Studying Pull Request Merges: A Case Study of Shopify's Active Merchant

Oleksii Kononenko*, Tresa Rose†, Olga Baysal†, Michael Godfrey*, Dennis Theisen‡, Bart de Water‡
*David R. Cheriton School of Computer Science, University of Waterloo, Canada
okononen@uwaterloo.ca,migod@uwaterloo.ca
†School of Computer Science, Carleton University, Canada
tresa.rose@carleton.ca,olga.baysal@carleton.ca
‡Shopify Inc., Canada
dennis@shopify.com,bart.dewater@shopify.com

ABSTRACT

Pull-based development has become a popular choice for developing distributed projects, such as those hosted on GitHub. In this model, contributions are pulled from forked repositories, modified, and then later merged back into the main repository. In this work, we report on two empirical studies that investigate pull request (PR) merges of Active Merchant, a commercial project developed by Shopify Inc. In the first study, we apply data mining techniques on the project's GitHub repository to explore the nature of merges, and we conduct a manual inspection of pull requests; we also investigate what factors contribute to PR merge time and outcome. In the second study, we perform a qualitative analysis of the results of a survey of developers who contributed to Active Merchant. The study addresses the topic of PR review quality and developers' perception of it. The results provide insights into how these developers perform pull request merges, and what factors they find contribute to how they review and merge pull requests.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Collaboration in software development**;

KEYWORDS

Pull request merges, review quality, survey, industrial case study

ACM Reference Format:
Oleksii Kononenko*, Tresa Rose†, Olga Baysal†, Michael Godfrey*, Dennis Theisen‡, Bart de Water‡. 2018. Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183542

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183542

1 INTRODUCTION

Pull-base software development has gained popularity in recent years, especially among open source systems (OSSs). Hosting services, such as GitHub and Bitbucket, have attracted a huge number of new and existing projects: GitHub alone is estimated to have more than 19.4 million active repositories[1]. In the pull-based model, developers make changes to an isolated copy of the project's repository, and then submit a *pull request* (PR) to the project owners; the owners then evaluate the suggested changes and decide whether or not to permit the proposed merge into the main codebase.

Since the vast majority of public software projects on GitHub are OSSs, the findings from those studies likely reflect the kinds of processes typically used in open source development. However, there are also commercial projects on GitHub that allow their repositories to be viewed by the public, while at the same time maintaining strict ownership and tight control over their ongoing evolution. Such repositories give researchers a unique opportunity to study industrial software development processes. We decided to perform an in-depth study on one such project.

We used three criteria for selecting an appropriate project to study. First, while many projects host their codebase on GitHub, it is also common practice to perform day-to-day development activities elsewhere using private external repositories, with only occasional large updates being made to the public repository [18]. Therefore, one criterion was that the selected project should be developed completely on the GitHub platform. A second criterion was that the selected project should be commercially successful, since successful projects are more likely to systematically employ similar practices that supported their success. Finally, we wanted to study a project that had been developed in physical proximity to our labs, in case we wanted to interview members of the main development team. Based on these criteria, we decided to study the Active Merchant project developed by Shopify Inc. which has offices in Ottawa, Toronto, Montreal, Waterloo, and San Francisco.

Shopify is an e-commerce company that provides a platform for online stores. As of February 2017, more than 377,500 merchants use this platform to sell commercial goods. Active Merchant is a part of that platform; it is a payment abstraction library that handles and unifies access to a variety of payment gateways with different internal APIs. The development of that project is done completely on GitHub; all PRs must pass a code review process and get approval before being merged into the main codebase. Although this project

[1] https://octoverse.github.com

is owned by Shopify, Spreedly Inc. has recently become an active contributor as well.

Our study is built around a quantitative analysis of the Active Merchant project repository, as well as an exploratory survey that we conducted with Shopify developers. We investigate three research questions:

RQ1: *Which merge strategies are used by developers? Do they affect the pull request review time?*

A submitted PR may consist of multiple commits; a developer can add yet more commits to address reviewers' comments. If the PR is accepted, developers can incorporate the commits in several ways [18]. We study how developers merge PRs, and analyze the effect the PR merge type has on the merge time.

RQ2: *What factors affect the PR review time and decision?*

Previous studies have looked into the effect of a variety of factors on the time needed to reach a decision about a PR, as well as the effect on the decision itself [11, 35]. We investigate what factors play a role in the studied commercial project.

RQ3: *How do developers perform and assess PR reviews?*

We recognized that the analyzing the data extracted from the project's repository would tell only part of the story. To get a more comprehensive view of the pull-based development process used by Active Merchant, we conducted a survey among Shopify developers. The analysis of the survey results provided us with a better understanding of the developers' perception of PRs and PR review quality.

This paper makes the following contributions:

- An in-depth study of new contributions in a pull-based software development model within a successful commercial software project.
- A survey with full-time professional developers that offers insight into their perception of the process of assessing of new PRs.
- A publicly shared dataset[2] that includes mined data with manually verified and labelled merged PRs, the survey questions used, and the anonymized survey responses.

The rest of the paper is organized as follows. Section 2 discusses prior related work. Section 3 describes the methodology we followed in our study. In Section 4, we present the results of our qualitative and quantitative analyses. Section 5 discusses our findings, while Section 6 addresses threats to validity. Finally, Section 7 summarizes our results.

2 RELATED WORK

The pull-based model is a relatively new approach to distributed software development; at the same time it is built on ideas that are well known and have been well studied, such as community participation, open source-style development and patch submission, and off-line code review. Mockus et al. [29] were one of the first to study open source software (OSS) communities; by investigating Apache and Mozilla projects, they identified key features of OSS development, including a heavy reliance on volunteer contributions as well as self-assignment of tasks. Rigby and German [31] studied the code reviews performed in four OSS projects; they identified multiple code review patterns, such as reviewer rewriting outsiders' patches instead of commenting on them. Rigby and Storey [32] studied review processes in the mailing lists of five OSS projects. They found that identifying and fixing technical problems is not the sole motivation behind the review process; for example, developers also try to resolve scope and process issues during review.

Weissgerber et al. [36] studied the email archives of two OSS projects; they found that smaller patches are reviewed faster and have a higher chance of being accepted. Baysal et al. [3] studied two industry-led software projects, WebKit and Blink, and found that technical, organizational, and personal factors affect both the review time and likelihood of a patch to be accepted. Jiang et al. [17] analyzed the relation between patch features and the probability of its acceptance in the Linux Kernel project. They found that patch writer experience, patch maturity, and prior subsystem churn affect the likelihood of a patch being accepted; they also showed that the time needed for review depends on the number of affected subsystems and on developer experience.

