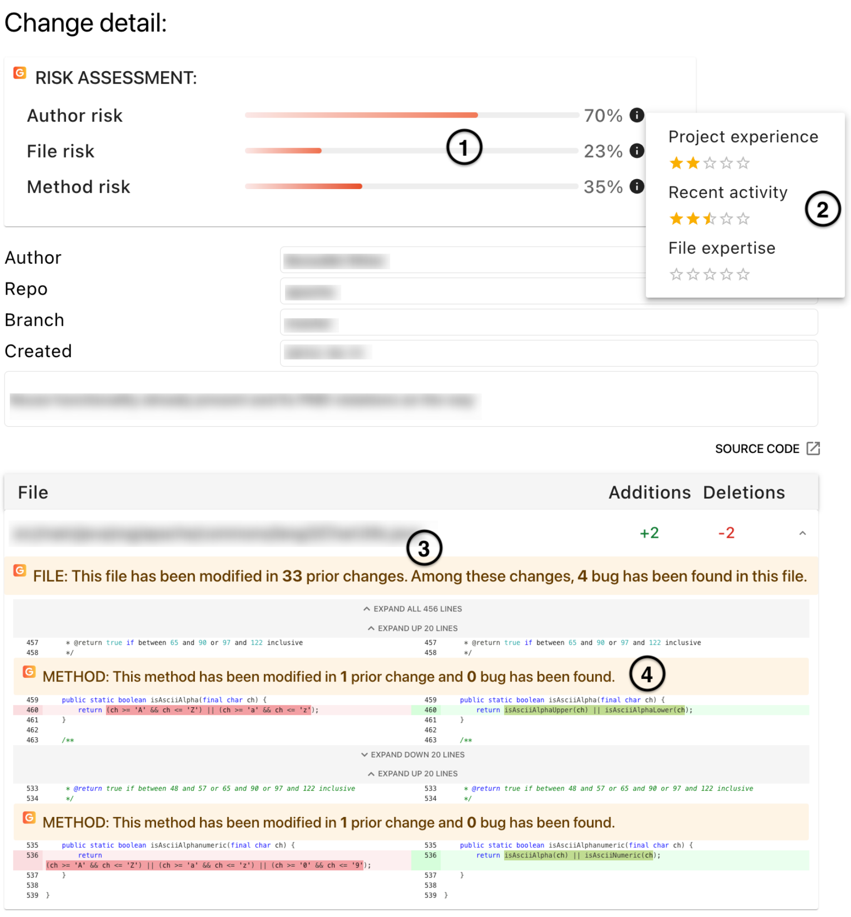
Participants in the treatment group were provided with access to Gherald during the experiment. Figure 2 shows the experiment platform interface for the treatment group. For each change, an overall risk assessment is presented (Figure 2 ①), indicating the riskiness of the author, files, and methods involved in the change. The risk percentages are relative scores compared to the other changes in the six months prior to the author date of the examined change. Gherald also offers more fine-grained information such as *author expertise and activity* (Figure 2 ②), as well as *file/method change history* and *bug tendency* (Figure 2 ③ ④).

4.4 Study Variables

This section discusses the variables that we collected and analyzed.

Table 6: The variables of the study.

Name	Description	Scale	Operationalization
Independent variable:			
Risk assessment support	Whether the participant is provided with risk assessment support	Nominal	See section 4.4.1.
Dependent variables:			
Risk awareness	Normalized pairwise agreement between the participant's rankings of changes based on the estimated risk level and the rankings of changes be their future defect density	Ratio	Computed at the end of type A tasks using the participant's rankings and the rankings by defect density. See section 4.4.2.
Code review effectiveness	Ratio of the total number of known defects correctly identified by the participants over the number of known defects in the change set	Ratio	Computed at the end of type B tasks using the number of detected known defects and the total number of known defects.
Code review efficiency	Number of known defects correctly identified per review hour	Ratio	Computed at the end of type B tasks using the number of detected known defects and the review time.
Confounding variables:			
Change set	The change set provided to the participants during the experiment	Nominal	Design: each participant is assigned to a change set selected from the 5 sample change sets
Review order	Order of code changes presented for review	Nominal	Measured: 3 types ("high risk to low risk", "low risk to high risk", "does not matter"); pre experiment questionnaire
Development experience	Years of participant's software development experience	Ordinal	Measured: 3-point scale ("less than a year", "1 year to 5 years", "5 years or more"); pre experiment questionnaire
Java experience	Years of participant's Java experience	Ordinal	Measured: 3-point scale ("less than a year", "1 year to 5 years", "5 years or more"); pre experiment questionnaire
Code review experience	Years of participant's code review experience	Ordinal	Measured: 3-point scale ("less than a year", "1 year to 5 years", "5 years or more"); pre experiment questionnaire
Coding hour per week	Participant's average coding hour per week	Ordinal	Measured: 3-point scale ("less than five hours", "five to ten hours", "ten hours or more"); pre experiment questionnaire
Review hour per week	Participant's average review hour per week	Ordinal	Measured: 3-point scale ("less than five hours", "five to ten hours", "ten hours or more"); pre experiment questionnaire
Fitness	Perceived energy level of the participant during the experiment	Ordinal	Measured: 3-point scale ("low", "moderate, "high"); post experiment questionnaire
Understandability of changes	Participant's overall understanding of the provided code changes	Ordinal	Measured: 3-point scale ("barely understand", "somewhat understand", "understand very well"); post experiment questionnaire
Difficulty of tasks	Participant's perceived difficulty of the assigned tasks	Ordinal	Measured: 3-point scale ("easy", "moderate", "very hard"); post experiment questionnaire


Change detail:

RISK ASSESSMENT:

Author risk	70%	
File risk	23%	
Method risk	35%	

Project experience ★★☆☆☆

Recent activity ★★⯪☆☆

File expertise ☆☆☆☆☆

Author	[redacted]
Repo	[redacted]
Branch	[redacted]
Created	[redacted]

[redacted]

SOURCE CODE

File	Additions	Deletions
[redacted]	+2	-2

FILE: This file has been modified in **33** prior changes. Among these changes, **4** bug has been found in this file.

```
                                    EXPAND ALL 456 LINES
                                    EXPAND UP 20 LINES
   * @return true if between 65 and 90 or 97 and 122 inclusive      * @return true if between 65 and 90 or 97 and 122 inclusive
   */                                                               */
```

METHOD: This method has been modified in **1** prior change and **0** bug has been found.

```
  public static boolean isAsciiAlpha(final char ch) {                   public static boolean isAsciiAlpha(final char ch) {
      return (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');          return isAsciiAlphaUpper(ch) || isAsciiAlphaLower(ch);
  }                                                                     }

  /**                                                                   /**
                                    EXPAND DOWN 20 LINES
                                    EXPAND UP 20 LINES
   * @return true if between 48 and 57 or 65 and 90 or 97 and 122 inclusive    * @return true if between 48 and 57 or 65 and 90 or 97 and 122 inclusive
   */                                                                          */
```

METHOD: This method has been modified in **1** prior change and **0** bug has been found.

```
  public static boolean isAsciiAlphanumeric(final char ch) {            public static boolean isAsciiAlphanumeric(final char ch) {
      return                                                                return isAsciiAlpha(ch) || isAsciiNumeric(ch);
(ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') || (ch >= '0' && ch <= '9');
  }                                                                     }

}                                                                     }
```

Fig. 2: Gherald experiment interface.

