Towards Understanding What Factors Affect Pull Request Merges

by

Tresa Rose

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Computer Science

in

School of Computer Science

Carleton University
Ottawa, Ontario

Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In Open Source Software (OSS), most of the contributions come from the user community than the project's core developers. The main goal of all contributors is to get their contributions accepted earlier. Pull-based development model has become a recent trend in distributed projects development. In patch-based model, contributions are submitted as patches while in pull-based model, contributions are submitted as pull requests. These pull requests which may consist of a set of commits, are pulled from the forked repositories, reviewed, and if accepted are merged into the project's main repository. The pull-based development model is widely used by the projects hosted on GitHub.

In this thesis, we perform two empirical studies – quantitative and qualitative – which investigate pull request (PR) merges of Shopify projects available in its public repository, Active Merchant. In the quantitative study, we performed data mining on the project's GitHub repository to understand the types of merges followed by Shopify developers. We conducted a manual inspection of pull requests and also investigated different factors contribute to PR merge time and outcome. In the second study, we performed a qualitative analysis of the results by performing a survey on the Shopify developers who contributed to Active Merchant. This qualitative study helps in understanding developers' perception of the factors affecting the time and outcome of pull request merges. The results of both studies provide insights on factors influencing time and outcome of PR merges and developers' perception about it.

Dedication

I dedicate my dissertation work to my parents, my husband and all my teachers.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Background

OSS (Open Source Software) is a software in which source code is accessible to public who are interested in it and having the license to research, manipulate and share it. Mozilla Firefox, Chrome, Unix, Drupal, Weka, JasperSoft are some of the examples of the OSS projects. Changes made to a code are known as patches. The main aim of all contributors is to get their patches accepted at the earliest. Major contributions to some OSS projects come from the user community than from the core developers of the company. So these contributions need to be carefully inspected for any bugs/defects before they are merged into the main codebase.

Code review is the process of inspecting the code of developers other than the owner of the code. This process is an important software development practice to reduce the errors in the code and thus, ensuring the quality of the software being developed. Many companies adopt this practice so that the overall performance of the project remains intact. There are generally three types of code review process - formal, email-based and light weight code reviews. While formal code review is the line by line manual inspection of code, email-based reviews are performed using emails exchanged between author and reviewers. Light weight code reviews are informal tool-based reviews. Modern code reviews or contemporary reviews are mostly informal, tool-based ones. Since formal code review requires more time, resources and

cost to complete, light weight reviews are highly recommended in the industry as such reviews need less time and thus, reduce the project's cost.

After the code change (also called a patch) is reviewed, it is merged to the master code repository. Code merging is the process of integrating all changes made in different files to form a single file. Each company has its own policies for code review and code merging. Many developers can work on the same project. So these codes are usually available on code hosting sites as publicly available data increases the efficiency of not only data collection but also the processing of the same. SourceForge [2] and GitHub [3] are two such examples of code hosting websites [4]. Overview of the GitHub's workflow [5] is shown in Figure 1.1.

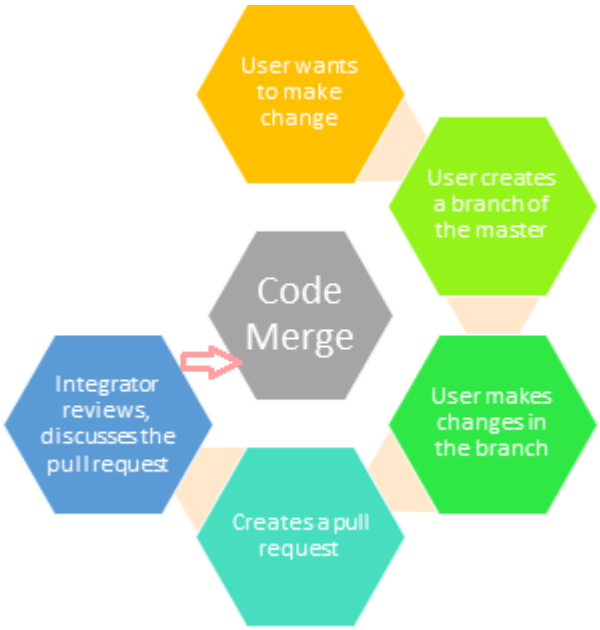


Figure 1.1: GitHub's Overview

1.2 Design of GitHub Code Merge

Git is an open version control system used for software development and other version control tasks. GitHub is a web-based git repository hosting

service where all the files for a particular project are stored and has its own additional features. These days pull-based software development becomes popular among software developers than more traditional development approaches. GitHub and Bitbucket are two hosting services having large number of new and existing projects, e.g., GitHub is estimated to have more than 19.4 million active repositories[1]. Being the largest code hosting service in the world, GitHub also allows its users to conduct code review, issue tracking, and even offers developers some social features like "follow" or "watch" their favourite projects. GitHub follows a fork and pull method where users create a copy of the main repository on their local machine and work on it. The work flow of GitHub is mainly based on the idea of "branching". When a user creates a repository, master branch becomes the default branch. Upon receiving a pull request, the project owners assess the changes made, and decide whether the new code can be merged into the project in the main repository.

A typical workflow can be explained as follows. The user who wants to submit a contribution, creates a branch of the project's repository she wants to modify. Even if changes can be made directly on the main branch, the best practise is to create a branch and do the modifications. After making necessary changes, she has to push the branch to the main repository. This process is also called *commit the changes*. The user's local branches won't be accessible to others until she commits her changes. The contributor, after committing the changes, notifies the core developers about the change by a pull request. Pull requests can be defined as requests given by the contributor to the core developers or project maintainers to review the changes made and committed. Pull request is the initial step of a discussion associated with a commit. A pull request allows interested contributors to review and discuss the changes made. Whenever a pull request is issued, GitHub provides a perfect medium between the contributor and project's maintainer to communicate with each other. Open Source Software (OSS) contributors are mainly volunteers whose contributions can be bug fixes (i.e., error corrections), additional features, etc. Discussions and further changes are made until all the requirements are met. The new code is merged to a master branch and deployed. Figure 1.2 demonstrates this workflow. Thus, code review of the submitted contributions confirms that they don't alter the main functionality of the project.

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

Figure 1.2: GitHub's Workflow.

Code changes can be merged using several techniques, while only few of them are practised by git/GitHub developers. Each of them has his/her own way to merge the branch to its master code. Main contributions being received from casual developers, most of them are either bug fixes or feature additions. While different software projects use pull-based development model, Google, Microsoft and Shopify are among early adopters of this model. Each project practices code merges as per the company policy and rules. A centralized internal tool, CodeFlow is used for peer reviews of submitted code changes in Microsoft. Gerrit is an OSS code review tool implemented internally and used in Google. These tools stand between developers private repository and centralized repository. To merge the new patch, it must be approved and verified by another developer. Some of the previous studies state that size of the code changes, author's experience, status of the reviewer in the project, etc., contribute to the time required and decision to merge [1].

There are different types of merges:

- **Git Merge**: is the simple merge option available in git itself. This merge can be done by pressing the merge button available in Git.
- **Fast-Forward and Merge**: If the master branch has not diverged since the feature branch is created, latest commit of the featured branch will become the master branch. This is called fast-forward(ff) merge.
- **Rebase and Merge**: Rebase merge performs a "replay". If multiple

changes have happened with a pull request, each commit will be merged one by one in the order it is committed.

- **Squash Merge**: In Squash merge, different changes made will be squashed together into a single unit before merging.
- **Cherry-pick Merge**: Another type of merge in which user can choose the commit she wants to merge from the group of commits and performs merging.

1.3 Research Questions and Case Study

Many qualitative and quantitative studies have been performed on OSSs and some of the previous research is done on the GitHub data. GitHub, being storage of both public and commercial projects has been subjected for various research questions. Since the majority of public software projects on GitHub are open source software systems (OSSs), the findings from those researches may be helpful in finding the processes generally followed in open source development. Some of the commercial projects on GitHub allow their repositories to be seen by the public, but maintain strict ownership and tight control over their progressing practices. Such repositories give researchers a unique opportunity to study industrial software development processes. We chose to perform an in-depth study on one of the projects.

An appropriate project was selected based on three criteria. Even if many projects store their master codebase in GitHub, everyday activities were stored in private repositories. Large updates were made to the public repository occasionally [1]. Thus, our first criterion was to select projects that were developed completely on the GitHub platform. Since successful projects are more likely to systematically employ similar practices that supported their success, we decided that second criterion should be selecting projects that were commercially successful. The third criterion was to study a project that had been developed in physical proximity to our lab, 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 develops software for online and

retail stores with the help of API platform and App store [2]. It provides services like shopping cart, payment processing, order management, product catalogs, etc., along with hosting [6]. Even merchants without any coding expertise can create a Shopify account and do sales. As of February 2017, more than 377,500 merchants use this platform to sell commercial resources [7].

Active Merchant is a payment abstraction library that manages and consolidates access to a more than 30 different payment gateways [8] with various internal APIs is maintained by Shopify and Spreedly. It is an open source project responsible for managing e-transactions from different merchants. It aids in credit card validation, payment processes, manage refunds, etc. It can work both as a stand-alone application and as a plug-in. The development of that project is done completely on GitHub.

All pull requests must go through a code review process and need to be approved before being merged into the master branch. Our research included a manual and quantitative analysis of the Active Merchant project repository, as well as an exploratory survey that we conducted with Shopify developers. Our goal is to answer the following research questions.

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

A submitted pull request 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 [1]. We study Shopify data to understand how developers merge PRs, different types of merges followed by Shopify's developers and the effect of merge type on merge time.

2. **[RQ2:] What factors affect the PR review time and decision?**

Previous studies have investigated the effect of a number of elements that influence the time needed to make a decision about a PR, as well as the effect on that decision itself [9], [10]. We investigate the role of these factors in the industrial study.

3. **[RQ3:] What factors affect the PR review time and decision**

[2]https://en.wikipedia.org/wiki/Shopify[28]

according to developer's perspective?

We believe that since the data is extracted from project's public code repositories we can understand the real time scenario partially. To get a better knowledge of the pull-based development process followed by Active Merchant, a survey is conducted among Shopify developers. The analysis of the survey results helped us to understand the developers' perception of pull requests and compare them with the actual results.

Several studies have been performed on different projects based on pull based development models. We conducted our study on a Shopify project and considered file-level metrics for investigating the factors influencing time and decision of PR merges. We followed manual, quantitative and qualitative analysis in our research. We performed a manual analysis of 798 pull requests to identify merge strategy used, built regression models for quantitative analysis and conducted a survey with the Shopify developers as a qualitative component of our research. Our survey received a response rate of 22% which is highly appreciable. While our research could identify the factors affecting time and decision of pull request merges of the ActiveMerchant project, we would highly recommend Shopify to provide researchers with an access to a private repository to investigate more realistic settings of a commercial project.

The rest of the thesis is organized as follows. Chapter II discusses related work. Chapter III describes the methodology we followed in our study. Chapter IV presents the results of our qualitative and quantitative analyses. Chapter V discusses our findings and Chapter VI details challenges and threats to validity. Chapter VII summarizes our research work.

Chapter 2

RELATED WORK

This chapter presents the existing qualitative and quantitative research in various areas of code review. Different approaches are followed by researchers and organizations to understand the factors affecting code merging, code review and OSS.

2.1 Background

Developers interested in contributing to OSS projects, submit them in the form of patches. Since major contributions come from the user community, main aim of some of the studies is to improve patch acceptance from casual contributors and thus, increase the number of contributions from them. Since developers involved in OSS project are geographically distributed, there are more chances for these projects to be successful than traditionally developed projects.

The pull-based model is a comparatively new approach to distributed software development which is developed on well-known and studied concepts, such as user community participation, open-source style development, patch submission, and code review. In pull based development model, code changes are submitted as pull requests. Pull requests are not a replacement for other git-based workflows but a more convenient and accessible addition to them. While submitting a pull request, a contributor needs to provide information

like source and destination branch and repository names. Pull-based model helps in performing activities like discussion, tool integration, evaluation of code quality, management of project milestones/deadlines, etc. Thus, pull requests are a proxy of a code review process.