Kemerer and Paulk [19] looked into the effect of the code review rate on the reviewers' ability to catch problems in new contributions, as well as on the quality of software products; as a result of their study, they recommended that to best ensure review quality, reviewers should not proceed faster than 200 LOC per hour. Hatton [16] found that reviewers differ in their capabilities to detect defects during the code review process; he also showed that there is a difference in defect detection in situations when developers review the same code together and separately (76% and 53% of found defects respectively). McIntosh et al. [25] investigated three OSS projects and found that code review coverage and participation affect system quality, as measured by the number of bugs. Kononenko et al. [21] studied the quality of the review process (i.e., the number of uncaught bugs) in Mozilla project; they found that 54% of reviewed patches still contain defects, and showed that metrics such as developers' involvement in patch discussion, their experience and workload, as well as patch's size-related metrics are good indicators of review quality.

In a qualitative study done at Microsoft [1], Bachelli and Bird studied the motivations, challenges, and outcomes of their code review process; they found that although the main goal of code review is detecting defects in submitted code, it also provide additional benefits, such as knowledge sharing among developers and increased team awareness. Rigby and Bird [30] did a quantitative study of six industry-led projects and seven OSS projects; while those projects differed in problem domain, team culture, and development processes, the study found that the code review processes employed by the projects had similar characteristics. Kononenko et al. [20] surveyed professional developers of Mozilla project to learn the developers' perception of code review process and its quality; they found that the developers associate the quality of a review with the thoroughness of the feedback they receive, the reviewer's knowledge of the codebase, and their inter-personal qualities.

Gousious et al. [11] were among the pioneers in research into pull-based development. They quantitatively investigated OSS projects hosted on GitHub to learn the factors that are influential to PR review time as well as to the acceptance of PRs; they found that

[2] https://cs.uwaterloo.ca/~okononen/shopify

the merge time is affected by several factors, including the developer's track record and the test coverage in the project, while the acceptance is primarily influenced by the "hotness" of code (i.e., the number of recent changes) that the PR proposes to modify. Tsay et al. [35] investigated the code contributions on GitHub to measure the effect of social and technical factors on the likelihood of a contribution being accepted; they found lengthier discussions tended to lead to rejection of PRs, while the developer's previous involvement in the project increased the likelihood of acceptance.

Gousious et al. [13] surveyed the *integrators* — i.e., the developers responsible for assessing/merging incoming contributions — of GitHub projects to understand their perspective on pull-based development practices; they found that integrators face multiple challenges, such as maintaining project quality and deciding which PRs to prioritize. Marlow et al. [24] studied how core developers from GitHub projects form their opinions of the incoming contributions; they found that integrators use signals such as the contributor's history of coding activity as well as their actions on GitHub (e.g., following other developers). In another study, Gousious et al. [12] surveyed the most active contributors on GitHub to learn their work practices and the challenges they face; they found that contributors are eager to maintain awareness of the projects to avoid submitting duplicate PRs, and that they communicate changes using PRs as well as issue trackers, emails, and instant messages. The main challenge identified by the contributors is poor responsiveness from core developers.

3 METHODOLOGY

To investigate our research questions, we performed a combination of quantitative and qualitative analyses. Our quantitative study consisted of mining software artifacts from Active Merchant's GitHub repository [33], pre-processing, and analyzing the extracted data. For our qualitative study, we surveyed project developers.

3.1 Data Mining

Data collection. Active Merchant is a library that provides a unified API that allows communication with many different payment gateways. The project is hosted on GitHub and employs a pull-based mechanism for submitting and accepting code contributions, as well as performing review of those contributions. GitHub provides APIs that allow users to access its data. We used an official library developed by GitHub to make API calls[3]. For our study, we looked at the contributions made to Active Merchant between January 1, 2012 and October 1, 2016; we extracted a total of 1,657 pull requests (PRs) that were submitted during this period. During the extraction process, we tracked a variety of information about each PR, including the unique ID of its author, the date the PR was added to the repository, the date the PR was closed, whether the PR was merged and (if so) the date of the merge, the natural language description of the PR, and its size statistics. For each PR, we also collected both PR-wide comments and in-code comments left by the developers.

GitHub user accounts of many Active Merchant contributors do not contain affiliation information or email address. To recover missing email addresses, we extracted the actual commits from the repository. For each commit, we analyzed commit author information recorded by GitHub (represented by unique user ID) and commit author information recorded in the header of the commit by Git (Git identifies users using their email address, so this field is always present). If GitHub user data was missing, we used the name (if available) and email address from the commit header. While it is possible that some of these addresses are inaccurate, this approach allowed us to identify many of the "anonymous" contributors. To recover developers' affiliation information, we parsed the email addresses and set it based on the domain name of the emails (except for "public" emails such as Gmail, Yahoo, etc.).

PR merge types. Kalliamvakou et al. noted that GitHub data is not always reliable regarding whether a PR has been merged or not [18]. The discrepancy between the recorded merge information and the actual merge status exists because developers can merge a PR using several different approaches. We used the heuristics proposed by Kalliamvakou et al. [18] to recover "missing" merge flags. These heuristics are based on commits in the master branch of a repository as well as on the content of the last comments left on a PR. For example, according to one heuristic, a PR was merged if there is a commit in the repository's master branch and that commit closed a PR using a specially formatted message appended to the commit message. By applying these heuristics, we were able to mark 798 "not merged" PRs (as reported by GitHub) as "merged" ones. Kalliamvakou et al. also warned that their heuristics may result in a considerable number of false positives. To reduce this risk, we performed a manual inspection of the merged PRs; the inspection also afforded us the opportunity to label each PR according the merge type labels suggested by Kalliamvakou et al.:

- *GitHub merge* — a merge performed using GitHub facility (i.e., using the "merge" button in the UI).
- *Cherry-pick merge* — a merge when a developer selects a subset of commits from a PR and adds it to the repository without any changes.
- *Commit squashing* — a situation when a developer creates a new commit that contains all commits from a PR, makes additional changes if needed, and adds this commit to the repository.

Manual inspection of pull requests. Two authors performed the inspection of all PRs marked as "merged" by the stated heuristics. To ensure that the authors had the same understanding of the merge types, we randomly selected 20 PRs and performed independent labelling of each PR; we then compared the assigned labels and calculated intercoder reliability score (i.e., percent agreement). The authors achieved high agreement (93%) between themselves: they differed only in two PRs. After that, the remaining PRs (778) were split in two sets; two authors separately inspected and labelled one of these sets. As a result of this inspection, we found seven PRs that were incorrectly marked as "merged".

