UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Marcela Bandeira Cunha

An Analysis of Git's Private Life and Its Merge Conflicts

Recife
2023

Marcela Bandeira Cunha

An Analysis of Git's Private Life and Its Merge Conflicts

A M.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Software Engineering

Advisor: Paulo Henrique Monteiro Borba

Co-advisor: Paola Rodrigues De Godoy Accioly

Recife
2023

Catalogação na fonte
Bibliotecária Mônica Uchôa, CRB4-1010

C972a Cunha, Marcela Bandeira
An analysis of git's private life and its merge conflicts / Marcela Bandeira Cunha. – 2023.
74 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2023.
Inclui referências e apêndices.

1. Collaborative software development. 2. Merge conflicts. 3. Empirical software engineering. 4. Repository mining. I. Borba, Paulo Henrique Monteiro (Orientador). II. Título.

004 CDD (23. ed.) UFPE - CCEN 2023 – 202

Marcela Bandeira Cunha

"An Analysis of Git's Private Life and Its Merge Conflicts"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Progamação.

Aprovado em: 31 de agosto de 2023.

Dedico a todos que acreditam no poder da ciência, da educação e da cultura.

ACKNOWLEDGEMENTS

Agradeço a todos que contribuíram de alguma forma com o desenvolvimento desta dissertação.

Primeiramente aos meus pais, Paulo e Mônica, por terem me guiado no caminho da educação.

Agradeço aos meus orientadores, Paulo e Paola, por terem me orientado com maestria, por todo o tempo dedicado ao nosso trabalho e por serem um apoio nesse complexo processo. Sou imensamente grata pela paciência e incentivo de vocês.

Agradeço também aos membros da banca examinadora, André e Catarina, pelo interesse e disponibilidade.

Agradeço a Humberto pelo apoio incondicional durante esta jornada.

Por fim, agradeço aos amigos e familiares pela compreensão e forças depositadas em mim quando eu precisava.

ABSTRACT

Collaborative development is an essential practice for the success of most nontrivial software projects. However, merge conflicts might occur when a developer integrates, through a remote shared repository, their changes with the changes from other developers. Such conflicts may impair developers' productivity and introduce unexpected defects. Previous empirical studies have analyzed such conflict characteristics and proposed different approaches to avoid or resolve them. However, these studies are limited to the analysis of code shared in public repositories. This way they ignore local (developer private) repository actions and, consequently, code integration scenarios that are often omitted from the history of remote shared repositories due to the use of commands such as `git rebase`, which rewrite Git commit history. These studies might then be examining only part of the actual code integration scenarios and conflicts. To assess that, we aim to shed light on this issue by bringing evidence from an empirical study that analyzes Git command history data extracted from the local repositories of a number of developers. This way we can access hidden integration scenarios that cannot be accessed by analyzing public repository data as in GitHub based studies. After identifying the visible and hidden integration scenarios, we investigate the relationship between the frequency of developers integrating code and how frequently these scenarios result in conflicts. This way, we can understand if these data are correlated. We analyze 95 Git reflog files from 61 different developers. Our results indicate that hidden code integration scenarios are more frequent than visible ones. We also find higher conflict rates than in previous studies. Our evidence suggests that studies that consider only remote shared repositories might miss integration conflict data by not considering the developer's local repository actions. Regarding the correlation study, our results indicate a statistically significant relationship between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts. This relationship is represented by a negative correlation (the higher values of one event are associated with the lower values of the other). From our study sample result, we suggest that if a developer integrates code often, the failed code integration frequency will tend to decrease.

Keywords: collaborative software development; merge conflicts; empirical software engineering; repository mining.

RESUMO

O desenvolvimento colaborativo é uma prática essencial para o sucesso da maioria dos projetos de software não triviais. No entanto, conflitos de mesclagem podem ocorrer quando um desenvolvedor integra, por meio de um repositório compartilhado remoto, suas alterações com as alterações de outros desenvolvedores. Tais conflitos podem prejudicar a produtividade dos desenvolvedores e introduzir defeitos inesperados. Estudos empíricos anteriores analisaram tais características de conflito e propuseram diferentes abordagens para evitá-los ou resolvê-los. No entanto, esses estudos se limitam à análise de códigos compartilhados em repositórios públicos. Dessa forma, eles ignoram as ações do repositório local (privado do desenvolvedor) e, consequentemente, os cenários de integração de código que muitas vezes são omitidos do histórico de repositórios remotos compartilhados devido ao uso de comandos como `git rebase`, que reescrevem o histórico de commits do Git. Esses estudos podem então estar examinando apenas parte dos cenários e conflitos reais de integração de código. Para avaliar isso, pretendemos lançar luz sobre essa questão, trazendo evidências de um estudo empírico que analisa dados do histórico de comandos Git extraídos dos repositórios locais de vários desenvolvedores. Dessa forma, podemos acessar cenários de integração ocultos que não podem ser acessados analisando dados de repositório público como em estudos baseados no GitHub. Após identificar os cenários de integração visíveis e ocultos, investigamos a relação entre a frequência dos desenvolvedores que integram o código e a frequência com que esses cenários resultam em conflitos. Dessa forma, podemos entender se esses dados estão correlacionados. Analisamos 95 arquivos Git reflog de 61 desenvolvedores diferentes. Nossos resultados indicam que os cenários de integração de código oculto são mais frequentes do que os visíveis. Também encontramos taxas de conflito mais altas do que em estudos anteriores. Nossas evidências sugerem que estudos que consideram apenas repositórios compartilhados remotos podem perder dados de conflito de integração por não considerar as ações do repositório local do desenvolvedor. Em relação ao estudo de correlação, nossos resultados indicam uma relação estatisticamente significativa entre a frequência de integração de código dos desenvolvedores e a frequência de cenários de integração que resultam em conflitos. Essa relação é representada por uma correlação negativa (os valores mais altos de um evento estão associados aos valores mais baixos do outro). A partir do resultado da amostra do nosso estudo, sugerimos que, se um desenvolvedor integra o código com frequência, a frequência de falha na integração do código tende a diminuir.

Palavras-chaves: desenvolvimento de software colaborativo; integração de código e seus conflitos; engenharia de software empírica; mineração de repositório.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In a software development environment, team members often work collaboratively through a shared remote repository. It is common for developers to work on tasks independently of each other. Each one uses their own local copies of a remote project repository. When a developer concludes a task, it is time to integrate the associated contributions and conflicts might then occur if developers changed overlapping areas in a common file. These are called merge conflicts (PERRY; SIY; VOTTA, 1998; ZIMMERMANN, 2007; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013; BRUN et al., 2013; MAHMOOD et al., 2020), as opposed to other kind of conflicts that might be detected during building (SILVA; BORBA; PIRES, 2022), testing (SILVA et al., 2020), or even at system production time.

