Code Review of Build System Specifications: Prevalence, Purposes, Patterns, and Perceptions

Mahtab Nejati, Mahmoud Alfadel, Shane McIntosh
Software REBELs
University of Waterloo, Canada
{mahtab.nejati,malfadel,shane.mcintosh}@uwaterloo.ca

***Abstract*—Build systems automate the integration of source code into executables. Maintaining build systems is known to be challenging. Lax build maintenance can lead to costly build breakages or unexpected software behaviour. Code review is a broadly adopted practice to improve software quality. Yet, little is known about how code review is applied to build specifications.**

In this paper, we present the first empirical study of how code review is practiced in the context of build specifications. Through quantitative analysis of 502,931 change sets from the Qt and Eclipse communities, we observe that changes to build specifications are at least two times less frequently discussed during code review when compared to production and test code changes. A qualitative analysis of 500 change sets reveals that (i) comments on changes to build specifications are more likely to point out defects than rates reported in the literature for production and test code, and (ii) evolvability and dependency-related issues are the most frequently raised patterns of issues. Follow-up interviews with nine developers with 1-40 years of experience point out social and technical factors that hinder rigorous review of build specifications, such as a prevailing lack of understanding of and interest in build systems among developers, and the lack of dedicated tooling to support the code review of build specifications.

***Index Terms*—build systems, build specifications, code review**

I. Introduction

Build systems orchestrate the process by which source code is transformed into deliverables. The build process is configured in build specifications, which describe the internal and external dependencies and set up the build dynamics. Build tools process build specifications to reason about the order- and configuration-dependent commands that must be invoked to produce the deliverables. Examples of such systems are Maven,[1] CMake,[2] and Gradle.[3]

Studies suggest that lax build maintenance can have severe and far-reaching consequences. A poorly maintained build system may underperform [1], which can hinder development progress as developers have to wait for slow feedback on their change sets. For example, Zhang et al. [2] reported that by skipping unnecessary re-compilations, builds can be accelerated up to 44.20%. A poorly maintained build system may also be prone to build breakages [3]–[5], which are disruptive for developers, who need to stop what they are doing and diagnose the failures. For example, Kerzazi et al. [6] reported that in a six-month period, 893–2,133 person-hours were lost due to build breakage in a commercial setting. Perhaps most seriously, a poorly maintained build system may introduce defects that impact the behaviour of the software in production environments. For example, the Knight Capital Group lost $440 million in less than an hour when an outdated piece of code in a module was resurrected due to an incorrect configuration setting [7].

Code review, i.e., the practice of developers critiquing each others' change sets, is a lynchpin of modern software quality assurance. Its technical and non-technical benefits are well-documented in the literature [8]–[13], including increases in code quality, peer mentorship, and knowledge transfer. However, it is still unclear whether such a well-established practice is applied to build specifications and what purposes it serves in this context. Spadini et al. [14] found that even in projects where code review is extensively performed, test files are less likely to be discussed. We suspect that this might also be true for build specifications.

Similar to test code, the review process for build specifications is perhaps even more important than production code. Build specifications are rarely (if ever) systematically tested themselves. Thus, the review process for build specifications is often the only quality assurance step that is applied. Prior work has also focused on the content of review discussions in different contexts [10], [14]–[17]. However, to the best of our knowledge, none have explored the problem in the context of build specifications. As build code is inherently different from source code (e.g., build code is often declarative rather than imperative in nature), the prior findings are unlikely to apply to the build context.

Therefore, we set out to conduct a mixed-method study [18] to address the following research questions:

RQ1. How rigorously are build specifications reviewed? Code review has been shown to improve code stability and early defect detection [8], [10], [19]. A more extensive review of changes increases the quality of the software [20], [21]. Prior studies have shown that as part of a software project, build specifications also need to be maintained and controlled for their quality [22]. To understand the extent to which code review is applied to changes to build specifications, we analyze 502,931 change sets from the large and active Qt and Eclipse communities. We compute popular measures

[1] https://maven.apache.org/
[2] https://cmake.org/
[3] https://gradle.org/

of review intensity [23] for build specifications and compare them to those of production and test code. We find that changes to build specifications are at least two times less likely to receive comments during code review, even when the change is solely focused on the build specifications.

RQ2. What are the purposes of the discussions on build specifications? In line with prior studies [10], [14], we aim to understand the purpose of the discussions on changes to build specifications. Bacchelli and Bird [10] discovered a set of categories that characterize the expectations and outcomes of code review. Spadini et al. [14] studied whether the same categories of concerns are discussed during the reviews of test code. Informed by those categories, we perform a closed coding [24] analysis on comments raised during the review of build specifications to investigate the purposes of the discussions and whether they are similar to those of production and test code. In some aspects, our study yields similar observations to the ones reported in the contexts of production and test code; however, in other aspects, our results show that the build context departs from the literature. For example, we observe that comments about defects in the code are 15.6–20.6 percentage points more prevalent in the build context when compared to the reported rates in the contexts of production and test code, where understanding and social communication categories are more prevalent than the defect category.

RQ3. What are the patterns of issues discussed during reviews of build specifications? Motivated by the fundamental differences in the nature of build specifications, aside from the purpose of the comments, we investigate what patterns of issues reviewers raise during the review of build specifications. We do so to understand whether maintenance of build specifications calls for attention to specific issues. We perform an open coding [25], followed by an open card sorting [26] analysis on the comments raised during the review of build specifications, resulting in a taxonomy of 14 issues, 10 of which are specific to build systems. We find that code evolvability and dependency-related issues are the most frequently raised issues.

RQ4. How do developers perceive the review of build specifications? We aim to understand developers' perspectives on code review of build specifications, the current policies for the practice, and the challenges that reviewers of build specifications face. To do so, we conduct semi-structured interviews [27], [28] involving nine practitioners, four with experience as core build maintainers and three from our studied projects. We thematically code [29] their responses to understand prevailing perceptions. We find that eight of the interviewees agree that the review of build specifications is critical for software projects. Unfortunately, they report that brunt of the reviewing burden is placed on a small team of dedicated build maintainers. Finally, interviewees mentioned a number of challenges that impede a more rigorous review of build specifications. A lack of developer attention to the maintenance of build specifications (eight interviewees), their lack of interest in and knowledge about the build technology being used (eight interviewees), and poor tooling to support build maintenance (six interviewees) were among the most poignant responses.

