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

by

Xueyao Yu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

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

Acknowledgements

I would like to express my deepest gratitude to my supervisors Prof. Michael Godfrey and Prof. Shane McIntosh for their invaluable guidance, support, and encouragement throughout my entire research process. As someone with no research background, I was initially overwelmed and apprehensive about taking on such a challenge. Their patient mentorship and assistance gave me a great sense of security and comfort, allowing me to develop a newfound passion for research and step out of my comfort zone to push myself further. I feel incredibly fortunate to have had the opportunity to work with them, and I will always be grateful for their invaluable support and assistance.

I am also appreciative of my project collaborator Dr. Filipe Cogo, for his valuable feedback and assistance with my research. Furthermore, I extend my gratitude to my thesis committee members, Prof. Mei Nagappan and Prof. Chengnian Sun, for their insightful feedback and suggestions that helped improve my research and thesis writing.

I would also like to thank those who participated in or helped me recruit participants for my user study. Their contributions were critical to the success of my research, and I am grateful for their time and effort.

I would especially like to thank my loving and supportive parents, whose unconditional love, encouragement and trust have been a constant source of strength throughout my life. Their unwavering support has given me the courage to overcome challenges and pursue my interest. I am forever grateful to them and love them.

Many heartfelt thanks to my friends and colleagues in the SWAG and REBELs group for their support and help. I really enjoy the time spent with all of them.

This experience has been invaluable to me and thanks everyone I have met.

Table of Contents

List of Figures

List of Tables

List of Publications

This thesis builds upon previous work submitted to the journal of Empirical Software Engineering (EMSE) and currently under review.

- Xueyao Yu, Filipe R. Cogo, Shane McIntosh, Michael W. Godfrey. Studying the Impact of Risk Assessment Analytics on Risk Awareness and Code Review Performance. submitted to Empirical Software Engineering (EMSE)

Chapter 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 [4, 42]. However, in practice, there are still a large proportion of reviewed changes that introduce bugs into the codebase [30]. Prior work [50, 23, 37] suggests that this may be mainly due to the lack of reviewer awareness of the riskiness of proposed code changes. Most code review tools provide reviewers with a view of fine-grained textual differences that highlight the proposed changes. However, 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.

Our proposed solution to promote risk awareness during code review is to provide developers with a risk assessment of each code change. In recent years, a plethora of studies has explored how to assess the risk of a code change. Some of the popular approaches are Just-In-Time (JIT) defect prediction [24, 15, 20, 21, 27] and automatic static analysis tools (ASATs) [43, 5, 3, 40, 22].

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 investigation on this topic, we attempted to apply defect prediction and ASATs to help with the code review process and learned the following lessons: first, while defect prediction approaches can identify risky changes effectively [20, 21, 38], they are not helpful in the code review process due to their lack of ability to provide clear reasoning and actionable messages [31]. Moreover, bug detection tools should ideally produce no more than 10% of false positives during code review [41, 13]. However, we applied one of the state-of-the-art JIT defect prediction approaches, JITLine [38], on our studied project and obtained a precision of 0.22, which falls significantly below the required level of performance for the integration of a JIT defect predictor with code review. ASATs, on the other hand, can suggest actionable solutions. However, they detect defects at a too fine-grained level and are inadequate in providing a comprehensive risk assessment of the change.

Therefore, 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 by using Gherald, reviewers can prioritize risky changes and conduct code reviews more efficiently and effectively.

To verify our hypothesis, we conduct a controlled experiment with 48 participants to investigate the impact of using Gherald on developers' risk awareness and code review performance. Similar to prior studies [9, 35, 16], we measure code review performance through the lenses of *effectiveness* (i.e., how many defects developers detect) and *efficiency* (i.e., how quickly developers detect defects). Our study employed a between-subjects experimental design [18], dividing participants into two groups: one with tool assistance from Gherald, and one without. We then compared the performance of the two groups on pre-assigned risk assessment and code review tasks. The goal of our study was to investigate three research questions:

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

We found that the risk rankings provided by participants in the treatment group were more closely aligned with the defect density of the experimental code changes, and that the use of Gherald was associated with statistically significant improvements in developer awareness of the riskiness of code changes.

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

We found that the participants in the treatment group were able to identify a larger proportion of known defects, and that there was a statistically significant improvement in code review effectiveness associated with the use of Gherald.

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

We found that participants in the treatment group had higher code review efficiency than the control group. However, the difference between the treatment and control groups was not statistically significant. Hence, we cannot conclude that the use of Gherald was associated with an improvement in 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.