Although many merge conflicts are easy to fix, some may demand significant effort and system knowledge before they can be resolved. Besides that, there is a risk of a developer incorrectly resolving a conflict. When this happens, the consequence is the introduction of unexpected defects in the system (BIRD; ZIMMERMANN, 2012; MCKEE et al., 2017). In fact, such conflicts may impair developers' productivity and compromise system quality (BIRD; ZIMMERMANN, 2012; SARMA; REDMILES; HOEK, 2012; MCKEE et al., 2017). Because of these negative consequences, and as merge conflicts might often occur (PERRY; SIY; VOTTA, 1998; MENS, 2002; ZIMMERMANN, 2007; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013; BRUN et al., 2013; MAHMOOD et al., 2020), a number of studies focus on understanding conflict characteristics, examining different mechanisms for proactive conflict detection (BRUN et al., 2011; HOEK; SARMA, 2008; aES; SILVA, 2012), and proposing tools to more effectively resolve conflicts (CAVALCANTI; BORBA; ACCIOLY, 2017; NISHIMURA; MARUYAMA, 2016; MENS, 2002; CLEMENTINO; BORBA; CAVALCANTI, 2021; TAVARES et al., 2019). This topic has been deeply studied in the literature, with the aim of seeking proper technical and organizational support to avoid the negative impact of conflicts on development productivity and software quality. However, evidence in the literature is limited to detecting and analyzing merge conflicts (ZIMMERMANN, 2007; APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY et al., 2018; CAVALCANTI et al., 2019; NGUYEN; IGNAT, 2018; ACCIOLY; BORBA; CAVALCANTI, 2018a; GHIOTTO et al., 2020; DIAS; BORBA; BARRETO, 2020) in shared remote repositories such as the ones available in GitHub, many of them considered to be the main project repositories.

By focusing only on merge scenarios visible in *shared remote* repositories, these studies ignore *local (private)* repository actions and, consequently, integration scenarios that are often omitted from the history of remote shared repositories due to the use of commands such as `git rebase` and `squash`, which rewrite Git commit history. This way integration scenarios and conflicts that locally occur and are resolved in the developers' private environment become hidden, that is, are not visible in remote repositories and, therefore, are not analyzed by these studies. As a consequence, these studies do not assess the influence that local actions might have on the occurrence of merge conflicts in remote repositories. In addition, these studies also miss integration scenarios and conflicts originated from the use of Git commands (rebase, cherry-pick, etc.) that perform code integration but leave no trace in remote shared repository histories. In the case of rebase, for example, it integrates changes from two branches but rewrites the history so that it's linear. As it does not explicitly signal that integration has occurred through a special (merge) commit, such integrations are not considered in previous studies. The failure to detect these integration scenarios might negatively impact research results, as, for instance, integration and conflict frequency could be higher than what is actually observed by considering only data from shared remote repositories.

Besides that, as merge conflicts often occur and are deeply studied in the literature, understanding how to decrease the merge conflict rate is a common desire between the software industry, software practitioners, and researchers. Moreover, there is evidence regarding the frequency of integrating code as a factor that could cause merge conflicts and also as a way to reduce such conflicts (BIRD; ZIMMERMANN, 2012; ADAMS; MCINTOSH, 2016; MCKEE et al., 2017; DIAS; BORBA; BARRETO, 2020).

As we identified the limitation of previous studies by analyzing code integration from public remote repositories and the desire to decrease the merge conflict rate, we dedicate this research to studying the private repositories of the developers. In other words, we aim to shed light on these issues by bringing evidence from an empirical study that analyzes Git commands history data extracted from the local (private) repositories of a number of developers. In this dissertation, we perform two main studies: an analysis of integration scenarios beside the merge and an understanding of code integration frequency and its association with merge conflicts in private repositories.

In particular, this research contributes with a new study that differs from what has been presented in the literature in two main ways: (a) examining developer local repository logs instead of remote shared repositories, and (b) going beyond `git merge` and analyzing additional git code integration commands and the conflicts they generate. The core idea is to evaluate developer's actions in local private repositories to understand their daily work integration practices, and how these differ from the `git merge` only actions visible in shared remote repositories. This way, we analyze here code integration scenarios that previous merge conflict studies were not able to analyze. We are then able to reveal here the *private* life of merge conflicts[1], and contrast it with the more widely known *public* life of merge conflicts, which has often been exposed by GitHub based studies that appear in the literature (CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY et al., 2018; CAVALCANTI et al., 2019; NGUYEN; IGNAT, 2018; ACCIOLY; BORBA; CAVALCANTI, 2018a; GHIOTTO et

[1] Hereafter we use the term *merge conflict* to denote any conflict reported during code integration time, no matter if using `git merge` or other git code integration commands.

al., 2020).

In the first study, we present evidence of visible and hidden integration scenarios in local repositories and their merge conflicts rate. We obtained our results by applying quantitative and qualitative techniques to answer our research questions regarding the integration scenarios. First, we use tools that we developed, we collect and analyze the local git history reference logs, or reflogs[2] of 95 private repositories owned by 61 developers. Our tools are able to identify the different kinds of integration scenarios mentioned before, and calculate their frequency and other derived measurements. Second, to understand which factors could influence the choice of git code integration commands and approaches, we interviewed 9 (of the 61) developers. Finally, the author acted as participant observant (SEDANO; RALPH; PéRAIRE, 2017) in a professional git based development context where this author works. Besides that, the author could observe the developer's routine related to the Git commands and the company policy towards the `git merge`. These related findings are described in Chapter 3.

The second study, in Chapter 4, we focus on evaluating the code integration frequency and how it is associated with the failed integration frequency. Thus, we dedicate the second part of this research to examining whether a relationship exists between developers' code integration frequency and how frequently these scenarios result in conflicts.

To assess that, we conduct a quantitative study to answer our research question. First, we reuse the quantitative study from Chapter 3 to mine the integration scenarios from local repositories to evaluate the frequencies of integration scenarios from Git commands. Then, we apply statistical tests to measure the correlation between these two frequencies.

Since we have data on visible and hidden integration, we analyze the correlation by considering the type of integration scenario. This way, we can examine if the result would differ between the strategies of integrating code. So, we consider 3 groups: all the developers' participants of our sample (which contains visible and hidden integration scenarios), the developers with high visible integration frequency, and the ones with high hidden integration frequency. For each group, we run statistical tests to verify the correlation.

The remainder of this study is organized as follows:

- Chapter 2 motivates our research and reviews the main concepts used in this study;
- Chapter 3 is about the empirical study that analyzes git command history data extracted from the local repositories of the developers.
  - It presents the research questions and describes our study setup. It presents our results and answers to our research questions. We also present the discussion and the threats to the validity of this study;

[2] <https://git-scm.com/docs/git-reflog>

  - We have an accepted paper referring to the study carried out in this chapter. Our paper was accepted on SBES 2022: Proceedings of the XXXVI Brazilian Symposium on Software Engineering. The paper link is available in Appendix A

- Chapter 4 is dedicated to examining whether a relationship exists between developers' code integration frequency and how frequently these scenarios result in conflicts;

  - It presents the research question and describes our study setup. It presents our results and answers to our research question. We also present the discussion and the threats to the validity of this study;

- Chapter 5 discuss related works;

- Chapter 6 details our conclusions and future work.