4.4.1 Independent Variable

Our overall goal was to investigate the degree to which code review performance can vary depending on whether risk assessment support is provided. Hence, we set *risk assessment support* as the independent variable.

Participants in the treatment group were exposed to risk assessment support from the experiment user interface. They were able to assess the riskiness of code changes and to conduct code reviews with the assistance of Gherald during the experiment. In contrast, participants in the control group were not provided with risk assessment support from the experiment user interface.

4.4.2 Dependent Variables

The dependent variables are metrics that we use to measure the participants' performance in the assigned tasks. We measured the dependent variable for RQ1 as the developer's *risk awareness* of code changes. For RQ2 and RQ3, we use *code review effectiveness* and *code review efficiency* as dependent variables, respectively.

Risk awareness is the degree to which reviewers recognize the potential for defects or failures that may be induced by a code change. The participants were asked to evaluate the perceived level of risk associated with a collection of code changes by arranging them in order of their perceived risk level. We estimated the risk awareness of a participant by computing the agreement between the ranking that they provided and the ranking of code changes by their future defect density. We used the normalized Kendall tau rank distance to measure the agreement of rankings; essentially, this counts the number of pairwise disagreements between two ranking lists and lies between 0 and 1 [74]. Before analyzing the measurements, we applied the complement operation, so that a higher score indicates greater agreement between the two rankings, i.e., more acute risk awareness.

Code review effectiveness. Finding defects has long been considered a primary motivation for investment in code review [1] and the number of defects discovered is a common measurement of the effectiveness of code review performance [37]. Hence, in our study, we also estimated review effectiveness using the proportion of known defects that a participant identified in their assigned change set.

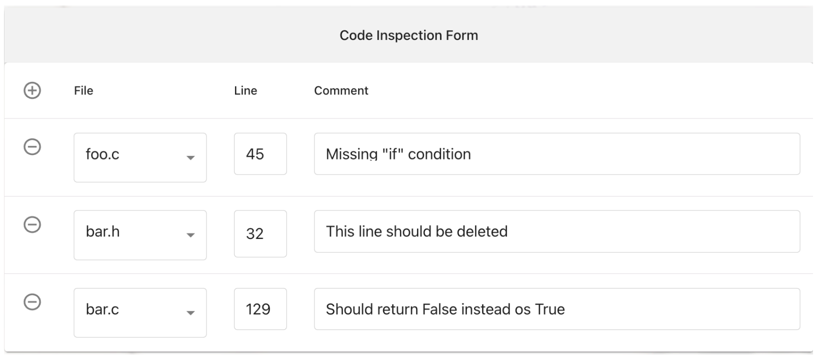
Code review efficiency. Code review efficiency has been defined as the number of defects found per unit of time [37]. In our study, we also estimated review efficiency using the number of detected known defects per unit of time the participant spent reviewing the assigned change set.

4.4.3 Confounding Variables

Apart from the supporting tools, the performance of code review could also be impacted by confounding factors related to the sample change sets and the recruited study participants (see Table 6). To assess their impact, we collected measures that associate with these confounding factors and we studied their correlation with the dependent variables. We selected five change sets for inclusion in our experiment to mitigate bias towards a specific change set. To mitigate the impact of study participant factors (e.g., development experience, code review experience, Java experience), we applied a matching strategy when assigning the change sets, so that for each participant in the control group, we assigned the same change set to a participant with similar development and review experience in the treatment group.

4.5 Experiment Tasks

To address our research questions, we asked participants to complete two code review-related tasks:



Code Inspection Form

	File	Line	Comment
	foo.c	45	Missing "if" condition
	bar.h	32	This line should be deleted
	bar.c	129	Should return False instead os True

Fig. 3: Example of a code inspection form.

Task A (RQ1) — Participants were asked to rank the three code changes in a change set based on their estimated riskiness.

Task B (RQ2, RQ3) — Participants were assigned to review code changes one at a time, with the stated goal of identifying functional defects. Participants recorded the suspected issues in a code inspection form (Figure 3), which facilitated our analysis of the results. To avoid bias, participants were not told how many known defects were present in the assigned code changes; they *were* told that it was possible that the code change contained no defects.

4.6 Experiment Flow

This section describes the flow of our experiment as shown in Figure 1.

4.6.1 Pre-experiment questionnaire

We first asked participants to complete a pre-experiment questionnaire. This questionnaire aimed to collect information about their development background (e.g., development experience, Java experience, code review experience) and code review preferences (i.e., preferred code review order). We use this information to evaluate the participants' qualification for the study, to group the participants while balancing these confounding factors, and to assign change sets to participants in their preferred order for the experiment.

Before collecting their information, we asked for the informed consent of the participants to use their data during the experiment.

4.6.2 Experiment

We analyzed the participants' responses from the pre-experiment questionnaire to filter out those who did not have prior experience in Java and code review. Next, we assigned the remaining participants to different tooling support groups. As described in Section 4.4.3, we controlled for the participants' development backgrounds (i.e., development experience, Java experience, code review experience, etc.) during the assignment to maintain a balanced distribution of these factors. The participants were then provided with a URL and a unique ID to access the web application and initiate the experiment. The procedure of the experiment in the application is presented as follows.

Welcome Page. The application starts with a welcome page introducing general information about the experiment and the estimated duration. The assigned tasks are explained and for those participants in the treatment group, the Gherald tool is explained.

Practice Task. To mitigate learning effects and reduce the impact that a lack of familiarity with the tasks or available tools has on the participants' performance, participants were assigned a practice task before their assigned task is shown. Participants were informed that this was a practice task to familiarize themselves with the interface and the type of tasks that they can expect during the live experiment.

Experiment Tasks. Once the participants completed the practice task, the live experiment began with the assigned tasks (see Section 4.5). A timer started after the task page had loaded. Participants were permitted to pause the timer at any time during the task if their work was interrupted.

4.6.3 Post-experiment Questionnaire

After the participants had completed the assigned tasks in the experiment application, they were prompted to complete a post-experiment questionnaire, which asked them to share their perceptions about the experiment (e.g., fitness and understandability).

4.7 Pilot Study

Before releasing the experiment to the public, we conducted a pilot study with 20 graduate students in computer science to identify problems with the experiment before recruiting a larger pool of participants.