Chapter 2

Related Work and Motivational Example

In this chapter, 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 [2, 1]. The concept of code review dates back to 1976 when Fagan proposed a structured process called *code inspection* [12], which required an in-person, manual, finely-grained, and process-heavy review of the code in question. In the past decades, the lightweight, tool-based, and asynchronous practice of *modern code review* has gradually replaced traditional code inspections [4]. Modern code review has been increasingly adopted in both industrial and open source projects and inspired numerous related studies [10, 42, 33, 30, 29, 8].

Traditionally, code review performance has been measured by the number of defects found (effectiveness) and the time taken to find them (efficiency) [9]. In recent years, significant research effort has been put on developing techniques to support the code review process and improve its performance. For example, Gonçalves et al. [17] studied whether explicit review strategies — such as using a checklist — improves the overall performance of code review. Zhang et al. proposed a tool that summarizes similar changes and detects potential defects based on the change content and context [51]. 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 [48]. Because of this, several other approaches were proposed to decompose composite changes into groups of cohesive and self-contained changes [19, 6, 49]. Additionally, there has been research examining the role of displayed file order in code review. For example, Baum et al. [7] 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. Also, Fragnan et al. conducted a user study to verify their hypothesis that displayed file order has an impact on the code review performance in defect detection [14]. This thesis builds on the idea that ranking changes by their associated risk will promote 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 various types of metrics to identify defect-prone modules (e.g., changes [28], files [52], subsystems [36]). Recently, the concept of Just-In-Time (JIT) defect prediction has emerged [34], 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. JIT defect prediction has been adopted by many industrial software teams, including Avaya [34], Blackberry [44], and Cisco [46]. Also, several recent studies have sought to improve JIT defect prediction models [15, 20, 21, 27, 38].

Automatic static analysis tools (ASATs) are another popular approach to finding potential defects in code changes and to help assure software quality [47]. 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. ASATs have been widely adopted into the software development process for defect detection and have supported a variety of programming languages and defect patterns [43, 5, 3, 40, 22].

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 existing ASATs, which detect defects at the line level by matching some pre-defined defect patterns, are not able to provide an overall risk assessment to the 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 [46]. 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 [41, 13]. Khanan et al. [26] 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 [13] 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 3.3.

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 changes, on the other hand, contain only minor modifications to the source code, so Alice quickly reviews and approves them. However, it happens that there are underlying risks of concern here. 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, has little experience with the project and has never modified this particular file before. Although Alice is an experienced developer herself, she has only recently joined the team and is unaware of these issues. Consequently, a bug is inadvertently introduced into the codebase as Alice, after only a brief inspection, 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. For example, 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. This type of risk analysis would complement Alice's understanding of the proposed changes and provide her with more contextual information when assessing patches.

It is important to note that such a risk assessment tool would not identify specific mistakes in the code as existing static code analysis tools do. Instead, it would provide contextual information that may be overlooked by reviewers but may nonetheless be critical in assessing the overall risk associated with changes. 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.

Chapter 3

Pre-experiment Data Collection

In this chapter, we discuss the rationale for selecting our studied systems and present our data preparation process and prototype tool for risk assessment. Figure 3.1 provides an overview of our study design.

3.1 Subject Systems/Communities

We perform our study on Apache Commons Lang,[1] which provides helper utility methods for the manipulation of Java core classes. We chose this system for several reasons: First, Apache[2] 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.