2 MOTIVATION AND BACKGROUND

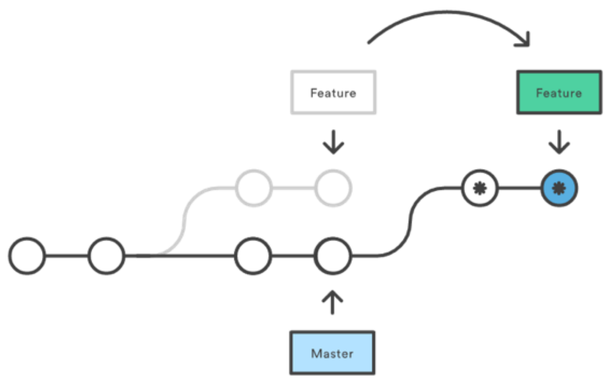
Each code change made by a developer has to be integrated into a project remote (possibly main) repository to be visible to the rest of the contributors. However, not all integration attempts are successful. Conflicts can arise when merging the code changes. Studies have reported that from 10% to 20% of all integration scenarios fail (BRUN et al., 2011; KASI; SARMA, 2013), with some projects achieving rates of almost 50% (BRUN et al., 2011; ZIMMERMANN, 2007), but this might vary depending on project practices and other factors (APEL; LEßENICH; LENGAUER, 2012; ACCIOLY; BORBA; CAVALCANTI, 2018a; DIAS; BORBA; BARRETO, 2020), with some projects observing no conflicts resulting from invoking commands like `git merge` (ACCIOLY; BORBA; CAVALCANTI, 2018a).

All of these empirical studies collect evidence by analyzing public Github repositories (CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018b; NGUYEN; IGNAT, 2018; ACCIOLY; BORBA; CAVALCANTI, 2018a; CAVALCANTI; ACCIOLY; BORBA, 2015; GHIOTTO et al., 2020; AHMED et al., 2017; BRUN et al., 2011; KASI; SARMA, 2013). Despite bringing significant evidence for software engineering researchers and practitioners, these studies suffer from a common threat to validity: all the integration scenarios they consider come from shared remote project repositories. These repositories only track integration scenarios created explicitly by the `git merge` command. However, there are other ways to integrate code using Git (CHACON; STRAUB, 2014), and they do not leave traces in the history of remote repositories. Besides that, other Git commands might rewrite repository history, which could then even erase `git merge` integration scenarios from the history of remote repositories. So remote repositories reflect only part of the code integration scenarios and merge conflicts that developers had to face during a project.

In fact, unlike the merge command, the following commands do not leave, or can erase, traces of code integration in the history of remote project repositories: (a) rebase, (b) cherry-pick, (c) squash, and (d) stash apply. The (a) rebase command is used to integrate changes from a branch by reapplying them, commit by commit, on top of the other branch's latest commit, which then represents the new base for the integrated changes. Different from the merge command, which creates an additional (merge) commit with the integrated code, rebase creates new "clone" commits for each commit in the original rebased branch. This way, the rebase command changes repository history keeping it linear and, therefore, easier to analyze, as we can see in Figure 1. The drawback of using it is not being able to identify, in the illustrated case, that Feature was actually independently developed and only later integrated to Master, as the light grey parts of the figure are not visible by inspecting the repository.

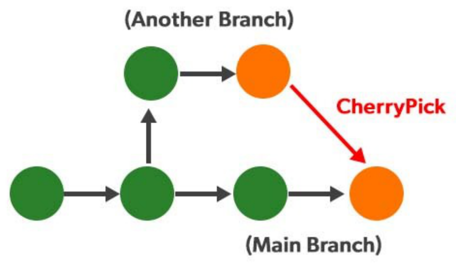
The (b) cherry-pick command is used to reapply selected changes (commits) from one branch into another branch. This is similar to rebase, but in a more constrained way, dealing with commits instead of branches. The result is also a linear history, leaving no

Figure 1 – Rebase scenario, via Atlassian.



Source: <https://tinyurl.com/rebasegitcommand>

Figure 2 – Cherry-Pick scenario, via GeeksforGeeks.



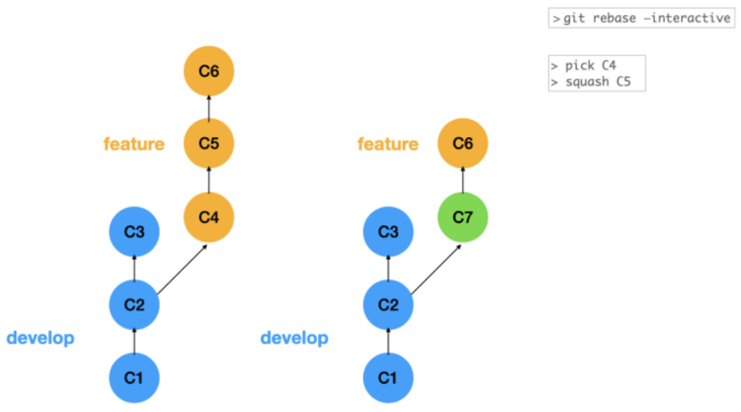
Source: <https://bit.ly/cherrypickgitcommand>

trace that the reapplied commits are, in fact, copies of commits independently developed in another branch and only later integrated into the target branch, as we can see in Figure 2.

The third command that can hide integration scenarios is (c) squash, which can, for instance, turn a sequence of commits into a single commit, joining their changes, but leaving no trace of the commits in the sequence. As shown in Figure 3, the squash command was executed on commits C5 and C4, creating a new commit C7. The specific scenario in which the squash can hide code integration is when a merge commit is among the squashed commits. The result is the loss of the merge commit record, that is, the code integration trace that was left in the original history.

Finally, the Git stash feature allows to save their changes in a stack temporarily (pull changes from another branch) and later reapply the stacked changes with the command (d) stash apply, effectively integrating code that was independently developed, as we can see in Figure 4.

Figure 3 – Squash scenario, via DEV Community.

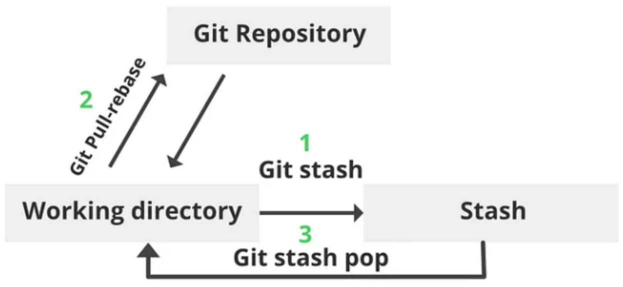


```
> git rebase -interactive
```

```
> pick C4
> squash C5
```

Source: <http://bit.ly/squashgitcommand>

Figure 4 – Stash apply scenario, via Medium.



Source: <https://bit.ly/stashapplygitcommand>

All of these commands represent different code integration situations that could happen without relying on the merge command; none of them is visible in remote repositories. Consequently, by failing to detect such cases, previous studies might be evaluating only a fraction of the code integration scenarios and the resulting merge conflicts. Besides that, since previous empirical studies only collect data from remote repositories, they might even miss conflicts that were detected and solved in a developer's machine, and never reached a remote repository.

For this reason, going beyond remote repositories, studies should also analyze the local repositories of the developers involved in the project. Such data would enable investigating which integration scenarios developers use more frequently. This way, we could understand the practices carried out by the development teams and their motivations for the approaches chosen to solve the integration problems. Finally, there would be a possibility to study the understanding of the factors that may or may not influence current conflicts and create new solutions to avoid them.

To better assess that, we dedicate this dissertation to investigating Git's private life and its merge conflicts. We commit to analyzing the local repositories of developers. This way, we can identify the integration scenario created explicitly by the merge command and some integration scenarios that are not traceable in the remote repositories.