We measured the estimated time for task completion to ensure that the experiment could be completed within a reasonable amount of time (i.e., 30–60 minutes). We also conducted casual post-experiment conversations with the participants to evaluate the perceived difficulty of the provided changes and to gather feedback on the experiment. From their feedback, it became apparent that three of the code changes were more challenging and required a deeper understanding of specific aspects of Java, particularly relating to hexadecimal and octal numbers. As a result, we decided to assign these changes only to participants with at least one year of experience in Java development. Furthermore, some participants found it difficult to grasp the Gherald risk assessment results. To address this, we improved the presentation of risk scores and adjusted the text explanations of the risk measures. We also enhanced the overall design of the experiment and the user interface based on the feedback to improve the participant experience.

The data collected from the pilot study were excluded from the results of the study.

4.8 Participants

We calculated the expected sample size for the study through the statistical power analysis proposed by Cohen [75], which is commonly used to determine the required sample size to verify the hypothesis with specified statistical power, significance criterion (α), and effect size. Although a smaller effect size indicates a greater opportunity to find the significant difference between the studied groups, it requires a larger number of participants, creating practical recruitment challenges for this study. Hence, we use the standard settings for uncovering a medium effect size in the context of applying a Mann-Whitney U test (unpaired, one-tailed, $\alpha = 0.05$, power = 0.8, $d = 0.5$). The estimated sample size is 106 participants, with 53 participants per group.

We sent out our recruitment invitation to graduate students at the University of Waterloo and posted the recruitment message on Java developer forums (e.g., Reddit) and social media platforms (e.g., Facebook). We targeted individuals who had experience with both code reviews and Java development. In appreciation of their time commitment, we provided each participant with $15 CAD as a token of our appreciation for their time.

In total, 161 participants sign up for the study.

4.9 Data Analysis

We first applied a list of filter steps to clean the data and remove anomalies. To ensure that the participants perform completed code reviews, we ignored participants who did not finish the assigned reviews and skipped the tasks. We also analyzed participants' activity logs and filtered out those who did not conduct reviews of the assigned changes (e.g., reported no defect without viewing the code diff). In addition, we filtered out the participants who declared in the post-experiment questionnaire that they did not fully understand the tasks or code changes and therefore had likely performed only superficial code reviews. We also filtered out time-based outliers, i.e., those who took a very long or very short time to perform the code reviews.

Next, we measured the value of dependent variables from the experimental data. As described in Section 4.4.2, the risk awareness of participants was measured by computing the agreement between the participant's ranking in Task A and the ranking of defect density of the examined code changes.

To measure code review effectiveness, we manually examined the defects reported by the participants and measured the proportion of known defects identified. It is possible that a participant could report a valid defect that has not been previously identified by the original developer. However, to prevent potential bias, we considered only "known" (i.e., previously identified) defects. This decision is justified for two reasons. First, the participants' inspection behaviors can vary widely, with some individuals reporting only functional defects that they are certain of, while others may report any types of defects they observe as many as they can. Also, assessing the validity of the new defects identified by the participants requires a manual interpretation of the author, which may introduce bias to the result. As such, new defects reported by the participants were excluded from the analysis and only known defects were compared with the participants' responses to measure code review effectiveness.

Similarly, for the measurement of code review efficiency, we considered only known defects that were identified by the participants. Then, we computed the ratio of the total number of identified known defects over the total time in hours that the participant spent on both task A and task B. We used the total time for both tasks to mitigate potential bias in the results. Due to different understanding of the task requirements and diverse code review habits, some participants may invest significant time inspecting the code during task A, which may result in a reduced review time for task B. Our observation of experimental data further confirms this assumption.

With the values of the dependent variables measured, we applied visualization techniques (e.g., bean plot) to present the descriptive statistics, and then applied the Shapiro–Wilk test to statistically examine the normality of the data. Also, we applied Spearman's rank correlation test to measure the pairwise correlations between the examined variables.

To address the research questions, for each dependent variable, we performed a one-tailed Mann-Whitney U test to identify whether there existed a significant difference between the results in the treatment group and the control group. We then applied effect-size measures (i.e., Cliff's Delta) to estimate the magnitude of difference between the groups if a significant difference is found.

5 Study Results

In this section, we present the results of our study. Before discussing the analysis for each research question, we briefly describe the general results regarding the participants and preliminary analysis.

A total of 161 participants signed up for the study, of which 90 completed the experiment. Upon conducting a check to filter out invalid participants as described in Section 4.9, the sample was reduced to 48 participants with valid experiment data. Table 7 provides an overview of the distribution of participants among change sets and groups. Overall, there were 26 participants in the treatment group and 22 in the control group. The participants were as evenly distributed among the two groups as was possible, and this balance was consistently maintained across each change set. Moreover, the difference in the participant experience, fitness during the experiment, and understandability of the code changes between the two groups were not statistically significant.

Figure 4 presents the Spearman pairwise correlations among the dependent and control variables introduced in Table 6. A significant strong correlation existed between only code review effectiveness and efficiency ($r = 0.94$), which is reasonable due to the computation of code review efficiency (i.e., number of known defects identified per unit of time). Moreover, we observed a positive relationship between devel-

Table 7: Distribution of participants among change sets and groups.

	Change set					**Total**
	1	**2**	**3**	**4**	**5**	
Gherald	8	9	4	3	2	26
No tool	8	7	2	4	1	22
Total	16	16	6	7	3	48

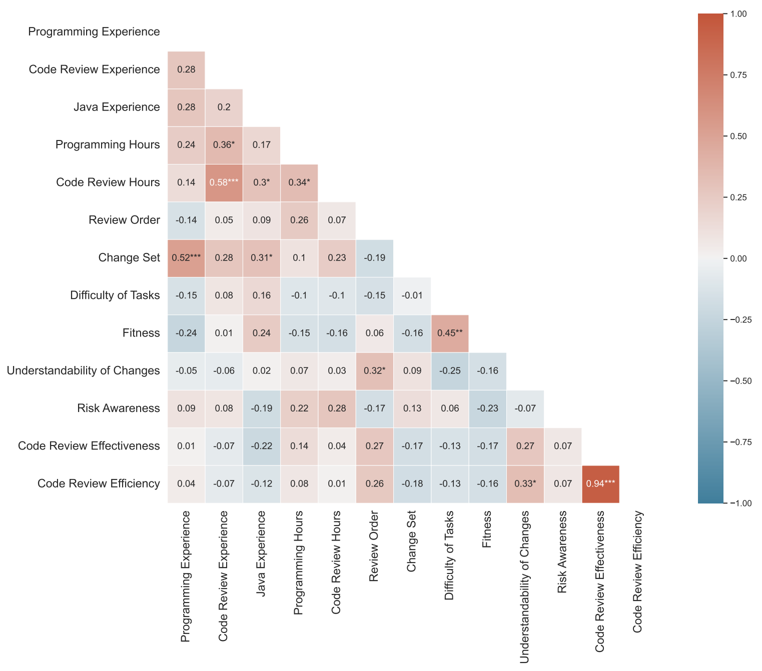


Fig. 4: Correlations among examined variables. Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

opers' understandability of changes and their code review efficiency ($r = 0.33$), which indicates, as expected, developers with a better understanding of changes spent less time identifying the known defects. No statistically significant strong correlation was found between the dependent variables and the remaining control variables.