Data pre-processing. To minimize potential noise in the collected data, we tried to eliminate outliers by applying three filters:

- We removed 5% of the PRs with the longest review time to account for PRs that struggled to catch developers' attention. Several PRs took an extremely long time to get reviewed; for example, the longest review took 637 days, while the median for review time is 3 days.

[3] https://github.com/octokit/octokit.net

- Some PRs are unusually large in terms of added/removed lines of code (LOC) — the biggest PR is nearly 1 million LOC, while the median is only 35 LOC — and to account for such PRs we removed the largest 5% of all PRs.
- Since we were interested only in studying those PRs that the developers had decided on, we removed all PRs that were not marked as "closed" (26 PRs in total).

After applying these filters at the same time, our dataset was reduced to 1,475 PRs.

3.2 Explanatory Factors

Previous research suggests a set of different metrics that can affect code review time and outcome [3, 11]. Table 1 lists the explanatory factors we considered in our study. The selection of each factor was governed by our ability to accurately calculate its values from the data (i.e., we did not include a factor if we could not collect the corresponding data, or if a heuristic was required to compute it).

Table 1: Overview of the factors studied.

Explanatory Factor	Description
PR size	Sum of added and removed LOC
# files	# files changed by a PR
# commits	# commits in a PR
PR author experience	# prior PRs submitted by PR author
# comments	# comments left on a PR
# author comments	# comments left by the PR author
# commenting developers	# devs participating in discussion
# in-code comments	# comments left on source code
# author in-code comments	# comments left on source code by author
# in-code commenting devs	# devs who commented on source code
PR author's affiliation	An org that a PR author affiliates with

Although GitHub allows a PR to be assigned to a specific developer, we found that this feature was rarely used within the Active Merchant repository. Therefore, one of the challenges we faced was determining the exact time boundaries of a review period. We considered the PR submission date to be the date that the review process started. Since a PR cannot be merged before it has passed the review, we considered the date a PR was closed as the date the review process ended. Thus, the time between these two dates (i.e., start and end dates of review) is defined as review process length.

Another challenge was the lack of standardized labels in GitHub to indicate the outcome of a PR review. We marked all closed and merged PRs as the ones that successfully passed the review, and all closed but not merged PRs as the ones that received a negative review. Similar assumptions were made in [11, 35].

3.3 Data Analysis

To understand the effect of the selected factors on review time and review outcome, we built Multiple Linear Regression (MLR) and Logistic Regression Models (LRM) respectively. These models try to capture the relationship between the explanatory variables — in our case, the factors described in Table 1 — and a response variable — i.e., the PR review time and review outcome [7]. Our goal of understanding the relationship between explanatory and dependent variables as well as our model construction process are similar to the ones in the previously published studies [5, 21, 25, 28].

Variable transformation. Empirical evidence suggests that software engineering data is rarely normally distributed [26]. To minimize any possible skewness, we applied a log transformation *log(x+1)* to all continuous variables (e.g., size, author experience, comments, etc.). Because categorical variables (e.g., `affiliation`) cannot be used directly in regression models, we employed a dummy coding technique to transform a categorical variable into a set of dichotomous variables that capture the same information.

Controlling Multicollinearity. Multicollinearity is defined as a high correlation among two or more explanatory variables in a regression model. We checked the models for multicollinearity using the variance inflation factor (VIF). A VIF score of each variable represents how much its variance is explained by the collinearity with other variables. We used the `vif` function from the `R car` package to calculate VIF scores [9]. As recommended by Fox [8], the threshold for VIF score was set to 5.

Model Evaluation. To evaluate our models, we considered R^2 values. For our MLR model, we used an *Adjusted* R^2 value [15]; unlike R^2, this value is affected by the extra variables with low explanatory power in the model: the more such variables, the lower the value. To reduce the number of "useless" variables, we used a bidirectional stepwise selection technique [8], a process of adding and removing independent variables for finding the best subset. There is no R^2 value for LRMs; instead several statistics, so called "pseudo R^2", have been proposed. We used the one proposed by Tjur — Tjur's D [34] as implemented by the R `binomTools` package [6]. This statistic is closely related to the definition of R^2 in MLR models, and it is designed for dichotomous dependent variables.

3.4 Survey Design and Participants

To understand developers' work practices and their vision of the PR review process used in the project, we conducted a survey. Similarly to previous studies [11, 20], our survey containing three groups of questions: nine questions related to the demographic information about participants and their work practices, three Likert-scale questions focused on PR review, and four open-ended questions asked participants to provide more information concerning their responses to the Likert-scale questions. Participants were informed that our survey would take 10–15 minutes to complete.

We targeted Shopify developers, since Active Merchant is a product of Shopify, and their developers are the main contributors to the project. Our dataset included 88 Shopify developers; 10 of these lacked valid email addresses, so we sent out personalized emails to the remaining 78, inviting them to participate in the survey. For five of those emails we received an automated response saying that the email address had been deactivated; we speculate that these developers are no longer with Shopify. The survey was open for two weeks — from February 27, 2017 to March 13, 2017 — and we received 16 responses. While the number of responses may seem small, the response rate of 22% (16/73) is higher than the suggested minimum response rate of 10% [14].

3.5 Card Sorting

We employed a grounded theory approach for analyzing the developers' responses to open-ended questions in the survey. Before our analysis we had no preconceived ideas or theories about the

survey responses, so we used an open coding approach to create categories and themes, and to group the data into them [27].

The first author split 16 survey responses into 181 isolated quotes (cards), i.e., each quote represents a single statement that differs from other statements in a particular answer to a question. After that, two authors, serving as coders, went through the cards grouping them into themes, and later grouping themes into broader categories. The coders used the following protocol on all but one of the open-ended questions:[4]

- The coders took the first 25% of cards that correspond to a particular question and — independently of each other — organized them into several groups. Once they were done with the first round of card sorting, they compared and discussed the emerged groups and the cards in them.
- In the next round, the coders took another 25% of cards and — again, independently — sorted them into the groups created in the previous step. If a coder believed that a card did not match any of the existing groups, they could create a new group for that card. To ensure the integrity of the process, we computed the intercoder reliability at this step.
- During the final round, the coders sorted the rest of the cards (i.e., the remaining 50%) together.