[1]https://commons.apache.org/proper/commons-lang/

[2]https://www.apache.org

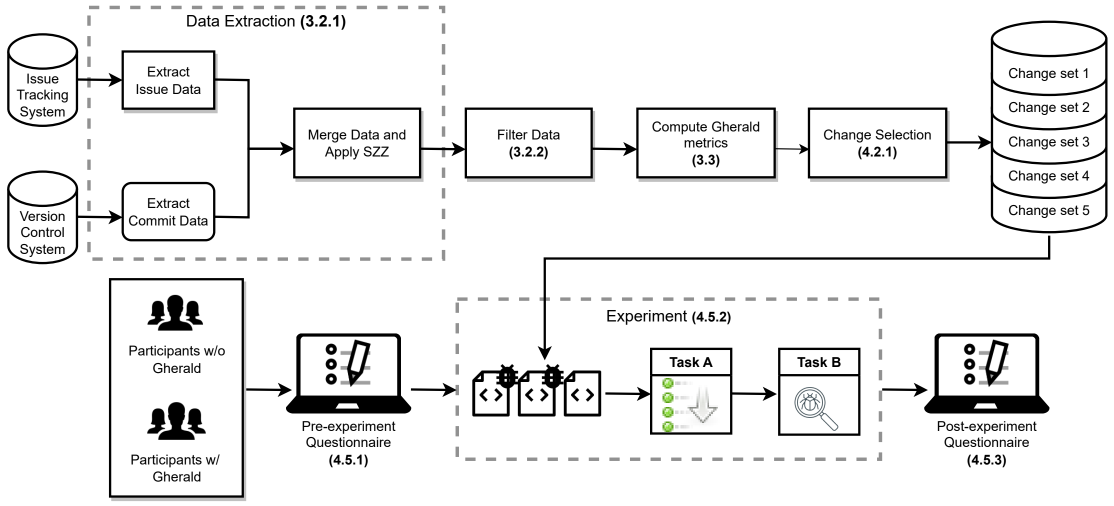


Figure 3.1: An overview of our study

3.2 Data Preparation

3.2.1 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[3] and selected the issues of the type Bug. Next, we extracted the commit data from the git version control system[4] (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 [45] to identify the *fix-inducing changes* — the set of changes that last "touched" the lines that were modified by the defect-fixing changes.

3.2.2 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 [32]. Table 3.1 shows the number of fix-inducing changes after applying each filter sequentially. First, we filtered out changes that update only code comments and whitespace (F_1). Next,

[3]https://issues.apache.org/jira/projects/LANG/issues/
[4]https://github.com/apache/commons-lang/

we removed the fix-inducing changes that were committed after the issue report date (F_2). We also removed the outliers that change at least 10,000 lines (F_{3a}) or at least 100 files (F_{3b}), or add no lines (F_{3c}). Then, we stratified the data into time periods and analyzed the rate of fix-inducing changes. We removed the periods that have unsteady fix-inducing rate (F_4). 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 3.1: 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%

3.3 Risk Assessment — **Gherald**

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

Prior works on JIT defect prediction suggested approaches to measure the risk of a change [24, 15, 20, 21, 27]. We evaluated the potential of incorporating JITLine [38], 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. Empirical study demonstrated that to be deemed suitable for adoption in code review, the bug detection tools need to produce less than 10% false positives [41, 13]. However, on our dataset collected in Section 3.2, JITLine yields a precision of 0.22, which is significantly 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.

Table 3.2: The metrics of risk assessment

	Metrics	Description
Author	Project experience	The ratio of the number of prior changes authored by an actor over the max number of prior changes authored by any actors.
	Recent activity	The ratio of the number of recent changes authored by an actor over the max number of recent changes authored by any actors.
	File expertise	The ratio of file awareness by an actor over the max file awareness by any actors in recent year.
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.

As indicated by prior works [32], `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 significantly 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.

3.3.1 Risk Metrics

We compute a broad range of metrics concerning the three categories (i.e., *author*, *file*, and *method*) as described below. Table 3.2 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. Similar to prior works [34, 32], for each change, we measure the author's (a) prior changes (i.e., the number of prior changes submitted by the author), (b) recent changes (i.e., the number of prior changes weighted by their age, which is measured in years), and (c) file awareness (i.e., the proportion of the prior changes to the modified file submitted by the author). To make the metrics comparable with the other authors, we divided the values by the max value in the six month period prior to the author date of the change to compute 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 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 recent changes and bugs by normalizing the value of the prior changes and prior bugs by their age.

3.3.2 Change Risk Score

With the risk metrics measured, we assessed the riskiness at the change-level; we performed simple calculations to measure the riskiness 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, 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 computde 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 prevent uncomputable values when the denominator is zero, we added one to its value; we also added one to the numerator as a counterbalance. To dampen overly inflated odds ratio induced by a recently added file/method with only few recent changes, we then multiplied the odds ratio by a positive factor $log(RecCng + 2)$. We added two to the file recent change to ensure that the factor is positive.

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.

Chapter 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 [18] by randomly 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).

4.1 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.

Table 4.1: Code changes selected for 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	-
G	`StringUtils`: gets the substring before the first occurrence of a separator	Add a new method	-

4.2 Experimental Artifacts

4.2.1 Code Changes

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.

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. The remaining two changes that did not induce any further fixes were labelled as *Clean 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 4.1.

With experiment code changes selected, we created five change sets, each consisting of three changes with varying Gherald risk scores and defect density. The assignment of code changes to their respective change sets is presented in Table 4.2. 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 these changes to participants with less than one year of development experience or Java experience.

[1]This experiment was reviewed by and received ethics clearance from the University of Waterloo Research Ethics Committee (ORE #44022).