Below, we present the results with respect to each research question.

(RQ1) Is the use of Gherald associated with greater awareness of the riskiness of changes?

The participants in the treatment group exhibited a greater awareness of the riskiness of code changes compared to participants in the control group. As shown in Table 8a,

Table 8: Descriptives of risk awareness, code review effectiveness, and code review efficiency in different groups and change sets.

Change set	**Gherald**		**No tool**	
	Median	**Mean**	**Median**	**Mean**
1	0.67	0.67	0.33	0.38
2	1.0	0.89	0.67	0.71
3	1.0	0.75	0.83	0.83
4	0.67	0.78	0.5	0.5
5	0.67	0.67	0.0	0.0
Total	1.0	0.77	0.67	0.53

(a) Risk awareness (RQ1)

Change set	**Gherald**		**No tool**	
	Median	**Mean**	**Median**	**Mean**
1	0.5	0.38	0.0	0.13
2	0.5	0.33	0.0	0.14
3	0.25	0.38	0.0	0.0
4	0.0	0.11	0.0	0.17
5	0.0	0.0	0.0	0.0
Total	0.17	0.30	0.0	0.12

(b) Code review effectiveness (RQ2)

Change set	**Gherald**		**No tool**	
	Median	**Mean**	**Median**	**Mean**
1	2.10	1.99	0.0	0.79
2	1.37	1.27	0.0	1.00
3	1.18	1.36	0.0	0.0
4	0.0	1.02	0.0	0.59
5	0.0	0.0	0.0	0.0
Total	0.69	1.38	0.0	0.71

(c) Code review efficiency (RQ3)

the median and mean risk awareness of participants in the treatment group were 1.0 and 0.77, respectively, which were higher than those in the control group (0.67 and 0.53, respectively). This observation applies to each change set except for change set 3. Although the median risk awareness in the treatment group was higher, participants with no tool support have a higher mean risk awareness than those with the assistance of Gherald.

This fact is also evident in Figure 5. For change set 3, the distribution of risk awareness for the control group showed a denser concentration of values around 0.83. On the other hand, for the treatment group, there was more density of values around 1. However, the distribution of data was more dispersed and less dense, which suggests that a low outlier may be overly influential in computing the average risk awareness

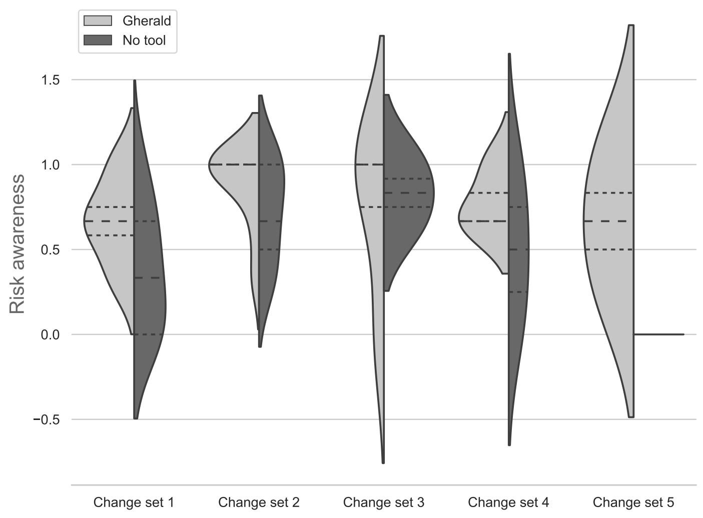


Fig. 5: Comparison of risk awareness between the treatment and control group across five change sets. The long (short) dash line represents the median (Q1/Q3) values.

score of the group. Upon closer inspection, we found that there was a participant in the treatment group with a risk awareness score of 0, indicating that the rankings provided by this individual are the exact inversion of the rankings based on defect density. A follow-up discussion with this participant revealed that they conducted the risk evaluation without accessing the Gherald results. Figure 5 shows that there is clearly a greater risk awareness for participants in the treatment group for all other change sets, suggesting that this outlier was a singular case.

We performed a one-tailed Mann-Whitney U test to determine whether the risk awareness of those in the treatment group was larger than those in the control group to a statistically significant degree The result was a p-value of 0.012, which indicates that the null hypothesis — that the risk awareness of both groups is sampled from the same distribution — can be rejected. We then used Cliff's delta to measure the effect size of the difference. The delta was 0.36, indicating a "medium" difference in risk awareness between the groups [76].

The use of Gherald was associated with improvements in developer awareness of the riskiness of code changes. In our experiment, the risk ranking provided by participants with the assistance of Gherald was more closely aligned with the actual defect density of the code changes.

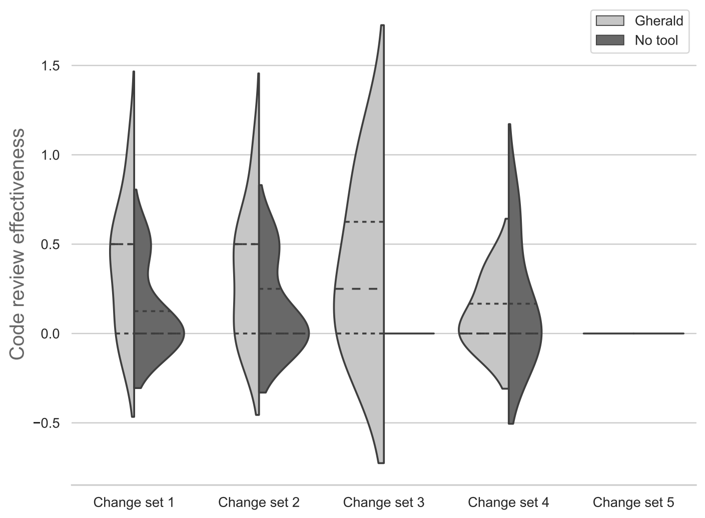


Fig. 6: Comparison of code review effectiveness between the treatment and control group across five change sets. The long (short) dash line represents the median (Q1/Q3) values.

Table 9: Number of participants identified the known defects in different groups and change sets.

	Change set					**Total**
	1	**2**	**3**	**4**	**5**	
Gherald	5	5	2	1	0	13
No tool	2	2	0	1	0	5
Total	7	7	2	2	0	18

(RQ2) Is the use of Gherald associated with an improvement in code review effectiveness?