We opted for percent agreement as our intercoder reliability coefficient as it is one of the most popular metrics. We used ReCal2 to compute percent agreement values for each question [10]. The values of the reliability coefficient were different among questions and ranged between 91.9% and 100%, with the average of 96.4%.

4 RESULTS

In this section we present the results of our quantitative and qualitative studies, and we answer our research questions.

4.1 RQ1: Merge types and effect on review time

In a more traditional development model, if a contribution (e.g., a patch) is approved, it will be added to the repository. If the contribution has undergone a series of revisions before being approved, only its final version will be incorporated into the codebase. Pull-based development, on the other hand, has created a new way of dealing with incoming contributions to a software project. When considering PRs, developers have more flexibility: they can choose to incorporate as little as they deem beneficial to the project or they can decide to merge a complete PR. If they decide on the latter, they have the further option of either leaving the commits from a PR untouched — which preserves more historical information — or "squashing" the commits into a single commit and add that into the repository — which results in a cleaner commit history on the master branch. While researchers have studied different aspects of pull-based development, we did not find any studies that analyzed the merge approaches used by developers. Thus, first we decided to take an exploratory look at PR merge strategies.

As described in Section 3.1, we applied several heuristics to determine merged PRs; later, we also manually inspected and labelled all merged PRs in our dataset. Table 2 reports the results of this manual classification. While we were surprised to see that only a small number of PRs — about 25% — were merged using the native GitHub UI, it might be due to the fact that the merging via the "merge button" is possible only if (a) there are no further changes required, and (b) GitHub can automatically resolve any merge conflicts it encounters. In fact, during the manual inspection we noticed that many merged PRs had additional changes made by a merger. When developers were merging a PR, they often updated `changelog`, `readme`, and/or `contributors` files to reflect the new change. Squashing was the most popular merge type used by developers. Although some historical information is lost during such a merge, there are some advantages too. The main benefit here is that the squashing merge helps developers to keep the commit history in the master branch simple and relatively clean: each commit represents a single PR. With such an organization of the branch, developers can ensure that the commits are accompanied by a descriptive log message; also, it is easier for developers to revert a merged PR if there is a need. Moreover, squashing naturally supports both the small additional changes the mergers make and the automatic closure of a PR.

Table 2: Classification of PRs by a merge type.

Merge type	Count	Percent	Median review time (in min)
Not merged	372	25.2%	10,111.5
GitHub	367	24.9%	1,107.8
Squashing	612	41.5%	4,241.8
Cherry-picking	124	8.4%	3,177.6

To study whether merge types have an effect on the time a PR stays open — i.e., until it is ultimately rejected or merged — we used two non-parametric statistical tests: Kruskal-Wallis analysis of variance [22], and a post-hoc Mann-Whitney U (MWW) test [23]. The Kruskal-Wallis test revealed that merge type has a statistically significant effect on time ($\chi^2(3)=89.02$, $p < 0.001$). Since this test does not show where the significance occurs, we followed up with pairwise comparison using MWW test with Bonferroni correction. The test showed significant difference among all pairs except one: the difference between *cherry-picking* and *squashing* was not statistically significant, although the median time for the former is smaller than the median time for the latter. We report the median time for each merge type in Table 2. The large difference in median time between unmerged PRs and merged PRs might indicate that if a PR is going to be accepted, it will be accepted quickly; otherwise, it will be shelved and "forgotten".

***RQ1**: While most developers merge pull request via squashing (41.5%), GitHub and cherry-pick PR merge types are also a common practice. Merge type has a statistically significant effect on PR merge time; cherry-picking and squashing merges take more time than the merges done via the GitHub UI.*

4.2 RQ2: Factors affecting merge time/decision

To investigate which factors influence the time developers take to make a decision about a PR, as well as the factors that affect the review decision (to merge or not to merge), we built two statistical models: Multiple Linear Regression Model (MLRM) for PR review time, and Logistic Regression Model (LRM) for PR review decision. The models used factors from Table 1 as independent variables; we removed `merge type` factor from the decision model because

[4]One of the questions received only few responses making it impractical for us to apply the described procedure.

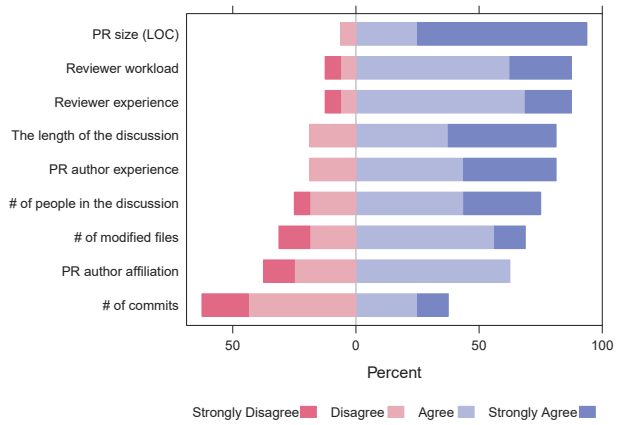

Figure 1: Factors influencing PR review time.

it implicitly reflects (i.e., correlates with) the dependent variable. Table 3 reports the regression coefficients for each of the studied factors. In addition to the qualitative analysis, in our survey we asked developers which factors they experienced to be influential to time and decision; we used 4-point Likert-scale questions, as well as open-ended questions to obtain developer insights.

Table 3: Models for fitting data.

	PR review time	**PR review outcome**
	Adjusted R^2: 0.28	Tjur's D: 0.14
Size (LOC)	0.110*	-0.187***
Number of files	‡	‡
Number of commits	·	‡
Writer experience	-0.285***	0.217***
# of comments	†	†
# of commenting devs	1.021***	-0.786***
# of author comments	‡	·
# of in-code comments	†	†
# of in-code commenting devs	0.389*	0.357*
# of in-code author comments	‡	‡
Affiliation with Shopify	-1.721***	1.160***
Affiliation with Spreedly	-1.980***	1.046***
Cherry-pick merge	0.654*	n/a
GitHub merge	-0.610**	n/a
Squashing merge	0.804***	n/a

†Removed during VIF analysis.
‡Removed during stepwise selection.
Stat. significance codes: *** < 0.001 < ** < 0.01 < * < 0.05 < ·

Merge time. The MLR model indicates that the PR size has a statistically significant effect on review time. The positive value of the regression coefficient means that the larger a PR is, the longer it takes for developers to review it. PR size was also seen as influential by almost all developers who participated in the survey (Figure 1). These findings are consistent with previous studies [3, 17, 20, 36].