Table 4.2: Five change sets selected for experiment

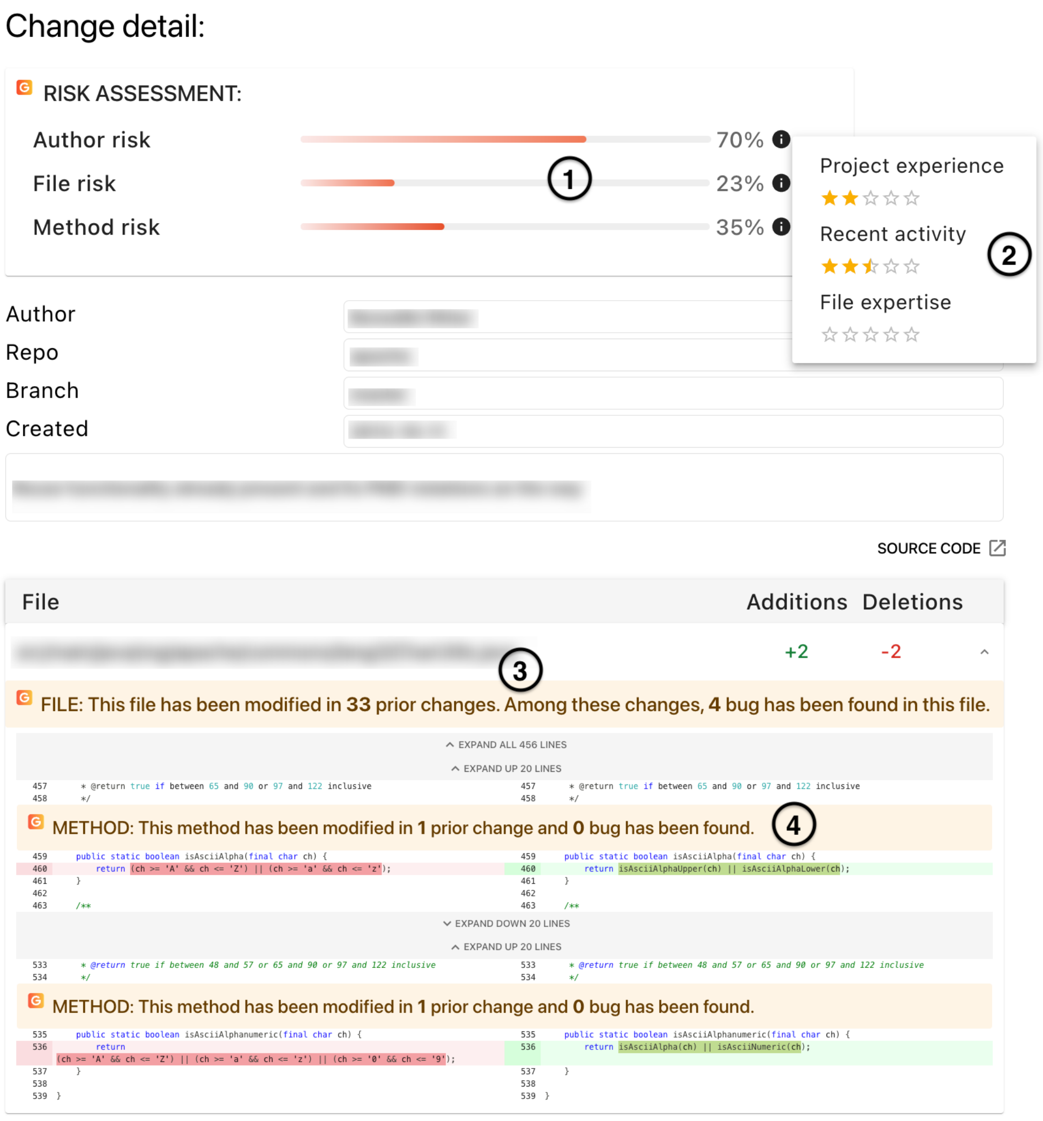
Change set	**Change**	**Defect Count**	**Defect Density**
	A	1	0.17
1	B	1	0.05
	F	0	0.00
	A	1	0.17
2	B	1	0.05
	G	0	0.00
	A	1	0.17
3	C	1	0.06
	F	0	0.00
	A	1	0.17
4	D	2	0.13
	F	0	0.00
	A	1	0.17
5	E	2	0.10
	F	0	0.00

4.2.2 Gherald

Participants in the treatment group were provided with access to Gherald during the experiment. Figure 4.1 shows the experiment platform interface for the treatment group. For each change, an overall risk assessment is presented (Figure 4.1 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 4.1 ②), as well as *file/method change history* and *bug tendency* (Figure 4.1 ③ ④).

4.3 Study Variables

This section discusses the variables that we collected and analyzed.



