Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack

Eman Abdullah AlOmar
Stevens Institute of Technology
Hoboken, New Jersey, USA
ealomar@stevens.edu

Moataz Chouchen
ETS Montreal, University of Quebec
Montreal, Quebec, Canada
moataz.chouchen.1@ens.etsmtl.ca

Mohamed Wiem Mkaouer
Rochester Institute of Technology
Rochester, New York, USA
mwmvse@rit.edu

Ali Ouni
ETS Montreal, University of Quebec
Montreal, Quebec, Canada
ali.ouni@etsmtl.ca

ABSTRACT

Modern code review is a widely used technique employed in both industrial and open-source projects to improve software quality, share knowledge, and ensure adherence to coding standards and guidelines. During code review, developers may discuss refactoring activities before merging code changes in the code base. To date, code review has been extensively studied to explore its general challenges, best practices and outcomes, and socio-technical aspects. However, little is known about how refactoring is being reviewed and what developers care about when they review refactored code. Hence, in this work, we present a quantitative and qualitative study to understand what are the main criteria developers rely on to develop a decision about accepting or rejecting a submitted refactored code, and what makes this process challenging. Through a case study of 11,010 refactoring and non-refactoring reviews spread across OpenStack open-source projects, we find that refactoring-related code reviews take significantly longer to be resolved in terms of code review efforts. Moreover, upon performing a thematic analysis on a significant sample of the refactoring code review discussions, we built a comprehensive taxonomy consisting of 28 refactoring review criteria. We envision our findings reaffirming the necessity of developing accurate and efficient tools and techniques that can assist developers in the review process in the presence of refactorings.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

refactoring, code review, developer perception, software quality

ACM Reference Format:
Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524842.3527932

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

1 INTRODUCTION

Refactoring is an essential practice to preserve code quality from degradation, as software evolves. Refactoring has grown from the act of cleaning the code, to play a critical role in modern software development. Therefore, refactoring has been attracting various researchers, with over three thousand research papers, according to recent surveys [2]. Another key practice in maintaining software quality is code review [15]. It has become another important to reduce technical debt, and to detect potential coding errors [15, 42, 70]. Code review represents the manual inspection of any newly performed changes to the code, for the purpose of verifying integrity, compliance with standards, and error-freedom [48]. Modern code review is a lightweight tool-based process that heavily relies on discussions between commit authors and reviewers to merge/abandon a given code change [91].

Similarly to any other code change, refactoring changes has to also be reviewed before being merged. That is, if not applied well, refactoring changes can have their side effects such as hindering software quality [8, 37, 38, 62] and inducing bugs [18, 28] making refactoring changes more challenging to review. However, little is known about how reviewers *examine* refactoring related code changes, especially when it is intended to serve the same *purpose* of improving software quality. According to recent industrial case study, AlOmar *et al.* [3] has found that reviewing refactoring related code changes takes a significantly longer time, in comparison with other code changes, demonstrating the need of refactoring review *culture*. Yet, little is known about what criteria reviewers consider when they review refactoring. Most of refactoring studies focus on its automation by recommending refactoring opportunities in the source code [49, 53, 86], or mining performed refactorings in change histories of software repositories [87]. Moreover, the research on code reviews has been focused on automating it by recommending the most appropriate reviewer for a given code change [15]. However, despite the the wide adoption of refactoring in practice, its review process is largely unexplored.

The goal of this paper is to understand what developers care about when they review code, *i.e.,* what are the main criteria developers rely on to develop a decision about accepting or rejecting

Table 1: Related work in refactoring-related code review.

Study	Year	Research Type	Research Method	Purpose	Evaluation Technique
Ge *et al.* [34, 35]	2014,2017	Tool-based code review	Case & formative study	Detect refactoring from non-refactoring changes	2 OSS & 35 developers
Alves *et al.* [13, 14]	2014,2018	Tool-based code review	User study	Inspect manual refactoring edits	3 OSS & 15 developers
Morales *et al.* [51]	2015	Empirical study	Case study	Understand the impact of code review on quality	3 OSS
Coelho *et al.* [24]	2019	Literature review	Systematic mapping study	Present refactoring solutions to MCR	N/A
Pascarella *et al.* [59]	2019	Empirical study	Case study	Study the effect of code review on code smells	7 OSS
Paixão *et al.* [56]	2020	Empirical study	Mining-based	Study refactoring in MCR	6 OSS
Pantiuchina *et al.* [57]	2020	Empirical study	Mining-based	Study refactoring in pull requests	150 OSS
Uchôa *et al.* [89]	2020	Empirical study	Mining-based	Study MCR & design degradation	7 OSS
Uchôa *et al.* [88]	2021	Empirical study	Mining-based	Predict design impactful changes in MCR	7 OSS
AlOmar *et al.* [3]	2021	Empirical & industrial case study	Case study & survey	Understand refactoring challenges in MCR	24 developers
Brito & Valente [20]	2021	Tool-based code review	User study	Detect refactoring during code review	8 developers
Kurbatova *et al.* [44]	2021	Tool-based code review	Mining-based	Enhance IDE representation of changes	4 OSS
Coelho *et al.* [25]	2021	Empirical study	Mining-based	Study refactoring-inducing pull requests	350 OSS
This work	**2022**	**Empirical & case study**	**Mining-based**	**Understand refactoring practices in MCR**	**2,225 OSS**

a submitted refactored code, and what makes this process challenging. This paper seeks to develop a taxonomy of all refactoring *contexts*, where reviewers are raising concerns about refactoring. We drive our study using the following research questions:

RQ1. *How do refactoring reviews compare to non-refactoring reviews in terms of code review efforts?*

RQ2. *What are the criteria that mostly associated with refactoring review decision?*

To answer these research questions, we first extracted set of 5,505 refactoring-related code reviews, from OpenStack ecosystem. Then, we compared this set of refactoring-related code reviews, with another set of code reviews, in terms of number of reviewers, number of review comments, number on inline comments, number of revision, number of changed files, review duration, discussion and description length, and code churn. Our empirical investigation indicates that refactoring-related code reviews take significantly longer to be resolved and typically triggers more discussions between developers and reviewers to reach a consensus. To understand the key characteristics of reviewing refactored code, we perform a thematic analysis on a significant sample of these reviews. This process resulted in a hierarchical taxonomy composed of 6 categories, and 28 sub-categories. We also externally validated our taxonomy using a survey of 11 questions related to our categories' correctness and representativeness. We also conducted a follow-up interview to further discuss the survey outcomes.

We provide our comprehensive experiments package [65] to further replicate and extend our study. The package contains the raw data, analyzed data, statistical test results, survey questions, interview transcription, and custom-built scripts used in our research.

The remainder of this paper is organized as follows: Section 2 reviews the existing studies related to refactoring awareness and code review. Section 3 outlines our empirical setup in terms of data collection, analysis and research question. Section 4 discusses our findings, while the research implication is discussed in Section 5. Section 6 captures any threats to the validity of our work, before concluding with Section 7.

2 RELATED WORK

Research on code review has been of importance to practitioners and researchers. A considerable effort has been spent by the research community in studying traditional and modern code review practices and challenges. This literature has been includes case studies (*e.g.*, [3, 34, 35, 47, 51, 66, 70]), user studies (*e.g.*, [14, 17, 63, 81, 92]), surveys (*e.g.*, [3, 15, 45, 80]), and empirical experiments (*e.g.*, [36, 47, 51, 61, 81]). However, most of the above studies focus on studying and improving the effectiveness of modern code review in general, as opposed to our work that focuses on understanding developers' perception of code review involving refactoring. In this section, we are only interested in research related to refactoring-aware code review. We summarize these approaches in Table 1.

In a study performed at Microsoft, Bacchelli and Bird [15] observed, and surveyed developers to understand the challenges faced during code review. They pointed out purposes for code review (*e.g.*, improving team awareness and transferring knowledge among teams) along with the actual outcomes (*e.g.*, creating awareness and gaining code understanding). In a similar context, MacLeod *et al.* [45] interviewed several teams at Microsoft and conducted a survey to investigate the human and social factors that influence developers' experiences with code review. Both studies found the following general code reviewing challenges: (1) finding defects, (2) improving the code, and (3) increasing knowledge transfer. Ge *et al.* [34, 35] developed a refactoring-aware code review tool, called ReviewFactor, that automatically detects refactoring edits and separates refactoring from non-refactoring changes with the focus on five refactoring types. The tool was intended to support developers' review process by distinguishing between refactoring and non-refactoring changes, but it does not provide any insights on the quality of the performed refactoring. Inspired by the work of [34, 35], Alves *et al.* [13, 14] proposed a static analysis tool, called RefDistiller, that helps developers inspect manual refactoring edits. The tool compares two program versions to detect refactoring anomalies' type and location. It supports six refactoring operations, detects incomplete refactorings, and provides inspection for manual refactorings.