While the change made to the code is known as commit, pull request (PR) is the request to merge this commit to a different branch. As changes done are never recommended to be on the master directly, commits should be done on branches. Commits should be associated with a commit message which explains the change made. Other contributors can also add commits to the PR by "push" to the same branch. If more commits added to the branch after initiating the pull request, the PR will be revised accordingly. During the code review process, reviewers can comment either on individual lines, as well as on pull requests.

The reviews of these pull requests provide the developers of the project with thorough knowledge of the master branch. Thus, only experts on the project that are familiar with the codebase can take a role of reviewers and conduct pull request merges. Thus, in pull-based development both integrators (i.e., developers who perform pull request merges) and contributors while equally important, have their distinct roles.

2.2 Related Work

Mockus et al. [11] were one of the first researchers to study open source software (OSS) communities. They studied data from Apache and Mozilla projects, by checking their email repositories. They compared Apache project with various commercial projects and developed several hypotheses and refined them based on the Mozilla data analysis. They identified processes like volunteer involvement, and task selection by developer himself as main features of OSS development.

Bird et al. [12] data mined patches that are submitted from casual contributors through project mailing lists to understand the working of OSS projects by studying the files in the project repositories, contributors, their submissions and acceptance level of patches.

Rigby and German [13] studied the code reviews performed in 11 open source

projects. While qualitative analysis helped in identifying multiple code review patterns, quantitative analysis was conducted on Apache server project by examining the developers and commiters in the mailing lists. They observed some inconsistencies in developer list such as developers having multiple email addresses, data inconsistencies like email detached from a thread, etc. The existence of 3 types of review — pre-commit, post-commit and secondary review were identified.

Baysal et al. [14] studied the life cycle of patches in the Mozilla project. Qualitative and quantitative analysis were performed by interviews and discussions with developers and comparison between pre-rapid and post-rapid release models. They found that contributions from casual developers were not accepted as often as those from core developers. Research showed that life cycle of both pre- and post-rapid models are similar and that rapid-release model implemented by Mozilla reduced the waiting time of patches to get reviewed.

Rigby and Storey [15] studied attributes of five OSS projects' review processes, by coding several reviews and core developers' interviews to understand the techniques of handling a large volume of peer reviews in OSS environment. They observed that the patches were sent to all subscribed developers in the mailing list. These developers filtered the emails received, reviewed interesting files and ignored the rest. The stakeholders communicated based on the scope of the project.

Weissgerber et al. [16] studied the email archives of two OSS projects to understand the number of emails having patches in it, time taken for a patch to get accepted and the influence of size of the patch in getting accepted. Data were extracted from emails and CVS repositories by implementing the parser technique used by Bird et al. [12]. They found that smaller patches are reviewed faster and have a higher chance of being accepted. They observed that the size of the patch does not have any significant influence in the duration of a patch to be accepted.

Baysal et al. [17] examined patch life cycle of two industry-led software projects – WebKit and Google Blink, extracted their code review data and compared the review processes to study the factors affecting time and probability of patch acceptance. They found that while technical, organizational, and personal factors affect both the review time and chances of a patch to be accepted, the most influential elements are the contributor's organization

and their experience in the project.

Jiang et al. [18] analyzed patch reviews from Linux kernel mailing lists and patch commits from GitHub repositories to study probability and the duration of patch acceptance. They found that patch writer experience, the amount of review discussion for a patch, 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 capability of reviewers to find bugs in patches and on the software products' quality. They used regression and mixed models to evaluate two datasets. They identified that the quality of product depends on the quality of effort taken throughout its lifecycle and review rate as important factors in identifying more defects. They recommended that reviewers should not proceed faster than 200 LOC per hour to provide high review quality.

Hatton [20] evaluated how valuable checklists are utilized in code assessments, by using formal statistical method. Hatton found that reviewers differ in their capacities to find defects while code reviewing. He also demonstrated that there is a distinction in identification of defects even when the developers review the same code together and independently (76% and 53% of found defects respectively). He also found that checklists used in his experiment were dealing only with code inconsistency.

McIntosh et al. [21] investigated three OSS projects to learn about the relationship between software quality and code review coverage, code review participation and reviewer experience quantitatively. And the findings provided empirical evidence for these factors having strong influence on code quality and bug proneness of software.

Kononenko et al. [22] executed a quantitative study on the quality of the review process in the Mozilla project; by exploring the relationship between quality code reviews and personal and social factors by applying SZZ algorithm. They found that 54% of reviewed patches introduce errors into the software, and showed that elements such as developers' involvement in patch discussion, their experience and workload, as well as patch size, number of developers involved are positively related to code review quality.

In a qualitative study done at Microsoft [23], Bachelli et al. surveyed, inter-

viewed, and conducted a manual inspection of review comments on different teams in Microsoft to find the expectations and real time outcomes of code reviews. They found that despite error identification being the fundamental point of code survey, it is more benefited in areas such as better understanding of project, team involved and in finding more accurate solutions. A few studies found the variables that influence the project review as size and priority of patches, involvement of contributor, time consumed for review, and reviewer's work load.

Kalliamvako et al. [1] researched GitHub repositories to understand features, data, performance and practices of GitHub. They found that majority of the projects were passive and just used for data storage purposes but not for software development. They found that only few of the projects possess pull requests and even if the PRs are merged, 40% of them appeared to be "not merged".

Rigby and Bird [24] performed a quantitative study of six industry-led projects and seven OSS projects which are different in scope of the issue, organizational environment, and development processes. They found that in spite of the differences in the projects, the characteristics of peer reviews are similar. The findings show that with code review there is significant increase in the level of knowledge sharing.

Later in 2016, Kononenko et al. [25] conducted a qualitative study on 88 Mozilla developers. They conducted a survey to understand the perception of the developers on quality of code, the factors affecting the time and decision of a review and challenges faced by them during code review. The study showed that code understanding level, experience of reviewer, workload, review time and technical factors such as patch size are correlated with code quality. Some of the survey participants also mentioned that one the main challenges encountered was the lack of proper tools.

Gousious et al. [9] were among the pioneers in research into pull-based development. They quantitatively investigated OSS projects facilitated on GitHub to understand the factors that are influential to PR review time as well as to the acceptance of PRs; they found that the merge time is affected by certain components, which includes the developer's reputation and the test scope in the project, while the acceptance is basically influenced by the number of recent changes of code that PR wants to change.

Tsay et al. [10] studied technical and social factors affecting pull request acceptance. They found social distance and contributor's status as some of the most important factors influencing pull requests. Pull requests with less comments were highly accepted compared to the ones with more comments. They found that unlike other developers, pull requests from the contributors having previous involvement in the project increased the likelihood of acceptance.

A recent qualitative study by Gousious et al. [26] also investigated the practices of pull-based development model from the perspective of integrators. They performed two rounds of survey with GitHub integrators to identify the factors influencing their decision to accept or reject a pull request. They found that while contribution quality and prioritization of the changes made are the most significant elements affecting the decision to accept or reject a pull request, reviewer availability and developer responsiveness contributes towards the time to make a decision. They additionally found that the integrators confront different difficulties, for example, keeping up project quality and choosing which PRs to organize.

Marlow et al. [27] performed a qualitative study to understand opinion forming in online peer production by interviewing GitHub users. They investigated how core developers from GitHub projects form their opinions of the incoming contributions and their developers. They found that integrators use information like contributor's track record of coding and their actions on GitHub like interactions with other users, etc., to assess a novice developer.

In another study, Gousious et al. [28] surveyed 645 most active contributors on GitHub projects to study their work practices and the challenges in pull-based development from the perspective of contributors. They found that integrator responsiveness, maintaining project awareness, external as well as pull request communication, and quality assessment as key factors motivating developers to contribute in the pull-based model. The main challenge identified by the contributors is poor responsiveness from core developers.

Rahman and Roy [29] conducted a comparative study on successful and unsuccessful pull requests of 78 GitHub projects and found that programming language and domain of a project, number of developers involved , developers experience and number of forks of a project affects the success and failure rates of pull requests.

Chapter 3

METHODOLOGY

In this chapter, we describe the methodology we followed to conduct our study. The methodology consists of following steps: data mining (Section 3.1), quantitative data analysis (Section 3.2), and survey design and setup (Section 3.3).

3.1 Data Mining

Data mining comprises of data collection, data pre-processing, data storage, manual analysis of data, manual analysis of pull requests, identification of PR merge types and creation of new dataset.

3.1.1 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 any of its data [3]. We used an official library developed by GitHub rather using them directly to make API calls [30] as it can handle the technical part of communication with servers.

Initially, we wanted to study the data during the latest 1 year and examined the contributions available in Active merchant since October 2015 to October 2016. Since the data points for the study was very less, we have considered our study period to 4 years, i.e., from January 1, 2012 and October 1, 2016; a total of 1,657 pull requests (PRs) were extracted. During the extraction of data, we observed several information associated with each PR (Refer Figure 3.1), such as the author's unique ID, the date in which the PR was added to the repository and closed, status of merge, i.e., whether the PR was merged or closed and its date, commits in each PR and its ID, commiter name, commit messages, total number of lines changed (number of lines added + number of lines deleted), the description of the PR, and its size statistics.[1] Both comments in pull requests and in-code from the developers are also collected for each PR; comments are the comments/messages associated with PR while in-code comments are comments seen in the file level of a PR.

3.1.2 Data Pre-processing

We attempted to discard any anomalies by applying three filters to limit any potential noise in the gathered data.

- 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.
- 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 data set was reduced to

[1]We did not calculate the size statistics ourselves; but entrusted upon the values provided by GitHub.

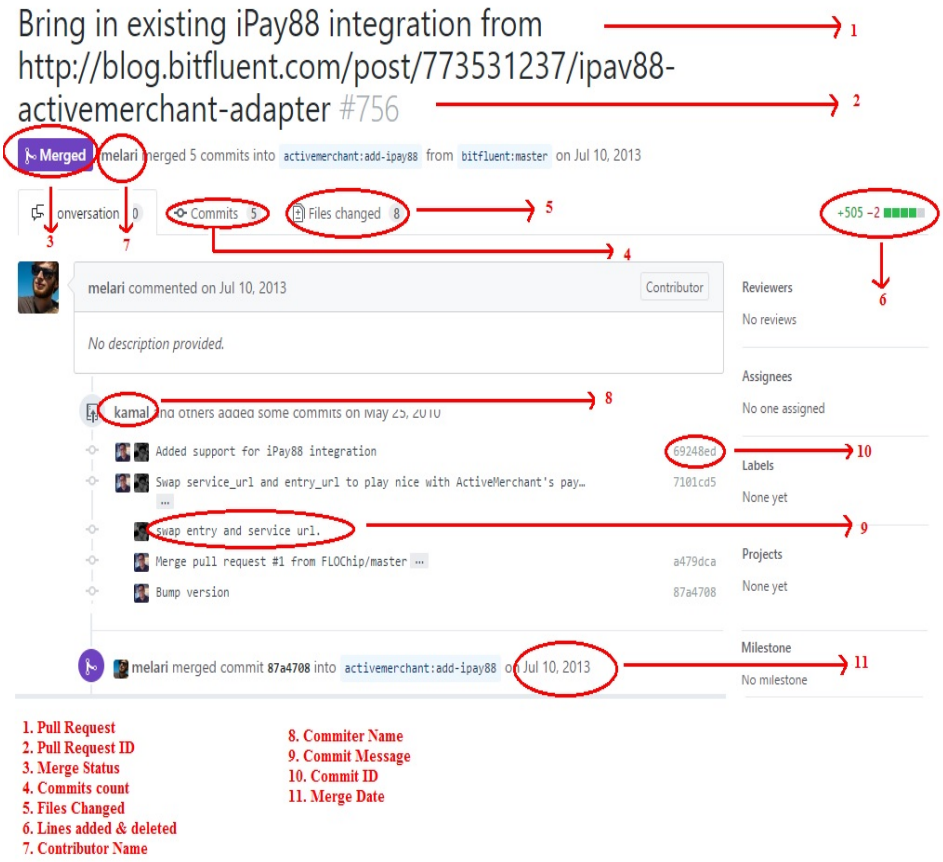


Figure 3.1: Sample Pull Request Page.