Our results show that discussions on changes to build specifications are infrequent, but often target a more pressing problem, i.e., defect detection. This, combined with our interview results, suggest that the criticality of well-maintained build specifications is still not fully perceived by the community. Moreover, we believe engaging more developers in the review of build specifications can have multiple positive impacts due to the non-technical benefits of code review including knowledge transfer and peer mentoring. It familiarizes developers with build specifications, leading to fewer mistakes in build specifications and knowledgeable reviewers. Finally, our study reveals that build maintainers are in need of dedicated tools that support the review of build specifications. For example, a tool to enforce the correct coding style for build languages was raised as a potential solution to the problem of lack of knowledge about build languages.

II. Related Work

In this section, we position our work in the current body of research on the topics of code review and maintenance of build systems.

A. Code Review

Prior studies have highlighted the benefits of code review in terms of both technical (e.g., code quality) and non-technical (e.g., team communication) properties [8]–[13]. Bacchelli and Bird [10] characterized the motivations for code review, as well as its outcomes, and discovered nine categories of purposes that code review serves, including code improvement, finding defects, and knowledge transfer. Rigby et al. [8] showed that up to 66% of the reviews successfully detect bugs. Mäntylä and Lassenius [15] reported that 75% of the discussions during code review raise concerns about the evolvability of the system and Beller et al. [16] found that 75% of the fixes that occur during code review successfully address maintainability concerns.

Noting the benefits of a well-conducted code review process, Spadini et al. [14] investigated whether this practice is equally applied to test code. They found that this shift of context from production to test code is associated with a decrease in the intensity of the code review process. They reported that when changes impact both production and test code, reviewers tend to favour review of production code over test code. However, they discovered that review of test code serves similar purposes to the review of production code. Inspired by Spadini et al.'s work, we study the code review

process in the context of build specifications to answer similar research questions.

B. Maintenance of Build Systems

Studies have shown that extensive effort goes into the maintenance of build systems. For example, Kumfert and Epperly [30] showed that build systems impose up to 30% of overhead in maintenance effort due to the maintenance of build specifications. However, despite the costs of build maintenance, its elimination can be even more cumbersome as poor maintenance of the build specifications can have far more expensive consequences. It can slow down the build process [1], lead to build breakages [3]–[5], or result in erroneous behaviour of the software.

Cao et al. [31] argued that a slow build negatively affects the developers' productivity and hinders the development progress. They proposed an approach to forecast build durations to aid developers in their scheduling of tasks. Other studies aimed to alleviate this issue by accelerating the build process through eliminating false dependencies from the build specifications [32], [33], eliminating redundant compilations [2], [33], or delaying test dependencies [34].

Kerzazi et al. [6] quantified the time developers expend to address build breakages and reported that in a six-month period, 893–2,133 person-hours were spent on this in a commercial setting. To mitigate this issue, studies have focused on automatically fixing build breakages. Macho et al. [22] focused on dependency-related breakages and managed to automatically address 54% of such build breakages. Hassan and Wang [35] used historical data on build specification patches and proposed an automated approach based on build-fixing patterns.

The detection of build defects has also been studied. Bezemer et al. [36] proposed an approach that could detect unspecified internal dependencies of a project by analyzing the specified dependencies in the build specifications and a concrete model of files in the project. Sotiropoulos et al. [37] proposed an approach that analyzed the dynamic behaviour of the build to find defects in build specifications.

Much of the aforementioned work focused on mitigating the consequences of poorly maintained build specifications. Our work complements prior studies by investigating the extent to which code review is leveraged to improve the maintenance of build specifications

III. Data Preparation

In this section, we describe how we prepare a meaningful corpus of data to address our research questions. Figure 1 provides an overview of the data preparation process, which is composed of three main steps: (A) selecting the subject communities, (B) collecting review data from the communities, (C) cleaning review data, and (D) grouping the changed files by their types. We elaborate on each of these below.

TABLE I: An overview of the subject communities.

Project	# Changes	# Unique Files	# Comments	# Reviewers
Eclipse	163,702	957,091	182,730	1,667
Qt	339,229	1,049,480	565,429	2,596

A. Select Subject Communities

To select our subject communities, we follow Spadini et al.'s [14] approach. We select communities that (1) perform code review intensively, (2) incorporate Gerrit as their main code review tool, and (3) review build specifications and test code. Gerrit is a modern code review tool for Git-based projects. This tool allows for a traceable code review process [20]. With the exception of the build specifications condition, Spadini et al. [14] applied the same criteria and studied Eclipse, Qt, and OpenStack communities. The first two communities satisfy our build specifications condition. However, OpenStack is a Python-based project and does not include build specifications. Therefore, we focus our analysis on Eclipse and Qt.

B. Collect Review Data

We extract the review data from the Gerrit instances of our subject communities using the `Gerrit REST API`.[4] We issue a request for each of the review records from initial adoption of Gerrit (2009 for Eclipse and 2011 for Qt) to the time of our data collection (March 1st, 2022). Fewer than 1% of our issued requests failed. Our inspection of the failure logs reveals that the original review records have been deleted from the Gerrit database. For each review, we store (meta)data about the status of the review (e.g., merged, abandoned, open), as well as the lists of modified files, reviewers, and comments with their exact location in the files.

C. Clean Review Data

To prepare our data for analysis, we perform a set of data cleaning steps. First, we select the `merged` reviews, since changes with other statuses are still subject to ongoing discussions. We only study changes that have been reviewed by at least one reviewer [38], [39]—comments recorded by bots[5] and the author of the change are omitted from our analysis. We filter out mega changes, i.e., changes affecting more than 50 files [14], [38], [39]. This is because studies have shown that code review is most effective when performed on small, independent, and atomic changes, which allows the reviewers to scrutinize the code closely [40]. We select the threshold of 50 affected files in line with prior work [14] and conduct a sensitivity analysis to confirm the suitability of the threshold in our dataset. Our investigation yields that among the merged changes with at least one reviewer, only 2.1% affect 50 or more files (mega-changes), 48% of which affect build specifications (1% of changes). For a threshold between 40 and 60 affected files, this number varies between 2.7% and

[4] https://gerrit-review.googlesource.com/Documentation/rest-api.html

[5] See the Data Cleaning section of our replication package[7] for the list of detected bot accounts.