Coelho *et al.* [24] performed a systematic literature mapping study on refactoring tools to support modern code review. They raised the need for more tools to explain composite refactorings. They also reported the need for more surveys to assess the existing refactoring tools for modern code review in both open source and industrial projects. Pascarella *et al.* [60] investigated the effect of code review on bad programming practices (*i.e.*, code smells). Their approach mainly focused on comparing code smells at file-level

before and after the code review process. Additionally, they manually investigated whether the severity of code smells was reduced in a code review or not. Their results show, in 95% of the cases, the severity of code smells does not decrease with a review. The reduction of code smells in remaining few cases was impacted by code insertion and refactoring-related changes.

Paixão *et al.* [56] explored if developers' intents influence the evolution of refactorings during the review of a code change by mining 1,780 reviewed code changes from 6 open-source systems. Their main findings show that refactorings are most often used in code reviews that implement new features, accounting for 63% of the code changes we studied. Only in 31% of the code reviews that employed refactorings the developers had the explicit intent of refactoring. Uchôa *et al.* [89] reported the multi-project retrospective study that characterizes how the process of design degradation evolves within each review and across multiple reviews. The authors utilized software metrics to observe the influence of certain code review practices on combating design degradation. The authors found that the majority of code reviews had little to no design degradation impact in the analyzed projects. Additionally, the practices of long discussions and high proportion of review disagreement in code reviews were found to increase design degradation. In their study on predicting design impactful changes in modern code review with technical and/or social aspects, Uchôa *et al.* [88] analyzed reviewed code changes from seven open source projects. By evaluating six machine learning algorithms, the authors found that technical features results in more precise predictions and the use of social features alone also leads to accurate predictions.

A couple of studies considered pull requests as a main source of the study code review process. Pantiuchina *et al.* [57] presented a mining-based study to investigate why developers are performing refactoring in the history of 150 open source systems. Particularly, they analyzed 551 pull requests implemented refactoring operations and reported a refactoring taxonomy that generalizes the ones existing in the literature. Coelho *et al.* [25] performed a quantitative and qualitative study exploring code reviewing-related aspects intending to characterize refactoring-inducing pull requests. Their main finding show that refactoring-inducing pull requests take significantly more time to merge than non-refactoring-inducing pull requests.

AlOmar *et al.* [3] conducted a case study in an industrial setting to explore refactoring practices in the context of modern code review from the following five dimensions: (1) developers motivations to refactor their code, (2) how developers document their refactoring for code review, (3) the challenges faced by reviewers when reviewing refactoring changes, (4) the mechanisms used by reviewers to ensure the correctness after refactoring, and (5) developers and reviewers assessment of refactoring impact on the source code's quality. Their findings show that refactoring code reviews take longer to be completed than the non-refactoring code reviews. Brito & Valente [20] introduced RAID, a refactoring-aware and intelligent diff tool to alleviate the cognitive effort associated with code reviews. The tool relied on RefDiff [72] and is fully integrated with the state-of-the-art practice of continuous integration pipelines (GitHub Actions) and browsers (Google Chrome). The authors evaluated the tool with eight professional developers and found that RAID indeed reduced the cognitive effort required for detecting and reviewing refactorings. In another study, Kurbatova *et al.* [44] presented RefactorInsight, a plugin for IntelliJ IDEA that integrates information about refactorings in diffs in the IDE, auto folds refactorings in code diffs in Java and Kotlin, and shows hints with their short descriptions.

To summarize, the study of open source projects that use either the Gerrit tools or GitHub pull requests has been extensively studied (*e.g.*, [57, 66, 85, 93]). Since notable open source organizations such as Eclipse and OpenStack adopted Gerrit as their code review management tool, we chose to analyze refactoring practice in modern code review from projects that adopted Gerrit as their code review tool. Although there are recent studies that explored the motivation behind refactoring in pull requests [25, 57], to the best of our knowledge, no prior studies have manually extracted all the criteria developers are facing when submitting their refactored code for review. To gain more in-depth understanding of factors mostly associated with refactoring review discussion and to advance the understanding of refactoring-aware code review, in this paper, we performed an empirical study on a rapidly evolving open source project, with large number of files with 100% review coverage. This study complements the existing efforts that are done in an industrial environment [3] and an open source systems [25, 57] using GitHub pull-based development.

3 STUDY DESIGN

The main goal of our study is to understand refactoring practice in the context of modern code review to characterize the criteria that influence the decision making when reviewing refactoring changes. Thus, we aim at answering the following research questions:

- **RQ1.** *How do refactoring reviews compare to non-refactoring reviews in terms of code review efforts?*
- **RQ2.** *What are the criteria that mostly associated with refactoring review decision?*

According to the guidelines reported by Runeson and Höst [69], we designed an empirical study that consists of three steps, as depicted in Figure 1. Since our research questions are both quantitative and qualitative, we used tools/scripts along with manual activities to investigate our data. Furthermore, the dataset utilized in this study is available on our project website [65] for extension and replication purposes.

Gerrit-based code review process. The code review process of the studied systems is based on Gerrit[1], collaborative code review framework allowing developers to directly tag submitted code changes and request its assignment to a reviewer. Generally, a code change author opens a code review request containing a title, a detailed description of the code change being submitted, written in natural language, along with the current code changes annotated. Once the review request is submitted, it appears in the requests backlog, open for reviewers to choose. Once reviewers are assigned to the review request, they inspect the proposed changes and comment on the review request's thread, to start a discussion with the author. This way, the authors and reviewers can discuss the submitted changes, and reviewers can request revisions to the code being reviewed. Following up discussions and revisions, a review decision is made to either accept or decline, and so the proposed