As discussed before, merge conflicts often occur, as previous studies have reported considerable rates of failures. Moreover, this type of conflict impacts developers' productivity because it is a demanding and error-prone task (BIRD; ZIMMERMANN, 2012; BRUN et al., 2013; KASI; SARMA, 2013; SARMA; REDMILES; HOEK, 2012; ZIMMERMANN, 2007). Because of that, understanding how to decrease the merge conflicts rate is a common desire among the software industry, software practitioners, and researchers.

Previous studies have shown the frequency of integrating code as a factor that could cause merge conflicts and also as a way to reduce their occurrence. Bird and Zimmerman (BIRD; ZIMMERMANN, 2012) interview developers that suggest that problems are caused by long delays in integrating contributions and moving them between teams. Adams and McIntosh (ADAMS; MCINTOSH, 2016) indicate that the best way to reduce conflicts is to keep branches short-lived and merge often. McKee et al. (MCKEE et al., 2017) informally claim that practitioners should attempt to commit often to prevent and alleviate the severity of merge conflicts. In addition, Dias, Borba, and Barreto (DIAS; BORBA; BARRETO, 2020) showed the higher the contribution duration (geometric mean of the number of days between the common ancestor and the last commit in each contribution), the higher the chances of merge conflict occurrence.

Those studies discovered an association between code integration frequency and its effect on merge conflicts. Moreover, their findings indicate that a way to reduce merge conflicts is to integrate often. Hence, in this research, we also seek to understand if this association is valid when analyzing the private repositories of developers.

The first study of this dissertation is responsible for bringing evidence about visible and hidden integration frequencies and their failure in developers' local repositories. This chapter is also responsible for indicating possible factors that influence the adoption of a code integration strategy. With this analysis, we can contrast our results with previous studies based on remote repositories and complement studies by showing evidence of visible and hidden integration scenarios in local repositories.

When analyzing the local repositories, we can examine the developer's actions and how the integration practices differ from those visible in shared remote repositories. By doing so, we decided to investigate the merge conflicts since we identified integration scenarios that previous merge conflict studies were not able to analyze.

The second study of this dissertation is dedicated to evaluating code integration frequency and its association with code integration conflicts. In other words, we examine if there is a correlation between the frequency of developers' code integration and how frequently these scenarios result in conflicts.

To better evaluate this relationship, we propose to verify the correlation considering the type of integration scenario. We consider only the developers that apply the merge command in the majority of its integration actions. Then, we do the same division with the developers and hidden integration scenarios. And finally, we consider both scenarios together (all developers).

Our expectation about running this analysis into these three groups is to examine if the result would differ between them. When including visible and hidden integration scenarios, we verify if the correlation result would converge to previous studies' findings: more frequent integrations cause fewer merge conflicts. When grouping developers with the same strategy of integrating code, we verify if the relationship would be statistically significant and if its degree would change between these groups. This way, we could analyze if a specific code integration strategy would strengthen this relationship and how one event would imply a change in the other.

In summary, the core idea we propose in this research is to show the importance of data that can be collected by including local repositories from developers in future studies. By having the local repositories, studies will present data closer to reality, as it will be possible to identify scenarios that are not traceable in remote repositories. Thus, it will be possible to contrast and contribute to these studies.

3 PRIVATE LIFE OF MERGE CONFLICTS

To assess the issues discussed in Chapter 2, we investigate and analyze the many forms of integrating code using Git in the developer's local repository. The main goal of this research is to shed some light on the *private* life of merge conflicts and how they relate to the code integration commands that generate them. In particular, we investigate the use of Git code integration commands, trying to assess whether previous code integration studies are examining only part of the actual code integration scenarios and their conflicts.

First, we explain the methodology used in this research. Then, we present the research questions that will guide this study in order to achieve our goal. To answer the research questions, we conduct a quantitative and a qualitative study that will be explained in Section 3.1. After presenting the study process, we describe how we identify the Git commands and their integration scenarios.

Then, we answer each research question in Section 3.2. We discuss the impacts of the research findings in Section 3.3. Finally, we consider some factors that might have influenced or limited our results in Section 3.4.

3.1 METHODOLOGY

In this research, we focus on studying developers' local repositories to understand a diverse group of code integration scenarios, as some of them cannot be detected by analyzing only remote repositories. To achieve this goal, we implement scripts to mine local code integration scenarios using Git, evaluate the frequency of each identified command, and investigate the possible reasons that lead developers to choose particular code integration commands and strategies. We answer the following research questions:

- **RQ1**: How frequently do developers integrate code using the merge command? How frequently do developers use commands (rebase, squash, cherry-pick) that hide integration from the development history?
- **RQ2**: How frequently do merge scenarios result in conflicts? How frequently do hidden integration scenarios result in conflicts?
- **RQ3**: Which factors can influence developers' choice of Git integration commands and strategies?

To answer RQ1 and RQ2 we measure the frequency of the integration scenarios we find in local repositories and how often they lead to conflicts. We conduct a *quantitative* and *retrospective* study by collecting developers' local logs from Git projects. From these logs, we analyze them using scripts that measure the frequency of different Git commands, such as the successful and failed code integration scenarios.

To answer RQ3, and understand which project characteristics lead to a more significant difference between the public and private lives of conflicts, we carry on a *qualitative* study by performing semi-structured interviews with developers, besides conducting a participant-observation process (SEDANO; RALPH; PéRAIRE, 2017) including the software companies that contributed to this research. In this qualitative study, the author observed Git usage in the development environment of one participant company since the author has worked there since 2018. Also, this author exchanged emails with a developer from another software company participant, asking which Git command is most used to integrate the code in the team and if there are company policies related to this matter. Meanwhile, the Master's co-advisor collected the same information from another software company participant during an informal conversation with the team leader while visiting.

3.1.1 Study Setup

To answer the research questions, we first define our sample. Then, we explain how our script works to measure the frequency of the Git commands. This script is responsible for the quantitative study and assists the results of all research questions.

To conclude, we explain how we carried out our qualitative study. We describe our semi-structured interviews and how we performed the participant-observation process. The qualitative study helps answer the RQ3 question.

The study sample, the script code and instructions, and the interview script are available online (see Appendix A).

3.1.1.1 Sample

To answer the research questions, we use a convenience sample of 95 Git log files (reference logs, or reflogs[1]; it will be detailed in the section 3.1.2.1) belonging to 61 developers. We collect these files mainly through a website that we developed for this study and that participants used to upload their files. The website URL is in Appendix A.

On this website, personal information (name, email, company name, repository URL) was optional. The required information was to include at least one Git log file. The website has a link explaining how it works with data encryption. For each Git log file, the website was responsible for encrypting the email and omitting the commit message. The volunteer had to agree with the disclaimer agreement before submitting all the data.

We use e-mail and social media (mainly LinkedIn, Facebook, and Twitter) to divulge our website and collect voluntary submissions. In an attempt to reach more participants, the author contacted managers from companies partners from the author's university

[1] <https://git-scm.com/docs/git-reflog>

and from projects from other universities. The author and her Master's advisor used professional and personal profiles on social media to engage volunteers by sharing our research and our website.

In the end, 4 software companies agreed to participate in this research, with their developers also submitting files using our website. One company is an open-source software company, where the author has worked since 2018. Two companies have projects and research in partnership with the authors' university. Finally, the last software company is specialized in game development for computer and mobile devices.