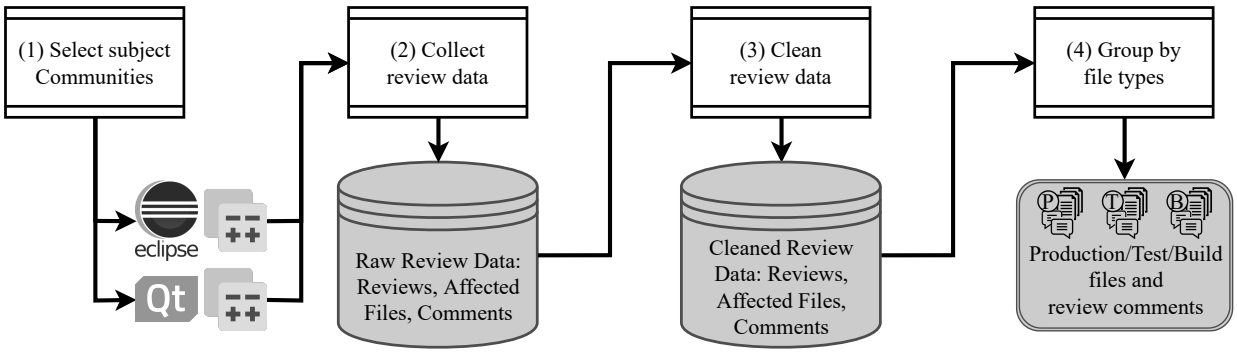

Fig. 1: An overview of our data preparation workflow.

1.7%. In fact, the number of affected files in changes follows a power-law distribution and only 4.6% of the changes affect 25 or more files.

Since each review may contain one or more revisions, and the same file can be modified in multiple versions, we focus on whether a file has been modified during a review at any revision rather than counting the number of times it is changed within a single review. However, we retain the comments from all versions, since these discussions are all relevant to the review. Finally, we trim leading and trailing whitespace from comments prior to measuring their length, since whitespace does not add meaning. Table I shows descriptive statistics about our corpus after this data cleaning step has been applied.

D. Group by File Types

Our quantitative analysis requires data on different file types. For a comprehensive comparison with prior work [14], we classify files into three categories: production, test, and build specification files. Any file that is not classified under one of these categories is disregarded.

To identify file types, we use file extension, naming and location conventions. First, in line with Spadini et al. [14], we assume that files containing source code are production and test files. For each project, we only consider files written in the project's primary programming language, i.e., `C++` for Qt and `Java` for Eclipse. To do so, we match file extensions `.h` and `.cpp` for Qt and `.java` for Eclipse as source code files. Then, following Spadini et al.'s approach, we use naming and file location conventions in the projects to design regular expressions that match test files. Source files that do not match this regular expression are labelled as production code. We evaluate our regular expression by applying it to the data in the replication package from Spadini et al.'s study.[6] We find that our approach yields similar production and test proportions (within 0.30–0.65 percentage points) as those reported in their prior work [14]. We suspect that the small discrepancies are likely due to the additional data that has accrued since the paper was published.

Lastly, to identify build specifications, we follow naming conventions. As we only consider the primary programming language in each project, we do the same for build systems. Eclipse is built using Maven, so we identify `pom.xml` files as build specifications. Qt was built using QMake, but is now built using CMake. For QMake files, we match all files with `.pri` and `.pro` extensions. For CMake, we match `CMakeLists.txt` and `.cmake` files.

[6] https://doi.org/10.5281/zenodo.4075318

IV. Quantitative Analysis (RQ1)

In this section, we aim to understand how rigorously build specifications are reviewed. Below, we first describe our approach to the quantitative analysis and then present our results.

A. Approach

To answer RQ1, we measure and compare the comment frequency and discussion intensity of production, test, and build code. We employ metrics that have been previously applied in the literature [23]. To measure discussion frequency, we report the proportion of files with comments and present the odds ratios for clear comparison among pairs of file types. As a measure of discussion intensity, we analyze the number of comments and their average length.

To draw comparisons and valid conclusions, we perform an analysis over a grid of settings along two dimensions. First, as our initial inspections over the corpus suggest that the majority of modifications to the files are not discussed (only 11.79% of modifications to the subject types of files receive comments), we perform our analysis in the following two settings:

- **Discussed reviews**: This setting selects only the reviews with at least one comment on one of the subject file types. This analysis controls for the skew from the large portion of non-discussed reviews, i.e., reviews in which all files have zero comments.
- **All reviews**: This setting does not omit any reviews.

Second, we conjecture that reviewing multiple software artifacts simultaneously may alter reviewer behaviour. Hence, we conduct our analysis in another two settings:

- **Cross-artifact setting**: For each comparison between file types, we only consider reviews that modify both of the file types. This will indicate how differently file types are treated by reviewers when their changes are coupled.
- **Isolated setting**: We focus on reviews that modify only one file type.

B. Results

Tables II and III provide an overview of the results of our quantitative analysis. We perform comparisons between pairs of production code, test code, and build specifications over the grid of our settings. We structure the discussion of results around our key observations. Detailed results for all settings are included in our online appendix.[7]

Discussion occurrence frequency. Reviews of both production and test code are at least two times more likely to include comments when compared to reviews of build specifications. This observation holds in both cross-artifact and isolated settings. Table II shows that the proportions of reviews receiving comments in the cross-artifact setting are 14.18% against 7.18% and 12.45% against 6.59% for production and test code, respectively, when compared against build specifications. Similarly, in the isolated setting, only 5.87% of the reviews of the build specifications receive comments, while 16.61% of the reviews of production code and 12.52% of the reviews of test code in this setting receive comments. The odds ratios reported in Table III further support this observation, ranging from 2.02 to 2.30 in all settings for production and test code compared against build specifications.

> ***Finding 1.*** *Build specifications are at least two times less frequently discussed during code review than production and test code, even when build specifications are the sole focus of the review.*

Discussion intensity. We estimate discussion intensity using the average length of discussion (in terms of the number of comments) and the average length of the comments. When analyzing the intensity of the discussions, we notice that the differences are much smaller in the *all reviews* setting than the *discussed reviews* setting. This is due to the multitude of the non-discussed reviews (only 11.79% of the modifications on production code, test code, or build specifications receive comments) for which all file types have zero comments. Therefore, we focus the remainder of the analysis of discussion intensity on the *discussed reviews* setting.

In the cross-artifact setting, the discussions of production and test code are composed of more comments than those of build specifications; however, the comments tend to be shorter. In the isolated setting, there is no statistically significant difference in discussion intensities, i.e., although we observe that in the isolated setting, review discussions about build specifications are less frequent than that of production and test code, when a discussion does occur, the intensity of the discussion is not substantially different for any of the file types.

In the cross-artifact setting, discussions on production and test code take 5.41 and 4.59 comments, respectively, whereas build specifications take 3.00 and 2.68 comments, respectively. The differences are statistically significant (Wilcoxon signed rank test, $p < 2.0 \times 10^{-16}$). When build specifications and production code are compared, the difference is large (Cliff's delta = 0.7465) and when build specifications and test code are compared, the difference is medium (Cliff's delta = 0.4199). In terms of the average length of comments in the cross-artifact setting, the comments on build specifications are longer than those of production code (95.24 for build vs. 89.39 for production) and test code (101.93 for build vs. 99.58 for test). The differences are statistically significant in the cross-artifact setting (Wilcoxon signed rank test, $p < 2.0 \times 10^{-16}$). For the comparison with production code, the difference is large (Cliff's delta = 0.6788) and when compared with test code, the difference is medium (Cliff's delta = 0.3804).