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
     */
```

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

```
    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
     */
```

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

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

}
```

Figure 4.1: Gherald experiment interface

Table 4.3: 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 4.3.1.
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 4.3.2.
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

4.3.1 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.

4.3.2 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 change sets 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 change sets 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 [25]. 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 [4] and the number of defects discovered is a common measurement of the effectiveness of code review performance [9]. 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 [9]. 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.

4.3.3 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 4.3). 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.

4.4 Experiment Tasks

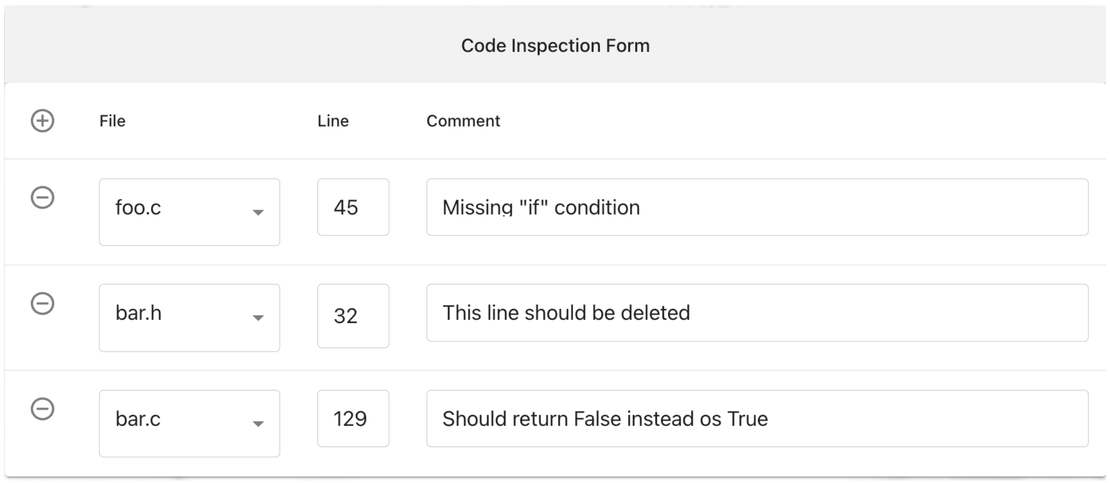
To address our research questions, we asked participants to complete two code review-related tasks:

Task A (RQ1) — Participants were asked to rank three change sets based on their estimated riskiness.

Task B (RQ2, RQ3) — Participants were assigned to review change sets one at a time, with the stated goal of identifying functional defects. Participants recorded the suspected issues in a code inspection form (Figure 4.2), which facilitated our analysis of the results. To avoid bias, participants were not told how many known defects were present in the assigned change sets; they *were* told that it was possible that the change set contained no defects.

4.5 Experiment Flow

This section describes the flow of our experiment as shown in Figure 3.1.


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

Figure 4.2: Example of a code inspection form

4.5.1 Pre-experiment questionnaire

We first asked participants to complete a pre-experiment questionnaire, in which we collected demographic information and code review preferences. Before collecting their information, we asked for the informed consent of the participants to use their data during the experiment.

4.5.2 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. Then, we assigned the remaining participants to different tooling support groups, and we provided them 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 4.4). 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.

4.5.3 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).

4.6 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 and evaluate the difficulty of the provided changes. After post-experiment interviews with these participants, we improved the risk assessment presentation and experiment design based on their feedback. The data collected from the pilot study were excluded from the results of the study.

4.7 Participants

We calculated the expected sample size for the study through the statistical power analysis proposed by Cohen [11], 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.

4.8 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 4.3.2, 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.

Chapter 5

Study Results

In this chapter, 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 4.8, the sample was reduced to 48 participants with valid experiment data. Table 5.1 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 well distributed among the studied change sets. 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.

Table 5.1: 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

Figure 5.1 presents the Spearman pairwise correlations among the dependent and control variables introduced in Table 4.3. A significant strong correlation existed between only code review effectiveness and efficiency ($r = 0.94$), which is reasonable due to the