1,475 pull requests.

Despite the fact that GitHub enables a PR to be allocated to a particular developer, we found that this aspect was seldom utilized inside the Active Merchant repository. In this manner, one of the challenges we confronted was deciding the correct time limits of the review period. We considered the PR submission date to be the date that the review began. Since a PR can't be merged before it has passed the review, we considered the date a PR was closed as the date the review finished. The time between the pull requests begin and end dates is calculated as review time duration.

Another challenge we needed to overcome was the absence of standardized flags or labels in GitHub to show the result of a pull request review. In the Active Merchant repository, a few commentators included a textual comment, while some use emoticons, and some others incorporate images of boats (meaning "ship it"). We marked all closed and merged PRs as the ones that successfully passed the review, and all closed yet not merged PRs as the ones that got a negative review. Tsay et al. and Gousios et al. made similar indications [9, 10].

3.1.3 Data Storage

The Active Merchant data available during the period of January 1, 2012 to October 1, 2016 and stored in a SQLite database. We have created six tables to store the data and named them as *commits*, *people*, *pull_request*, *pr_comments*, *pr_incode_comment*, *pr_state*. Table 3.1 shows the number of records in each table.

Table Name	**Records Count**
commits	3,617
people	976
pr_comments	5,365
pr_incode_comments	2,723
pr_state	2
pull_requests	1,824

Table 3.1: Tables and Records Count in Raw Dataset.

Table *"commits"* contains all details of each of the commits such as *sha*, *pull request ID*, *author*, *commiter*, *date*, *comments*, *LOC*. The *sha* variable is a hash identifier which is unique for each commit. While *pull_request_id* is the ID for each pull request, *author* and *author_date* shows who has made the change and on what date. The *commiter* variable stores commiter ID and *commiter_date* shows the date on which the commit was submitted. The *message* variable stores the description/comment in the commit. *LOC* accounts towards the number of lines added and deleted.

Table Name	**Variables**
commits	sha, pull_request_id, author, author_date, commiter, commiter_date, message, additions, deletions
people	id, name, email, company, login, not_existing, name_git, email_git, company_custom
pr_comments	id, pr_id, created_at, author, comment
pr_incode_comments	pr_id, id, created_at,text,author
pr_state	id, name
pull_requests	id, title, body, submitted_on, is_merged, is_merged_custom, merged_on, author, merger, state, additions, deletions, files_cnt, closed_at, commits_cnt, comments_cnt, author_exp, author_exp_v2

Table 3.2: Table Details in Raw Dataset.

Table *"people"* details about the *name*, *email ID*, *company*, *login name*, *git name* and *git email ID* , *company_custom* , etc. While *company* variable holds the organization to which author is affiliated, *company_custom* variable is created to filter Shopify and Spreedly developers. Table *"pr_comments"* contains details like *pull request comments ID*, *author ID* , *comment* , and *created time* . Table *"pr_incode_comments"* accounts to the comments within the code associated with a pull request. This table also contains *incode comment ID* , *author ID* , *created time* and a text inside the code which is termed as *incode_comment*. Table *" pr_state"* holds only two variables, *ID* and *name* . Variable *name* stores 2 values - "open" and "close" - which shows whether the pull request is merged or not merged. Table *"pull_requests"* holds all

details such as a *unique ID* , *title* , *body* which holds a detailed message, *date* shows the PR submited date, *is_merged* and *is_merged_custom* shows merged status , *merged_on* indicates the merged date, *author* of pull request , *merger* of pull request. *state* variable shows the pull request state, i.e., whether PR is closed or open, *additions* accounts for number of lines added , *deletions* for the number of lines deleted, *files_cnt* accounts for the number of files, *closed_at* indicates pull request closed time, *commits_cnt* and *comments_cnt* in each pull request, *author_exp* holds the author's experience based on pull request before that submitting the current pull request and *author_exp_v2* shows the author's experience based on commits before submitting the current pull request. Table 3.2 shows the summary of each table in the raw dataset.

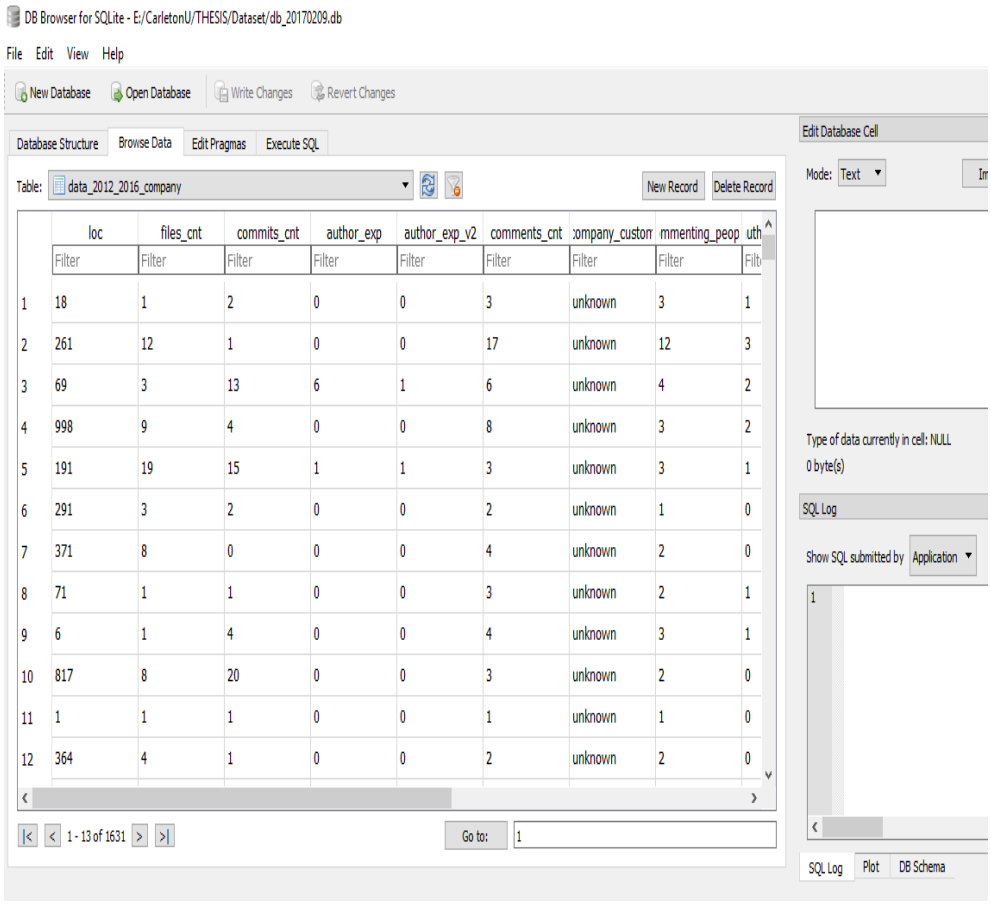
3.1.4 New Table and Explanatory Factors

Three tables are primarily used to create the new table for the study. Join command is used to join ‘‘pull_requests”, ‘‘pr_comments” and ‘‘pr_incode_comments” tables.

```
select pr.id,
       (pr.additions + pr.deletions) as pr_size, pr.files_cnt,
          pr.commits_cnt, pr.author_exp, pr.author_exp_v2,
          pr.comments_cnt, p.company_custom,
       COALESCE(prc_.cnt_comment_authors, 0) commenting_people,
          COALESCE(prc_.cnt_comments, 0) cnt_comments,
       COALESCE(prc_author.author_comments_cnt, 0) author_comments,
       coalesce(incode.cnt_comments, 0) incode_comments,
          coalesce(incode.cnt_comment_authors, 0) incode_people,
       COALESCE(incode_author.author_comments_cnt, 0)
          incode_author_comments,
       pr.is_merged, pr.is_merged_custom,
       coalesce(is_merged_custom, is_merged) is_merged_combined,
       ( julianday(pr.closed_at) - julianday(pr.submitted_on) ) *
          24 * 60 review_time_minutes
from pull_requests pr
inner join people p on pr.author = p.id

left join (SELECT prc.pr_id, count(DISTINCT prc.author)
```

```
    cnt_comment_authors, count(prc.id) cnt_comments
            FROM pr_comments prc
            GROUP BY prc.pr_id) as prc_ on prc_.pr_id = pr.id
left join ( SELECT t1.id, count(t2.id) author_comments_cnt
            from pull_requests t1 inner join pr_comments t2 on t1.id
                = t2.pr_id
            WHERE t1.author = t2.author
            group by t1.id
          ) as prc_author on pr.id = prc_author.id

left join ( SELECT t.pr_id, count(DISTINCT t.author)
    cnt_comment_authors, count(t.id) cnt_comments
            FROM pr_incode_comments t
            group by t.pr_id
          ) as incode on incode.pr_id = pr.id
left join ( SELECT t1.id, count(t2.id) author_comments_cnt
            FROM pull_requests t1 inner join pr_incode_comments t2
                on t1.id = t2.pr_id
            WHERE t1.author = t2.author
            GROUP BY t1.id
          ) as incode_author on pr.id = incode_author.id

WHERE pr.closed_at between '2012-01-01' and '2016-10-01'
```

Filtered dataset is stored in a new table *data_2012_2016_company* that contains filtered data from all raw tables. Thus, the newly created table variables are *pr_size* (sum of added and removed LOC), *files_cnt*, *commits_cnt*, *author_exp* (author experience based on pull requests), *author_exp_v2* (author's experience based on commits), *comments_cnt* (total number of comments in each pull request), *commenting_people* (number of people who have commented for each pull request as many people can comment on each pull requests), *authors_comments* (among all the comments for each pull requests we look at the count of comments received from the author of pull request), *incode_comments* (number of comments within changed codein the file level), *incode_people* (count of all contributors who have written code change in the file level), *incode_author_comments* (number of comments inside code from the author of pull request), *is_merged* (merged status), *review_time_minutes* (review time in minutes format). Review time duration is calculated by find-



	loc	files_cnt	commits_cnt	author_exp	author_exp_v2	comments_cnt	:ompany_custom	mmenting_peop	uth
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filt
1	18	1	2	0	0	3	unknown	3	1
2	261	12	1	0	0	17	unknown	12	3
3	69	3	13	6	1	6	unknown	4	2
4	998	9	4	0	0	8	unknown	3	2
5	191	19	15	1	1	3	unknown	3	1
6	291	3	2	0	0	2	unknown	1	0
7	371	8	0	0	0	4	unknown	2	0
8	71	1	1	0	0	3	unknown	2	1
9	6	1	4	0	0	4	unknown	3	1
10	817	8	20	0	0	3	unknown	2	0
11	1	1	1	0	0	1	unknown	1	0
12	364	4	1	0	0	2	unknown	2	0

Figure 3.2: A Screenshot of a New Dataset.

ing difference between submitted date and closed date and converted into minutes format. The SQL query shows that we have selected records associated with unique authors. The new table contains 1,631 records. Figure 3.2 shows a screen shot of a new table. Null values in all variables in the new table is replaced with 0 as NULL value can create errors while doing quantitative analysis. Almost all variables in *"people"* table is omitted. But *"people"* table is mainly used to study research question 1. The variable *sha* in commits table, variables in *pr_state* table, state variable in *pull_requests* table are ignored as it holds less importance in the research.