The participants provided with Gherald had higher code review effectiveness compared to participants without tooling support. As shown in Table 8b, participants in the treatment group exhibited a median and mean code review effectiveness of 0.17 and 0.3, respectively, which exceeds the values from the control group, i.e., 0 and 0.12, respectively. For change sets 1, 2, and 3, participants with the assistance of Gherald achieved higher median and mean code review effectiveness. This pattern is also evident from the data and quartile distribution depicted in Figure 6, where a noticeable difference could be observed for change sets 1, 2, and 3.

Table 9 displays the number of participants from different groups that correctly identified the known defects for each change set. Out of the 48 valid participants, 18 were able to identify at least one known defect, and three were able to identify all

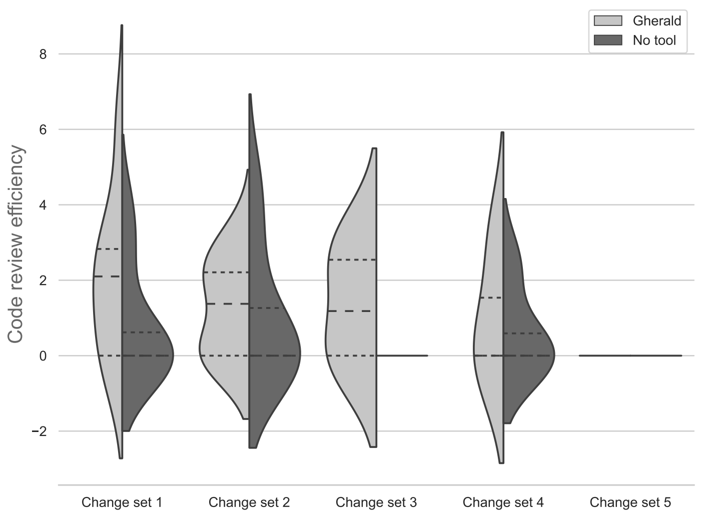


Fig. 7: Comparison of code review efficiency between the treatment and control group across five change sets. The long (short) dash line represents the median (Q1/Q3) values.

known defects. For each of the change sets 1 and 2, seven out of sixteen participants were able to identify at least one known defect, respectively. However, for change sets 3, 4, and 5, only two, two, and zero participants, respectively, were able to correctly identify the known defects. We believe that these results occurred because the defects in change sets 3, 4, 5 were more complex and required recognizing an edge case with respect to hexadecimal and octal numbers. Additionally, the sample size for these changes was small. Nevertheless, although only a small proportion of participants identified the known defects, from Table 9, we can still observe that Gherald participants outperformed the control group in terms of defect detection.

The Mann-Whitney U test results indicate that the null hypothesis can be rejected, i.e., there is a statistically significant improvement of code review effectiveness associated with Gherald ($p = 0.03$). The Cliff's delta effect size is 0.27, indicating a "small" difference in participants' code review effectiveness between the groups [76].

The use of Gherald was associated with an improvement in code review effectiveness. In our experiment, a larger proportion of known defects were identified by participants with the assistance of Gherald.

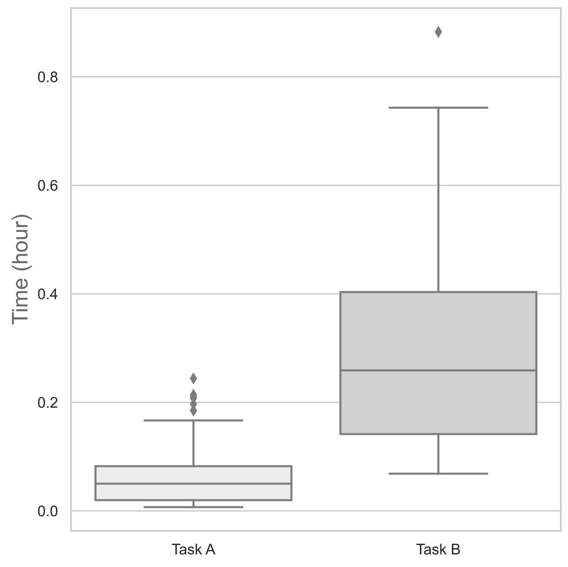


Fig. 8: The completion time for experiment tasks.

(RQ3) Is the use of Gherald associated with an improvement in code review efficiency?

The participants provided with Gherald had higher code review efficiency compared to participants without tooling support. As shown in Table 8c and Figure 7, the treatment group displayed a higher median and mean code review efficiency, with 0.69 and 1.38 known defects identified per hour, respectively. Notably, we observed a significant increase in code review efficiency associated with Gherald in most change sets, except for change set 5 with a code review efficiency of zero as none of the participants identified the known defects.

Figure 8 depicts the task completion time of participants for each task. On average, participants spent 4.2 minutes on task A and 18 minutes on task B, which is consistent with our anticipated experiment duration of 30 minutes. The time taken to complete task A ranges from 36 seconds to 14.4 minutes, while the completion time for task B ranges from 4.2 minutes to 52.8 minutes. Interestingly, we observed that some participants invested a long time on task A, while allocating a comparable amount of time to task B. This pattern of behavior indicates that they may have devoted some review effort to conducting code inspection during task A. This finding supports our decision to calculate the code review time by summing the time spent on both tasks.

The Mann-Whitney U test yields a p-value of 0.0503, which is not less than our pre-established confidence level (0.05). Therefore, we are unable to reject the null hypothesis, which indicates that there is insufficient evidence to suggest the use of Gherald is associated with an improvement in code review efficiency.

In our experiment, participants in the treatment group had higher code review efficiency compared to those in the control group. However, the difference between the two groups is not statistically significant, so we cannot draw a conclusion that the use of Gherald is associated with an improvement in code review efficiency.

6 Discussion & Practical Implications

In this study, we found that the use of Gherald had a strong impact on the developer awareness, effectiveness, and efficiency during code review processes. The study demonstrated that the introduction of risk assessment tools significantly enhanced developers' awareness of code change riskiness and improved code review effectiveness. It implies that developers, especially those with limited project-specific experience, might not always have the necessary information to evaluate the potential risks associated with a code change. The risk assessment tool fills this gap effectively by providing them information about author experience, change history, and past defects, leading to better risk assessment, more thorough evaluations to the risky areas, and more effective issue identification. Although the study did not establish statistical significance in code review efficiency, it observed higher mean and median efficiency for participants using Gherald. This further suggests there is potential for efficiency gains, which can be particularly valuable in fast-paced development environments.