However, results from the isolated setting show that for both production and test code, the differences in both heuristics are not statistically significant (Mann-Whitney U test, $p > 0.05$) with one exception for the comparison of the number of comments on production code and build specifications where the difference is still negligible (Mann-Whitney U test, $p = 3.04 \times 10^{-9}$, Cliff's delta = 0.0708).

> ***Finding 2.*** *The discussions on build specifications take fewer but lengthier comments when changes to build specifications are coupled with production or test code. When the reviews modify only one type of file, the differences in both heuristics of discussion intensity are not statistically significant.*

V. Qualitative Analysis for Purposes (RQ2)

In this section, we aim to understand the purposes of the discussions on build specifications. To do so, we analyze the content of the review comments of build specifications to characterize the concerns of the reviewers. Below, we describe our approach, followed by our results.

A. Approach

Table IV provides an overview of the nine categories that Bacchelli and Bird [10] used to characterize the purposes of code review. Spadini et al. [14] classified comments from the reviews of test code using the same categories. In this study, we also analyze the content of the comments to characterize the purposes that reviews of build specifications serve.

To do so, we first draw a random sample of the comments raised during the reviews of production code, test code, and build specifications. We then apply closed coding [24] to our sampled set of comments to label them with previously discovered categories of purposes in code review [10]. The following describes the sampling and closed coding steps in more detail.

Sampling: We focus on the discussion initiations and exclude comments made in reply to other comments. This is because the first comment of a thread raises the main concerns of the discussion and sets the topic of the discussion. We also exclude comments that refer to the content of other comments, e.g., "Same here", "Ditto", because their content does not clarify

[7] https://doi.org/10.5281/zenodo.7042930

TABLE II: Prevalence of reviews and their intensity in different file types and settings.

Setting	File Type	Avg. # of reviewers	# Files	# Files w/ Comments	Percent of Files w/ Comments	# Files wo/ Comments	# Comments	Avg. Discussion Length (# Comments)	Avg. Length of Comments
Cross-artifact (P vs. B)	Production	2.76	122,461	17,366	14.18	105,095	93,946	5.41	89.39
	Build		35,763	**2,567**	**7.18**	33,196	7,692	**3.00**	**95.24**
Cross-artifact (T vs. B)	Test	2.90	16,789	2,091	12.45	14,698	9,524	4.59	99.58
	Build		17,234	**1,136**	**6.59**	16,098	3,048	**2.68**	**101.93**
Cross-artifact (P vs. T)	Production	2.62	137,936	21,214	15.38	116,722	108,043	5.09	106.05
	Test		52,086	6,214	11.93	45,872	24,033	3.87	101.01
Isolated	Build	**2.52**	52,344	**3,070**	**5.87**	49,274	10,310	**3.36**	**108.26**
	Production	2.17	341,902	56,775	16.61	436,846	205,244	3.62	98.74
	Test	2.13	23,464	2,937	12.52	20,527	11,106	3.78	98.32

TABLE III: Odds ratios of receiving comments by different file types in different comparison settings.

Setting	Pairs	Odds Ratio	Upper 95% CI	Lower 95% CI
Cross-artifact	P vs B (B vs P)	**2.14 (0.47)**	2.23 (0.49)	2.05 (0.45)
	T vs B (B vs T)	**2.02 (0.50)**	2.17 (0.53)	1.87 (0.46)
	P vs T (T vs P)	1.34 (0.75)	1.38 (0.77)	1.30 (0.72)
Isolated	P vs B (B vs P)	**2.09 (0.48)**	2.17 (0.50)	2.01 (0.46)
	T vs B (B vs T)	**2.30 (0.44)**	2.42 (0.46)	2.18 (0.41)
	P vs T (T vs P)	0.91 (1.10)	0.95 (1.15)	0.87 (1.06)

the concern. These comments were previously categorized as *Miscellaneous* by Spadini et al.[6] (57 of the 600 samples).

Closed Coding: We apply a closed coding analysis [24] on the randomly sampled set of comments from the reviews of build specifications. We take the set of labels proposed by Bacchelli and Bird [10] as our code booklet. The coding task is performed by the first two authors as coders. When labelling the comments, coders focus their attention on the content of the comment and refer to the changes to build specifications when additional context is required to make a category assignment. First, to gain a solid understanding of the labelling guidelines and label definitions, the coders discuss a set of 113 comments and collaboratively label them.[8] Then, as an evaluation of the agreement among the coders, the first and second authors independently label 387 comments in batches of 37–50 samples. Note that our sample size of 500 review comments is similar to the samples used in the exploratory analyses in prior studies (570 comments on production code [10] and 600 comments on test code [14]). After labelling each batch, the coders discuss the conflicts to reach an agreement. In cases where a consensus could not be reached, the last author casts the deciding vote. The coders achieve a Cohen's Kappa score of 0.8647, indicating a strong agreement among the coders [41].

Coding was completed in multiple rounds. First, as the coding process takes weeks to complete and the discussions lead to more refined labelling guidelines, coders must make sure that the lessons learned from later coding activities are also applied to earlier coding activities. To ensure this, after completing a coding pass, the entire set of comments are re-examined and labels are adjusted as necessary. Second, as the results show a substantial deviation from previous studies for the understanding category, we perform a third coding pass. Inspection of Spadini et al.'s replication package[6] shows that they took a more lenient approach when applying this label. Therefore, during the third coding pass, we apply this more lenient approach to produce results that are more comparable to the prior work.

[8]This set of 113 comes from the exploratory coding process described in Section VI, where we reach saturation after coding 113 comments.

TABLE IV: Prevalence of purposes in code review.

Purpose	f_{Build}	$f_{Prod} - f_{Build}$	$f_{Test} - f_{Build}$
Code Improvement	41.2	-12.2	-6.2
Defect	**29.6**	**-15.6**	**-20.6**
Understanding	22.6	-0.6	+9.4
Knowledge Transfer	5.4	-2.9	-1.4
Review Tool	1.2	+1.8	-1.2
Social Communication	0.0	+16.0	+11.0
External Impact	0.0	+5.0	+0.0
Testing	0.0	+5.0	+0.0
Miscellaneous	0.0	+7.0	+10.0

f_x refers to the frequency of comments with the subject purpose appearing in review of x code.

B. Results

Table IV provides a summary of our results from the closed coding and compares them with the reported ones in the literature for production [10] and test [14] code comments. We did not observe any comments related to *social communications*, *external impact*, or *testing*. Finally, we exclude *miscellaneous* comments from this analysis. We account for this by scaling the reported rates for production and test code over only the first five categories—those that have also been discovered in the context of build specifications. We structure our discussion of results around the following observations.