Previous research suggests a set of different metrics that can affect code

review time and outcome.While Baysal et al. [17] found that contributor's organization and their experience in the project as the influencing factors, Gousios et al. [9] found that developer's reputation and the test scope in the project influences code review time, and the number of recent changes of code affects the outcome. Table 3.3 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 mined data (i.e., we did not include a factor if we could not collect the data corresponding to that factor or if we needed to apply some heuristic to compute its value). In our study developer experience was calculated not based on his experience in project but by calculating the number of PRs and commits submitted by PR author prior to the current PR.

Explanatory Factor	**Description**
PR size	Sum of added and removed LOC
files count	Number of files changed by a PR
commits count	Number of commits in a PR
author experience on PR	Experience based on the number of prior PRs submitted by PR author
author experience on commit	Experience based on the number of commits submitted by PR author
comments count	Number of comments left on a PR
author comments count	Number of comments left by the PR author
commenting developers count	Number of developers participating in discussion
in-code comments count	Number of comments left on source code
author's in-code comments count	Number of comments left on source code by author
in-code commenting developers count	Number of developers who commented on source code
PR author's organization	An organization that a PR author affiliates with

Table 3.3: Overview of the Factors Studied.

3.1.5 Manual Analysis of Data

To identify the Shopify developers and merge types used by them, initially a manual analysis is done on each table data. Tables *"people"*, *"pr_comments"*, and *"pr_incode_comments"* are studied in detail to know about the Shopify developers and the merge types used by them. We found that, among the total 1,631 records in the newly created table *data_2012_2016_company*, we could only identify 88 (5.3 %) contributors with any affiliation. As per GitHub, a user can offer a contribution only if she creates an account with a username, email ID and password. But anyone can inspect and download from public repositories. An account can be created by individual users, as well as organizations. In registered organization account, users can be added into organization's GitHub account. GitHub policy allows only registered users to create, operate and control contributions, repositories and review changes in code. But the detailed study of *"people"* table shows that there are quite a few commits which cannot be verified for authorship. Majority of the contributor's organization was also not specified. Table 3.4 presents the number of contributors with the identified affiliation.

Organization	**Contributor Count**
Shopify	10
Spreedly	4
Others	74
Unknown	1,543

Table 3.4: Contributors' Affiliation - Initial Data.

As Shopify and Spreedly are the maintainers of Active Merchant, these values cannot be accurate. This analysis indicates that many users are not using their authentic account details for contributions. Even though the list of developers was requested from Shopify, it was incomplete. And this turned out to be one of the main challenges as we could not identify all Shopify developers.

GitHub allows its users to create dummy accounts, i.e., creating without even a valid email ID. So GitHub user accounts of several Active Merchant contributors do not contain user information or a valid email address, or some accounts do not even have an email address itself. To recover missing email addresses, actual commits are extracted using API from the repository. One

field of API response is used to extract all commits, and the other one to get commit's details like count of lines added and removed. Github provides a unique ID for each commiter and git stores commit author details in the commit header as it always preserves author email address.

User information is first retrieved from GitHub. Since this research is on the Shopify data, the easiest way to get a list of commiters who worked at Shopify was to use the *git shortlog* command in the Active Merchant repository which gets you all users and the number of commits under their name. After this step, *grep* command is executed on the email addresses of the users and looked for `shopify.com` and `jadedpixel.com` email addresses. "jadedpixel" mail domain is also from Shopify users as it used to be the parent company of Shopify back in the days. Whenever a missing data is encountered, name and email address is extracted from commit header in git. Thus, we are able to reduce the number of unknown authors, even if there are several possibilities of incorrect email addresses. Emails are parsed and set on the basis of domain names except the public ones like Gmail, Yahoo mail, etc., for retrieving the email addresses. Thus, we were able to identify the organizations of more contributors. Table 3.5 presents the contributor's affiliation after the detailed analysis. Contributors from Shopify were increased to 22% and those from Spreedly became 20%, whereas "unknown" authors demonstrated a decline to 57%.

Organization	**Contributor Count**
Shopify	362
Spreedly	338
Unknown	931

Table 3.5: Contributors' Affiliation - Final Data.

3.1.6 Manual Inspection of Pull Requests

We wanted to identify the merge types used by Shopify developers. As an initial step, a manual inspection is performed on the "merged" pull requests since there are chances for the pull requests can be marked as merged even though they are not merged. Among the total 1,824 records in pull_requests table (Refer to Table 3.1), 798 records are having "merged" status. Keeping

the heuristics of Kalliamvakou et al. [1] as basis, we performed a manual inspection.

First step of this inspection was to select 20 pull requests and let two coders (one being the author of this thesis and the other one — a PhD student) label them individually. This was to make sure that both of the coders understand the heuristics to identify the merge types in the same manner. Next step was to compare the labels of both coders and calculated percentage agreement as a mode of reliability score. Both of the coders reached 93% of agreement between them in their findings, while they differed in their opinions in two pull requests. We analyzed both pull requests and reached to a final agreement. Later, the rest of the 778 pull requests were divided into two batches and both coders inspected and tagged one batch each separately. Seven pull requests were identified to be incorrectly marked as "merged".

3.1.7 PR Merge Types

Kalliamvakou et al., in one of their studies mentioned that GitHub data is not always reliable regarding whether a PR has been merged or not [1]. Since integrators can merge the commits in different ways there exists the chances of displayed merge information and actual merge status may differ. Kalliamvakou et al. [1] has suggested some heuristics to find the merge flags which are displayed. These heuristics are developed on the basis of master branch commits and last commit contents in a PR. The heuristics [1] are as follows:

By applying these heuristics, among the total 1,657 pull requests 798 pull requests mentioned as "not merged" by GitHub were marked as "merged" ones. One of the perils mentioned by Kalliamvakou et al. [1] was that the heuristics suggested by them can generate large numbers of false positives. Keeping this in mind we did a manual inspection of the merged pull requests so that the risk of false positives can be minimized. This inspection provided a chance to label each of the pull requests according to the merge type labels suggested by Kalliamvakou et al. [1]. Thus, we identified 3 types of merges that are performed by the Shopify developers.

- **GitHub Merge** : Github merge is a regular default commit which is executed by using GitHub facility - using "Merge" button available.

Heuristic	Description
H1	At least one of the commits in the pull request appears in the target project's master branch.
H2	A commit closes the pull request using its log (e.g., if the log of the commit includes one of the closing keywords, see above) and that commit appears in the project's master branch. This means that the pull request commits were squashed onto one commit and this commit was merged.
H3	One of the last 3 (in order of appearance) discussion comments contains a commit unique identifier, this commit appears in the project's master branch and the corresponding comment can be matched by specifically formatted expression.
H4	The latest comment prior to closing the pull request matches the regular expression above.

Table 3.6: Heuristics to Determine Merge Flags [1].

The merge button on GitHub retains every commit in the working branch as well as insert them along with commits on the master branch. Figure 3.3 illustrates the working of Git Merge. New features added will be unified and committed to the master merge. The newly merged commit will become the new master. The merge can be performed using the command *git merge –no-ff*. Even if the merge output looks complicated, it shows a detail and precise commit log and it portrays how changes from a feature branch landed in the base branch.

- **Cherry-pick Merge** : Developer chooses a subset of commits from a PR and adds it to the repository without any changes; thus, in turn merging pull request branch with the main branch. Figure 3.4 shows "D", "E", "F" as the features added from different branches and the developer cherry-picks "D" and merge it to the master branch and "D" becomes the new master head. When the commit is merged into the new branch, it will get a new commit ID and it will have a new parent commit. Cherry picking does not store any commit log history, but it keeps authorship details of commits.

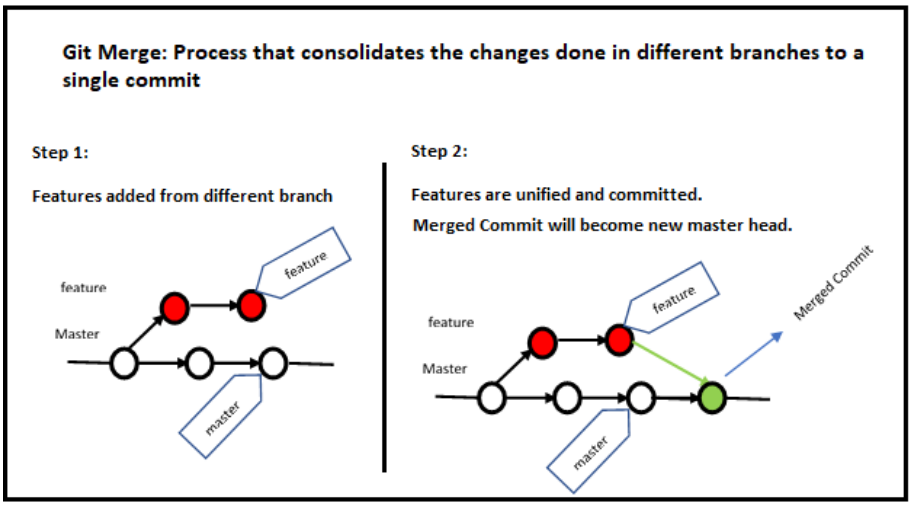


Figure 3.3: Git Merge.

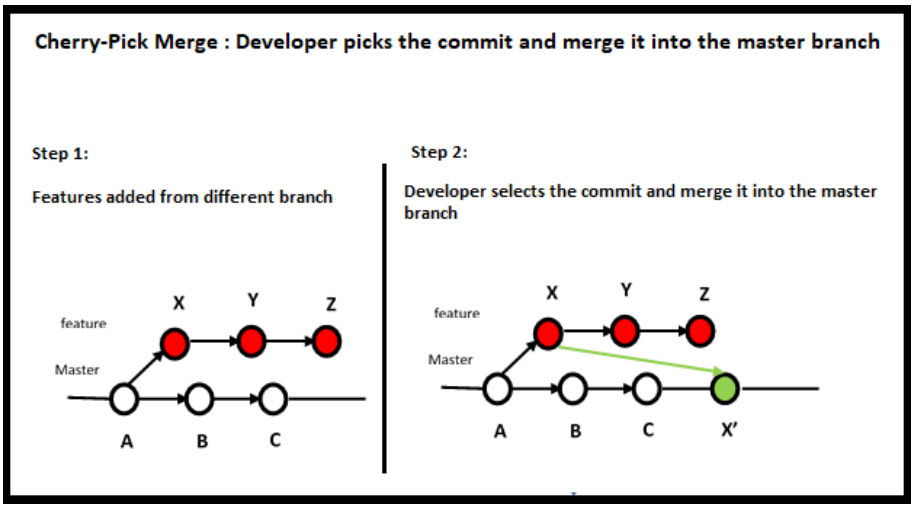


Figure 3.4: Cherry-Pick Merge.

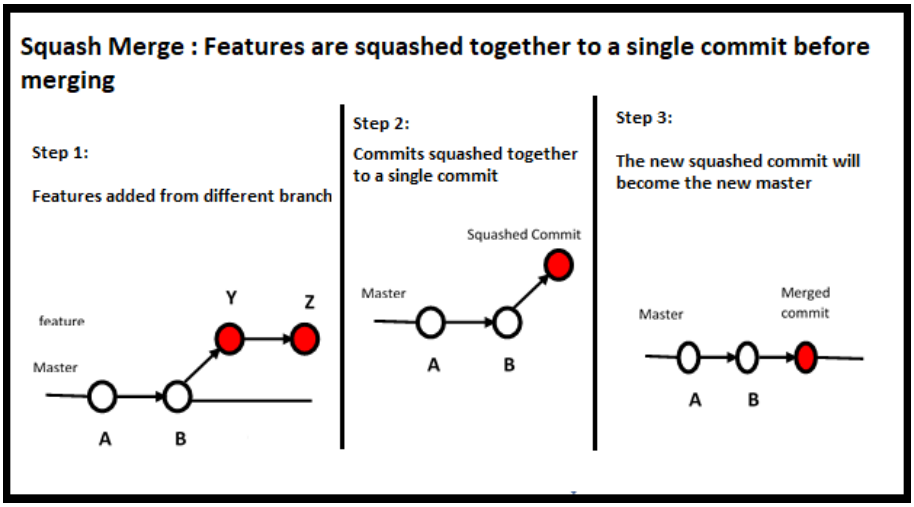


Figure 3.5: Commit Squashing Merge.