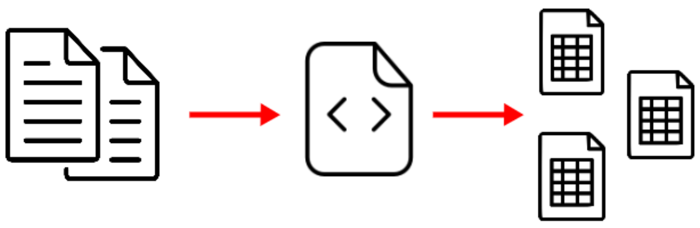
Of 61 developers, 17 were independent volunteers, and 44 were volunteers from the software companies we contacted. The independent volunteers participated via our website. We don't know in which context they worked since we made the insertion of personal data on our website optional. Unfortunately, many of them did not include this type of information. Of the volunteers from the software companies, all of them were developers. Overall, no specific technical skill was requested as a requisite for participating. The only requisite was to use Git in their work routine.

We obtained more log files than participants because they were allowed to share their log files from as many projects as they participated. For the analysis of this research, the script identifies this scenario and aggregates all the log information related to each developer.

3.1.1.2 *Script for log analysis*

To automatically analyze our dataset, we implement a script to identify and count the Git commands of interest to this study, as discussed in Chapter 2. The script takes reflog files as input and generates tables with the occurrences of the commands and related metrics, as illustrated in Figure 5. Since a developer could have worked on more than one project or repository, the script aggregates all the log information related to each developer. We decide to aggregate the data by the developer because it facilitates the evaluation from an individual perspective.

Figure 5 – Study illustration.



Source: The author (2023)

The reflog file structure will be detailed in Section 3.1.2. In resume, the reflog file consists of a sequence of lines, each representing a different git action performed by the developer. The script identifies the Git command by reading the lines and counting each command per developer.

The script creates two folders in the local environment: one with a collaborator's perspective and the other with a total sample perspective. Each folder has files of CSV extension to represent the calculations made from the occurrences of each Git command. A CSV file is a comma-separated values file, which allows data to be saved in a tabular format. For example, one of the files generated is a CSV file with the Git commands as columns and the occurrences as its value.

The calculations were based on the occurrence result. From the occurrences, we calculate the frequencies by changing the event (Git command) we want to analyze in our sample. For example, calculate the merge command frequency (visible integration scenario) among the integration scenarios (visible and hidden). To generate a file with this frequency, the script counts all occurrences of the merge command (successful and failed scenarios) and divides it by the integration scenarios occurrences (derived from visible or hidden scenarios, regardless of whether it was successful or not). Besides the file generation, we use the Git commands frequencies to generate graphics (bean plot) and tables.

In addition, we do not run statistical tests to answer the research questions. Since the questions related to the quantitative study (RQ1 and RQ2) focus on the frequency of the integration scenarios, we do not identify the need to run such tests. Also, we did not run this test on the last research question because it relates to a qualitative study.

3.1.1.3 Interviews

To complement the log analyses, we conducted semi-structured interviews with 9 developers from the same software company. This software company has a project in partnership with the authors' university. The interviews were conducted in one afternoon, and we created an interview script containing 18 questions to explore the developer experience using Git. You can check the questions in Appendix B.

Although this is, in general, considered a small panel, it is adequate for our simpler goal of using the interviews to better understand the results of the log analysis. The interview script begins with questions to determine whether the developer knows and uses each Git command analyzed in the study and why they use (or not) the commands. It then moves forward to questions about how they usually integrate their code with the shared remote repository and if they had to deal with merge conflicts.

The last questions ask whether there is any configuration management and git usage policy or standard in the daily work of the software teams to integrate code using Git. The interviews were carried out individually, and we recorded each one of them. The following is an example of some of the questions we asked in the interview.

- Do you have experience with or know the rebase command?
- When using it, did you have to deal with any conflicts?
- Have you ever had to deal with conflicts resulting from the merge command?
- How did you solve them?
- Is there a workflow pattern in your company to work with Git?

3.1.1.4 Observations

As part of our qualitative study, we conducted a participant-observation involving the software companies that contributed to this research. From the 4 software companies that participated in this research, we collected information from one of them by performing semi-structured interviews. For the remaining 3 participant companies, we collected the information through the participant-observation process.

Concerning the participant observation analysis, the author collected notes while working as an engineer on one of the software company participants of this study and collected information by exchanging emails with one developer from another software company participant. The Master's co-advisor collected information from a different software company by having informal conversations with the team leader while visiting. All the information collected was related to analyzing if any company has a policy or guidance toward using Git integration commands.

To answer the RQ3, we gather information from the quantitative and qualitative studies. This way, we could apply the data from the qualitative study to understand the frequencies of the Git integration commands collected from the quantitative study. For example, given a developer who integrates code mainly using the merge command (high merge command frequency), we would analyze if we have information that helps to explain this behavior from the semi-structured interviews or the company guidance about Git usage from the observation process.

3.1.2 Git Log Analysis

After studying the Git commands that hide code integration scenarios (see Chapter 2), we know that to answer our research questions we need to automatically identify these commands using data from local private repositories.

3.1.2.1 Command identification

Our strategy is to analyze the local history of the project. To keep the developer's local history, Git maintains a temporary local history of changes made to the HEAD and branch references. This tracking records reference logs, or "reflogs", whenever the branches' references are updated in the *local* repository.

The reflog is strictly local, then cannot be pushed or pulled from a remote repository. The reflog is stored as a file and is automatically saved and updated by Git as the user performs Git commands. This file contains information about the Git commands performed that changed branch references.

A Git command changes a branch reference when the "HEAD" (CHACON; STRAUB, 2014) is updated. For example, for each commit performed by the developer in a local repository, Git creates a unique key to reference it, known as an SHA-1 hash (CHACON; STRAUB, 2014). Each Git commit stores your unique key. After a commit action, Git has to update the branch reference to point to that last commit. This update occurs by maintaining a pointer known as "HEAD". This pointer indicates which branch the developer is in and the last performed git command. HEAD is updated automatically by Git whenever the user runs a Git command, and this information is captured in the reflog. That way, Git does not lose track of the repository history.

Since our goal is to analyze integration scenarios, we were able to identify these scenarios in the reflog file because many Git commands update the reference of a branch, such as `git merge` and `git rebase`. The `git merge` updates the reference by adding a new commit, a merge commit. And the `git rebase` updates the reference by reapplying the commits to make the history linear.

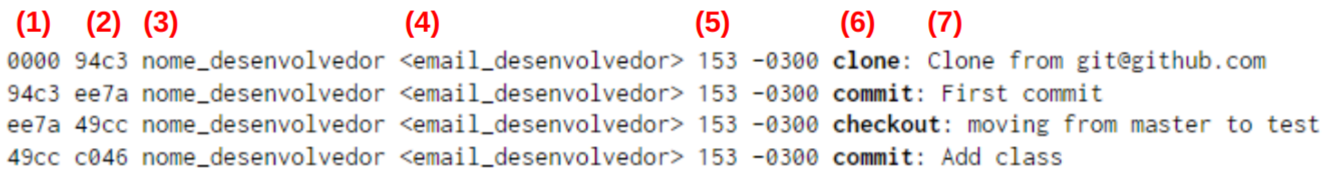
The reflog file is located in the repository's Git directory and can be found through the following directory path: ".git/logs/HEAD". It represents the reflog of the Git repository, and it is a record of all commits that were referenced in the repository. This file has an expiry subcommand that cleans up old or unreachable reflog entries. The reflog expiration date is set to 90 days by default, but this date can be configurable. Also, developers can access the reflog through the command `git reflog` in the terminal.