[1] https://www.gerritcodereview.com/

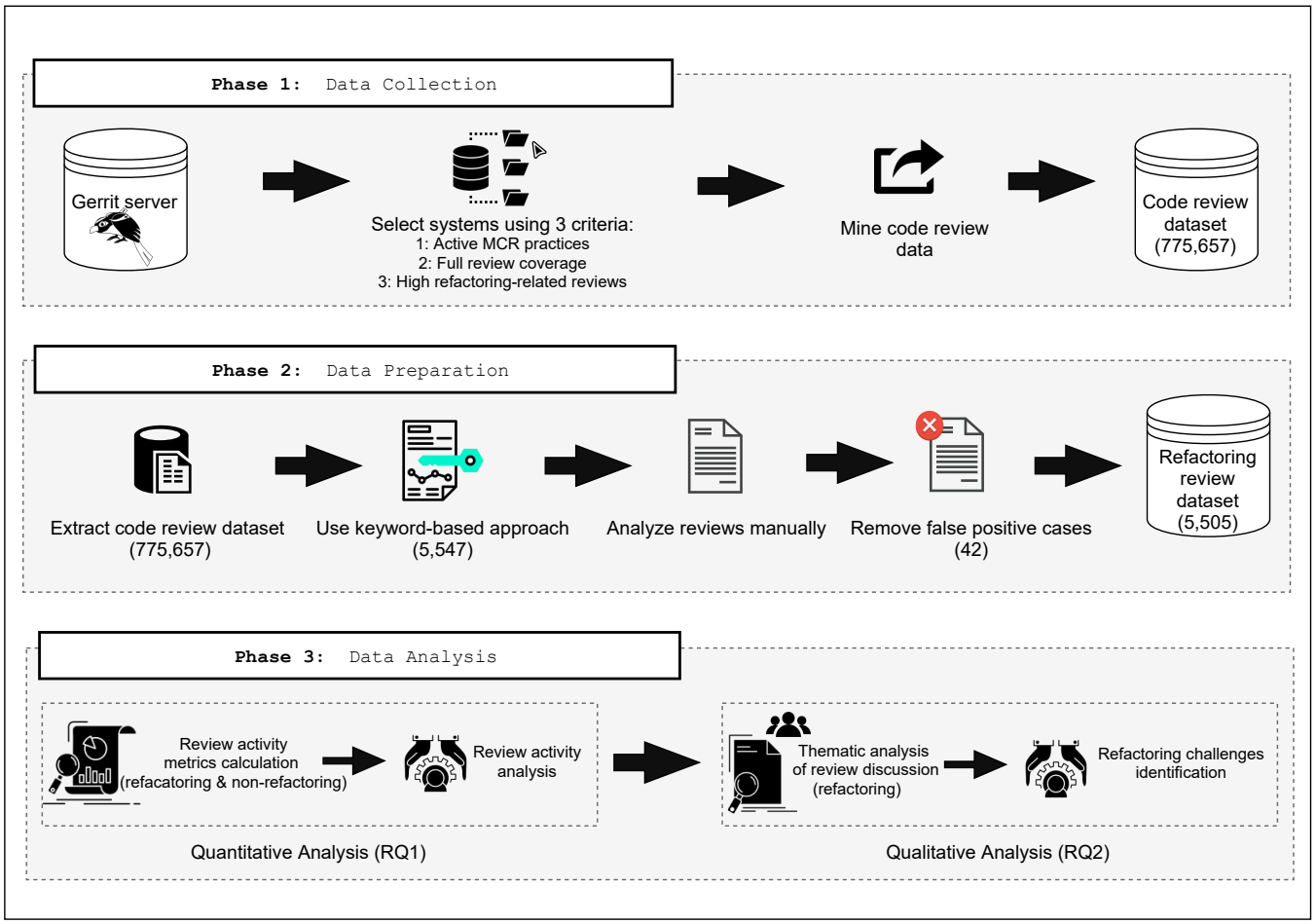

Figure 1: Overview of our experiment design.

code changes are either "*Merged*" to production or "*Abandoned*". A diagram, modeling a simplified bird's view of the Gerrit-based code review process, is shown in Figure 2.

3.1 Data Collection

3.1.1 Studied Systems. To select the subject systems, we identified three important criteria:

Criterion #1: Active MCR practices. Our goal is to study a system that actively examines code changes through a code review tool. Therefore, we focus on systems where a number of reviews are performed using a code review tool (*i.e.*, systems which have review procedures in place), similar to [51, 83, 84].

Criterion #2: Full review coverage. Since we investigate the practice of refactoring-related code reviews, we focus on systems that have many files with 100% review coverage (*i.e.*, files where every change made to them is reviewed before they are merged into the repositories), similar to the studies that explored code review practices in defective files [47, 83, 84].

Criterion #3: High number of refactoring-related reviews. Since we want to study refactoring practices in MCR, we need to ensure that the subject systems have sufficient refactoring-related instances to help us perform our statistical analysis. So, we selected the project with the highest number of refactoring reviews.

To satisfy criterion 1, we started by considering five systems (*i.e.*, OpenStack, [2] Qt ,[3] LibreOffice, [4] VTK, [5] ITK [6]) that use Gerrit code review tool and have been widely studied in previous research in MCR, *e.g.*, [21, 32, 54, 82]. We then discarded VTK and ITK since Thongtanunam *et al.* [84] reported that the linkage rate of code changes to the reviews for VTK is too low and ITK does not satisfy criterion 2. As for criterion 3, after mining the code review data, we found that OpenStack has a higher number of refactoring-related code review instances than Qt and LibreOffice. Due to the human-intensive nature of carefully studying and analyzing refactoring practice in MCR, we opt for performing an in-depth study on a single system. With the above-mentioned criteria in mind, we select OpenStack, an open-source software for cloud infrastructure

[2] https://review.opendev.org/
[3] https://codereview.qt-project.org/
[4] https://gerrit.libreoffice.org/
[5] http://vtk.org/
[6] http://itk.org/

service that is developed by many well-known companies, *e.g.*, IBM, VMware, and NEC.

3.1.2 Mining code review data. We mined code review data using the `RESTful API`[7] provided by Gerrit, which returns the results in a `JSON` format. We used a script to automatically mine the review data and store them in `SQLite` database. All collected reviews are closed (*i.e.*, having a status of either "*Merged*" or "*Abandoned*"). In total, we mined 775,657 code changes between December 2012 and April 2021 from OpenStack projects. An overview of the project's statistics is provided in Table 2.

Table 2: Overview of the OpenStack studied system.

Item	Count
Review period	12.16.2012 to 04.27.2021
Number of projects	2,225
Version	Folsom to Stein
Line of code	31,680,274
No. of commits	4,137,446
No. of code changes	775,657
No. of developers	15,432
No. of files	705,982
Reviews with keyword '*refactor**' in title or description	10,440
Reviews with keyword '*refactor**' in title and description	5,547
False positive reviews	42
Refactoring review dataset	5,505
Final refactoring and non-refactoring reviews dataset	11,010

3.2 Data Preparation

To extract the set of refactoring-related code reviews, we follow a two-step procedure: (1) automatic filtering, and (2) manual filtering.

(1) Automatic Filtering. In the first step, we utilize a keyword-based mechanism to filter out all entries that do not contain the keyword *refactor** in both the title and description of the submitted code change. Specifically, we start by searching for the term '*refactor**' in the title or description (we use * to capture extensions like refactors, refactoring etc.). The keyword-based approach has been widely used in prior studies related to identify refactoring changes or defect-fixing or defect-inducing changes [4, 6–9, 11, 12, 25, 39, 41, 43, 48, 50, 57, 64, 76, 79, 84], as it allows the pruning of the search space to only consider code changes whose documentation matches a specific intention. The choice of '*refactor*', besides being used by various related studies, is intuitively the first term to identify ideal refactoring-related code changes [4, 7, 9, 10, 43]. Also, the choice of enforcing the existence of the keyword in both code change's title or description was piloted by a first trial of only applying the filter to description. This initial filtering gives us a total of 10,440 review instances. After performing a manual inspection of a subset of these reviews, we realized that there exist several false positive cases. To reduce them, we only kept code changes having the term '*refactor**' in both the title and description. The keyword-based filtering resulted in only selecting 5,547 code changes and their corresponding reviews. We notice that the ratio of these reviews is very small in comparison with the total number of the mined reviews, *i.e.*, 775,657. However, these observations aligned with previous studies [52, 77] as developers typically do not provide details when documenting their refactorings. Yet, despite this strict filtering, this approach is still prone to high false positiveness, and therefore, the second step of manual analysis is needed.

[7] https://gerrit-review.googlesource.com/Documentation/rest-apichanges.html

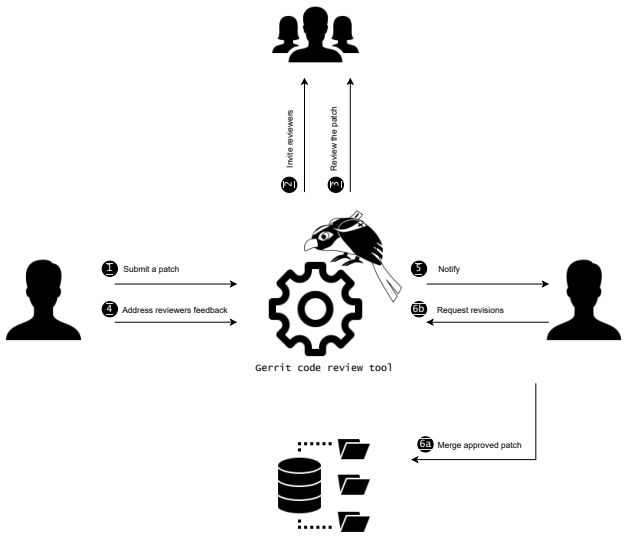

Figure 2: Gerrit-based code review process overview.

(2) Manual Filtering. To ensure the correctness of data, we manually inspected and read through all these refactoring reviews to remove false positives. An example of a discarded review is: "*Revert "Refactor create_pool." and "Add request_access_to_group method"*" [1]. We discarded this code review because the refactoring action is undone by developers. This step resulted in only considering 5,505 reviews. Our goal is to have a *gold set* of reviews in which the developers explicitly reported the refactoring activity. This *gold set* will serve to check later criteria that are mostly associated with refactoring review discussion.