- **Commit Squashing** : Contributors always prefer short and frequent commits as they are accountable for better testing, reviews, quality analysis, etc. As the project progresses, there are more chances for the contributions get piled up in the commit log. Squash merging helps in keeping the log history short. In "Squash and Commit" merge, the developer integrates every intermediate commit associated with a pull request and creates a new commit. If required, additional changes like adding or deleting features can also be done during this process of squashing. Once squashing gets completed, the newly created commit will be merged to the master branch and it will become the master node. Figure 3.5 explains squashing in 3 steps. First step shows new features getting added. Second step shows squashing of all commits into a single commit. In the last step the new commit is merged to the master branch and becomes the master. During squash merge, the changes are retained but individual commits (the details on how the feature branch developed throughout) can be lost [5]. Thus, squash merging is chosen depending on whether the commit history needs to be maintained or not.

3.2 Quantitative Data Analysis

To study the data quantitatively, we performed a regression analysis for evaluating the connections among factors. In statistics, regression analysis aims to find the best relationship between a response random variable `Y` (regressant) and `n` independent variables $x_1 - x_n$ (regressors) in a number of observations [31]. Regression analysis is generally used for "prediction" and to find the relationship between independent and dependent variables and the effect of multiple independent variables on any single dependent variable. Thus, regression is one of the main tool for modelling and data analysis. Different regression techniques available are Linear Regression, Logistic Regression, Polynomial Regression, Step-wise Regression, Lasso Regression, Elasticnet Regression, etc. Simple linear regression and multiple linear regression is similar except the former has one independent variable, whereas the latter has multiple independent variables.

We used Multiple Linear Regression and Logistic Regression to learn the effect of different variables selected on review time and decision respectively. The reasons for selecting these regression models for our quantitative analysis are as follows. Linear regression is the most popular modeling technique used by those learning regression modeling. Logistic Regression is mainly used for classification problems [32]. This regression models aids in finding the success or failure probability of a dependent variable. In another previous case study, Tsay et al. [10] also used same logistic regression statistical model to find the factors that influence the decision to merge a pull request.

In our study we analyze the influence of factors described in Table 3.3 with variables such as *pull request review time* and *merge decision*. Multiple linear regression model with *company* variable is as follows (note: we used R statistical software for doing our analysis):

```
Multiple Linear Regression Model:

mlr_time = lm(review_time_minutes ~ custom_log(loc) + files_cnt +
    commits_cnt + author_exp + author_exp_v2 + comments_cnt +
    company_custom + commenting_people + author_comments +
    incode_comments + incode_people + incode_author_comments +
    merge_type, data = data ) }
```

In this model *review_time_minutes* becomes the dependent variable, and *loc*, *files_cnt*, *commits_cnt*, *author_exp*, *author_exp_v2*, *comments_cnt* , *company_custom*, *commenting_people*, *author_comments*, *incode_comments*, *incode_people*, *incode_author_comments*, *merge_type* become the independent variables. Step-wise regression modelling is also performed to remove the less important variables from the model.

Since merge decision variable can be either "true" or "false", we selected logistic regression modelling. The logistic regression model using R looks as follows:

```
Logistic Regression Model:

fit_positivity_company_logistic =
    glm(is_merged_combined.f~custom_log(loc) + files_cnt +
    commits_cnt + author_exp + author_exp_v2 + comments_cnt +
    commenting_people + author_comments + incode_comments +
    incode_people + incode_author_comments + company_custom +
    review_time_minutes, data = data, family=binomial() )
```

In this model *is_merged_combined* variable becomes the dependent variable, and *loc*, *files_cnt*, *commits_cnt*, *author_exp*, *author_exp_v2*, *comments_cnt* , *commenting_people*, *author_comments*, *incode_comments*, *incode_people*, *incode_author_comments*, *company_custom* and *review_time_minutes* become the independent variables. Anova model is also used to ensure the correctness of logistic regression results.

Previous studies of Cataldo et al. [33], Mockus et al. [34], McIntosh et al. [35], and Kononenko et al. [22] also focused on finding the correlation among different variables and the models we created for training the classifier are similar to the ones in these studies.

Variable Transformation.

Previous studies offer empirical evidence that software engineering data is seldom normally distributed [21]. Transformation of variables can change the distorted variable distribution to a normal one. There are mainly two types of transformations: linear and non-linear. While linear transformation retains the linear transformation between variables, non-linear transforma-

tion alters it by either rising or declining the linear relationship of variables. Transformation is usually applied on the independent variables as the dependent variable transformation can affect the distribution of residuals in the model. To reduce any possible imbalance in the data, we used a log transformation – the most commonly used non-linear transformation – in our study.

```
custom_log <- function(x) ifelse(x <= 0, 0, base::log(x))
```

This customized log function is applied to the continuous variable *file size* which turns out to be LOC in our data. 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. Dummy or dichotomous variables are independent variables which take the value of either 0 or 1 [36].

Controlling Multicollinearity. Multicollinearity shows that two or more predictor variables are highly correlated. Multicollinearity describes how each independent variables can influence dependent variables. Multicollinearity in models is inspected using the variance inflation factor (VIF) which is the most commonly used one. The variance of a variable with respect to the collinearity level of other variables is represented by VIF score. We used `vif` function from the `R car` package to calculate VIF scores [37]. With reference to [38], VIF score threshold was set to 5. The VIF is measured for every predictor by performing a iterative linear regression of that predictor on all the other predictors. In this iterative process, we examined that our models contain only predictors with VIF values less than the starting value. In every iteration, if predictors with higher VIF value than the threshold is found, we discarded that variable and the VIF values are recalculated.

3.3 Survey Design and Participants

We conducted a survey with Shopify developers to understand the practices they follow in Shopify and their perception about the pull request reviews. We conducted a survey similar to some of the previous studies by Gousios et al. [9] and Kononenko et al. [25]. The 10 survey questions could be categorized into two groups. While queries regarding demographic information

and work practices of participants dominated the 8 questions, the information regarding factors affecting time and outcome of pull requests in Shopify covered the remaining questions. Even if Spreedly and Shopify are the maintainers of Active Merchant, we targeted mainly Shopify developers as Active Merchant is their product and they are the main contributors.

As an initial step we sent a common email to all Shopify people for which we couldn't get any responses. We identified 88 developers from Shopify who are involved in pull requests and send 78 personal invitation emails to participate in the survey as our database was missing email addresses of rest of the 10 developers. All the questions were mandatory and we informed the participants that our survey would take 10 – 15 minutes to complete. We got automated responses for five of the 78 emails mentioning that the email address had been deactivated. We figured out that those developers are not associated with Shopify any more. We kept the survey open for two weeks – from February 27, 2017 to March 13, 2017. We received 16 responses from the 78 selected participants. Even though the responses count look small, the response rate of 22% (16/73) is higher than the minimum response rate of 10% mentioned by Groves et al. [39].

The first question was about the role of participant in the organization. The second query was about the experience of participant on the basis of years. This is to check whether the experience of developers influence the time and outcome of merge. While third question was about the number of pull requests the participant submit every week, the next one was about the number of pull requests they review every week. The fifth question examined the participants experience in the domain of pull request. Sixth and seventh ones analyzed the type of merges the participant do. Next question seeks an answer for the pull request modes of discussion. While the 9th question investigated the factors affecting time required for pull requests, the last question seek participants' perception about the factors influencing the outcome of pull request merge.

We have included our survey form in Appendix A.1.

Chapter 4

RESULTS

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

4.1 RQ1: Merge Types and Their Effect on Review Time.

In traditional patch based development model, the contributions made are sent for approval. Once the contribution is approved, it will be added to the project's master repository. In the situations where a contribution has undergone a number of revisions before getting approval, the final version will only get added to the main repository. Pull-based model has developed a new method to manage the incoming contributions to any software projects. Developers using pull-based models are privileged with more options to handle pull requests. They can choose either a portion or a complete pull requests while integrating them to the project. Developers are equipped with more options if they merge the full pull request: they can either leave commits from a PR untouched thereby preserving more historical information or squash the commits into a single commit and merging to the main repository resulting in a cleaner commit history on the main branch. Even though many researches focus on the different aspects of pull-based development, we could not find any study that investigates the types of merges used by developers. As an

initial step, we took an exploratory study on PR merge strategies.

We performed different heuristics to understand the merged pull requests as specified in Section 3.1. To evaluate our findings using these heuristics, we checked each pull requests in the dataset manually and marked the merged ones as *"merge"*. Results of the manual inspection are reported in Table 4.1. We found that about 25% of pull requests were only merged using GitHub User Interface which was very much contradictory to our perception. The reason behind the least preference of GitHub merge can be because the merge process using "merge" button can happen only if (a) the changes made are final (there are no further changes required), and (b) GitHub has the feature of resolving merge conflicts automatically, if it occurs. In our manual study, we observed that merger makes several further changes in many of the pull requests. While merging a PR, developers usually updated `changelog`, `readme`, and/or `contributors` files to reflect the new change.

Merge type	**Count**	**Percent**
Not merged	465	28.49%
GitHub	385	23.59%
Squashing	657	40.25%
Cherry-picking	124	7.59%

Table 4.1: Manual Inspection of Pull Requests by a Merge Type.

In our study we identified squash merge as the most preferred merge type by the developers. Even though some historical information is lost during this merge process, several benefits are also there. The main advantage of squash merging is that developers preserve the commit history in master branch transparent and clean. Since each commit depicts each pull request, contributors can guarantee descriptive log message with each commit, reverting a merged pull request (if needed) will be easier for developers. Likewise, squashing consistently accommodates both further changes made by mergers as well as automatic closing of a pull request.

The manual inspection of pull requests on the basis of merge type showed that median time (in minutes) for cherry-picking merge is 13,549.9, GitHub merge is 6,914.45, squashing is 1,475.30, not_merged is 7,750.476 (as shown in Table 4.2). Cherry-picking showed highest median time, whereas squashing needed least time. We performed two non-parametric statistical tests to

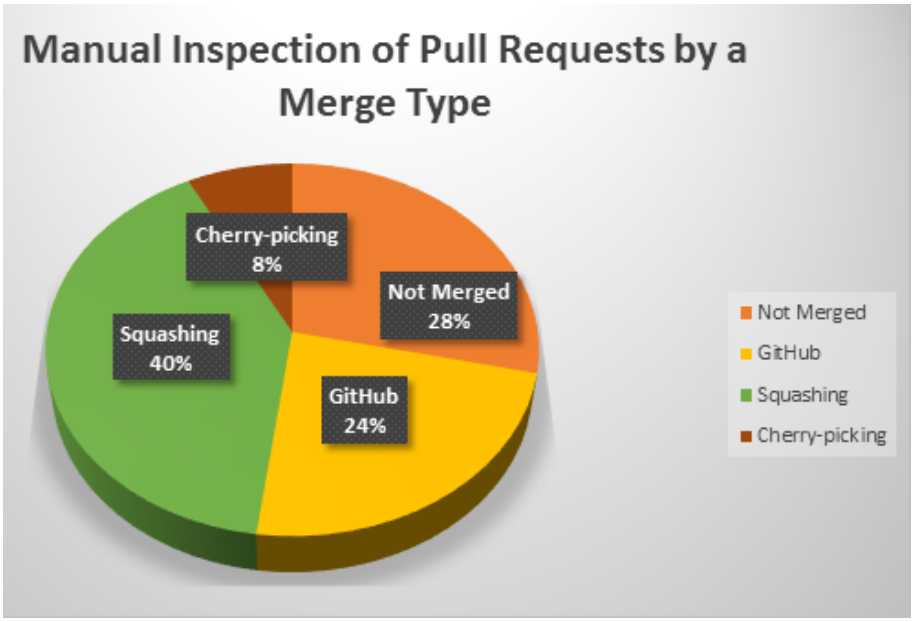


Figure 4.1: Graphical Representation of Merge Type Count.

validate our findings in the manual inspection — 1) Kruskal-Wallis analysis of variance [40], 2) a post-hoc Mann-Whitney U (MWW) test [41].