Writer experience appears to have a statistically significant effect on review time. A negative regression coefficient for this factor demonstrates that more experienced developers tend to have quicker turnaround time for their PRs. One possible explanation for this finding is that experienced developers might be more familiar with the codebase and project culture, and thus are likely to submit PRs that fit better into the project. While we did not consider reviewer experience as a factor — because it was not feasible to calculate it accurately — it was one of the factors we asked developers about. Developers believe that both author and reviewer experience are important contributors to PR review time (81% and 87% of positive answers respectively).

Out of six discussion-related factors only two made it to the final model: the number of developers who left comments at a PR level and at a source code level. Both of these metrics have positive regression coefficients, indicating that each new developer participating in a discussion delays the decision on a PR. Discussion of new contributions is vital to the health of software project, thus a thorough discussion should likely be welcomed although it can cause a delay. At the same time, we were surprised to see that the PR author comments factor was not present in the model. It is the author's job to explain a proposed change and address reviewer comments; our intuition is that the lack of such comments would only delay the final decision. Contrary to the results of the model, developers indicated that the length of a discussion is a critical factor (81% positive responses).

The model shows that the PR author's affiliation influences review time as well. Pull requests submitted by developers from Shopify (owners of the project) or Spreedly (who work very closely with Shopify) receive faster reviews on their PRs than those submitted by external developers. This finding is somewhat similar to the one made by Baysal et al. [2] who studied code review of the Mozilla project. Surprisingly, few developers agreed with the statement that author affiliation affects PR review time. Perhaps, these developers did not participate in a review of external PRs, or because they believe that the established process is impartial.

Merge type was also included in the final model; the findings here are similar to those presented in Section 4.1. A merge performed using GitHub UI correlates with shorter review time, while the other two merge types are associated with longer review time.

The open-ended questions of the survey provided developers an opportunity to discuss factors not covered by the Likert-scale questions. The open coding analysis of the open-ended questions also revealed several additional factors that developers believe have an impact on the PR review time. The biggest theme identified in the responses is *PR quality*, which includes *PR description* and *PR complexity* categories. As explained by D9, *"bad descriptions are the biggest factor; someone submitting a 1,000 LOC PR with a good description is much better than someone submitting a 100 LOC PR with only 'Added XXX integration'"*. Several developers believe that the *type of change* (e.g., *"new feature, refactor, bugfix"*) and whether the change affects code's *architecture/design* (e.g., *"big refactoring"* (D14)) are also important factors affecting review time. *Human factors* such as *"trust you have in author"* (D6) and reviewers' familiarity with *"that part of the codebase"* (D2), PR discussions that may take place across multiple channels *"in a GitHub issue, face-to-face, etc."* (D10), the *"set up of the tophat"* (i.e., *testing*) are all considered by developers to contribute to the PR review time.

Merge decision. As with the previous model, the PR size metric is included in the final model for review outcome (i.e., merge decision). Its negative regression coefficient indicates that larger PRs are more likely to receive a negative merge decision (i.e., a PR is not merged) than smaller ones. However, when we asked developers about the influence of this factor, their answers were split: only

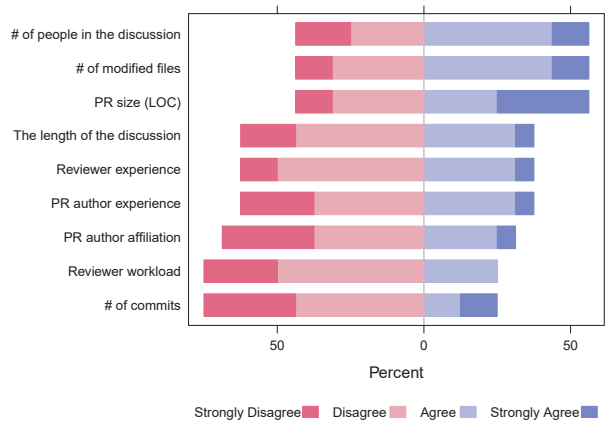

Figure 2: Factors influencing PR review outcome.

56% of developers agreed that the size affects the review outcome (Figure 2). A possible explanation for this is that a larger PR has a higher chance of containing more than one "atomic" change, which is against PR submission policies in many software projects. While developers may not mind accepting a large PR that is coherent and single-purpose, they may feel more negatively about a large PR that is a collection of loosely related changes.

The PR author experience has a statistically significant effect on merge decision. The more PRs a developer has submitted in the past, the more likely their PR will be accepted again. Developers actively contributing to the project are often well known to other developers, and therefore the acceptance of a PR might be affected by their reputation [4] or interpersonal relationship with other developers [20]. Surprisingly, when we asked developers about impact of experience on merge decision, they believed that neither author nor reviewer experience affects it (63% of negative responses for each factor).

Similar to the model on time, the only discussion-related factors here are the number of people leaving comments and the number of people commenting on the source code. However, in this model, these factors have the opposite effect on the merge decision: the higher number of developers commenting on a PR leads to a lower chance being approved, while the number of developers leaving comments on the source code is likely to increase the chance of a PR being accepted and merged. We speculate that this happens in the situations when PR comments are likely to be more 'high level' (e.g., the need for such a change, alignment with project's goals, etc.) while the comments on the source code, by definition, are more 'low level' (e.g., implementation, API choice, etc.). At the same time, the discussions regarding high-level issues are likely to be controversial; therefore, bringing more developers to such discussions might prevent them from reaching an agreement. Developers agreed that the number of people involved in a PR discussion affects its acceptance (56% of positive responses); however, they disagreed the length of that discussion plays a role (63% of negative responses).

The affiliation of the PR author has a statistically significant impact on the merge decision. Both Shopify's and Spreedly's regression coefficients are positive, indicating that the PRs that come from the developers of these two organizations have a higher chance of being accepted.

In analyzing developer responses to the open-ended survey question related to review decision, we observed that several categories emerged. The highest impact on the PR review decision is perceived to be *PR quality* including its complexity, *"can we instead generalize this feature? can you do instead something simpler using the current functionality of the code?"* (D12). Developers also argue that PR author's responsiveness and workloads (we put these under *human factors*) affect reviewer's decision on whether to merge the PR. As D6 explains, *"mostly if people have time, most PRs seem to eventually get merged as long as the author has time to fix things."* Some developers find that overall project schedule such as *"release plan"*, *"code freeze"* (D14) may also impact merge decisions.