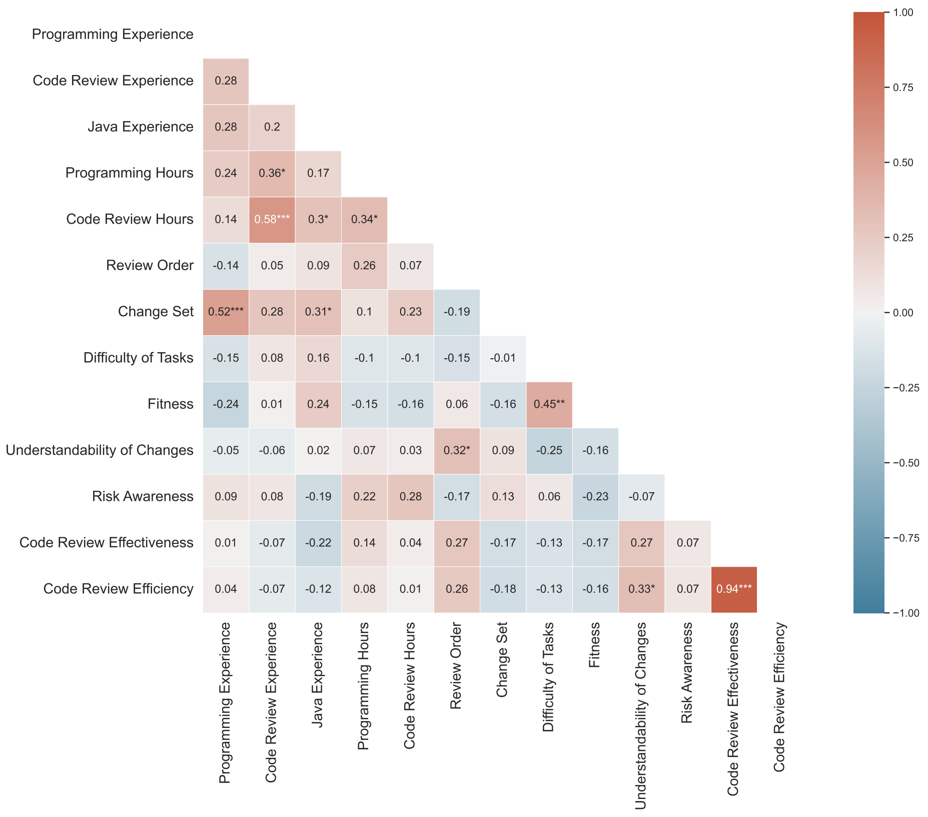


Figure 5.1: Correlations among examined variables. Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

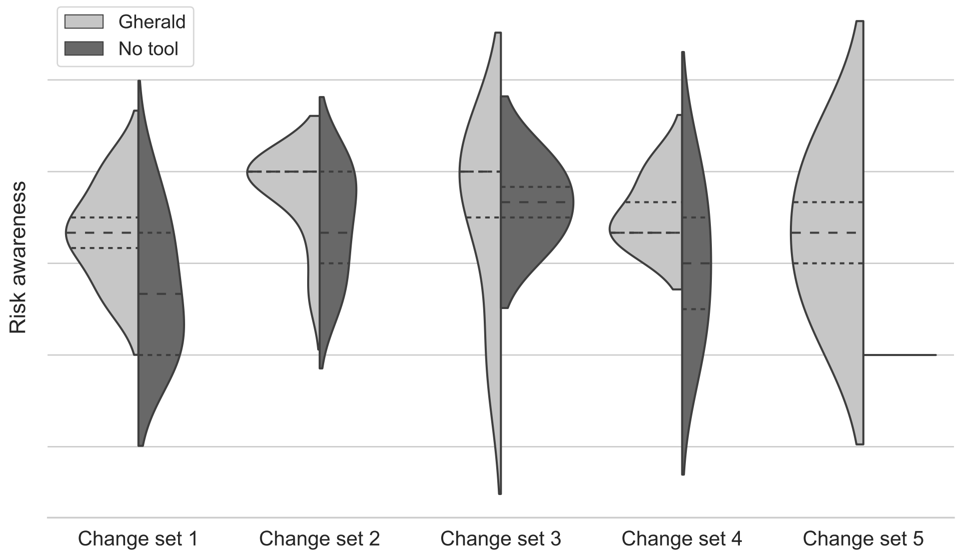


Figure 5.2: Comparison of risk awareness between the treatment and control group across five change sets. The long (short) dash line represents the median (Q1/Q3) values.

computation of code review efficiency (i.e., number of known defects identified per unit of time). Moreover, we observed a positive relationship between developers' 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 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 5.2a, 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

Table 5.2: 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)

a higher mean risk awareness than those with the assistance of Gherald.

This fact is also evident in Figure 5.2. 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 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 5.2 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 [39].

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.