Merge Type	**Median Review Time (in min)**
Not merged	10,111.5
GitHub	1,107.8
Squashing	4,241.8
Cherry-picking	3,177.6

Table 4.2: Merge Types and Median Review Time.

The Kruskal-Wallis test assesses the important differences on a continuous dependent variable based on an independent variable. The Kruskal-Wallis test showed that merge type has a statistically significant importance on time ($\chi^2(3)$=89.02, $p < 0.001$). But this test does not reveal the significance of the variable, we preformed pairwise comparison using MWW test with Bonferroni correction. We assessed the following pairs – *GitHub* & *squash*, *GitHub* & *cherry*, *Squash* & *cherry*, *not_merged* & *GitHub*, *not_merged* & *squash*, *not_merged* & *cherry*. The test showed statistical difference among all except *cherry* & *squash* pair. Even though the median time for cherry

picking is lesser than that of squashing there is no statistical relevance in the difference found. The biggest difference shown in median time between not merged pull requests and merged pull requests might show that if a PR is going to be accepted, it will be accepted quickly; if it is utilizing more time there are more chances for those pull requests not to get merged and will be eventually discarded.

RQ1: While 40.25% of developers merge pull request via squash mege technique, PR merge types such as GitHub(23.59%) and cherry-pick(7.59%) are also a typical practice. Our findings show that merge type has a statistically significant impact on PR merge time. GitHub merging scored less time to merge than cherry-pick and squash merges.

4.2 RQ2: Factors Affecting Merge Time and Decision

Several investigations [25] are done earlier to find the factors affecting time to merge a pull request and decision to merge or not a pull request. We build two regression models to study the factors influencing the time to merge a pull request and outcome of merge. Multiple Linear Regression Model is used for PR review time, and Logistic Regression Model is used for PR review decision (Section 3.2). These models adopted variables mentioned in Table 3.3 as independent variables.

Merge Time. As an initial step, we developed the multiple linear regression model for time factor without the independent *company* variable as we couldn't gather information regarding the developers affiliation. We found that *merge type*, *file size*, and *commenting people* are factors affecting time to merge a pull request (as shown in Table 4.3).

We decided to train the model with *company* variable after finding the affiliation of contributors. Table 4.6 shows the regression coefficients for each of the studied factors. The results are depicted in Table 4.4 that *file size*, *commenting people*, *author comments* and *company* are factors affecting time to merge a pull request. The MLR model indicates that the *PR size* has a statistically significant effect on review time. The positive value of the regression coefficient implies that as the *PR size* increases, *review time* for pull

Coefficients	Estimate	Std. Error	t value	p-value
(Intercept)	51803.5	6753.9	7.670	2.94e-14 ***
custom_log(loc)	3238.4	1206.2	2.685	0.00733 **
commenting_people	13517.0	1884.2	7.174	1.10e-12 ***
author_comments	-2918.2	1637.3	-1.782	0.07488 ·
incode_comments	274.3	399.1	0.687	0.49203
merge_typecherry	-56560.0	9180.8	-6.161	9.11e-10 ***
merge_typegithub	-73685.8	6221.8	-11.843	< 2e-16 ***
merge_typesquash	-48183.4	5516.3	-8.735	< 2e-16 ***

Table 4.3: MLR Model without "Company" Variable.

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 88460 on 1623 degrees of freedom
Multiple R-squared: 0.1416, Adjusted R-squared: 0.1379
F-statistic: 38.25 on 7 and 1623 DF, p-value: < 2.2e-16

requests also increases. The number of people involved in commenting is also statistically important. The model shows that the *PR author's affiliation* influences review time as well. Pull requests submitted by Shopify developers (owners of the project) or Spreedly developers receive faster reviews on their PRs than PRs submitted for review by external developers. This finding is somewhat similar to the one made by Baysal et al. [14] who studied code review of Mozilla project. During VIF analysis we found that *Writer experience* also have a statistically significant influence on review time(Refer Table 4.6). Negative regression coefficient for this factor demonstrates that highly experienced developers merge their pull requests fast. We assume that the familiarity of the experienced developer on the project and code can be the reason behind this.

Merge Decision. To study the factors affecting the decision to merge or not merge a pull request, we assigned *is_merged_combined* variable the independent variables leaving other variables as dependent ones. We eliminated *merge type* factor from the decision model because it implicitly reflects (i.e., is correlated with) the dependent variable. The results shown in Table 4.5 indicate that *file size*, *commits count*, *commenting people*, *author's company* are factors affecting the decision to merge a pull request or not. Similar to the previous model for estimating the factors affecting review time, the *PR size* variable is involved in the final model for merge decision. Its negative

Coefficients	Estimate	Std. Error	t value	p-value
(Intercept)	29010.6	5324.9	5.448	5.87e-08 ***
custom_log(loc)	4097.7	1189.1	3.446	0.000583 ***
commits_cnt	792.0	431.1	1.837	0.066356 ·
commenting_people	17267.3	1923.7	8.976	< 2e-16 ***
author_comments	-5651.6	1568.3	-3.604	0.000323 ***
company_customShopify	-70126.4	5663.3	-12.383	< 2e-16 ***
company_customSpreedly	-51793.7	5696.1	-9.093	< 2e-16 ***

Table 4.4: MLR Model with "Company" Variable.

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 87150 on 1624 degrees of freedom
Multiple R-squared: 0.1663, Adjusted R-squared: 0.1632
F-statistic: 53.99 on 6 and 1624 DF, p-value: < 2.2e-16

regression coefficient shows that larger PRs are more prone to get a negative effect on merge decision (i.e., a pull request is not merged) than the smaller pull requests.

Out of six discussion related factors only two made it to the final model: the number of developers who left comments at a pull request level and at a source code level. Unlike the model for time, in this model, these factors have the opposite effect on the merge decision. Both of these metrics have positive regression coefficients, indicating that more discussion leads to the delay of pull request decision. As discussion of new contributions is very important in any software project, a detailed discussion should be promoted even if 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 author's responsibility to explain any proposed change and address reviewer comments; our assumption is that the lack of such comments can result in the delay of final decision.

The *PR author experience* has a statistically significant effect on merge decision. If a developer has a long history of submitting pull requests there are more chances for those PRs to get accepted again. Developers actively contributing to the project are quite known to other developers, and thus, the PR acceptance could be affected by their reputation [42] or interpersonal relationship with other developers [25]. While the higher number of develop-

ers commenting on a PR brings down odds of it being approved, the number of developers leaving comments on the source code is probably going to expand the possibility of a PR being accepted and merged. We estimate this to happen in such cases as the need for such a change, alignment with project's goals, etc., at the point when PR comments are probably going to be more "high level", while the comments on the source code, by definition such as implementation, API choice, etc., are more "low level". In the mean time, the discussions regarding high-level issues are prone to be disputable; therefore, bringing more developers to such discussions might prevent them from reaching an agreement. The *affiliation of the PR author* has a statistically significant impact on the merge decision. Both Shopify's and Spreedly's regression coefficients are certain, demonstrating that the PRs that originated from the developers of these two associations have a higher shot of being accepted.

Coefficients	**Estimate**	**Std. Error**	**z value**	**p-value**
(Intercept)	1.412e+00	1.565e-01	9.026	<2e-16 ***
custom_log(loc)	-1.268e-01	3.449e-02	-3.675	0.000238 ***
files_cnt	-6.700e-03	5.088e-03	-1.317	0.187875
commits_cnt	-6.076e-02	1.752e-02	-3.467	0.000526 ***
author_exp	5.249e-03	6.583e-03	0.797	0.425232
author_exp_v2	1.814e-02	1.259e-02	1.441	0.149624
comments_cnt	3.680e-02	4.342e-02	0.847	0.396734
commenting_people	13517.0	1884.2	7.174	1.10e-12 ***
author_comments	-2918.2	1637.3	-1.782	0.07488 ·
incode_comments	274.3	399.1	0.687	0.49203
incode_people	1.998e-01	9.549e-02	2.092	0.036394 *
incode_author_comments	1.213e-02	5.887e-02	0.206	0.836782
company_customShopify	1.005e+00	2.017e-01	4.983	6.27e-07 ***
company_customSpreedly	9.248e-01	2.491e-01	3.712	0.000205 ***
review_time_minutes	-4.713e-06	7.961e-07	-5.920	3.23e-09 ***

Table 4.5: Factors Affecting the Decision to Merge or Not Merge a PR.

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Null deviance: 1949.7 on 1630 degrees of freedom
Residual deviance: 1609.1 on 1616 degrees of freedom
AIC: 1639.1

Table 4.6 shows the regression coefficients for each of the studied factors.

	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

Table 4.6: Models for Fitting Data.

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

RQ2: The statistical models revealed that both PR review time and merge decision are affected by a PR size, the discussion, *as well as* author's experience *and* affiliation.

4.3 RQ3: Developer Perception on Factors Affecting Merge Time and Decision

Sixteen developers participated in our survey. All participants identified themselves as software developers, while two of them have an additional role of a project manager. Two participants have an experience of 1-2 years, seven of 3-6 years, three with 7-10 years of experience, and another three have 10+

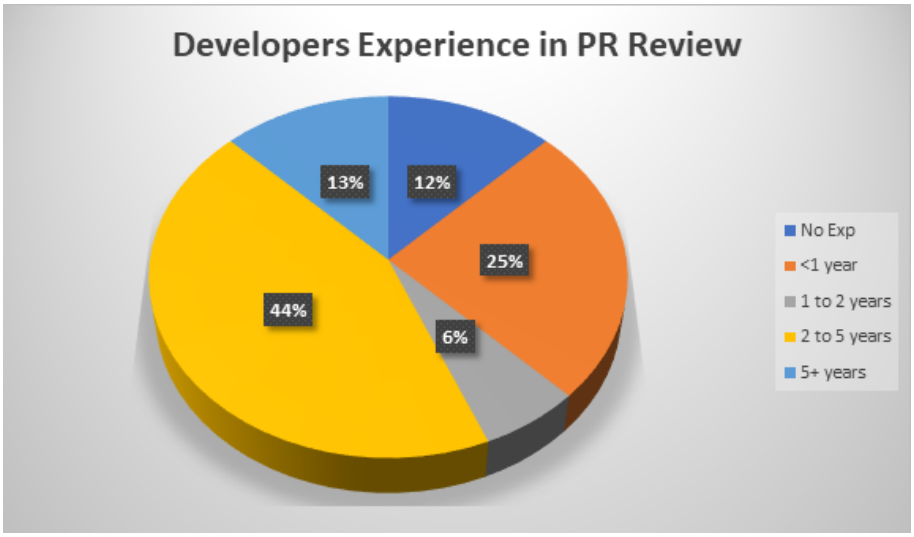


Figure 4.2: Developers Experience in PR Review.

years of software development experience. Participants' experience in the area of pull request reviews differs. Figure 4.2 illustrates the developers' experience in conducting pull request reviews.

While they were queried about average number of pull requests they submit every week, 12.5% answered that they never submitted pull request. 43.74% voted for 3 to 4 pull requests, 25% told they do 5 to 10 submissions, 12.5% claimed 10 to 20 submissions. While 43.74% of developers review 5 to 10 PRs every week, 25% do 10 to 20 reviews, 12.25% of developers review 1 to 2 reviews and 12.25% do no reviews. 6.25% do more than 20 reviews. Figure 4.3 depicts the average PR submission count and PR review count per week.

When they are asked about the type of merge used in PR review, 87.5% claimed that they use GitHub merge most often. Only 6.25% of developers use squash merge most often while the rest 6.25% voted for Git Rebase and Git fast forward merge. For the query related to where they discuss pull request, 81.25% of the developers answered GitHub. Along with GitHub most of them prefer face to face discussions, VOIP/Online chats for discussions. Only 6.25 % prefers only face to face discussions while another 12.5% uses only VOIP/Online chat and face to face discussions.