> ***RQ2**: The statistical models revealed that both PR review time and merge decision are affected by PR size, discussion, and author experience and affiliation. Developers believe that PR quality, type of change, and responsiveness of an author are also important factors.*

4.3 RQ3: How developers perform and assess PR review process

During the open coding process, 22 key categories (including "irrelevant") emerged; Table 4 presents these categories in detail reporting the number of quotes, the number of respondents, the question numbers, and the totals for each question.

PR review process of Active Merchant. Since each organization adopts its own code review guidelines and practices, we first wanted to understand how developers conduct reviews of the Active Merchant PRs. In particular, in our survey we asked developers about the steps they typically follow when they are asked to review a PR.

The analysis of the survey answers shows that developer see *testing* as the key feature of the PR review process: new code must pass manual user testing ("tophatting") , as well as automated unit tests and then peer reviewed. Thus, one of the first steps is to "tophat". To do so, reviewers check whether the PR author has tested the code, *"did you tophat the change? what steps did you follow?"* (D10). D10 further explains that Shopify requires *"separate tophats (testing) from someone other than the PR author before submission"*. Therefore, reviewers *"look for tests and what kind of tests and where they cover"* (D1) and *"make sure that test cover all changes"* (D10).

The next key step of the process is to *understand the scope of the change*. First, reviewers read the PR title and description and *skim through*, i.e., *"an initial pass to get a sense of what it's about"* (D6). And next, they try to *"understand the full extent of the change, not simply the changes in the* `diff` *"* (D10). D1 reports that *"I read the what, how and why you are trying to do with your PR."* To understand why a change was made, sometimes developers need to *"jump to different parts of the code as necessary to reference other changes"* (D8). At the end of this step, reviewers pay close attention to *"touchy code"* (D1), i.e., *"things that look weird"* (D6).

Code inspection is an integral part of the PR review. Reviewers look at the code and evaluate *code quality* according to code guidelines. For example, D10 checks if PR author *"named everything correctly and in an intuitive way: variables, tests"*. Some reviewers apply different code inspection strategies depending on the type of change they review. D8 reflects on his approach: *"If it's a bugfix or feature, start reading through the changes in one window (top to*

Table 4: The categories emerged during open coding.

Category	Q9		Q11		Q13		Q15		Q16	
	#Q	#devs	#Q	#devs	#Q	#devs	#Q	#devs	#Q	#devs
Understanding context/rationale/scope	19%	60%	-	-	25%	50%	16%	33%	7%	20%
Code inspection	15%	70%	-	-	-	-	-	-	-	-
Touchy code	4%	20%	-	-	-	-	-	-	-	-
Testing	22%	70%	5%	10%	8%	17%	6%	20%	14%	33%
PR complexity / granularity	4%	20%	30%	50%	25%	33%	12%	33%	-	-
PR description	-	-	25%	30%	-	-	20%	47%	-	-
Skimming through	4%	20%	-	-	-	-	-	-	-	-
Catching bugs	2%	10%	-	-	-	-	-	-	5%	13%
Refactoring	2%	10%	-	-	-	-	-	-	-	-
Comments/discussion	11%	40%	10%	10%	-	-	4%	13%	7%	20%
Code quality	2%	10%	-	-	-	-	18%	47%	7%	20%
Architecture/design	9%	10%	10%	20%	-	-	-	-	5%	13%
Type of a change	-	-	10%	20%	-	-	-	-	-	-
Familiarity/knowledge of codebase	-	-	5%	10%	-	-	-	-	-	-
Release schedule	-	-	-	-	17%	17%	-	-	-	-
Human factors (e.g., experience, trust)	-	-	5%	10%	17%	33%	2%	7%	2%	7%
Time	-	-	-	-	-	-	2%	7%	11%	33%
Feedback	-	-	-	-	-	-	-	-	32%	60%
Conformance to project goals	-	-	-	-	-	-	6%	13%	9%	13%
Performance	-	-	-	-	-	-	2%	7%	-	-
Revertability	-	-	-	-	-	-	6%	13%	-	-
Irrelevant	7%	30%	-	-	8%	17%	8%	27%	2%	7%
Total	54	10	20	10	12	6	51	15	44	15

Notes: #Q: the number of quotes, #devs: the number of developers, Q9: PR review process, Q11: factors affecting time, Q13: factors affecting decision, Q15: characteristics of PR quality, Q16: characteristics of PR review quality.

bottom). If it's a refactor, open the code in two windows side-by-side, so additions and deletions can be browsed independently".

Reviewers typically provide their feedback — in a form of comments or questions — to a PR author to *"discuss the solution and the approach, not just the implementation"* (D10). Such *discussions* are critical as they are seen as communication mechanisms between PR author and reviewer. Reviewers check whether the PR author has addressed their questions, as D8 elaborates *"I circle back on questions and comments to see if they've been answered by code later in the PR".*

Apart from code quality, reviewers also check for any violations related to *architecture and design*. Such inspections can be performed on code itself (*"are any abstractions leaked into code?"*), tests (*"are the tests tightly coupled making refactoring harder in the future?"*) or use cases (*"is any complexity added from trying to anticipate future use-cases of the code?"*). Also, reviewers check whether the PR author has used the *"best methods"* of implementing a piece of functionality.

Developer perception of a PR quality. Enforcing quality standards is key to code review; standards of both code and project development must be met by proposed PRs. With this research question, we explore how developers define and view PR quality.

One of the main attributes for developers when evaluating PR quality is its *description*. Developers believe that the PR description should be thorough, explaining *"what it's solving and why"* (D8), describing *"happy/unhappy paths"* and *"possible errors/problems that are not fully solved"* (D11). Some respondents also said they wanted PR descriptions to include *"potential alternatives"* to the solution and *"decision why current solution was chosen"* (D12). Useful commit messages can help reviewers to decide whether *"PR is large [and needs to] be broken down into smaller bits"* (D8).

Code quality is another top property that Shopify reviewers look for. PR must follow *"coding guidelines"* (D14), *"clean and in a mergeable (not draft) state (no commented out lines, syntax follows convention, etc.)"*, *"annotated if necessary (why certain things are changed; foreseeing where a reviewer might have questions)"* (D8).

PR complexity is also an important indicator of quality. Developers check whether a PR is *"too large for people to review in one go?"* (D2) and *"can be split into smaller PRs"* (D4). Reviewers want a PR to be *"small and easy to understand"* (D1) and to *"solve only one issue, and make the smallest change possible (doesn't get carried away doing non-targeted changes"* (D8).