The reflog file consists of a sequence of lines, each representing a different git action performed by the developer. In each line, it is possible to observe the following data, as Figure 6 depicts: (1) the hash of the previous HEAD commit (first number sequence), (2) the hash of the current HEAD commit (second number sequence), (3) developer's Git name, (4) developer's Git e-mail, (5) date and time equivalent in epoch time[2], (6) the command performed and (7) linked message.

In Figure 6, we can see that the developer first cloned the repository in their local machine. After that, supposing this is the beginning of the project history, they created

[2] The epoch time is defined as the number of seconds passed since January 1st, 1970 at 00:00 UTC.

Figure 6 – Structure of reflog file lines.



```
0000 94c3 nome_desenvolvedor <email_desenvolvedor> 153 -0300 clone: Clone from git@github.com
94c3 ee7a nome_desenvolvedor <email_desenvolvedor> 153 -0300 commit: First commit
ee7a 49cc nome_desenvolvedor <email_desenvolvedor> 153 -0300 checkout: moving from master to test
49cc c046 nome_desenvolvedor <email_desenvolvedor> 153 -0300 commit: Add class
```

Source: The author (2023)

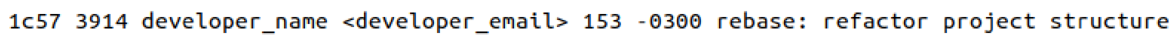
a first commit. Finally, a new branch was created and then the developer committed changes to that secondary branch.

As not all integration commands are clearly represented in this kind of log file, we manually analyzed a number of files to understand how to obtain the integration scenarios, including those that do not appear in the main project history. The commands explored were the following: (a) merge, (b) rebase, (c) cherry-pick, (d) squash, and (e) stash apply (see Chapter 2 for Git command details). The first attempt at identification was made by reading the log files to differentiate the various types of Git commands. Soon after, tests were carried out in local repositories to reproduce the commands mentioned above to recognize how Git records them in reflog files.

After finishing this analysis, it was possible to identify the occurrence of the following commands: (a) merge, (b) rebase, (c) cherry-pick, and (d) squash. We will describe each identified command with an example.

In Git, there are two ways to perform the rebase command: using the non-interactive version or the interactive version (CHACON; STRAUB, 2014). The non-interactive version of rebase executes immediately. The interactive version allows the developer to change individual commits, squash commits together, drop commits or change the order of the commits. Respectively depicted in Figures 7 and 8, we see two examples of how Git records the rebase command in reflog files, both in its non-interactive form and in an interactive way. The rebase can also be used together with the pull action.

Figure 7 – Log line using rebase.



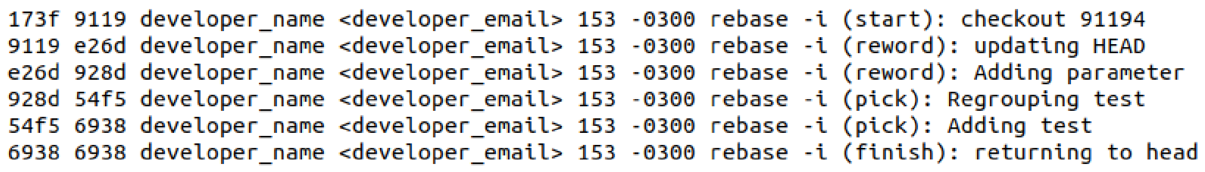
```
1c57 3914 developer_name <developer_email> 153 -0300 rebase: refactor project structure
```

Source: The author (2023)

Git records the cherry-pick command in the reflog file as illustrated in Figure 9. In contrast, Git does not record the squash with a single line command, but we can identify it through the interactive rebase. At the interactive rebasing, the developer can choose a list of commits to perform the squash action. Figure 10 shows an example of this command.

The examples illustrated in the figures follow an abstract and consistent pattern that

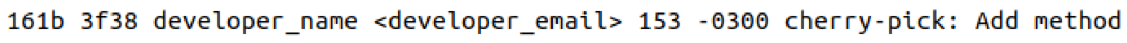
Figure 8 – Log line using interactive rebase.



```
173f 9119 developer_name <developer_email> 153 -0300 rebase -i (start): checkout 91194
9119 e26d developer_name <developer_email> 153 -0300 rebase -i (reword): updating HEAD
e26d 928d developer_name <developer_email> 153 -0300 rebase -i (reword): Adding parameter
928d 54f5 developer_name <developer_email> 153 -0300 rebase -i (pick): Regrouping test
54f5 6938 developer_name <developer_email> 153 -0300 rebase -i (pick): Adding test
6938 6938 developer_name <developer_email> 153 -0300 rebase -i (finish): returning to head
```

Source: The author (2023)

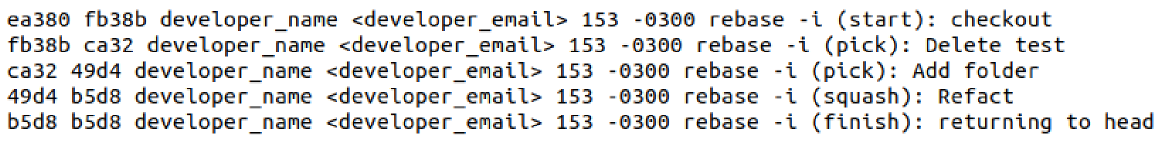
Figure 9 – Log line using cherry-pick.



```
161b 3f38 developer_name <developer_email> 153 -0300 cherry-pick: Add method
```

Source: The author (2023)

Figure 10 – Log line using squash.



```
ea380 fb38b developer_name <developer_email> 153 -0300 rebase -i (start): checkout
fb38b ca32 developer_name <developer_email> 153 -0300 rebase -i (pick): Delete test
ca32 49d4 developer_name <developer_email> 153 -0300 rebase -i (pick): Add folder
49d4 b5d8 developer_name <developer_email> 153 -0300 rebase -i (squash): Refact
b5d8 b5d8 developer_name <developer_email> 153 -0300 rebase -i (finish): returning to head
```

Source: The author (2023)

applies to other instances of the invoked commands, allowing us to uniformly and systematically process reflog patterns by matching these patterns.

Regarding the stash-apply command, we could not recognize it in the log file history. Git even records the moment of stash creation in the reflog file, but does not refer to when the developer reapplies the saved code from the stack into the current branch. No trace of this command is left in reflogs because this command simply applies the Git stash to the current working directory, so no reference is updated. For this reason, we do not consider this command in this work, although it can be used to integrate code.

Even though the reflog file does not contain the occurrence of the stash-apply command, the command still represents a way to integrate code. When this command is applied, independently developed code is integrated into the actual code.

As discussed later, we do not consider all possible integration scenarios in this study, but the ones we consider are still sufficient to support our conclusions.

3.1.2.2 Integration identification

As our goal is to investigate integration scenarios in the developer's local repository, we must identify when an integration scenario occurred. For this study, we aim to identify the successful and failed integration scenarios from the collected Git commands (merge, rebase, cherry-pick, and squash).