The defect category occurs 15.6–20.6 percentage points more often in the build setting than was observed in the production and test setting of prior work. Unlike review of production and test code, in reviews of build specifications, the defect category tends to occur more frequently, appearing in 29.6% (36.2% with our stricter "understanding" coding approach) of the reviews, 9.7 percentage points more than production code and 18.3 percentage points more than test code even when accounting for categories that we did not encounter.

The most prevalent concern is code improvement (41.2%, 47.6% with our stricter "understanding" coding approach).

These comments aim to improve the readability and maintainability of the code. This conforms with the findings of both Bacchelli and Bird [10] and Spadini et al. [14], where it is reported that code improvement is the purpose of the majority of the comments made on production and test files. While the exact rates fluctuate, accounting for the categories we did not encounter, the differences are only 0.1 percentage points lower for production and 2.6 percentage points higher for test code.

Studies have previously reported that code improvement and the maintainability of the software are the most prominent outcomes of code review [10], [15], [16]. It is not surprising that build specifications also follow the same pattern as they are part of a software project.

> ***Finding 3.*** *Discussion of defects is more prevalent in reviews of build specifications than reviews of production and test code. However, code improvement remains the most prevalent category of comments.*

VI. Qualitative Analysis of Issue Patterns (RQ3)

To understand the patterns of issues that are raised during the review of build specifications, we inspect the content of the comments on build specifications. In the following, we describe our approach and results.

A. Approach

We perform open coding [25] on a randomly sampled set of review comments on build files. Then, we apply open card sorting [26] to the codes to construct a taxonomy of the issue types discovered. The following describes the sampling, open coding, and open card sorting processes.

Sampling: We use the same sample of comments from our content analysis in Section V. Thus, we use a sample of 500 review comments, 113 of which we analyzed through the exploratory coding and the remaining 387 during the evaluation process.

Open Coding: Our open coding process is composed of code discovery and code evaluation steps. As was done in Section V, when making coding decisions, we prioritize the content of the comments over the content of the change to the build specifications. In this analysis, we encountered instances where the content of the comment did not contain enough information to identify the issue type. These comments were tagged as "not self-explanatory" (199 of the comments, 39.8%, see this comment[9] for an example). In such cases, we defer our attention to the changes in the build code and the discussion in the comment to identify the issue type.

For the code discovery step, the first and second authors randomly draw and inspect comments on changes in build specifications, one at a time. For each comment, the coders discuss the content of the comments to label them with an issue pattern, creating new patterns as they emerge. Throughout the code discovery process, coders define and refine the patterns through discussions, iteratively establishing coding guidelines. The coders continue to draw and inspect comments until they reach saturation [42] for the issue patterns. We set our saturation criterion to 50 consecutive comments where new patterns are not discovered. We reached saturation after inspecting 113 comments.

To evaluate the reliability of the coding approach, after saturation is achieved, the coders independently label 387 comments in batches of 37–50 samples. Similar to the evaluation for the closed coding process (Section V), after labelling each batch, the coders discuss disagreements in search of a consensus. In cases where a consensus is not reached, the last author casts the deciding vote. During this code evaluation step, despite reaching our saturation criterion, new issue patterns may emerge. In our case, only two new issue patterns emerged during the code evaluation step. The coders achieve a Cohen's Kappa score of 0.9093, indicating a strong agreement among the coders [41].

Finally, to mitigate errors in the coding process, we perform coding in multiple passes. In particular, coders do so to ensure that the labels that emerged later in the coding process do not apply to the comments coded earlier in the process. Moreover, as coders progress through the set of comments and the patterns become more clear, we iteratively refine the labels. These iterative improvements may decompose overly general patterns to more specific ones or merge overly specific patterns into more applicable general patterns. Such improvements to the set of patterns call for re-analyzing the previously labelled comments to ensure consistency and integrity. A final pass to correct the miscoded comments concludes the process. Although the patterns are not inherently mutually exclusive, we find that multi-pattern comments are rare—we only apply more than one pattern to two of the inspected comments. In these cases, we select the more prominent label or an arbitrary one and note the instances in our labelling sheet.[7]

Open Card Sorting: Similar to prior work [6], [10], [43]–[45], we apply open card sorting to construct a taxonomy of issue patterns that reviewers focus on when reviewing changes to build specifications. During the card sorting process, we first group patterns based on topic similarity and then assign descriptive names to each of these groups to refer to their higher level category.

B. Results

Table V presents an overview of our taxonomy, which is composed of 14 patterns that span six categories. Examples are available in our online appendix.[7] Below, we describe each category in more detail.

(C1) Evolvability. We find that 35.0% of the comments focus on evolvability of build specifications. Among the comments in this category, 77.71% are concerned with the maintainability of build specifications (IP1). Such comments discuss the coding style and readability, redundant code, and the placement and visibility of methods. The remaining 22.29% of the patterns in this category discuss documentation issues

[9] https://codereview.qt-project.org/c/qt/qtdeviceutilities/+/208369/3/src/settingsui/settingsuiplugin/settingsuiplugin.pro#29

TABLE V: Taxonomy of issue patterns in build code review.

Category	Count	Frequency
C1: Evolvability	*175*	*35.0%*
Maintainability (IP1)	136	27.2%
Documentation Issues (IP2)	39	7.8%
C2: Externals	*108*	*21.6%*
Platform Configuration (IP3)	46	9.2%
Tool Configuration (IP4)	30	6.0%
Libraries and Plugins (IP5)	16	3.2%
Artifact Versioning (IP6)	16	3.2%
C3: Behavioural	*93*	*18.6%*
Dynamic Settings (IP7)	90	18.0%
Logging (IP8)	3	0.6%
C4: File System	*73*	*14.6%*
Logical File System (IP9)	58	11.6%
Physical File System (IP10)	15	3.0%
C5: Build Language	*38*	*7.6%*
Syntactic Issues (IP11)	29	5.8%
Semantic Issues (IP12)	9	1.8%
C6: Miscellaneous	*13*	*2.6%*
Patch Content Reorganization (IP13)	11	2.2%
Review Tool (IP14)	2	0.4%

(IP2) such as misleading, missing, or unclear documentation, all of which could impede the evolution of build specifications.