Developers also checked PRs for their *revertability*, i.e., whether a PR is *"self-contained"* (D2), and *"can be reverted (in most cases) with no side effects"* (D4). *Conformance to project goals* is seen as important for a PR; for example, D13 states that *"a good PR either adds features as per the goals of the project or rectifies errors and inconsistencies in existing code and documentation"*. Other reviewers look at whether a PR's functionality satisfies performance requirements, i.e., it *"satisfies SLA the code is going to be run under"* (D6).

Several respondents suggested that *testing* is a good indicator of the PR quality. *"Does PR have tests?"* (D6), *"can [PR] be tested on its own?"* (D10), *"how good PR in terms of tests coverage?"* (D14) are some of the questions that developers try to answer when reviewing PRs. *"Good discussions going on"* (D4), author experience submitting PRs, and time it takes to review a PR as explained by D5 *"the minimum time I can spend reviewing it, the better is the PR"* are also attributed to the PR quality.

Perception of the PR review quality. We now offer insights related to understanding developer perception of the main characteristics that contribute to a PR review quality.

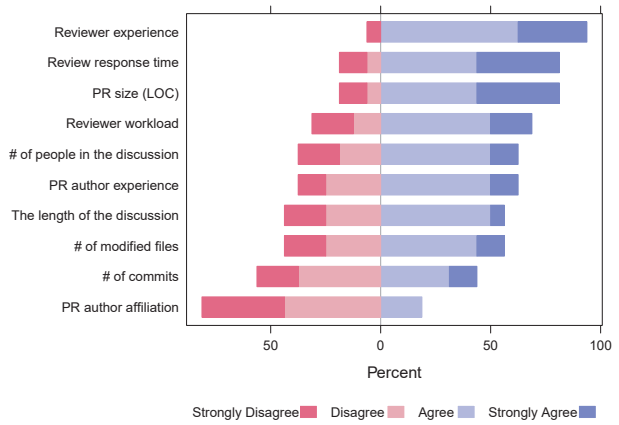

Figure 3: Factors influencing PR review quality.

The responses to the relevant Likert-scale questions are shown in Figure 3. There was no strong prevalence of positive or negative responses concerning most of the factors we asked developers about. Reviewer experience was the only factor to receive overwhelming support (94% positive). Many developers also agree that response time is a strong indicator of PR review quality. Surprisingly, only one size-related metric (PR size) received a large number of positive responses (81%). The vast majority of developers disagreed with the statement that the affiliation of the PR author affects the quality of the review process. This might indicate that developers are confident that the established practices are fair, and do not differentiate based on where a contribution came from.

Applying manual coding analysis to the open-ended question, we found that the majority of survey respondents indicated that *constructive feedback* remains the key attribute of a high quality review. Developers expect from reviewers to 1) maintain *"a good balance between asking questions and being clear about what you, as a reviewer, want changed"* (D2); 2) offer *"actionable comments"* (D2) to PR authors, so it is easy for them to address these comments; 3) express their feedback in an appropriate manner, *"strong opinions weakly held — a weak opinion is useless"* (D4); 4) take time to *"educate when you point out a mistake"* (D6); and 5) ask detailed questions to PR authors to get them *"thinking about what they are trying to accomplish"* (D10). When a PR is "rejected", developers want reviewers to offer *"the list of what is missing of the user story/task functionality, coding style, architectural approaches"* (D15).

Testing is viewed as a process that helps reviewers to conduct PR reviews. Reviewers typically need to pull a PR into their local repository and test it to make sure it does not cause any issues. Thus, reviewers check for presence of *"automated tests and the quality of the tests"* (D1). Other factor that developers attribute to review quality is *time* taken to review PRs, reviewers are advised to *"take time to read it all"* (D6) and not *"rush through"* (D4).

Conformance to project goals, once again, was also an important property. Developers noted that review quality is associated with whether reviewers understand that *"we are all on the same team"*, as D2 suggests *"let's keep things moving, as long as they are moving in the right direction"*. Reviewers are expected to evaluate PRs based on project priority and *"charter"*, as well as how they improve *"maintainability of the project"* (D13).

PR reviews are also assessed by how well developers *understand context/rationale/scope* of the change and whether *"its impact is well acknowledged"* (D7), proper syntax/language (D8), and whether the PR review facilitated a good *discussion*, e.g., such as *"generated from important stakeholders"* (D7).

> ***RQ3**: Developer perception of PR quality is defined by its description, complexity, and revertability. PR review quality is seen as a function of constructive feedback, quality of tests (unit tests and tophats), generated discussion between reviewer and PR author, and moving the project forward.*

5 DISCUSSION

Merge types. One of our findings is that developers use squashing more often than any other merge technique. While this approach has the largest median merge time associated with it, developers clearly see benefits in using it. Squashing also leads to some loss of historical information, and hence it has been suggested that it is a "bad" development practice[5]. We argue that further research is needed to better understand the considerations that developers make when deciding how to integrate a pull request. With such an understanding, researchers can better address the shortcomings of current merge techniques, which in turn may improve the quality of future research of git-stored data.

Merge time. Recent work by Gousios et al. [11] has also studied merge time and the factors affecting it. The authors analyzed similar factors to the ones studied in this paper; however, their findings are somewhat different. Both studies showed that developer experience/reputation impacts merge time. However, in Gousios et al.'s study the project-level metrics were shown to be influential, while we did not investigate those factors. Furthermore, the factors that were significant in our model were not found to be significant in their study. This suggests that there is likely no unified model that will work consistently across the disparate landscape of all GitHub projects, and that each project/domain should be studied individually.

Merge decision. The factors that affect the merge decision were also studied by Gousios et al. [11] and Tsay et al. [35]. Our findings align well with those of Tsay et al.: both sets of studies showed similar effects of PR size, discussion (although via different metrics: number of commits vs. number of people involved), and affiliation on decision to merge a PR. Gousios et al.'s study showed the effect of the number of files changed as well as some project-level metrics; in our work, those metrics were either insignificant or not studied. Here, the claim we made for the differences in findings regarding merge time appears to be unjustified. However, it is worth pointing out that both Tsay et al. and we relied on the same statistical model — logistic regression — while Gousios et al. used a random forest classifier. We wonder to what degree the choice of statistical model affects the results, which in turn might warrant a call for unifying the way in which statistical models are used by software engineering researchers.

[5]Certainly, software engineering researchers may be expected to be unhappy with the loss of information.