It is important to mention that we do not aim to consider fast-forward merges in our study. It means that if the script can identify this type of integration, it will not count its occurrence. A fast-forward integration is when Git moves the pointer of the branch forward. An example is when you try to merge one commit with a commit that can be reached by following the first commit's history, and Git moves the pointer forward because there is no divergent work to merge (CHACON; STRAUB, 2014). This scenario is not interesting for our research because it is not about a merge scenario, simply a branch update. This means there is nothing to integrate, and the branch pointer will move straight forward, resulting in a linear history.

After understanding the difference between integration and fast-forward integration, we must analyze the possible scenarios regarding integration. When integrating code, it can be a successful or failed action. If the integration happens without conflict and is not a fast-forward action, we count it as a successful scenario. When merge conflicts occur, the user has three options: try to fix the conflicts and continue the integration process; skip, meaning that you will drop the commit that caused the failure (none of the modifications made by the conflict commit will be applied); and abort, meaning that you will undo the operation. Regardless of the decision made in such a situation, the conflict makes us count this as a failed integration scenario. For each integration scenario recognized in the Git log analysis, we identify the action results to count how many times the integration was successful or not.

To identify if the integration was a successful or failed scenario, we ran local tests in a local repository to reproduce both scenarios for each Git command (merge, rebase, cherry-pick, and squash) and check how Git records them in reflog files. After this investigation, we could automate our script to count the Git command occurrence and differentiate if it was a successful or failed integration. We summarize our analysis of the integration scenario identification from the Git commands in Table 1. We will detail further each result from Table 1.

Concerning the successful integration scenario, our script can identify all the occurrences when the Git command is merge or rebase (as illustrated by the symbol "=" in the column Successful Integration from Table 1).

However, there are successful scenarios in which we could not differentiate them from the fast-forward because the log line of both scenarios had the same characteristics. It means that both scenarios were registered with the same command and linked message (see Figure 6). We could affirm this after realizing manual testing and checking if the log line would be registered differently. When this situation happens, our script counts both scenarios as successful integrations.

This situation happened with the cherry-pick and the squash commands. The consequence is an overestimated result of the occurrence of these commands when measuring

Table 1 – The identified integration scenarios of the Git commands. Arrows indicate whether the number is underestimated (↑, meaning the numbers could be greater in practice) or overestimated (↓, meaning the numbers could be smaller in practice)

Git command	**Successful Integration**	**Failed Integration**
Stash Apply	Not Available	Not Available
Merge	=	↑
Cherry-Pick	↓	↑
Squash	↓	↑
Rebase		
- Normal	=	↑
- Interactive	=	=
- Pull –rebase	=	↑

Source: The author (2023)

the successful integration scenarios (as illustrated by the symbol "↓" in the column Successful Integration from Table 1). This means that the occurrence number of the successful integration scenario for these commands could be smaller in practice.

Regarding the failed integration scenario, our script can identify all the occurrences when the Git command is the rebase interactive (as illustrated by the symbol "=" in the column Failed Integration from Table 1).

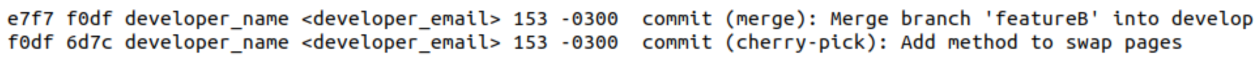
However, there are failed scenarios in which we could not identify if the abort or the skip happened because the log file does not record this information. We could affirm this after realizing manual testing and checking if the reflog file would register any information about the abortion of the action or the skip of the commit. When this situation happens, our script cannot identify and count it.

This situation happened with the merge, the cherry-pick, the squash, and some varieties of the use of the rebase. The consequence is underestimated result for most of the commands when measuring the failed integration scenario (as illustrated by the symbol "↑" in the column Failed Integration from Table 1). It means that our script is losing merge conflict data since it only identifies one of the three pieces of evidence representing that the command failed by a merge conflict. The occurrence number of the failed integration scenario for these commands could be bigger in practice.

A failure integration scenario of the merge and the cherry-pick commands can be identified with a single line, as shown in Figure 11. However, it is different with the rebase and squash commands because these are identified in a block of commands. Figure 12 shows an example of these commands.

To sum up, some of our identified scenarios indicate an approximated result. Some results are overestimated, meaning that the actual result could be smaller. This happens with successful integration scenarios. Meanwhile, other results are underestimated,

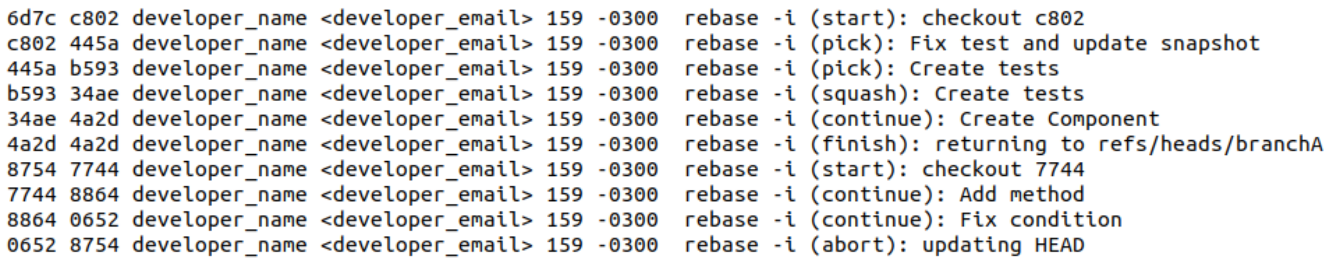
Figure 11 – Failed merge and cherry-pick scenario.



```
e7f7 f0df developer_name <developer_email> 153 -0300  commit (merge): Merge branch 'featureB' into develop
f0df 6d7c developer_name <developer_email> 153 -0300  commit (cherry-pick): Add method to swap pages
```

Source: The author (2023)

Figure 12 – Failed squash and rebase scenario.



```
6d7c c802 developer_name <developer_email> 159 -0300  rebase -i (start): checkout c802
c802 445a developer_name <developer_email> 159 -0300  rebase -i (pick): Fix test and update snapshot
445a b593 developer_name <developer_email> 159 -0300  rebase -i (pick): Create tests
b593 34ae developer_name <developer_email> 159 -0300  rebase -i (squash): Create tests
34ae 4a2d developer_name <developer_email> 159 -0300  rebase -i (continue): Create Component
4a2d 4a2d developer_name <developer_email> 159 -0300  rebase -i (finish): returning to refs/heads/branchA
8754 7744 developer_name <developer_email> 159 -0300  rebase -i (start): checkout 7744
7744 8864 developer_name <developer_email> 159 -0300  rebase -i (continue): Add method
8864 0652 developer_name <developer_email> 159 -0300  rebase -i (continue): Fix condition
0652 8754 developer_name <developer_email> 159 -0300  rebase -i (abort): updating HEAD
```

Source: The author (2023)

meaning the actual result could be bigger. This occurs with failed integration scenarios. Therefore, the number of occurrences will not be exact for some commands and analyses, for which we can only provide estimations in this study. We can observe the summary of the estimations in Table 1.

Since one of our main interests in this study is to provide evidence on integration scenarios happening in the developer's local environment, having an estimated result will not change the conclusion of our research questions' findings because the evidence exists even though it is not an exact number. Similarly, our results are also estimated for the failed integration scenarios, which is enough to show evidence of their occurrence. We only need evidence that these scenarios happen, even if it is an estimated number of occurrences. In other words, our results are not accurate but follow the direction of our conclusion.