In addition to the queries regarding demographic information and work prac-

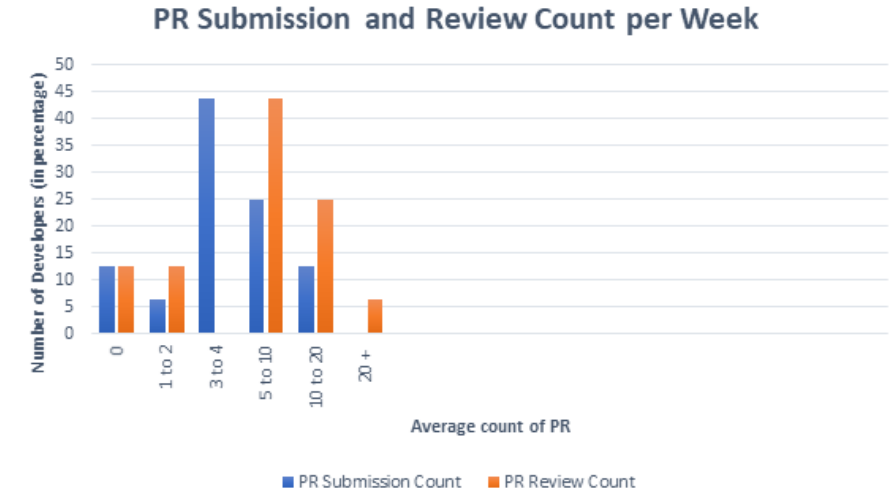


Figure 4.3: PR Submission and Review Count per Week.

tices of participants, we surveyed developers to understand their perspective about the factors that influence the time and decision. 4-point Likert-scale questions were asked to gain developer insights. *PR size* was seen as influential by almost all developers who participated in the survey These findings are similar to the previous research studies [16–18, 25].

This section offers insights related to understanding developer perception of the main characteristics contributing to a PR review. For the majority of questions we asked developers about the factors they think that affect a PR review time and decision, there was no strong prevalence of either positive or negative responses. For the factors affecting the time for code review, the only factor that received the overwhelming positive response is *file size* (94% of positive responses). Majority of the developers disagreed with the statement that the *commits count* of the PR author affects the time for pull request review. The reason behind this can be that they are satisfied with the existing review practices in the organization. Since *reviewer experience* is not easy to calculate accurately, we did not consider reviewer experience as an independent variable in our MLR model. But we asked the developers about their perception on the significance of this factor in our survey. Developers believe that both *author* and *reviewer experience* are important contributors to PR review time. While 81% believed *author experience* matters, 87% of the subjects believed that *reviewer experience* also counts along

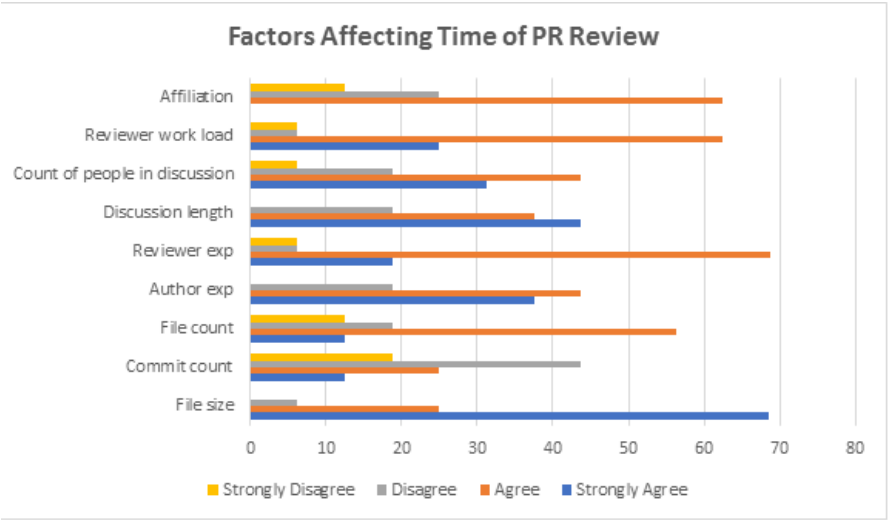


Figure 4.4: Factors Influencing PR Review Time.

with author's experience. Our model indicated that more discussions of pull requests negatively affect the decision. Contrary to the results of the model, 81% of developers responded that the *length of a discussion* is a critical factor in decision making. Like our model shows that the *author's affiliation* matters the time to review a pull request, 63% of developers agreed with the statement of significance of author's affiliation. One reason behind this thought may be because these developers did not participate in a review of external PRs.

In our survey, when we asked developers about their perspective on the factors affecting pull request decision, we did not receive much positive responses. While the influence of file size on the decision is questioned, the answers were split: only 56% of developers positively agreed that the *size* affects the review outcome (refer to Figure 4.5). A possible explanation for this is that a larger PR has a more probability of changes, which is against PR submission policies in many software projects. Even though developers accept a large PR with many changes, they prefer small pull request with less changes as it can be an easy fix. Similar to the response of influence of *commits count* in PR review time, majority of the developers (75%) believe that commits count has less importance in PR review output. 56% of developers concurred that the *commenting_people* (i.e., number of people involved in a PR discussion) and *files count* influence its acceptance, whereas, 63%

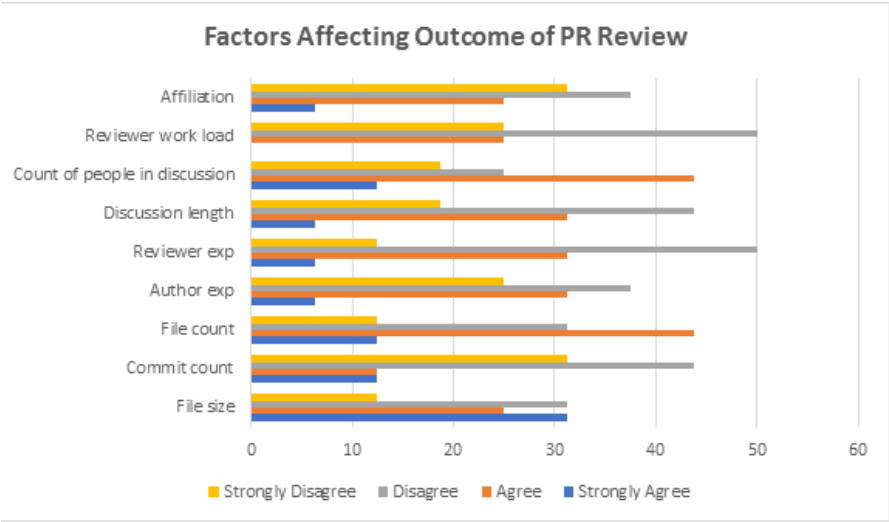


Figure 4.5: Factors Influencing PR Review Outcome.

of negative responses from them showed that they disagreed that the length of that discussion plays a role. Regarding the importance of developers and reviewers experience, only 37% agreed positively, while the rest didn't. Only 25% consider that the work load influences the decision, while 69% of developers believe that the affiliation of the reviewer is not accountable for the PR output.

RQ3: Developer perception of factors affecting time are PR size, author and reviewer experience, count of files and people *involved in discussion,* discussion length, author affiliation *and those affecting decision of PR are* PR size, files count *and* number of people involved in discussion.

Chapter 5

DISCUSSION

In this chapter, we discuss our findings and offer suggestions for future work. In particular, we discuss the merge types used by Shopify developers, and the factors affecting the time and decision to merge a pull request in Shopify comparing with the previous studies.

5.1 Merge Types

From our findings, we could see that the developers use "squash" merge more often than "git" merge or "cherry-pick" merge. Even though squash merging holds the largest median time for pull request merges, our manual analysis shows that developers prefer using it. While our survey results show that developers prefer GitHub merge, we cannot rely on it completely as only a few developers participated in the survey. Squash and merge has a disadvantage of not preserving the historical information which in turn makes it a "not recommended practice". Software engineering researchers prefer the details on how the feature branch developed throughout until it got merged to the master branch than losing the archival data. While 931 records about contributor's affiliation remain "unknown" (refer to Table 3.5), identifying the contributor's organization is important for achieving more accurate results. Further research can be conducted to study the elements considered while decision on pull request merges are made. This can help in

understanding the flaws in the existing practices, thus, can help to improve the git data research quality.

5.2 Merge Time

As a part of data pre-processing, we cleaned data by removing pull requests which have longest duration for pull request merges, therefore eliminating a few records from the study. Gousios et al. [9], in one of their recent publication, studied the time required for pull request merges and the factors affecting it. Our study also analyzed time required for merges and the factors influencing time for pull request merges. While both studies indicated developer experience is significant, most of our study results are different from that of Gousious et al.'s. While Gousios et al. found the project-level metrics such as comments in discussion and project size as influential factors, whereas we did not use them for our study. And the file-level metrics we studied were not important in their study as well. This shows that each project needs to be studied separately and the likeliness of using a generalized model may not be good idea as the domain and elements of each project differ.

5.3 Merge Decision

Gousios et al. [9] and Tsay et al. [10] also studied the factors that influence the decision to merge a pull request. Results of our research coordinate well with the findings of Tsay et al. which showed *pull request size*, *number of people involved in discussion*, *commits count* and *organization of the contributor* are the factors that affect the decision to merge a pull request. Gousios et al.'s research on 88 projects exhibited the influence of count of files changed and some project-level factors, while in our research we didn't study these factors as we concentrated on file-level factors of a single project. So we could not reason our statement about the discrepancies in both of our merge time results. While comparing both our and Tsay et al.'s case studies, the obvious similarity is the usage of same logistic regression statistical model. Gousious et al. used a random forest classifier in their study. the influence of

statistical models selected on the results of each study (this would be a good project to conduct in the future), it is recommended to have a generalized approach to use statistical models by researchers. Another recommended practice is to choose few of the models used by previous researchers and compare the findings. We opted for the simplest models used in the past by various researchers.

5.4 Implications

Developers: Knowledge about the factors affecting the time and decision to merge a pull request can help the developers to identify the ways to accelerate the code review process. Our studies shows PR review time and outcome is influenced by factors such as PR size, discussion, author's experience and affiliation. Based on the results of our study, we would like to offer a list of recommendations for developers.

- Adopt to a practice of writing and submitting smaller pull requests so that the pull requests will be accepted faster.
- Detailed comments also speaks well with reviewer and it can increase the probability of PR acceptance.
- Another recommended practice for developers is to build their reputation by submitting more patches and gaining more experience and thus improve your chances in reviewers accepting your pull requests.
- Encourage other developers to participate on pull request discussions
- Practicing patch submissions through organization's official email account rather a dummy account is always a highly recommended practice as it helps reviewers, as well as researchers to identify the affiliation of the contributor and thus avoid identity resolution problems and ambiguity. Contributions from unidentified developers take more time to be merged and delay reviewers' decisions.

Researchers: In our study we combined both quantitative and qualitative methods for investigating how pull requests get reviewed and merged. Our recommendations for researchers are as follows.

- Detailed study can be done on the analysis of the pull request review process and merge strategies.
- Research on practices in other organizations can help in studying more about pull-based development.
- Personal interviews can also be conducted with the developers in the organization. This can help in extending the qualitative study of understanding developer perception on pull request reviews and merges.
- Developing some technique to measure the trustworthiness of survey responses can also help in validating the findings of qualitative studies.