3.3 Data Analysis

To address our research questions, a structured mixed-method study was designed to combine elements of both quantitative and qualitative research.

3.3.1 Quantitative data analysis. We leverage the data collected to compare refactoring and non-refactoring reviews using review efforts, *i.e.*, code review metrics. As we calculate the metrics of refactoring and non-refactoring code reviews, we want to distinguish, for each metric, whether the variation is statistically significant. We first test for normality using Shapiro-Wilk normality test [78] and we observe that the distribution of code review activity metrics does not follow a normal distribution. Therefore, we use the Mann-Whitney U test [26], a non-parametric test, to compare between the two groups, since these groups are independent of one another. The null hypothesis is defined by no variation in the metric values of refactoring and non-refactoring code reviews. Thus, the alternative hypothesis indicates that there is a variation in the metric values. Additionally, the variation between values of both sets is considered

significant if its associated p-value is less than 0.05. Furthermore, we use the Cliff's Delta (δ) [23], a non-parametric effect size measure, to estimate the magnitude of the differences between refactoring and non-refactoring reviews. As for its interpretation, we follow the guidelines reported by Romano *et al.* [68]:

- Negligible for $|\ \delta\ | < 0.147$
- Small for $0.147 \leq |\ \delta\ | < 0.33$
- Medium for $0.33 \leq |\ \delta\ | < 0.474$
- Large for $|\ \delta\ | \geq 0.474$

To measure the extent of the relationship between these metrics, we conducted a Spearman rank correlation test (a non-parametric measure) [90]. We chose a rank correlation because this type of correlation is resilient to data that is not normally distributed.

3.3.2 Qualitative data analysis. To answer RQ2, two of the authors perform the analysis of the data. One of the author manually inspects refactoring review discussions by considering both the general comments and the inline comments, and the other author reviews the taxonomy. As the complete list of refactoring review data is too large to be manually examined, we select a statistically significant sample for our analysis by considering the reviews with higher review duration, and we annotated 384 reviews. This quantity roughly equates to a sample size with a confidence level of 95% and a confidence interval of 5. The manual analysis process took approximately 40 days in total. Next, we describe the methodology for building and refining the taxonomy, followed by the validation method.

Taxonomy Building and Refinement. When analyzing the review discussions, we adopted a thematic analysis approach based on guidelines provided by Cruzes *et al.* [27]. Thematic analysis is one of the most used methods in Software Engineering literature (*e.g.*, [73]), which is a technique for identifying and recording patterns (or "themes") within a collection of descriptive labels, which we call "codes". For each refactoring review, we proceeded with the analysis using the following steps: i) Initial reading of the review discussions; ii) Generating initial codes (*i.e.*, labels) for each review; iii) Translating codes into themes, sub-themes, and higher-order themes; iv) Reviewing the themes to find opportunities for merging; v) Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence.

The above-mentioned steps were performed independently by two authors. One author performed the labeling of review discussions independently from the other author who was responsible for reviewing the currently drafted taxonomy. By the end of each iteration, the authors met and refined the taxonomy. At the time of the study, one of the author had 4 years of research experience on refactoring, while the other author had 9 years of research experience on refactoring.

It is important to note that the approach is not a single step process. As the codes were analyzed, some of the first cycle codes were subsumed by other codes, relabeled, or dropped all together. As the two authors progressed in the translation to themes, there was some rearrangement, refinement, and reclassification of data into different or new codes. For example, we aggregated, into "*Refactoring*", the preliminary categories "*incorrect refactoring*", "*behavior preservation violation*", "*separation of other changes from refactoring*", "*interleaving other changes with refactoring*", and "*domain constraint*" that were discussed by different reviewers. We used the thematic analysis technique to address RQ2.

Taxonomy Validation. In addition to the iterative process of building the taxonomy, we need to also externally validate it from a practitioner's point of view [30, 58]. The aim of this validation is to investigate whether it reflects actual MCR practices. To do so, we validated the taxonomy with a senior developer, with 8 years of industrial and refactoring experience, and with 6 years of experience in code review. The survey contained 11 questions related to the correctness and representativeness of our taxonomy. We also conducted a follow-up interview to further discuss the survey outcomes. The interview took an hour and was recorded for further analysis. The interview summary is available in the replication package.

4 RESULTS AND DISCUSSION

4.1 How do refactoring reviews compare to non-refactoring reviews in terms of code review efforts?

Approach. To address RQ1, we intend to compare *refactoring reviews* with *non-refactoring reviews*, to see whether there are any differences in terms of code review efforts or metrics listed in Table 3. Since our refactoring set contains 5,505 reviews, we need to sample 5,505 non-refactoring reviews from the remaining ones in the review framework. To ensure the representativeness of the sample [22], we use stratified random sampling by choosing reviews from the rest of reviews.

Results. By looking at the statistical summary in Table 3, we found that reviewing refactoring changes significantly differ (*i.e.*, more reviewers (μ = 5.56), more review comments (μ = 20.87), more inline comments (μ = 6.61), more revisions (μ = 4.79), more file changes (μ = 5.98), lengthier review time (μ = 928.21), discussions and descriptions (μ = 3450.29, μ = 327.61, respectively), and more added and deleted lines between revisions (μ = 367.63) from reviewing non-refactoring changes. As shown in Table 3, we performed a non-parametric Mann-Whitney U test and we obtained a statistically significant p-value when the values of these two groups were compared (p-value < 0.05 for all review efforts), and accompanied with a small, medium, or large effect size depending on the review effort/metric.

We speculate that reviewing refactoring triggers longer discussions between the code change authors and the reviewers as we notice that several refactoring-related actions are being extensively discussed before reaching an agreement. While previous studies have found a similar pattern in GitHub's pull requests in open-source systems [25] and using code review tools in industry [3], there is not a study that looked at what are the main reasons for refactoring-related discussions to take significantly longer effort to be reviewed. Therefore, the findings of RQ1 has motivated us to manually analyze these reviews and extract the main criteria related to reviewing refactored code (RQ2).

Table 3: Statistics of code review activity efforts.

Metrics	*Refactoring code review*						*Non-refactoring code review*						*Statistical difference*	
	Min	**Q1**	**Median**	**Mean**	**Q3**	**Max**	**Min**	**Q1**	**Median**	**Mean**	**Q3**	**Max**	***p*-value**	**Cliff's delta (δ)**
Number of reviewers	0	2	4	5.65	6	12	0	2	3	4.45	5	9	**6.750458e-43**	small (0.15)
Number of review comments	0	5	9	20.87	19	40	0	3	6	13.05	12	25	**2.730193e-82**	small (0.22)
Number of inline comments	0	0	0	6.61	5	12	0	0	0	3.26	1	2	**2.066649e-98**	medium (0.33)
Number of revisions	1	1	3	4.79	6	13	1	1	2	3.19	3	6	**1.252695e-126**	small (0.3)
Number of changed files	0	1	3	5.98	6	13	0	1	1	3.69	3	6	**6.006899e-190**	medium (0.33)
Review duration (seconds)	0	37	170.68	928.21	606.22	1458.68	0	12.85	75.98	738.93	351.03	852.99	**2.572584e-66**	small (0.23)
Length of discussion (characters)	0	168	451	3450.29	1563	3653	0	110	296	1962.39	1044	2436	**6.966752e-42**	small (0.15)
Length of description (characters)	62	171	264	327.61	388	711	56	124	218	276.97	354	699	**4.071541e-41**	small (0.13)
Code churn	-1	40	114	367.63	309	711	0	3	11	201.58	53	128	**0.000000e+00**	large (0.64)

Further, we observe that refactoring related code reviews impact larger code churn and more changes across files than non-refactoring code changes. These results are expected and are in agreement with prior works [25, 40, 55], which found that refactored code has higher size-related metrics and larger changes promote refactorings. We also notice that the number of developers who participated in the refactoring code review process is also higher due to the high number of added, modified, or deleted lines between revisions. In contrast to a previous finding [25], however, no evidence of the correlation between the number of reviewers and refactoring was detected.