(C2) Externals. We find that 21.6% of the comments on build specifications focus on factors that are external to the project. External factors include artifacts outside of the project that are used in or interact with the software to satisfy needs. Platforms, both the hardware (e.g., the architecture of CPU) and software (e.g., operating system, language toolchain), are common examples of such externals. Configuring the build for different platforms (IP3) is the focus of 42.59% of the comments in this category. In 27.79% of the comments, reviewers raise concerns about configuring external tools (IP4) by changing their dynamic behaviour or invoking the tool from within the build system. Sixteen of the 30 comments are specifically about configuring the compiler toolchain. Issues configuring libraries and plugins (IP5) represent 14.81% of the comments in this category. Finally, versioning of any of these artifacts (IP6) represents the remaining 14.81% of the comments in this category.

(C3) Behavioural. In 18.6% of the comments, reviewers focus on the behaviour of the build system, which is configured with (environment) variables and flags that alter the commands that must be invoked for the build process to complete. The majority of these comments (96.77%) are concerned with the setting and configuration of this dynamic behaviour (IP7). The remaining 3.23% focus on logging (IP8), a passive behaviour that does not affect artifacts produced by the build system.

(C4) File System. We find that 14.6% of the comments discuss issues related to the project's file system schema and content, both on the logical and physical layer. Comments discussing the file system on a logical layer (IP9) are concerned with requiring project modules as sources and headers for the build invocations and consist of 79.45% of the comments in this category. The physical layer of file systems (IP10) refer to the existence of files and directories in certain locations and cover the remaining 20.55% in this category. Such comments discuss moving, renaming, deleting, or creating files or directories in the project.

TABLE VI: Interviewees' experience (in years) and working context (OSS project or company)

ID	Years of Experience as				Working Context	Build Maint.
	Dev.	Rev.	Build Dev.	Build Rev.		
P1	20	13	13	13	Qt	Yes
P2	10	7	7	7	Co. A	Yes
P3	40	15	10	4	Qt	Yes
P4	11	5	8	3	OSS	No
P5	8	8	3	3	Qt	Yes
P6	5	2	2	1	Co. B	No
P7	4	2	1	1	Co. C	No
P8	14	5	1	0	OSS	No
P9	1	1	0	0	Co. D	No

(C5) Build Language. In 7.6% of the comments, reviewers point out issues related to the use of the build language. These issues consist of syntactic issues (IP11) and semantic issues (IP12), pointed out in 76.32% and 23.68% of the comments in this category, respectively. Syntactic issues include the best practices for assignments, invocations, and conditionals. For semantic issues, since most build languages are declarative by nature, issues that could cause a failure in an imperative programming language, e.g., function call before definition, are mostly considered improvements towards better readability and maintainability of the system. Therefore, semantic issues in this context refer to cases, such as the use of undefined identifiers and redefining existing identifiers.

(C6) Miscellaneous. Finally, we categorize the comments that discuss the ordering and the content of patches (IP13), as well as those focusing on misuse of review or version control tools (IP14) under the miscellaneous category, which account for only 2.60% of the comments. These comments are not particularly discussing build-specific concerns and mainly focus on the atomicity of the changes and the order of patches.

> ***Finding 4.*** *Evolvability and dependency-related issues are the most frequent patterns of issues raised during the review of build specifications.*

VII. Interview Analysis for Developers' Perceptions (RQ4)

To further explain our observations and complement the results from our prior analysis, we set out to understand how developers perceive the review of build specifications. To do so, similar to prior work [10], [14], we conduct semi-structured interviews, focusing on the code review process and our findings as the themes of the interview questions [14]. We then transcribe and code interviewee responses to reveal common perceptions among developers about the review of build specifications. Below, we present our interview structure and analysis approach, followed by our findings.

A. Approach

We conduct semi-structured interviews [27], [28] that span the following four dimensions: (1) the policies and practices of the code review process in which the interviewee participates, (2) specific policies and practices for the review of build specifications, and interviewee reactions to our observations from (3) RQ1 (Section IV) and (4) RQ2 (Section V). Instead of sharing our results for RQ3, we indirectly investigate whether the patterns of issues that we have detected are among the specific issues that reviewers pay attention to when reviewing build specifications. Focusing on these four dimensions as the outline of the interviews, we leverage the semi-structured nature of the interviews to allow interesting responses to prompt follow-up questions that explore emergent themes. A full description of our interview protocol is included in our online appendix.[7] Once all four topics had been covered, interviewees were invited to share any thoughts or personal experiences before we conclude the interview.

To recruit interviewees, we employ the following procedure. We first obtain ethics approval to conduct the study.[10] Then, we invite members of the subject communities to participate using posts on their developer mailing lists. We invite other practitioners to participate through posts on social media. We obtain consent to participate in the interviews through an online form. We then schedule the interviews based on the interviewees' availability.

Each online interview lasted for 20 to 60 minutes. We record the audio of the interviews upon the interviewees' consent. We then transcribe the interviews for further analysis. To analyze the responses, we employ a thematic analysis [29]. First, we code the relevant pieces of information in the responses with a brief description of their content that captures the core idea. Then, we group codes according to common themes. Finally, to ensure the quality of the generated themes, we perform an editorial pass over the generated themes and codes. We also account for the interviewees' level of expertise and engagement with the maintenance of build specifications, i.e., we make note of the differences in perceptions of build maintainers and other developers.

Participants. We interview nine practitioners with different levels of experience in the development, maintenance, and review of build specifications. Table VI provides an overview of the demographics of our interviewees. Three of the interviewees are active members of the Qt community. Moreover, based on the positions interviewees hold in their projects, we label them as build maintainers and developers. Build maintainers are experts who are responsible for the maintenance of build specifications. Developers work with and review build specifications, but do not self-identify as experts.

B. Results

Based on our analysis of the interviews, we extrapolate three main themes.

[10]We obtained clearance for this study from the University of Waterloo Research Ethics Board (application # 44388).

Importance of review of build specifications. Eight of the 9 interviewees reported that review of the build specifications is just as crucial, if not more so, than the review of other software artifacts. In particular, interviewees pointed out that review of build specifications helps to mitigate build breakages. Indeed, P9 explained "*if [builds] break, they block the pipeline*", propagating the problem to all other components of the system. All interviewees either explicitly or implicitly pointed to a lack of expertise and knowledge in build systems, which could lead to more defects in changes to build specifications. In particular, P3 stated that "*Anybody who changes the code does have to change the build code from time to time [. . .] because it has contact with every part of the code [. . .] and most people aren't familiar with it [...]*". This makes review of build specifications even more crucial for the purpose of finding defects, especially since running a build can be slow (P3, P4, P5) and "*the only way to test build changes is to run the build and see if the build is successful*" (P5)—a brute-force testing method that was mentioned by all of the interviewees.