These findings also have practical implications for software development teams. Integrating a risk assessment tool like Gherald into the development workflow can greatly enhance the quality and security of the codebase. Organizations can consider the integration of risk assessment tools like Gherald into their existing development workflows, making them easily accessible to developers. For instance, this tool can be linked to the version control system to automatically analyze incoming code changes and inform developers of the defect-prone areas in the history. It can also be integrated with continuous integration tools to provide instant feedback on the risk level of new code changes. Additionally, connecting this tool to code review systems can help reviewers to prioritize high-risk changes and identify potential defects more effectively. By combining risk assessment tools like Gherald with these existing practices, developers can experience a more streamlined and efficient workflow that naturally incorporates risk assessment as a crucial part of the development process.

Consider a real-world scenario where a risk assessment tool like Gherald is integrated into an open-source GitHub repository. Due to the nature of open-source repositories, there is frequent developer turnover and many contributors are unfamiliar with each other. When a new developer submits their code for review, the tool analyzes the historical data and informs the developer of the files and methods that have historically been prone to defects. This information empowers the developer to proactively make necessary adjustments during the development phase, improving the quality of the code before it is sent to the reviewers. As the code change moves through the review process, the tool continues to play a role and provides the reviewers with valuable context. Armed with the knowledge of a code change's historical context and the author's risk assessment information, reviewers are better equipped to conduct a thorough evaluation of the code changes that pose a greater risk, identify potential defects, and make more informed decisions about where to invest their effort. This will, in turn, reduce the chances of introducing defects into the codebase. By employing a risk assessment tool like Gherald in such a scenario, software development teams can maintain the security and stability of their codebase, even during rapid development cycles or when collaborating on open-source repositories.

7 Threats To Validity

7.1 Construct Validity

The results of our study may be impacted by the accuracy of risk assessment tools. We choose to focus on whether an accurate risk signal would help developers, and leave the analysis of noise in the risk assessment signal to future work. To control for that noise in our study, for each change set, we selected changes with defect density values that are aligned with Gherald assessment.

Limitation of the SZZ algorithm accuracy in locating fix-inducing changes may result in false positives and false negatives, which may lead to incorrect historical records about defects. To mitigate noise in our labels of fix-inducing changes, we followed McIntosh and Kamei's approach [66] to apply a series of filtering steps and manual verification to the initially produced label set.

7.2 Internal Validity

The participants recruited for the experiment were outsiders — not developers or reviewers of the projects under study. Thus, their behavior might differ from that of the actual code reviewers. To mitigate this threat, we selected code changes that do not necessitate additional contextual learning, which allows us to approximate the actual code review conditions as closely as possible. Nonetheless, it may be safest to interpret our findings as reflecting the newcomer experience.

Some participants encountered technical issues during the experiment, such as unresponsive buttons, application crashes, and a laggy user interface. These technical problems could potentially impact the participants' concentration, mood, and overall sense of disorientation when engaging in the experiment tasks. Furthermore, they may have led to inaccuracies in tracking and logging task times and responses. Upon a closer examination of these participants' activities and feedback, it was observed that these technical difficulties were not exclusive to a particular experimental group; rather, they were experienced by participants from both groups. More importantly, before analyzing the experiment data, we have filtered out the anomalous data as outlined in Section 4.9. Participants who were considerably affected by technical problems had already been excluded due to the likelihood that such issues would result in incomplete tasks or exceptionally abnormal task completion times. Of the 48 valid participants after this exclusion process, five reported relevant technical issues. Up to this point, we have not been able to pinpoint the exact cause of the problems as we were not able to reproduce them on our end. Our initial assessment suggests that it might be related to the participants' devices and software configuration. Nonetheless, further investigation is necessary to ensure the experiment platform's stability for future studies.

The remuneration that we provided also presents a potential threat to the validity of our study. It is possible that participants in the treatment group might be influenced to perceive Gherald as a useful tool and trust the risk assessment result due to this compensation. To address this potential bias, we deliberately did not inform participants that we were evaluating the tool's effectiveness and assigning them into distinct tooling support groups.

7.3 External Validity

The sample of the code changes that we included in our study is small when compared to the history of the studied project. In our study, each participant was shown three code changes. However, due to a limited number of participants, only a small proportion of changes were included in the experiment. To mitigate this threat, we selected code changes for inclusion that impacted different types of functionality and present different types of defects as our sample for experiment.

Although we selected different types of defects, it is still logistically impractical to include all types of defects that may occur in real-world scenarios. To generalize the results, future studies that include other kinds of defects are necessary. We recruited participants from our local student population and from broadcasts on our social networks. As such, our sample of participants may not be entirely representative of the broader software development community.

In addition, to achieve a statistical significant result ($\alpha = 0.05$, power = 0.8, d = 0.5), we aimed to recruit at least 106 participants for our study. However, due to the challenges associated with conducting a human-intensive study, we obtained only 48 valid responses after filtering. Further studies with a larger, more diverse population of developers are needed to confirm our findings and to increase the generalizability of the results.

Generalizability concerns also apply to the selection of projects and programming languages, as we experimented only with code changes from Apache Commons Lang written in Java. To address these limitations, further studies are necessary to verify whether our findings are still valid for more projects with different programming languages.

8 Conclusions

Modern code review is an essential procedure in software development. However, in practice, there are still a large proportion of reviewed changes that introduce bugs into the codebase [3]. This can occur due to a lack of historical context and awareness of the riskiness of the proposed code changes.

In this study, we aimed to investigate whether providing developers with historical context and risk assessment information regarding code changes can enhance their awareness of the riskiness of such changes and result in an improvement in code review effectiveness and efficiency. To accomplish this, we introduced a risk assessment prototype called Gherald, which analyzes the riskiness of code changes based on historical data. We conducted a controlled experiment with 48 participants assigned to two groups, with or without Gherald. We investigated whether the use of Gherald is associated with greater risk awareness and an improvement of code review performance of the participants.

Through the experiment with 48 participants, we found that Gherald has a positive impact on the code review practice:

- The use of Gherald was associated with a statistically significant improvement in developer awareness of the riskiness of code changes.
- The use of Gherald was associated with a statistically significant improvement in code review effectiveness.
- Although the difference in code review efficiency between the treatment and control groups was not statistically significant, in our experiment, we observed a higher mean and median code review efficiency for participants with the assistance of Gherald.

Future Work. As existing risk assessment approaches do not provide sufficient contextual information regarding the riskiness of the code changes, our study introduced a risk assessment prototype and found that its use had a positive impact on code review practices. Future work may propose risk assessment approaches that provide more precise defect proneness prediction with explainable risk indicators for code changes.

Furthermore, while we sampled Java code changes from Apache Commons Lang and conduct one-time code review tasks with participants recruited from various sources, future studies could benefit from a more project-specific approach involving the actual developers and reviewers of the repository. With the risk assessment tool embedded into the code review system, the risk awareness and code review performance of the reviewers can be evaluated over a long-term to explore whether risk assessment has a positive and sustainable effect on code review practices.

9 Declarations

Funding. This study was funded by Waterloo-Huawei Joint Innovation Lab.

Conflict of Interest. The authors declared that they have no conflict of interest.