Moreover, our correlation analysis is measured using the Spearman rank correlation test reveals that the number of review comments, discussion length, and number of revisions are highly correlated with the review duration. The Spearman rank correlation test yielded a statistically significant (*i.e.*, p-value < 0.05) correlation coefficient of 0.57, 0.53 and 0.49, respectively. Further, the number of reviewers are highly correlated with the discussion length and number of review comments with p-value < 0.05 and correlation coefficient of 0.73 and 0.77, equating to a strong correlation. Additionally, we observe that very high number of deleted or added lines are correlated with very high number of file changes (*i.e.*, p-value < 0.05 and correlation coefficient of 0.58). In contrast, the Spearman's correlation values detect no significance in the relationship between the number of files and review duration.

4.2 What challenges do developers face when reviewing refactoring tasks?

Approach. To get a more qualitative sense, we manually inspect the OpenStack ecosystem using a thematic analysis technique [27], to study the challenges that reviewers catch when reviewing refactoring changes, so we understand the main reasons for which refactoring reviews take longer compared to non-refactoring reviews.

Results. Upon analyzing the review discussions, we create a comprehensive high-level categories of review criteria. Figure 3 shows the proposed taxonomy of the criteria related to reviewing refactored code. The taxonomy is composed of two layers: the top layer contains 6 categories that group activities with similar purposes, whereas the lower layer contains 28 subcategories that essentially provide a fine-grained categorization. Due to space constraints, we made the iterative taxonomy available in our replication package [65]. These refactoring review criteria are centered around six main categories as shown in the figure: (1) quality, (2) refactoring , (3) objective , (4) testing, (5) integration, and (6) management. It is worth noting that our categorization is not mutually exclusive, meaning that a review can be associated with more than one category. An example of each category is provided in Table 4. Further, according to the interview with the senior developer, they conformed that our taxonomy is representative of the reviews cases they have experienced. In the rest of this subsection, we provide more in-depth analysis of these categories.

Category #1: Quality. The design quality is found to be a vital part of the refactoring review process. According to the review discussions, reviewers enforce the adherence to *coding convention*, optimization of *internal quality attribute*, *external quality attribute*, the avoidance of (*i.e.*, *code smell*, resolution of *technical debt*, correctness of *design pattern* implementations), and *lack of documentation*. For instance, developers recommend appropriate ways to write code, optimize *internal and external quality attributes* as developers may not *draw the full picture* of the software design, which makes their decision adequate locally, but not necessarily at the global level. Moreover, developers only reason on the actual *snapshot* of the current design, and there is no systematic way for them to recognize its evolution by, for instance, accessing previously performed refactorings. This may also narrow their decision making, and they may end up *reverting* some previous refactorings. Moreover, providing a clear and meaningful explanation of the reasons behind the proposed changes are equally important in review decisions. The requested documentation change includes (but is not limited to) expanding code comments, removing code smells, and reusing existing functionality. Further, developers typically refactor classes and methods that they frequently change. We observe this as various reviews were containing similar recurrent files. So, the more they change the same code elements, the more familiar with the system they become, and so it improves their design decisions. While the follow-up interview indicates the importance of each sub-category in real life, it was pointed out by the senior developer that *technical debt* was not allowed to be submitted since the company has a strict policy preventing developers from submitting near optimal code. Additionally, according to the senior developer's experience, reviewers did allow the relaxation of design patterns strict implementations, as long as it is properly justified.

Category #2: Refactoring. This category gathers reviews with a focus on evaluating the correctness of the code transformation and checking whether or not the submitted changes lead to a safe and

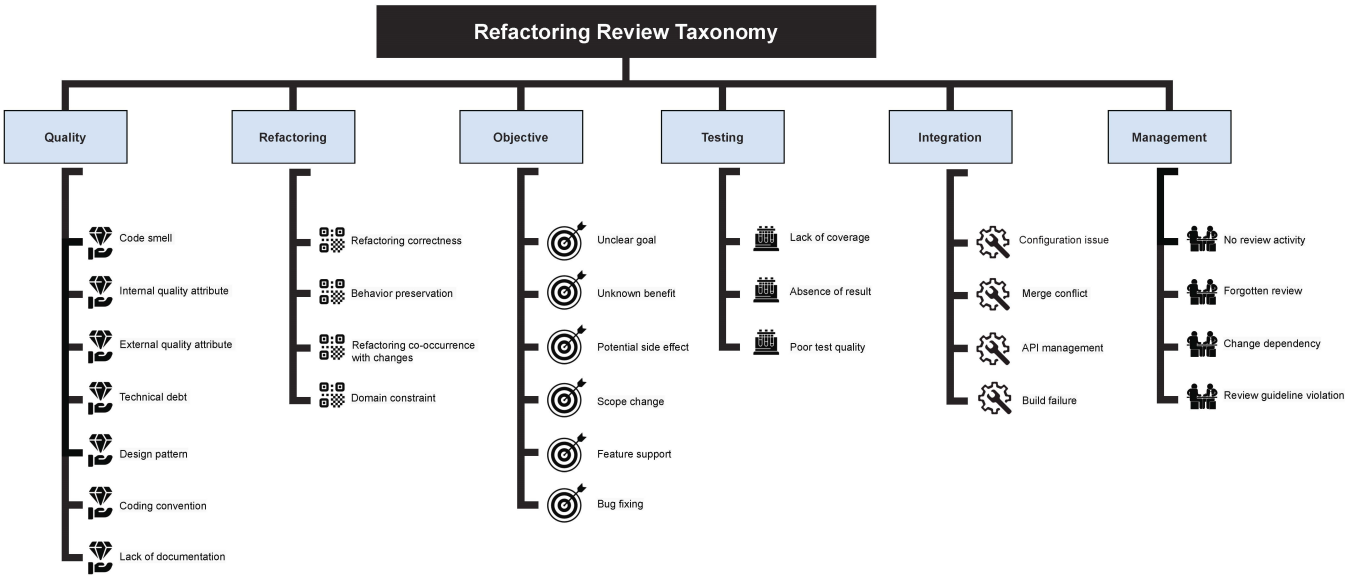

Figure 3: Refactoring review criteria in modern code review.

trustworthy refactoring. These reviews discuss *refactoring correctness*, *behavior preservation*, *refactoring co-occurrence with changes*, and *domain constraint*. Because developers interleave refactorings with other tasks, reviewers highlighted that mixing refactoring with other changes may lead to overshadowing errors, and so the intrusion of bugs. Similar concerns were also raised by the senior developer during our interview. In their regular reviews, they would typically recommend the separation of refactoring from other code changes, whenever possible.

Category #3: Objective. In this category, we have gathered cases where reviewers eventually ask to clearly document the *goal*, *benefit*, *side effects*, *scope*, *feature-related*, and *bug fix-related* activities to better understand the rationale of the submitted code changes. This reveals how reviewers keep proposing areas of improvement in developers' documentation practices, pertaining to the perception and the rationale of the change. It appears that the clarity of the documented changes is of paramount importance as reviewers also struggle with identifying the *benefit* and/or *side effects* and spend time understanding them. We realized that the clarity of the explanation of what is being changed and why affects review time and decision. Missing rationale and documentation are also frequently reported as confusing during code review [31]. Reviewers are requested to *clarify the scope of the change*. Changing the scope is one of the influential factors for reviewers to make their decisions. Reviewers check the actual appropriateness of the change before it is merged into the code base. Further, we realized that source code is not the only artifact that developers refactor. Other artifacts, such as database elements, are also subject to refactoring. During the follow up interview, the senior developer indicated that *unclear goal* and *unknown benefit* were less likely to be encountered due to the company review guidelines. Also *bug fixing* is not frequent in the expert's current company because of the enforcement of performing minimal refactoring changes when fixing bugs.

Category #4: Testing. Refactoring is supposed to preserve the behavior of the software. Ideally, using the existing unit tests to verify that the behavior is maintained should be sufficient. However, since refactoring can also be interleaved with other tasks, then there might be a change in the software's behavior, and so, unit tests may not capture such changes if they were not revalidated to reflect the newly introduced functionality. We have seen various reviews discussions raising this concern, especially when developers are unaware of such non behavior preserving changes, and so, deprecated unit tests will not guarantee the refactoring correctness. Based on our analysis of the discussion, reviewers suggested adding unit tests before the refactoring, so that they can be more confident the code has not broken anything. They also recommend adding test cases when the refactored code lowers the test coverage (*e.g.*, extracting new methods). Moreover, when developers submit a review, they can include the results of running the tests as reviewers expect code changes to be accompanied by a corresponding test change. So, to capture these various cases, under the *Testing* category, we have the following sub-categories: *lack of coverage*, *absence of result*, and *poor test quality*. The outcome of the interview shows the existence of this type of review criteria.