A Qt build maintainer, P5, saw beyond build breakages and clarified that even if the build is successful, this is "*[...] an indicator that maybe this is an OK change.*" P5 alluded to concerns of the decay of maintainability of build specifications in the absence of adequate reviews—a concern that was echoed in the responses of three of the 4 build maintainers (P2, P3, P5), as well as P4. Only one participant (P7) objected to rigorously reviewing build code, arguing that "*All I care about is the ability of building my program*". In P7's development setting, every pull request is built prior to landing in production, and any serious problem will result in a build breakage. P4 held a similar opinion for small projects, but stated that "*Build files are not the center of the attention, but if you want to have a large long-term project, you definitely need to get it right*".

> ***Finding 5.*** *Eight of the 9 interviewees acknowledged that reviewing build specifications is at least as important as reviewing production and test code. Main concerns for those reviews included avoiding costly build breakages (especially in projects that are slow to build) and the maintainability of build specifications.*

Current policies for review of build specifications. Eight of the 9 interviewees mentioned that *significant* changes to build specifications are always reviewed by *build maintainers*; however, other developers rarely engage in this process. P5 says that in teams "*if [developers] see that there is a build change, they might just add us, [...] the build maintainers, and defer the [review] work to us*". P1, P3, and P5 clarified that minor and *straightforward* changes are reviewed by the maintainers of the code modules. When a change to the build specifications is substantial enough to affect users or the whole build process, the change will also be reviewed by a build maintainer. P7 maintained that due to the complexities of build specifications, reviewing build files is "*inefficient*" and it is best

to simply rely on the outcome of the build.

The interviewees also explained that when reviewing build files, they search for specific patterns of issues tailored to build specifications. Among the concerns were changes in external dependencies and their versions, module configurations and their versions, changes to (environment) variables, as well as code style and naming conventions.

> ***Finding 6.*** *Eight of the 9 interviewees reported that review of build specifications is mainly performed by build maintainers who are familiar with the build system. When reviewing build specifications, build maintainers focus on build-specific issues, such as dependencies, versioning, and module configuration.*

Social and technical challenges. According to eight of our 9 interviewees (all four build maintainers and four of the 5 developers), one of the major challenges that hinders the review of build specifications is that most developers do not pay much mind to build systems and their maintenance. Build systems are not the primary focus of developers because they are not visible to the end user. Indeed, P1 stated that "*Build tools are seen as necessary evil and [developers] tend to ignore them until they need them*".

Moreover, all build maintainers stated that most developers do not take an interest in learning the build technology of the project, which is required for a deep review of build specifications as build languages are fundamentally different than source code languages. Indeed, seven of the 9 interviewees remarked about the lack of build system expertise leading to the lower heuristics of review intensity that we observe in RQ1 (P1, P2, P3, P4, P5, P8, and P9).

Seven of the 9 interviewees also point out that build files are inherently complex, especially when configured for cross-platform builds (P1, P3, P4, P5, P7, P8, and P9). The sheer scale of the build dependency graph (i.e., the data structure that build tools use to reason about the commands that should be (re-)invoked), its intra- and inter-module dependencies, external package dependencies, as well as platform- and configuration-specific behaviour make modifying build specifications a daunting task. Moreover, as running the build is the only way to test changes,[11] slow builds are also raised as a challenge.

Finally, the lack of dedicated tooling was a challenge brought up by six of the 9 interviewees (P1, P2, P3, P5, P6, and P7). Indeed, interviewees remarked about the need for automation of the build process to reduce the complexity of build specifications (P1, P2, P3, and P6) and for more tool support for build languages, such as linters and IDE plugins (P2, P5, and P7). Interestingly, when given the choice between solving the social problem of the knowledge gap among developers or the lack of better automated tools for build languages, P5 opted for "better tooling, because people come and go but tools stay".

> ***Finding 7.*** *Interviewees believe that a pervasive lack of interest in and knowledge about build systems, the complexity of build specifications, and lack of tools to support the maintenance of build specifications are prevailing challenges that complicate the review of build specifications.*

VIII. Threats to Validity

Below, we discuss the threats to the validity of our study.

A. Internal Validity

Our manual coding analyses introduce the subjectiveness of the inspectors to our results. To mitigate this, we follow common practices for all of our manual analyses [24]–[26], [29], [42] and run multiple passes on our results. Every disagreement is discussed in a team of three authors until consensus is reached. The coders achieve Cohen's Kappa scores of 0.8647–0.9093 for the manual coding tasks.

Moreover, it is possible that the list of purpose labels in RQ3 is not exhaustive. We chose to adhere to a closed-coding methodology to preserve comparability with the prior work. The coders did discuss borderline samples and considered alternative labels; however, similar to when the Bacchelli and Bird taxonomy [10] was applied to the testing context in prior work [14], we did not observe any emergent purpose labels specific to build specifications.

Furthermore, we interview nine practitioners, which inherently introduces subjectivity. To account for this, we solicited participation from a broad sample of developers with different levels of experience with build systems. We also account for their roles with respect to the build system as an indicator of their expertise when reporting our observations.

Finally, we address the discrepancies between our coding approach and the prior works' by inspecting the samples in their replication package if available[6] and running extra passes on the set of labelled comments to ensure our conformance with their labelling approach. We further account for this factor by reporting results over only the encountered categories, as our sampling approach excludes comments from the *Miscellaneous* category.

B. Construct Validity

We measure discussion intensity using heuristics such as the average number of comments per discussion and the average length of comments. Descriptive statistics often do not tell the whole story; however, the employed heuristics have been validated and applied in prior work [23]. We also investigate if the greater length of the comments on build specifications is attributed to the inclusion of lengthy code snippets in the comments. In our manual analyses, we observe that only 5% of the sampled comments contain code snippets. Given that this might also be the case for comments on production and

[11] While there are proposals in the literature about other testing approaches for build systems [36], [37], to the best of our knowledge, they have not yet found their way into practice-ready tools.

test code, we believe that this could not have a substantial impact on our results.

Finally, we verified our scripts by running them against the data from the replication package of prior work[6] and comparing our results on our collected data with the reported findings in their study [14]. We also make our replication package available online for fomenting future work.[7]

C. External Validity

In the first three RQs, we study two large and active communities, Qt and Eclipse. Projects in these two communities are in two different programming languages and are built with different build technologies, i.e., Maven and CMake. The Qt community also has historical data on the QMake build system. Nevertheless, these examples may not guarantee the generalizability of our findings. More specifically, the intensity of the results might vary from one system to another. Researchers are encouraged to reproduce our study considering other build systems and communities. All materials necessary to reproduce the analyses are included in our online appendix.[7]

IX. Conclusion and Practical Implications

In this paper, we perform a mixed-method study to explore the code review process in the context of build specifications. We first quantified the frequency and intensity of review discussions on build specifications. Then, we inspected these discussions to identify main purposes and patterns of the issues that reviewers raise. Finally, we interview practitioners to understand how they perceive and perform reviews of build specifications, as well as the challenges that they face.