(RQ2) Is 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 5.2b, 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

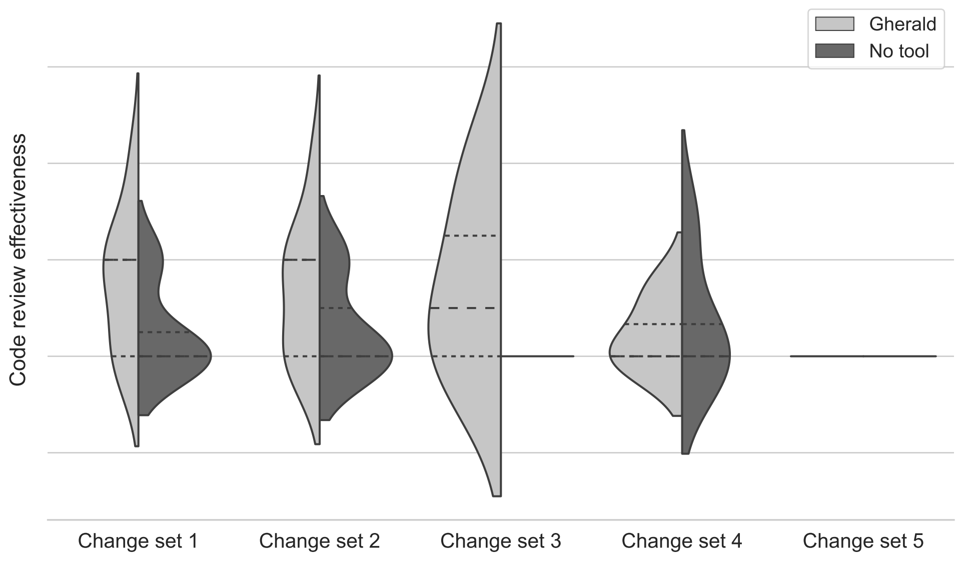


Figure 5.3: Comparison of code review effectiveness between the treatment and control group across five change sets. The long (short) dash line represents the median (Q1/Q3) values.

Table 5.3: 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

distribution depicted in Figure 5.3, where a noticeable difference could be observed for change sets 1, 2, and 3.

Table 5.3 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 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 5.3, 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 [39].

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.

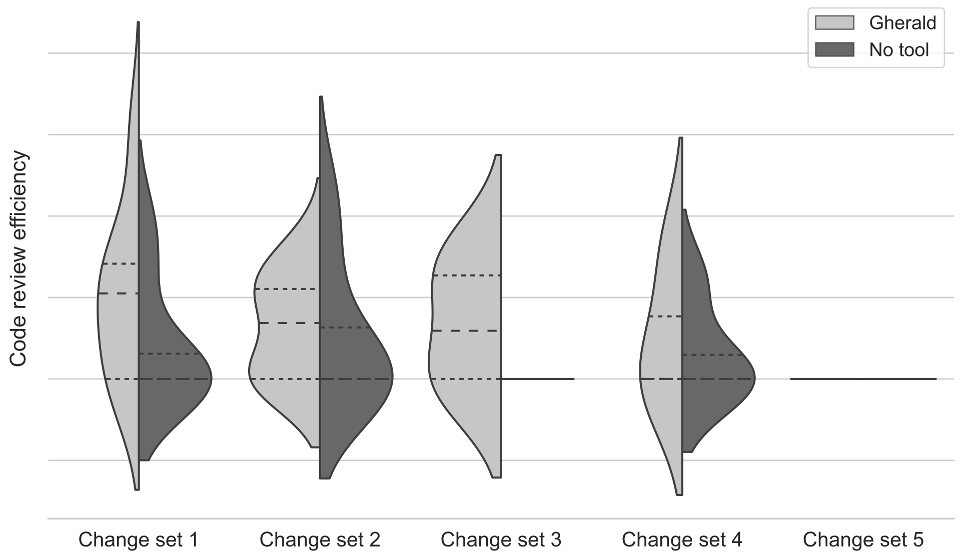


Figure 5.4: Comparison of code review efficiency between the treatment and control group across five change sets. The long (short) dash line represents the median (Q1/Q3) values.

(RQ3) Is 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 5.2c and Figure 5.4, 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 5.5 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.