Ethics Approval. This study was reviewed by and received ethics clearance from the University of Waterloo Research Ethics Committee (ORE #44022).

Data Availability. To facilitate reproduction and foment further research on the field, we make a replication package publicly available.[6] We also publish Gherald as a Python Package on `pip`. The source code is available online on our public GitHub repository.[7]

Acknowledgments. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Huawei and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of Huawei's products.

References

1. A. Bacchelli, C. Bird, 2013 35th International Conference on Software Engineering (ICSE) (2013). DOI 10.1109/icse.2013.6606617
2. C. Sadowski, E. Söderberg, L. Church, M. Sipko, A. Bacchelli, in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)* (2018), pp. 181–190
3. O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, M.W. Godfrey, in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2015), pp. 111–120. DOI 10.1109/ICSM.2015.7332457
4. P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015), pp. 168–179. DOI 10.1109/MSR.2015.23
5. A. Janes, M. Mairegger, B. Russo, in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2018), ASE 2018, p. 876–879. DOI 10.1145/3238147.3240488. URL `https://doi.org/10.1145/3238147.3240488`
6. M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, M. Harman, in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 95–105. DOI 10.1109/ASE.2017.8115622
7. Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, IEEE Transactions on Software Engineering **39**(6), 757–773 (2013). DOI 10.1109/tse.2012.70

[6] `https://doi.org/10.5281/zenodo.7838135`
[7] `https://github.com/filipe-cogo/gherald`

8. T. Fukushima, Y. Kamei, S. Mcintosh, K. Yamashita, N. Ubayashi, Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014 (2014). DOI 10.1145/2597073.2597075
9. T. Hoang, H.K. Dam, Y. Kamei, D. Lo, N. Ubayashi, 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) (2019). DOI 10.1109/msr.2019.00016
10. T. Hoang, H.J. Kang, D. Lo, J. Lawall, Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (2020). DOI 10.1145/3377811.3380361
11. S. Kim, E.J. Whitehead, Y. Zhang, IEEE Transactions on Software Engineering **34**(2), 181–196 (2008). DOI 10.1109/tse.2007.70773
12. C. Pornprasit, C.K. Tantithamthavorn, 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (2021). DOI 10.1109/msr52588.2021.00049
13. C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, E.J. Whitehead, in *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 372–381. DOI 10.1109/ICSE.2013.6606583
14. C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, Commun. ACM **61**(4), 58–66 (2018). DOI 10.1145/3188720. URL `https://doi.org/10.1145/3188720`
15. E. Fregnan, Assessing review outcomes and cognitive factors to improve code review. Ph.D. thesis (2023)
16. C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, C. Winter, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1 (2015), vol. 1, pp. 598–608. DOI 10.1109/ICSE.2015.76
17. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, A. Ustuner, in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (Association for Computing Machinery, New York, NY, USA, 2006), EuroSys '06, p. 73–85. DOI 10.1145/1217935.1217943. URL `https://doi.org/10.1145/1217935.1217943`
18. N. Ayewah, W. Pugh, in *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Association for Computing Machinery, New York, NY, USA, 2010), ISSTA '10, p. 241–252. DOI 10.1145/1831708.1831738. URL `https://doi.org/10.1145/1831708.1831738`
19. N. Rutar, C. Almazan, J. Foster, in *15th International Symposium on Software Reliability Engineering* (2004), pp. 245–256. DOI 10.1109/ISSRE.2004.1
20. D. Hovemeyer, W. Pugh, SIGPLAN Not. **39**(12), 92–106 (2004). DOI 10.1145/1052883.1052895. URL `https://doi.org/10.1145/1052883.1052895`
21. T. Baum, O. Liskin, K. Niklas, K. Schneider, in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2016), FSE 2016, p. 85–96
22. J. Hampton, in *Laboratory Psychology* (Psychology Press, 2018), pp. 15–37
23. A. Ackerman, L. Buchwald, F. Lewski, IEEE Software **6**(3), 31 (1989). DOI 10.1109/52.28121

24. A.F. Ackerman, P.J. Fowler, R.G. Ebenau, in *Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives* (Elsevier North-Holland, Inc., USA, 1984), p. 13–40
25. M.E. Fagan, IBM Systems Journal **15**(3), 182 (1976). DOI 10.1147/sj.153.0182
26. S. Kollanus, J. Koskinen, The Open Software Engineering Journal **3**(1), 15–34 (2009). DOI 10.2174/1874107x00903010015
27. F. Shull, C. Seaman, IEEE Software **25**(1), 88 (2008). DOI 10.1109/MS.2008.7
28. D.G. Feitelson, E. Frachtenberg, K.L. Beck, IEEE Internet Computing **17**(4), 8 (2013). DOI 10.1109/MIC.2013.25
29. A. Alami, M. Leavitt Cohn, A. Wasowski, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (2019). DOI 10.1109/icse.2019.00111
30. L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, A. Bacchelli, Proc. ACM Hum.-Comput. Interact. **2**(CSCW) (2018). DOI 10.1145/3274404. URL `https://doi.org/10.1145/3274404`
31. A. Bosu, M. Greiler, C. Bird, in *Proceedings of the 12th Working Conference on Mining Software Repositories* (IEEE Press, 2015), MSR '15, p. 146–156
32. S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, in *Proc. of the Working Conference on Mining Software Repositories (MSR)* (2014), pp. 192–201
33. O. Kononenko, O. Baysal, M.W. Godfrey, in *Proceedings of the 38th International Conference on Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2016), ICSE '16, p. 1028–1038. DOI 10.1145/2884781.2884840. URL `https://doi.org/10.1145/2884781.2884840`
34. O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, Empir. Softw. Eng. **21**(3), 932 (2016). DOI 10.1007/s10664-015-9366-8. URL `https://doi.org/10.1007/s10664-015-9366-8`
35. G. Gousios, M. Pinzger, A.v. Deursen, in *Proceedings of the 36th International Conference on Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2014), ICSE 2014, p. 345–355. DOI 10.1145/2568225.2568260. URL `https://doi.org/10.1145/2568225.2568260`
36. Y. Tao, Y. Dang, T. Xie, D. Zhang, S. Kim, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2012), FSE '12. DOI 10.1145/2393596.2393656. URL `https://doi.org/10.1145/2393596.2393656`
37. S. Biffl, in *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000* (2000), pp. 136–145. DOI 10.1109/APSEC.2000.896692
38. Y. Tymchuk, A. Mocci, M. Lanza, in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015), pp. 151–160. DOI 10.1109/SANER.2015.7081825
39. S. Oosterwaal, A.v. Deursen, R. Coelho, A.A. Sawant, A. Bacchelli, in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2016), FSE 2016, p. 1038–1041. DOI 10.1145/2950290.2983929. URL `https://doi.org/10.1145/2950290.2983929`