Below, we distill three lessons for the community and propose the following implications based on them.

Both practitioners and researchers should be more aware of the importance of the review of build specifications. Our interviewees report that review of build specifications is essential for the correctness of the build system, as well as its maintainability and quality (Section VII). Moreover, we find that a non-negligible portion (36.2%) of reviews on build specifications are concerned with defects (Section V). Omitting a careful review of build specifications could result in builds that are likely to break, some of which being highly costly to fix. Even when the build is successful, without a rigorous review of build specifications, the maintainability of the build system will likely decay over time. On the other hand, our qualitative analysis (Section IV) shows that changes to build specifications are at least two times less likely to be discussed than production and test code.

Practitioners can reduce the review burden for build specifications by educating their teams about their build technology. Our interviewees broadly believed that there is a lack of knowledge in build systems among developers (Section VII). This can impact the review of build specifications in two ways. First, changes to build specifications by less knowledgeable developers are more likely to be prone to defects. Second, developers with little knowledge about build systems cannot contribute much to the reviews of build specifications, shifting the review burden on the small teams of build maintainers. Educating developers about the build technology in use would mitigate both concerns. Using code review as a means of knowledge transfer and engaging more developers in the review of build specifications can help with this education. Moreover, structuring educational efforts based on our discovered patterns of issues (Section VI) would be a promising direction for future work.

Researchers and tool developers can improve the review and maintenance of build specifications by focusing on the needs of build maintainers. Our interviewees noted a lack of dedicated tools for the development and maintenance of build specifications. They pointed out their needs spanning tools that support writing and reviewing code in build languages and enforce coding style (e.g., linters and IDEs), tools that minimize the complexity of build specifications through automation, and tools that accelerate builds. They also suggest that more sophisticated testing methods for build specifications are needed, arguing that running a build is the only way to verify the correctness of build specifications. While research solutions have been proposed to many of these challenges (e.g., [31]–[34]), production-ready tools have yet to reach developers. Future research and tool development that aims to bridge this gap would likely be fruitful.

References

[1] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Identifying and understanding header file hotspots in c/c++ build processes," *Automated Software Engineering*, vol. 23, no. 4, pp. 619–647, 2016.

[2] Y. Zhang, Y. Jiang, C. Xu, X. Ma, and P. Yu, "Abc: Accelerated building of c/c++ projects," in *the Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, 2015, pp. 182–189.

[3] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2014, pp. 724–734.

[4] M. Sulír and J. Porubän, "A quantitative study of java software buildability," in *the Proceedings of the International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2016, pp. 17–25.

[5] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[6] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 41–50.

[7] C. Saltapidas and R. Maghsood, "Financial risk the fall of knight capital group," 2018.

[8] P. Rigby, D. German, and M.-A. Storey, "Open source software peer review practices," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 541–550.

[9] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 1, pp. 56–75, 2016.

[10] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[11] M. di Biase, M. Bruntink, and A. Bacchelli, "A security perspective on code review: The case of chromium," in *the Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 21–30.

[12] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 1039–1050.

[13] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 111–120.

[14] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 677–687.

[15] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 3, pp. 430–448, 2008.

[16] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *the Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 202–211.

[17] E. Fregnan, F. Petrulio, L. Di Geronimo, and A. Bacchelli, "What happens in my code reviews? an investigation on automatically classifying review changes," *Empirical Software Engineering (EMSE)*, vol. 27, no. 4, pp. 1–43, 2022.

[18] J. W. Creswell and V. L. Plano Clark, *Designing and Conducting Mixed Methods Research*. Sage Publications, 2017.

[19] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[20] S. McIntosh, Y. Kamei, B. Adams, and A. 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 *the Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 192–201.

[21] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: Evaluating contributions through discussion in github," in *the Proceedings of the ACM Special Interest Group on Software Engineering International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, 2014, pp. 144–154.

[22] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *the Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 106–117.

[23] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review," *Empirical Software Engineering (EMSE)*, vol. 22, no. 2, pp. 768–817, 2017.

[24] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. Sage Publications, 2018.

[25] K. Charmaz, *Constructing Grounded Theory*. Sage Publications, 2014.

[26] P. Morville and L. Rosenfeld, *Information Architecture for the World Wide Web: Designing Large-scale Web Sites*. O'Reilly Media, Inc., 2006.

[27] W. C. Adams, "Conducting semi-structured interviews," in *Handbook of Practical Program Evaluation*. John Wiley & Sons, 2015, ch. 19, pp. 492–505.

[28] T. R. Lindlof and B. C. Taylor, *Qualitative Communication Research Methods*. Sage Publications, 2017.

[29] E. E. Lyons and A. E. Coyle, *Analysing Qualitative Data in Psychology*. Sage Publications, 2007.

[30] G. Kumfert and T. Epperly, "Software in the doe: The hidden overhead of the build," Lawrence Livermore National Lab., CA (US), Tech. Rep., 2002.

[31] Q. Cao, R. Wen, and S. McIntosh, "Forecasting the duration of incremental build jobs," in *the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 524–528.

[32] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, "Removing false code dependencies to speedup software build processes." in *the Proceedings of the Premier Industrial and Academic Conference for Advanced Studies in Computer Science and Software Engineering (CASCON)*, 2003, pp. 343–352.

[33] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos, "Reducing build time through precompilations for evolving large software," in *the Proceedings of the International Conference on Software Maintenance (ICSM)*, 2005, pp. 59–68.

[34] J. Bell, E. Melski, G. Kaiser, and M. Dattatreya, "Accelerating maven by delaying test dependencies," in *the Proceedings of the International Workshop on Release Engineering (RelEng)*, 2015, pp. 28–28.

[35] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *the Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 1078–1089.

[36] C.-P. Bezemer, S. McIntosh, B. Adams, D. M. German, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empirical Software Engineering (EMSE)*, vol. 22, no. 6, pp. 3117–3148, 2017.

[37] T. Sotiropoulos, S. Chaliasos, D. Mitropoulos, and D. Spinellis, "A model for detecting faults in build specifications," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[38] D. L. Parnas and D. M. Weiss, "Active design reviews: Principles and practices," *Journal of Systems and Software (JSS)*, vol. 7, no. 4, pp. 259–265, 1987.

[39] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[40] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *IEEE software*, vol. 29, no. 6, pp. 56–61, 2012.

[41] M. L. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[42] M. B. Miles and A. M. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*. Sage Publications, 1994.

[43] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. Van Deursen, "Communication in open source software development mailing lists," in *the Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 277–286.

[44] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *the Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014, pp. 1–10.

[45] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "The review linkage graph for code review analytics: A recovery approach and empirical study," in *the Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 578–589.