Category #5: Integration. This category gathers reviews highlighting how refactoring has complicated the merging process, or triggered configuration issues. Several sub-categories raise, namely, *configuration issue*, *merge failure*, *API management*, and *build failure*. These categories show that refactoring may complicate the review process. As per our analysis, it appears that the code changes involving refactoring reviews tend to be more problematic for failure or issue, compared to non-refactoring reviews. This observation is partially in lined with previous studies [29, 46]. They found that merge conflicts that involve refactoring are more complex than conflicts with no refactoring changes. Moreover, we noticed that API upgrades and migrations typically trigger discussions, mainly

when there are API breaking changes. Developers tend to ask about the appropriate migration plans and the necessary changes to preserve the software's behavior during the migration process. Despite the existence of various tools to support the detection of breaking changes, discussions revealed that this process is still manual. Besides agreeing with the existence of these types subcategories, the participant also reported that his current company supports developers with review *bots* to early detect any merge-related issues.

Category #6: Management. Another category emerged from the manual coding analysis is review management. The subcategories we found were *no ongoing review activity*, *forgotten review*, *change dependency*, and *review guideline violation*. In other words, we observe that some of the reviews take longer due to a lack of reviewer attention, as there is a case of little prompt discussion about the proposed changes. For instance, some reviews received a review score of +1 from a reviewer, but there was no other activity after waiting a couple of days or months. The follow-up interview confirms that this category is common in industry. Reviewers have their own workload, and some discussions can be subject to delays depending on the developer and reviewers' participation [83].

5 IMPLICATIONS

5.1 Implications for Practitioners

Establishing guidelines for refactoring-related reviews. Our taxonomy shows reviewing refactoring goes beyond improving the code structure. To improve the practice of reviewing refactored code, and contribute to the quality of reviewing code in general, managers can collaboratively work with developers to establish customized guidelines for reviewing refactoring changes which could establish beneficial and long-lasting habits or themes to accelerate the process of reviewing refactoring. Additionally, since our RQ2 findings show that integration and testing are one of the challenges caught by developers when reviewing refactoring changes, it is recommended to utilize continuous integration to keep the testing suite in sync with the code base during and after refactoring. Further, adherence to coding conventions is considered one of difficulties when reviewing refactoring tasks. To cope with this challenge, we recommend project leaders to educate their developers about the coding conventions adopted in their systems. Considering the above-mentioned characteristics not only save developers time and effort, but also can assist taking informed decision and bring a discipline toward reviewing code involving refactorings within a software development team.

5.2 Implications for Researchers

Exploring the potential of combining multiple behavior preservation strategies when reviewing refactorings. Our study shows that preserving the behavior of software refactoring during the code review process is a critical concern for developers, and developers determine whether a behavior is preserved based on the context. Recently, AlOmar *et al.* [5] aggregated the behavior preservation strategies that have been evaluated using single or multiple refactoring operations, and some of these refactorings are applied using multiple strategies. In order to accelerate the process of reviewing code involving refactoring, future researchers are advised to explore the potential of combining several behavior preservation approaches and use those which would be useful in the context of modern code review according to a defined set of criteria. For instance, Soares *et al.* [75] have implemented the tool 'SafeRefactor' to identify behavioral changes in transformation. It would be an interesting idea to verify the correctness after the application of refactoring by embedding test results and generating a test suite for capturing unexpected behavioral changes in code review board.

Supporting for the refactoring of non-source code artifacts. From RQ2, we discover that refactoring operations are not limited to source code files. Artifacts such as databases and log files are also susceptible to refactoring. Similarly, we also observed discussions about refactoring test files. While it can be argued that test suites are source code files, recent studies by [9, 57] show that the types of refactoring operations applied to test files are frequently different from those applied to production files. Hence, future research on refactoring is encouraged to introduce refactoring mechanisms and techniques exclusively geared to refactoring non source code artifact types and test suites.

5.3 Implications for Tool Builders

Developing next generation refactoring-related code review tools. Finding that reviewing refactoring changes takes longer than non-refactoring changes reaffirms the necessity of developing accurate and efficient tools and techniques that can assist developers in the review process in the presence of refactorings. Refactoring toolset should be treated in the same way as CI/CD tool set and integrated into the tool-chain. Researchers could use our findings with other empirical investigations of refactoring to define, validate, and develop a scheme to build automated assistance for reviewing refactoring considering the refactoring review criteria as review code become an easier process if the code review dashboard augmented with the factors to offer suggestions to better document the review.

Moreover, we noticed that poorly naming the code elements is one of the major bad refactoring practices typically catch by developers when reviewing refactoring changes. Constructing tools that enforce code conventions aid in speeding up reviewing refactoring changes and benefiting reviewers to focus on deeper design defects. Furthermore, to accelerate code review process and limit having a back-and-forth discussion for clarity on the problem faced by the developer, tool builders can develop *bots* for the integration, testing, and management categories. Additionally, it would be interesting to use a popular and widely adopted quality framework, *e.g.*, Quality Gate of SonarQube [33], as part of quality verification process by embedding its results in the code review. This might facilitate convincing the reviewer about the impact and the correctness of the performed refactoring.

6 THREATS TO VALIDITY

In this section, we describe potential threats to validity of our research method, and the actions we took to mitigate them.

Internal Validity. Concerning the identification of refactoring related code review, we select reviews with the keyword '*refactor**' in their title and description. Such selection criteria may have resulted in missing refactoring-related reviews and there is the

Table 4: A taxonomy of the refactoring review criteria in Modern Code Review.