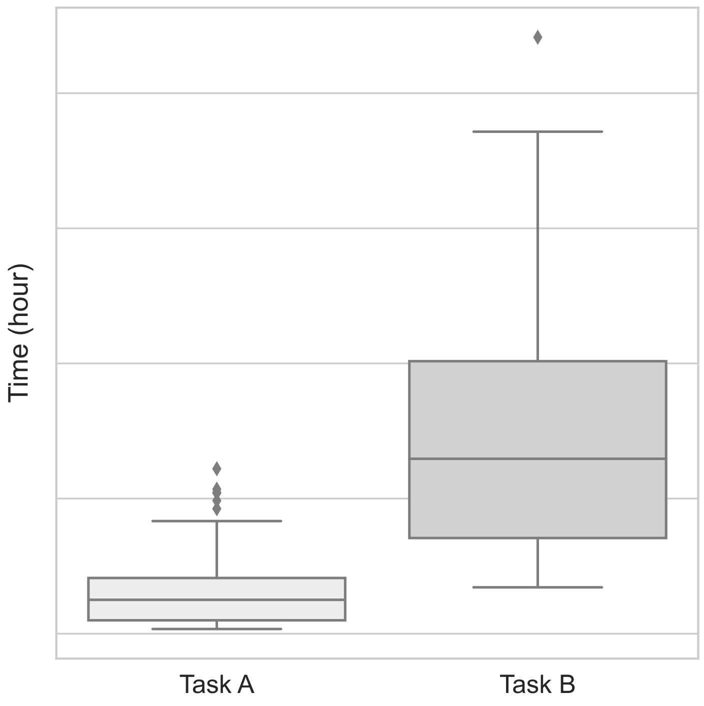


Figure 5.5: The completion time for experiment tasks

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.*

Chapter 6

Threats To Validity

6.1 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 [32] to apply a series of filtering steps and manual verification to the initially produced label set.

6.2 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.

6.3 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.

Chapter 7

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 [30]. 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.

Data Availability. To facilitate reproduction and foment further research on the field, we make a replication package publicly available.[1]

[1]https://doi.org/10.5281/zenodo.7838135

References

[1] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives*, page 13–40, USA, 1984. Elsevier North-Holland, Inc.

[2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, 1989.

[3] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 241–252, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, page 73–85, New York, NY, USA, 2006. Association for Computing Machinery.

[6] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144, 2015.

[7] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the optimal order of reading source code changes for review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 329–340, 2017.

[8] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empir. Softw. Eng.*, 21(3):932–959, 2016.

[9] S. Biffl. Analysis of the impact of reading technique and inspector capability on individual inspection performance. In *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, pages 136–145, 2000.

[10] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 146–156. IEEE Press, 2015.

[11] Jacob Cohen. Statistical power analysis. *Current directions in psychological science*, 1(3):98–101, 2013.

[12] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[13] Enrico Fregnan. *Assessing Review Outcomes and Cognitive Factors to Improve Code Review*. PhD thesis, 2023.

[14] Enrico Fregnan, Larissa Braz, Marco D'Ambros, Gül Çalıklı, and Alberto Bacchelli. First come first served: The impact of file position on code review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 483–494, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Takafumi Fukushima, Yasutaka Kamei, Shane Mcintosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 2014.

[16] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Do explicit review strategies improve code review performance? In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 606–610, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Do explicit review strategies improve code review performance? towards understanding the role of cognitive load. *Empirical Softw. Engg.*, 27(4), jul 2022.

[18] James Hampton. The between-subjects experiment. In *Laboratory Psychology*, pages 15–37. Psychology Press, 2018.

[19] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[20] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019.

[21] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[22] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[23] Andrea Janes, Michael Mairegger, and Barbara Russo. Code_call_lens: Raising the developer awareness of critical code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 876–879, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[25] Maurice G Kendall. *Rank correlation methods.* Griffin, 1948.

[26] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. Jitbot: An explainable just-in-time defect prediction bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1336–1339, 2020.

[27] Sunghun Kim, E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[28] Sunghun Kim, E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[29] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1028–1038, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120, 2015.

[31] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 372–381, 2013.

[32] Shane Mcintosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2018.

[33] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proc. of the Working Conference on Mining Software Repositories (MSR)*, pages 192–201, 2014.

[34] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[35] Yukasa Murakami, Masateru Tsunoda, and Hidetake Uwano. Wap: Does reviewer age affect code review performance? In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 164–169, 2017.

[36] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005.

[37] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. Are developers aware of the architectural impact of their changes? In

2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 95–105, 2017.

[38] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021.

[39] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen's d indices the most appropriate choices? In *Annual Meeting of the Southern Association for Institutional Research*, pages 1–51, 2006.

[40] N. Rutar, C.B. Almazan, and J.S. Foster. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256, 2004.

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

[42] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 181–190, 2018.

[43] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608, 2015.

[44] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[45] D. Spadini and A. Bacchelli. Pydriller: Python framework for mining software repositories. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, pages 528–532, 2020.

[46] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108, 2015.

[47] Hao Tang, Tian Lan, Dan Hao, and Lu Zhang. Enhancing defect prediction with static defect analysis. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, page 43–51, New York, NY, USA, 2015. Association for Computing Machinery.

[48] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 180–190, 2015.

[50] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 168–179, 2015.

[51] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 111–122, 2015.

[52] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9, 2007.