6 THREATS TO VALIDITY

Threats to *internal validity* concerns the quality of study design and rigorousness of its execution. In our study, these threats are related to data mining, model construction, as well as survey design and analysis. We extracted data using GitHub's own API; this ensured that we had the most up-to-date dataset at our disposal. To limit the number of inaccurate records, we performed a manual inspection of the merged PRs. In addition, we filtered out the obvious outliers from the dataset. While the choice of factors selected for the models might be a threat, we relied on the metrics previously used by the research community. In designing our survey, we tried to ensure that our questions were clear and easy to answer. We might have introduced some research bias during the analysis of open-ended questions; however, we also tried to minimize any such bias by following a strict protocol described in Section 3 and reporting the intercoder reliability scores.

External validity concerns the generalizability of the findings. We focused on a single software project and the developers working on it. While Active Merchant is a successful commercial project and the survey respondents are highly experienced full-time software developers, we cannot claim that our findings generalize across all projects hosted on GitHub based only on our study. At the same time, we do believe that any medium-sized software project with similar structure (i.e., a commercial project that is open to external contributions) is likely to exhibit many similar features of pull-based development. Nevertheless, further research is required to enhance the understanding of pull-based software development model and to create a unified body of empirical knowledge.

7 CONCLUSIONS

Pull-based software development is a popular model of modern distributed software development. In this work we provide an in-depth study of a commercial software project that employs the pull-based model. First, we study PR merge types by manually classifying PRs. We then built statistical models to investigate the effect of a variety of factors on the PR review time and PR merge decision. We found that PR size, the number of people involved in the discussion of a PR, author experience and their affiliation were significant in both models. Finally, we surveyed Shopify developers to understand their perception of the PR quality and PR review process. The analysis of the survey responses showed that the developers associate the quality of a PR with the quality of its description, its complexity and revertability, while the quality of the review process is linked to the feedback quality, tests quality, and the discussion among developers.

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proc. of the Int. Conf. on Soft. Eng.* 712–721.

[2] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2012. The secret life of patches: A firefox case study. In *Proc. of the Working Conf. on Reverse Eng.* 447–455.

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

[4] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. 2007. Open borders? immigration in open source projects. In *Proc. of the Int. Workshop on Mining Soft. Repositories.*

[5] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Soft. Eng.* 35, 6 (2009), 864–878.

[6] Rune Haubo B Christensen and Merete K Hansen. 2011. *binomTools: Performing diagnostics on binomial regression models.* R package version 1.0-1.

[7] J. Cohen. 2003. *Applied Multiple Regression - Correlation Analysis for the Behavioral Sciences.*

[8] J. Fox. 2008. *Applied Regression Analysis and Generalized Linear Models.*

[9] John Fox and Sanford Weisberg. 2011. *An R Companion to Applied Regression* (second ed.).

[10] Deen Freelon. [n. d.]. ReCal2: Reliability for 2 Coders. http://dfreelon.org/utils/recalfront/recal2/. ([n. d.]).

[11] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proc. of the Int. Conf. on Soft. Eng.* 345–355.

[12] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-based Development: The Contributor's Perspective. In *Proc. of the Int. Conf. on Soft. Eng.* 285–296.

[13] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proc. of the Int. Conf. on Soft. Eng.* 358–368.

[14] R.M. Groves, F.J. Fowler, M.P. Couper, J.M. Lepkowski, E. Singer, and R. Tourangeau. 2009. *Survey Methodology* (2 ed.).

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference and prediction* (2 ed.).

[16] Les Hatton. 2008. Testing the Value of Checklists in Code Inspections. *IEEE Software* 25, 4 (2008), 82–88.

[17] Yujuan Jiang, Bram Adams, and Daniel M. German. 2013. Will My Patch Make It? And How Fast?: Case Study on the Linux Kernel. In *Proc. of Working Conf. on Mining Soft. Repos.* 101–110.

[18] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proc. of the Working Conf. on mining Soft. Repositories.* ACM, 92–101.

[19] Chris F. Kemerer and Mark C. Paulk. 2009. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Trans. Soft. Eng.* 35, 4 (July 2009), 534–550.

[20] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: how developers see it. In *Proc. of the Int. Conf. on Soft. Eng.* 1028–1038.

[21] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating Code Review Quality: Do People and Participation Matter?. In *Proc. of the Int. Conf. on Soft. Maintenance and Evolution.* 111–120.

[22] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *Journ. of the American Statistical Ass.* 47, 260 (1952), 583–621.

[23] E.L. Lehmann and H.J.M. D'Abrera. 2006. *Nonparametrics: statistical methods based on ranks.*

[24] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. 2013. Impression Formation in Online Peer Production: Activity Traces and Personal Profiles in Github. In *Proc. of the Conf. on Computer Supported Cooperative Work.* 117–128.

[25] 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 *Proc. of the Working Conf. on Mining Soft. Repos.* 192–201.

[26] 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 Soft. Eng.* 21, 5 (Oct. 2016), 2146–2189.

[27] M.B. Miles and A.M. Huberman. 1994. *Qualitative Data Analysis: An Expanded Sourcebook.*

[28] Audris Mockus. 2010. Organizational volatility and its effects on software defects. In *Proc. of the Int. Symposium on Foundations of Soft. Eng.* 117–126.

[29] Audris Mockus, Roy T Fielding, and James D Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Soft. Eng. and Methodology* 11, 3 (2002), 309–346.

[30] Peter C. Rigby and Christian Bird. 2013. Convergent Contemporary Software Peer Review Practices. In *Proc. of the Joint Meeting on Foundations of Soft. Eng.* 202–212.

[31] Peter C Rigby and Daniel M German. 2006. *A preliminary examination of code review processes in open source projects.* Technical Report. DCS-305-IR, University of Victoria.

[32] Peter C. Rigby and Margaret-Anne Storey. 2011. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proc. of the Int. Conf. on Soft. Eng.* 541–550.

[33] Shopify. 2017. Active Merchant. https://github.com/activemerchant/active_merchant/. (2017).

[34] Tue Tjur. 2009. Coefficients of determination in logistic regression models - A new proposal: The coefficient of discrimination. *The American Statistician* 63, 4 (2009), 366–372.

[35] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proc. of the 36th Int. Conf. on Soft. Eng.* 356–366.

[36] Peter Weissgerber, Daniel Neu, and Stephan Diehl. 2008. Small patches get in!. In *Proc. of the Working Conf. on Mining Soft. Repos.* 67–76.