Category	Sub-category	Example (Excerpts from a related refactoring review discussion)
Quality	Code smell	"*I do understand the desire to refactor some code to **eliminate duplicate code**. The purpose of the common class was to contain all of the duplicate code between the 2 drivers. This seems like a half baked approach to refactoring to accomplish that goal, when the common class should have been used as a new base. Now with this patch there are 2 classes (base and common) that contain common code.*"
	Internal quality attribute	"*I think there is still an **inheritence problem** with the Base2 class (see inline) and the way python handles multiple inheritence means we have to be careful when creating the UTF8 versions of the unittest classes.*"
	External quality attribute	"*Looking through the series i found enough such code to see that it does **not look easy readable**.*"
	Technical debt	"*I was trying to use exception.ARQInvalidState with assertRaises, but **found we had some technical debts**, so use Exception instead to check the error message.*"
	Design pattern	"*I think the alternative should be what other refactor possibilities are. The proposed change indicates some of the methods that are being planned to be implemented. As an alternative to evolve this library in future **we need another design pattern**.*"
	Coding convention	"*We've seen how **difficult** it is to arrive at **a single naming convention** that works for all OpenStack projects, but as Rally expands beyond benchmarking OpenStack alone, it is quite simply impossible to create a single name format that will work for literally everything.*"
	Lack of documentation	"***Is there a documentation** somewhere explaining how to define "extra" dependencies? It would be nice to have a link somewhere. IHMO dependencies are a very complex problem, and extra dependencies look even more complex to me.*"
Refactoring	Refactoring correctness	"*I think, this piece is **incorrectly refactored**. You should create neutron.conf.agent.linux with IP_LIB for linux and corresponding change for neutron.conf.agent.windows with IP_LIB for windows. Then you could import it, similar to this, how it was done previously.*"
	Behavior preservation	"*Refactoring involves "behavior-preserving transformations", but **this change does change behavior**. Describing it as refactoring is misleading and suggests the change is less risky than it really is*"
	Refactoring co-occurrences with changes	"*I would have **separate this change from your refactor**. The value returned by this function is transferred to the compute manager and to the virt drivers. Some of them could have considered True/False/None to handle different behavior and even if that is not the case, to have this isolated can help reviewer to ensure all is OK.*"
	Domain constraint	"*Looking through the series i found enough such code to see that it does not look easy readable. It operates with StoragePolicy's properties which are initialized there and here we can guess only what they mean (e.g. external_boot). **StoragePolicy knows all about disks, what kinds of disks exists, what disks are used by instance**, etc. It would be consistent to move such operations into that class as well. Or at least to add an iterator over disks there and to use it*"
Objective	Unclear goal	"*with regards to the refactor. The plan was to make a progressive refactor that would keep the old API around while the new one is adopted. The **motivations behind this refactor** are related to the lack of any kind of architecture and design in the current code. The current code, as it is, is not easily consumable by other services, which is the whole point of this library.*"
	Unknown benefit	"*I would like understand **what is the added value** or improvement with these changes to l3-agent scheduler.*"
	Potential side effect	"*It seems promising, but the change in pattern, if not thoroughly thought out, may **lead to other unforeseen side effects**.*"
	Change scope	"*Well I think IloISCSIDeploy name wins over IloPXEDeploy. This is no longer tied to a particular boot interface and can work with other boot interfaces (like virtual media when other driver is refactored). But I missed what Ruby pointed out, it's doing pxe.<something> for now. Needs to be refactored but may be better in another patch. We have enough content in this patch alone which refactors the generic upstream driver. So I think it's better to refactor in next patch.*"
	Feature support	"*Isn't this a whole **new feature**, and not just a refactor? IMHO this should be its own spec.[...] Yes, this is new feature and cannot be implemented by only this refactoring. This feature depends on the refactoring. I agree with you to create a spec*"
	Bug fixing	"*I don't think some of this patch is needed since we merged Gage's fix. On the other hand, this change seems to be doing quite a bit of refactoring. Is there a reason to continue pursuing that refactor, or was it mainly to **fix the bug?***"
Testing	Lack of coverage	"***Is there enough coverage** on this to ensure that all changes are tested?*"
	Absence of result	"*I should read more closely... this was intended according to the commit message: "for patches that get 'rechecked' a lot, you'll see the whole **history of test results** on current patch".*"
	Poor test quality	"*What do you think of continuing to clean up the test infrastructure as you've started here and create good examples of what to do? We could document **how to write good tests** so that authors and reviewers could ensure the quality of new additions, and require rewriting of existing tests only as needed rather than attempting the monumental effort of fixing the entire unit test tree.*"
Integration	Configuration issue	"*IMO also some detailed configuration example and illustration is needed in etc/designate.conf - My concern is that, not specifically against patch but for whole project, we add all these new functionalities but people don't know how to use/configurate it. An example is option 'options' in many config sections. It varies a lot from different types of backend/pool_target, but we didn't document them well. People(at least me) need to read the source code to figure out **how to configure that correctly**. Maybe we should come up with a patch fix this in short future.*"
	Merge conflict	"*This seems like it's mostly just going to **cause merge conflicts** with a lot of outstanding code.*"
	API management	"*This **seems to be a breaking change** in the interface, meaning this new cli wouldn't work with the old api and vice versa. I think it would be appropriate if this new code (in the API) produced dag_execution_date on the action dictionary returned, so as to be backwards compatible with old clients, and this new client should probably sniff this for the new way, and revert to the old way if needed*"
	Build failure	"***Build failed** (check pipeline). For information on how to proceed, see http://docs.openstack.org/infra/manual/developers.html#automated-testing.*"
Management	No review activity	"*This patch has been **idle for a long time**, so I am abandoning it to keep the review clean sane. If you're interested in still working on this patch, then please unabandon it and upload a new patchset.*"
	Forgotten review	"*I see that this change has not been updated since April 23rd. I also see that there have been **no responses** from you to the questions Mark asked inline on the most recent patchset. Do you intend to continue working on this change?*"
	Change dependency	"*This is a significant enough change that I think it warrants **discussion with the drivers team and a spec**.*"
	Review guideline violation	"*This is much more readable than before, but I would like to see a satisfactory answer to the question of why this is necessary, before this is merged. As far as I am aware, there is **no consensus** from OpenStack on **following this guideline**; do you have a spec for this work?*"

possibility that we may have excluded synonymous terms/phrases. However, even though this approach reduces the number of reviews in our dataset, it also decreases the false positiveness of our selection. While our data collection may results in missing some reviews, our approach ensures that we analyze reviews that are explicitly geared towards refactoring. In other words, these are reviews where developers were explicitly documenting a refactoring action and they wanted it to be reviewed. Additionally, upon performing the manual inspection on review discussions, we realized that refactoring is heavily emphasized on discussions that start with a title or a description containing the keyword '*refactor**'. Yet, this does not prevent other discussions from bringing refactoring into the picture, and these will be missed by our selection (*i.e.*, false negatives). We opted for such picky selection to only consider discussions when code authors explicitly wanted their refactored code to be reviewed, and so reviewers eventually propose a refactoring-aware feedback, which is what we are aiming for in this study. Therefore, it would be interesting to consider scenarios where reviewers have raised concerns about refactoring a code change that was not intended to be associated with refactoring. Since refactoring can easily be interleaved with other functional changes, it would be interesting to extract scenarios where reviewers thought it was misused. Study can also help developers better understand not only how to refactor their code, but also how to document it properly for easier review.

Further, we focus on the code review activity that is reported by the tool-based code review process, *i.e.*, Gerrit, of the studied systems due to the fact that other communication media (*e.g.*, in-person discussion [19], a group IRC [71], or a mailing list [67]) do not have explicit links of code changes and recovering these links is a daunting task [16, 84].

Construct Validity. About the representativeness and the correctness of our refactoring review criteria, we derive these criteria from a manual analysis of a subset of refactoring-related reviews that have lengthier review duration. This approach may not cover the whole spectrum of all the review criteria done with refactoring in mind. To mitigate this threat, we reviewed the top 7.3% lengthier refactoring reviews assuming that these reviews will capture the most critical challenges. Additionally, to avoid personal bias during the manual analysis, each step in the manual analysis was conducted by two authors, and the results were always cross-validated. Another potential threat to validity relates to refactoring reviews. Since refactorings could interleave with other changes [52] (*i.e.*, developers performed changes together with refactorings), we cannot claim that the selected refactoring reviews are exclusively about refactoring. Nevertheless, during our qualitative analysis, we identified this activity as one of the challenges that contribute to slowing down the review process.

External Validity. We focus our study on one open-source system, due to the low number of systems that satisfied our eligibility criteria (see Section 3). Therefore, our results may not generalize to all other open-source systems or to commercially developed projects. However, the goal of this paper is not to build a theory that applies to all systems, but rather to show that refactoring can have an impact on code review process. Another potential threat relates to the proposed taxonomy. Our taxonomy may not generalize to other open source or commercial projects since the refactoring review criteria may be different for another set of projects (*e.g.*, outside the OpenStack community). Consequently, we cannot claim that the results of refactoring review criteria (see Figure 3) can be generalized to other software systems where the need for improving the design might be less important. To mitigate this threat, we validate the taxonomy with an experienced software developer, by conducting a follow-up interview to gather further insight and possible clarification. Yet, performing the validation with only one developer brings its own bias. The choice of one developer was driven by their experience with code review. We mitigated the bias by selecting a developer that does not belong to the software systems we analyzed. This brings an external opinion that has no conflict of interests with the current projects.

Conclusion Validity. To compare between two groups of code review requests, we used appropriate statistical procedures with p-value and effect size measures to test the significance of the differences and their magnitude. A statistical test was deployed to measure the significance of the observed differences between group values. This test makes no assumption that the data is normally distributed. Also, it assumes the independence of the groups under comparison. We cannot verify whether code review requests are completely independent, as some can be re-opened, or one large code change can be treated using several requests. To mitigate this, we verified all the reviews we sampled for the test.

7 CONCLUSION

Understanding the practice of refactoring code review is of paramount importance to the research community and industry. Although modern code review is widely adopted in open-source and industrial projects, the relationship between code review and refactoring practice remains largely unexplored. In this study, we performed a quantitative and qualitative study to investigate the review criteria discussed by developers when reviewing refactorings. Our results reveal that reviewing refactoring changes take longer to be completed compared to non-refactoring changes, and developers rely on a set of criteria to develop a decision about accepting or rejecting a submitted refactoring change, which makes this process to be challenging.

For future work, we plan on conducting a structured survey with software developers from both open-source and industry. The survey will explore their general and specific review criteria when performing refactoring activities in code review. This survey will complement and validate our current study to provide the software engineering community with a more comprehensive view of refactoring practices in the context of modern code review. Another interesting research direction is to link refactoring-related reviews to refactoring detection tools such as the Refactoring Miner [87] or RefDiff [74] to better understand the impact of these reviews on refactoring types specifically.