40. V. Uquillas Gómez, S. Ducasse, T. D'Hondt, Sci. Comput. Program. **98**(P3), 376–393 (2015). DOI 10.1016/j.scico.2013.08.002. URL `https://doi.org/10.1016/j.scico.2013.08.002`
41. L. Gasparini, E. Fregnan, L. Braz, T. Baum, A. Bacchelli, in *2021 Working Conference on Software Visualization (VISSOFT)* (2021), pp. 115–119. DOI 10.1109/VISSOFT52517.2021.00022
42. T. Zhang, M. Song, J. Pinedo, M. Kim, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1 (2015), vol. 1, pp. 111–122. DOI 10.1109/ICSE.2015.33
43. K. Herzig, A. Zeller, in *2013 10th Working Conference on Mining Software Repositories (MSR)* (2013), pp. 121–130. DOI 10.1109/MSR.2013.6624018
44. M. Barnett, C. Bird, J. Brunet, S.K. Lahiri, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1 (2015), vol. 1, pp. 134–144. DOI 10.1109/ICSE.2015.35
45. Y. Tao, S. Kim, in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015), pp. 180–190. DOI 10.1109/MSR.2015.24
46. T. Baum, K. Schneider, A. Bacchelli, Empirical Software Engineering **24**(4), 1762–1798 (2019). DOI 10.1007/s10664-018-9676-8
47. F. Paas, J. Tuovinen, H. Tabbers, P. Van Gerven, Educational Psychologist - EDUC PSYCHOL **38**, 63 (2003). DOI 10.1207/S15326985EP3801_8
48. P.W. Gonçalves, E. Fregnan, T. Baum, K. Schneider, A. Bacchelli, Empirical Softw. Engg. **27**(4) (2022). DOI 10.1007/s10664-022-10123-8. URL `https://doi.org/10.1007/s10664-022-10123-8`
49. T. Baum, K. Schneider, A. Bacchelli, in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), pp. 329–340. DOI 10.1109/ICSME.2017.28
50. E. Fregnan, L. Braz, M. D'Ambros, G. Çalıklı, A. Bacchelli, in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2022), ESEC/FSE 2022, p. 483–494. DOI 10.1145/3540250.3549177. URL `https://doi.org/10.1145/3540250.3549177`
51. P.L. Li, J. Herbsleb, M. Shaw, B. Robinson, in *Proceedings of the 28th International Conference on Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2006), ICSE '06, p. 413–422. DOI 10.1145/1134285.1134343. URL `https://doi.org/10.1145/1134285.1134343`
52. T. Zimmermann, R. Premraj, A. Zeller, in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)* (2007), pp. 9–9. DOI 10.1109/PROMISE.2007.10
53. Y. Kamei, S. Matsumoto, A. Monden, K.i. Matsumoto, B. Adams, A.E. Hassan, in *2010 IEEE International Conference on Software Maintenance* (2010), pp. 1–10. DOI 10.1109/ICSM.2010.5609530
54. E. Shihab, A.E. Hassan, B. Adams, Z.M. Jiang, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2012), FSE '12. DOI 10.1145/2393596.2393670. URL `https://doi.org/10.1145/`

2393596.2393670
55. A. Mockus, D.M. Weiss, Bell Labs Technical Journal **5**(2), 169 (2000). DOI 10.1002/bltj.2229
56. J. undefinedliwerski, T. Zimmermann, A. Zeller, SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005). DOI 10.1145/1082983.1083147. URL https://doi.org/10.1145/1082983.1083147
57. M. Tan, L. Tan, S. Dara, C. Mayeux, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2 (2015), vol. 2, pp. 99–108. DOI 10.1109/ICSE.2015.139
58. H. Tang, T. Lan, D. Hao, L. Zhang, in *Proceedings of the 7th Asia-Pacific Symposium on Internetware* (Association for Computing Machinery, New York, NY, USA, 2015), Internetware '15, p. 43–51. DOI 10.1145/2875913.2875922. URL https://doi.org/10.1145/2875913.2875922
59. M. Beller, R. Bholanath, S. McIntosh, A. Zaidman, in *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), pp. 470–481
60. D.A. Tomassi, C. Rubio-González, in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), pp. 292–303. DOI 10.1109/ASE51524.2021.9678535
61. B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, in *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 672–681. DOI 10.1109/ICSE.2013.6606613
62. S. Heckman, L. Williams, in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Association for Computing Machinery, New York, NY, USA, 2008), ESEM '08, p. 41–50. DOI 10.1145/1414004.1414013. URL https://doi.org/10.1145/1414004.1414013
63. S. Heckman, L. Williams, in *2009 International Conference on Software Testing Verification and Validation* (2009), pp. 161–170. DOI 10.1109/ICST.2009.45
64. S. Kim, M.D. Ernst, in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2007), ESEC-FSE '07, p. 45–54. DOI 10.1145/1287624.1287633. URL https://doi.org/10.1145/1287624.1287633
65. C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, T. Sunetnanta, in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), pp. 1336–1339
66. S. Mcintosh, Y. Kamei, IEEE Transactions on Software Engineering **44**(5), 412–428 (2018). DOI 10.1109/tse.2017.2693980
67. S. Matsumoto, Y. Kamei, A. Monden, K.i. Matsumoto, M. Nakamura, in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (Association for Computing Machinery, New York, NY, USA, 2010), PROMISE '10. DOI 10.1145/1868328.1868356. URL https://doi.org/10.1145/1868328.1868356

68. T. Graves, A. Karr, J. Marron, H. Siy, IEEE Transactions on Software Engineering **26**(7), 653 (2000). DOI 10.1109/32.859533
69. Z. Wan, X. Xia, A.E. Hassan, D. Lo, J. Yin, X. Yang, IEEE Transactions on Software Engineering **46**(11), 1241 (2020). DOI 10.1109/TSE.2018.2877678
70. D. Spadini, A. Bacchelli, in *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)* (2020), pp. 528–532. DOI 10.1145/3379597.3387455
71. S. Kim, T. Zimmermann, K. Pan, E.J. Jr. Whitehead, in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)* (2006), pp. 81–90. DOI 10.1109/ASE.2006.23
72. J. undefinedliwerski, T. Zimmermann, A. Zeller, (Association for Computing Machinery, New York, NY, USA, 2005), MSR '05, p. 1–5. DOI 10.1145/1083142.1083147. URL `https://doi.org/10.1145/1083142.1083147`
73. C. Rezk, Y. Kamei, S. McIntosh, IEEE Transactions on Software Engineering **48**(9), 3297–3309 (2022)
74. M.G. Kendall, *Rank correlation methods* (Griffin, 1948)
75. J. Cohen, Current directions in psychological science **1**(3), 98 (2013)
76. J. Romano, J.D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, in *Annual Meeting of the Southern Association for Institutional Research* (2006), pp. 1–51