5.5 Future Work

Our study was on a Shopify's public repository data and the fidelity of the this data can very much contradictory from real-time private repository dataset, it is highly recommended to perform more case studies so that researchers can find a generalized approach about pull-based development. Getting access to the developer's identity and their affiliation can help in identifying more merge types (if any) practised by developers in respective organizations. Survey can be send extensively to all developers to get their feedback as majority of them are still "unknown". Further studies can also be done to get developers concerns during pull request integration decision making.Finding an automated method for determining merge strategies can help in saving time and resources used for the manual analysis of pull request merges. Future work can also include studying the influence of the quality of code in PR time and outcome. Project level metrics considered by previous researchers can also be studied on our dataset which may help in identifying common factors that affect time and outcome of PR merges. Developing better metrics for measuring developer expertise on a project/code and PR complexity (e.g., functional complexity and dependency) are also possible future steps that could improve our work.

5.6 Contributions

This thesis makes the following contributions:

1. An in-depth study of pull request merges in a pull-based software development model within a successful commercial software project.

2. Our study provides a publicly shared dataset that includes mined data with manually verified and labeled merged PRs, the survey questions used, and the anonymous survey responses.

3. A development of statistical models such as multiple linear regression and a logistic regression to study the affects of various factors on PR merge time and outcome. The models used by Tsay et al. and our findings turned out to be similar in this study. This can help in developing assumptions about some of the ground metrics that affect decision of pull based development models.

4. In our study we have used file level metrics which were not the subject of previous studies.

5. A survey with professional Shopify developers offers insights into their perception of PR merges and strategies used. The survey data is then compared with the quantitative findings extracted by mining the development repository.

6. Our findings can help Shopify developers to better understand the factors affecting time and decision of PR merges and thus improve the contribution and review practices within the organization.

Chapter 6

THREATS TO VALIDITY

In our study we analyzed the extend to which the results are accurate. Two aspects of validity are reviewed – internal and external validity. We came across both internal and external validity threats in our study which can alter the results of our study. While internal validity threats are specific to our study and can be controlled, the latter one relates to the hindrances that pulls back our study from being generalized.

6.1 Internal Validity

Threats to *internal validity* involve the quality of our study design and thoroughness of its execution. These threats in our study are linked with data mining, building of model, survey design and analysis. We extracted data from Shopify's public repository – Active Merchant, which can be far from the real time data. By analyzing the data we could see that most of the pull requests are in the merged category. Thus, there is great chance for the linear model to be biased towards the merged category. Consequently we could not guarantee the accuracy of the output as a result of data imbalance. We can consider overcoming this challenge by studying on a dataset from a private repository which is having more real time values.

We extracted the data using GitHub APIs to ensure that we got access to the most up-to-date dataset. We did manual inspection of merged pull

requests to reduce the count of inaccurate records. We cleaned the anomalies from the dataset such as removal of the pull request with largest files and longest review time, and "closed" status ones. There are chances of internal validity threats in the models selected as we have trusted only on the metrics used by researchers in their previous studies. Since GitHub allows users to create dummy accounts, majority of the contributors' affiliation couldn't be identified. The identification of "unknown" users can help us in finding more merge types, if any, used by Shopify developers. While designing our survey, we adapted simple and easy to answer questions.

6.2 External Validity

External validity threats deal with the generalizing of the findings in our study. We focused on a single software project in Shopify and the developers associated with it. Even if Active Merchant is a successful commercial project and the survey respondents are highly experienced full-time employees, it is not possible to generalize the findings across all projects hosted on GitHub. Especially we consider the possibility of different findings for Shopify's private repository data itself. At the same time, there is possibility for similar features for any medium-sized GitHub projects with similar setup. For example pull-based commercial projects in public repositories can depict similar characteristics in its development. So it is highly recommended to do further research so that pull-based software development model can be understood well and the possibility of developing a unified model can be investigated. We created an anonymous dataset, classified pull requests based on merge types, a survey design, and the survey responses publicly available so that our study can be replicated.

Chapter 7

CONCLUSION

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.The raw data set we extracted from Active Merchant public repository and used SQL query to filter the dataset needed for study. This dataset is loaded into R for analysis. We have mainly used manual and quantitative analysis on the dataset found. Manual analysis was done to identify PR merge types and found that squash merging is scored high among other merge types. Statistical models are developed to study the influence of different 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 his/her affiliation were important factors that influence both merge time and merge decision models. Another study is done by conducting a survey on Shopify developers. This survey helped in understanding the perception of Shopify developers about the factors that affect time and decision to merge and the quality of pull request process. The analysis of the survey responses showed that the developers consider the PR size, author and reviewer experience, number of files, count of people involved in discussion, discussion length, and author's affiliation affects time while PR size, files count and number of people involved in discussion affect merge decision.

Bibliography

[1] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[2] A. Mysore, "Github alternative: Top 7 best source code hosting sites. beebom," Mar 2015. [Online]. Available: https://beebom.com/github-alternatives/

[3] GitHub, "GitHub API v3," https://developer.github.com/v3/.

[4] "Github." [Online]. Available: https://en.wikipedia.org/wiki/GitHub

[5] N. Paolucci, "Atlassian developers- pull request merge strategies: The great debate," Dec 2014. [Online]. Available: https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/

[6] M. Larkin, *Shopify Application Development*. Packt Publishing Ltd, 2014.

[7] Shopify, "Shopify Announces Fourth-Quarter and Full Year 2016 Financial Results," https://investors.shopify.com/Investor-News-Details/2017/Shopify-Announces-Fourth-Quarter-and-Full-Year-2016-Financial-Results/default.aspx.

[8] N. Plante and D. Berube, *Practical Rails Plugins*. Apress, 2008.

[9] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th*

International Conference on Software Engineering. ACM, 2014, pp. 345–355.

[10] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 356–366.

[11] 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.

[12] C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in oss projects," in *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*. IEEE, 2007, pp. 26–26.

[13] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," Technical Report DCS-305-IR, University of Victoria, Tech. Rep., 2006.

[14] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The secret life of patches: A firefox case study," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 447–455.

[15] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 541–550.

[16] P. Weissgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proc. of the 2008 Int. Working Conf. on Mining Soft. Repos.*, 2008, pp. 67–76.

[17] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "Investigating technical and non-technical factors influencing modern code review," *Empirical Software Engineering*, vol. 21, no. 3, pp. 932–959, 2016.

[18] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: Case study on the linux kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 101–110.

[19] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 534–550, Jul. 2009.

[20] L. Hatton, "Testing the value of checklists in code inspections." *IEEE Software*, vol. 25, no. 4, pp. 82–88, 2008.

[21] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 2146–2189, Oct. 2016.

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

[23] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 2013, pp. 712–721.

[24] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 202–212.

[25] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: how developers see it," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 1028–1038.

[26] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 358–368.

[27] J. Marlow, L. Dabbish, and J. Herbsleb, "Impression formation in online peer production: Activity traces and personal profiles in github," in *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, 2013, pp. 117–128.

[28] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 285–296.

[29] M. M. Rahman and C. K. Roy, "An insight into the pull requests of github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 364–367.

[30] GitHub, "GitHub API Client Library," https://github.com/octokit/octokit.net.

[31] M. Zabarankin and S. Uryasev, "Statistical decision problems," *AMC*, vol. 10, p. 12, 2014.

[32] S. Ray, G. Blog, and K. Jain, "7 types of regression techniques you should know." [Online]. Available: https://www.analyticsvidhya.com/blog/2015/08/comprehensive-guide-regression/

[33] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 864–878, 2009.

[34] A. Mockus, "Organizational volatility and its effects on software defects," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 117–126.

[35] 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 *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 192–201.

[36] S. Garavaglia and A. Sharma, "A smart guide to dummy variables: Four applications and a macro," in *Proceedings of the Northeast SAS Users Group Conference*, 1998, p. 43.

[37] J. Fox and S. Weisberg, *An R Companion to Applied Regression*, 2nd ed. Thousand Oaks CA: Sage, 2011. [Online]. Available: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

[38] J. Fox, *Applied Regression Analysis and Generalized Linear Models*. SAGE Publications, 2008.

[39] R. Groves, F. Fowler, M. Couper, J. Lepkowski, E. Singer, and R. Tourangeau, *Survey Methodology*, 2nd ed. Wiley, 2009.

[40] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. pp. 583–621, 1952.

[41] E. Lehmann and H. D'Abrera, *Nonparametrics: statistical methods based on ranks*. Springer, 2006.

[42] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? immigration in open source projects," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 6.

[43] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 541–550.

[44] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German, "Management of community contributions," *Empirical Software Engineering*, vol. 20, no. 1, pp. 252–289, 2015.

[45] J. Asundi and R. Jayant, "Patch review processes in open source software development communities: A comparative case study," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 2007, pp. 166c–166c.

[46] "Shopify." [Online]. Available: https://en.wikipedia.org/wiki/Shopify

[47] "Github guides." [Online]. Available: http://guides.github.com/

[48] D. Doomen, "Git like a pro (how to use it as it was meant to)," Jan 2016. [Online]. Available: http://www.slideshare.net/dennisdoomen/git-like-a-pro-how-to-use-it-as-it-was-meant-to

Appendix A

A.1 Survey Form

A Study Investigating Pull Request Reviews at Shopify

* Required

1. **1. How would you describe your role on the project(s)?** *
   Check all that apply.
   Check all that apply.

   - [ ] Software Developer/Engineer
   - [ ] Project Manager/Lead
   - [ ] QA/Test Engineer
   - [ ] Other: _____

2. **2. How many years of experience do you have in software development?** *
   Mark only one oval.

   - ◯ < 1 year
   - ◯ 1-2 years
   - ◯ 3-6 years
   - ◯ 7-10 years
   - ◯ 10+ years

3. **3. On average, how many pull requests do you submit every week?** *
   Mark only one oval.

   - ◯ 1 to 2 pull requests
   - ◯ 3 to 4 pull requests
   - ◯ 5 to 10 pull requests
   - ◯ 10 to 20 pull requests
   - ◯ 20+ pull requests
   - ◯ I do not submit

4. **4. On average, how many pull requests do you review every week?** *
   Mark only one oval.

   - ◯ 1 to 2 pull requests
   - ◯ 3 to 4 pull requests
   - ◯ 5 to 10 pull requests
   - ◯ 10 to 20 pull requests
   - ◯ 20+ pull requests
   - ◯ I do not review

5. **5. How long have you been involved in pull request reviews?** *

Mark only one oval.

- ◯ Less than 6 months
- ◯ 6 to 12 months
- ◯ 1 to 2 years
- ◯ 2 to 5 years
- ◯ 5+ years
- ◯ I do not review

6. **6. As a developer, what types of merges do you do?** *

Check all that apply.

- ☐ GitHub merge (using GitHub UI)
- ☐ Cherry-pick merge (e.g., git cherry-pick)
- ☐ Squash merge
- ☐ Other: _____

7. **7. What type of merge do you use most often?** *

Mark only one oval.

- ◯ GitHub merge
- ◯ Cherry-pick merge
- ◯ Squash merge
- ◯ Other: _____

8. **8. Where do you discuss pull requests?** *

Check all that apply.
Check all that apply.

- ☐ GitHub
- ☐ Email
- ☐ VoIP/online chat (e.g., Skype, Hangout)
- ☐ Face to face discussions
- ☐ Other: _____

9. **9. In your opinion, which of the following factors affect TIME of pull request review?** *

Mark only one oval per row.

	Strongly Disagree	Disagree	Agree	Strongly Agree
Pull request size (LOC)				
Pull request size (# of commits)				
File count				
PR author experience				
PR reviewer experience				
Length of the discussion				
Number of people involved in the discussion				
Reviewer work load				
Affiliation of a PR author (e.g., Shopify dev, Spreedly dev, etc.)				

10. **10. Which of the following factors do you think affect pull request review DECISIONS (i.e., whether you merge/not merge the pull request)** *

Mark only one oval per row.

	Strongly Disagree	Disagree	Agree	Strongly Agree
Pull request size (LOC)				
Pull request size (# of commits)				
File count				
PR author experience				
PR reviewer experience				
Length of the discussion				
Number of people involved in the discussion				
Reviewer work load				
Affiliation of a PR author (e.g., Shopify dev, Spreedly dev, etc.)				