REFERENCES

[1] [n. d.]. *OpenStack*. https://review.opendev.org/640051
[2] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 2020. 30 Years of Software Refactoring Research:A Systematic Literature Review. arXiv:2007.02194 [cs.SE]
[3] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox. In *2021 IEEE/ACM 43rd*

International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 348–357.

[4] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the documentation of refactoring types. *Automated Software Engineering* 29, 1 (2022), 1–40.

[5] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology* (2021), 106675.

[6] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *International Workshop on Refactoring-accepted. IEEE*.

[7] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2021. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software* 171 (2021), 110821.

[8] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.

[9] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. 2021. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications* 167 (2021), 114176.

[10] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. 2021. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process* (2021), e2395.

[11] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. 2020. How Do Developers Refactor Code to Improve Code Reusability?. In *International Conference on Software and Software Reuse*. Springer, 261–276.

[12] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. Refactoring for Reuse: An Empirical Study. *Innovations in Systems and Software Engineering* (2021), 1–31.

[13] Everton LG Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 751–754.

[14] Everton LG Alves, Myoungkyu Song, Tiago Massoni, Patrícia DL Machado, and Miryung Kim. 2017. Refactoring inspection support for manual refactoring edits. *IEEE Transactions on Software Engineering* 44, 4 (2017), 365–383.

[15] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *International conference on software engineering*. 712–721.

[16] Alberto Bacchelli, Michele Lanza, and Romain Robbes. 2010. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 375–384.

[17] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *International Conference on Software Engineering-Volume 1*. 134–144.

[18] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 104–113.

[19] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *Proceedings of the 11th working conference on mining software repositories*. 202–211.

[20] Aline Brito, Andre Hora, and Marco Tulio Valente. 2020. Refactoring Graphs: Assessing Refactoring over Time. *arXiv preprint arXiv:2003.04666* (2020).

[21] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. 2021. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing* 100 (2021), 106908.

[22] Kenneth L Clarkson and Peter W Shor. 1989. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry* 4, 5 (1989), 387–421.

[23] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (1993), 494.

[24] Flavia Coelho, Tiago Massoni, and Everton LG Alves. 2019. Refactoring-aware code review: a systematic mapping study. In *International Workshop on Refactoring*. 63–66.

[25] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton LG Alves. 2021. An Empirical Study on Refactoring-Inducing Pull Requests. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[26] William Jay Conover. 1998. *Practical nonparametric statistics*. Vol. 350. John Wiley & Sons.

[27] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*. IEEE, 275–284.

[28] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. 2020. On the relationship between refactoring actions and bugs: a differentiated replication. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 556–567.

[29] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. 2007. Refactoring-aware configuration management for object-oriented programs. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 427–436.

[30] Emre Doğan and Eray Tüzün. 2022. Towards a taxonomy of code review smells. *Information and Software Technology* 142 (2022), 106737.

[31] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2021. An exploratory study on confusion in code reviews. *Empirical Software Engineering* 26, 1 (2021), 1–48.

[32] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. 2018. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering* 23, 6 (2018), 3346–3393.

[33] Olivier Gaudin. 2013. *Continuous Inspection A Paradigm Shift in Software Quality Management* (3 ed.). 10, Vol. 4. SonarSource.

[34] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. 2014. Towards refactoring-aware code review. In *International Workshop on Cooperative and Human Aspects of Software Engineering*. 99–102.

[35] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. 2017. Refactoring-aware code review. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 71–79.

[36] Bo Guo and Myoungkyu Song. 2017. Interactively decomposing composite changes to support code review and regression testing. In *Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 118–127.

[37] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel O Cinnéide, and Mohamed Wiem Mkaouer. 2021. An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications. (2021), 28–39.

[38] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. 2021. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology* 140 (2021), 106699.

[39] Ahmed E Hassan. 2008. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM symposium on Applied computing*. 837–841.

[40] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology* 95 (2018), 313–327.

[41] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[42] Yutaro Kashiwa, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi. 2022. An empirical study on self-admitted technical debt in modern code review. *Information and Software Technology* 146 (2022), 106855.

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

[44] Zarina Kurbatova, Vladimir Kovalenko, Ioana Savu, Bob Brockbernd, Dan Andreescu, Matei Anton, Roman Venediktov, Elena Tikhomirova, and Timofey Bryksin. 2021. RefactorInsight: Enhancing IDE Representation of Changes in Git with Refactorings Information. *arXiv preprint arXiv:2108.11202* (2021).

[45] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code reviewing in the trenches: Challenges and best practices. *IEEE Software* 35, 4 (2017), 34–42.

[46] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 151–162.

[47] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Working Conference on Mining Software Repositories*. 192–201.

[48] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[49] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 1–45.

[50] Audris Mockus and Lawrence G Votta. 2000. Identifying Reasons for Software Changes using Historic Databases.. In *icsm*. 120–130.

[51] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 171–180.

[52] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[53] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 23.

[54] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-based peer reviewers recommendation in modern code review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 367–377.

[55] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2019. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering* (2019).

[56] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 125–136.

[57] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–30.

[58] Luca Pascarella, Franz-Xaver Geiger, Fabio Palomba, Dario Di Nucci, Ivano Malavolta, and Alberto Bacchelli. 2018. Self-reported activities of android developers. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 144–155.

[59] Luca Pascarella, Davide Spadini, Fabio Palomba, and Alberto Bacchelli. 2019. On The Effect Of Code Review On Code Smells. *CoRR* abs/1912.10098 (2019). arXiv:1912.10098 http://arxiv.org/abs/1912.10098

[60] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 135.

[61] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. 2019. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 74–85.

[62] Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 350–357.

[63] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical Software Engineering* 27, 1 (2022), 1–43.

[64] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the Relation of Refactorings and Software Defect Prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (Leipzig, Germany) *(MSR '08)*. ACM, New York, NY, USA, 35–38. https://doi.org/10.1145/1370750.1370759

[65] Self-Affirmed Refactoring. 2022. *ReplicationPackage*. https://smilevo.github.io/self-affirmed-refactoring/refactoring-review/

[66] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 202–212.

[67] Peter C Rigby and Margaret-Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 541–550.

[68] Jeanine Romano, J Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data. In *Proceedings of the Annual Meeting of the Florida Association of Institutional Research*. 1–3.

[69] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131–164.

[70] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *International Conference on Software Engineering: Software Engineering in Practice*. 181–190.

[71] Emad Shihab, Zhen Ming Jiang, and Ahmed E Hassan. 2009. Studying the use of developer IRC meetings in open source projects. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 147–156.

[72] Danilo Silva, João Silva, Gustavo Jansen De Souza Santos, Ricardo Terra, and Marco Tulio O Valente. 2020. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* (2020).

[73] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[74] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.

[75] Gustavo Soares, Diego Cavalcanti, Rohit Gheyi, Tiago Massoni, Dalton Serey, and Márcio Cornélio. 2009. Saferefactor-tool for checking refactoring safety. (01 2009).

[76] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, 10–10.

[77] Gábor Szóke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 95–104.

[78] Dirk Taeger and Sonja Kuhnt. 2014. *Statistical hypothesis testing with SAS and R*. John Wiley & Sons.

[79] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 238–250.

[80] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 51.

[81] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *Working Conference on Mining Software Repositories*. 180–190.

[82] Patanamon Thongtanunam and Ahmed E Hassan. 2020. Review dynamics and their impact on software quality. *IEEE Transactions on Software Engineering* (2020).

[83] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2015. 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*. IEEE, 168–179.

[84] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*. 1039–1050.

[85] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150.

[86] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 329–331.

[87] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 483–494.

[88] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. 2021. Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 471–482.

[89] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 511–522.

[90] Clark Wissler. 1905. The Spearman correlation formula. *Science* 22, 558 (1905), 309–311.

[91] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. 2016. Mining the modern code review repositories: A dataset of people, process and product. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 460–463.

[92] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. 2015. Interactive code review for systematic changes. In *International Conference on Software Engineering-Volume 1*. 111–122.

[93] Xin Zhang, Yang Chen, Yongfeng Gu, Weiqin Zou, Xiaoyuan Xie, Xiangyang Jia, and Jifeng Xuan. 2018. How do multiple pull requests change the same code: A study of competing pull requests in github. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 228–239.