3.2 RESULTS

Following the study design presented in the previous section, we analyze 95 Git reflog files from 61 developers to investigate the frequency of integration commands in their local repositories. From this sample, we collected a total of 34504 Git commands. We now present the results of our analysis for each research question.

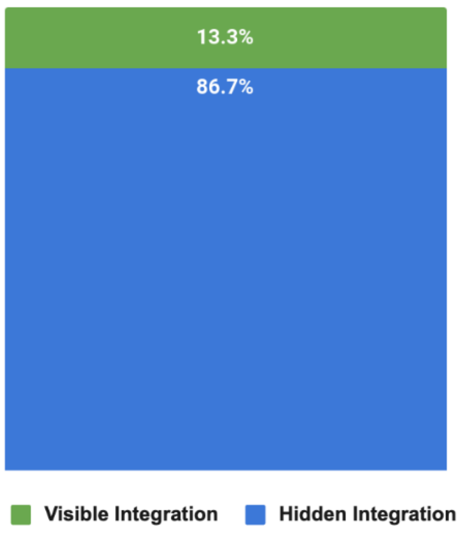
3.2.1 RQ1: How frequently do developers integrate code using the merge command? How frequently do developers use commands (rebase, squash, cherry-pick) that hide integration from the development history?

3.2.1.1 *How frequently do developers integrate code using the merge command?*

To answer this question, our script identifies merge occurrences for each developer. This frequency is the sum of the successful and the failed scenarios. Since our script could not identify the aborted or skipped scenarios (see Table 1), our result represents the underestimated frequency of merge occurrences compared to the developer's actual numbers.

From 34504 Git commands, 4131 of them were Git integration scenarios, representing 12% of all actions. Regarding the merge command, 548 scenarios were identified, representing only 13.3% of the 4131 integration scenarios. The major part of the integration scenarios in our sample, that is 86.7%, is carried out by git code integration commands such as `git rebase` and `squash`, which leave no trace on remote shared repositories. This result indicates that hidden integration scenarios are more frequent than visible ones, at least six times bigger in our sample. The distribution of integration frequency is shown in Figure 13.

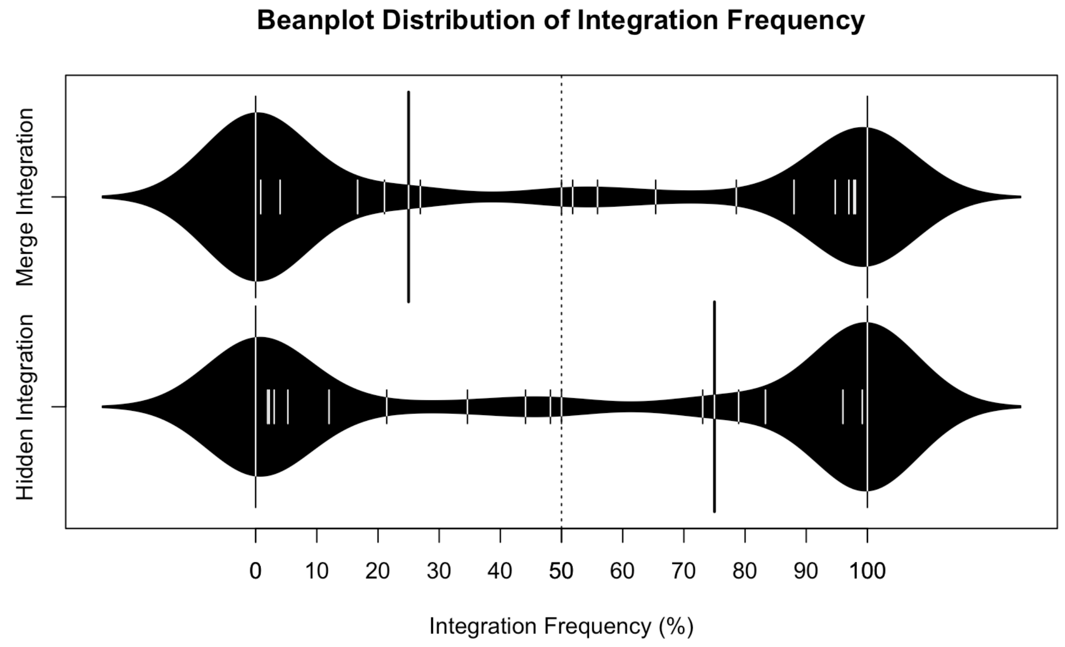
Figure 13 – Distribution of integration frequency in the sample.



Source: The author (2023)

We also analyze the distribution of the integration scenarios per developer. The percentage of merge commands substantially varies depending on the developer, as can be seen in the top part of Figure 14, which shows the distribution per developer. This graph presents the distribution of the percentage of merge command scenarios in relation to the total of integration commands per developer, which includes both the visible integration commands (merge) and the hidden integration commands (rebase, cherry-pick, and squash).

Figure 14 – Distribution of integration frequency per developer.



Source: The author (2023)

In the top part of Figure 14, we can see that the merge action is more concentrated in the extremes of the graph, meaning that there are developers that often use this integration command, whereas others rarely use it. There are only a few points in the middle of the graph, showing that these developers do not seem to have a well-defined preference for visible or hidden integration commands.

According to our study on identifying integration scenarios from the Git commands, the scenarios we could not identify regarding the merge command are when the integration has failed and the developer chose to abort or skip the conflict resolution. Then, this result represents an underestimated frequency of its occurrences. It means that each developer could have used this command more than the result presented here. This limitation of identifying all types of failed scenarios of the merge command also happens with studies that analyze merge conflicts based only on the history of the remote GitHub repository. Thus, our study represents only a fraction of the merge conflict occurrences.

3.2.1.2 How frequently do developers use commands (rebase, squash, cherry-pick) that hide integrations from the development history?

In this question, we compute the frequency of the commands that hide integrations. Then, our script considers the occurrence of the following commands: rebase, cherry-pick, and squash. To answer this question, we must count the successful and failed integration scenarios for these commands.

Regarding the hidden integration scenarios, our script does not identify the fast-forward integration when the scenario is successful for the rebase and cherry-pick commands. Also, the script does not identify when the user opted for skipping or aborting when it is about a failed integration. As explained before, our result represents an estimated frequency of hidden integration commands compared to the developer's actual numbers (see Table 1).

As mentioned before, from 34504 Git commands, 4131 of them were Git integration scenarios, representing 12% of all actions. If we analyze only the integration scenarios, 3583 were identified as hidden scenarios, representing 86.7%. Concerning only hidden integration scenarios identified, rebase was the most used command among the developers with 2353 scenarios (65.7%), followed by cherry-pick with 1092 scenarios (30.5%) and squash with 138 scenarios (3.8%). According to our sample, hidden integration is more frequent than the visible ones (13.3%) in the private repositories.

Although these commands frequencies are estimated numbers, our results maintain that hidden integration frequency is more frequent than the visible ones in the private repositories. This is possible because, even in the worst scenario, the results would follow the same direction. For example, if we only consider the successful integration scenarios, we know that the occurrences identified by the cherry-pick and the squash commands are overestimated while the rebase command is exact (see Table 1). In this case, our worst scenario is that all the occurrences by the cherry-pick and the squash commands were fast-forwarded scenarios, leaving our results only counting the rebase command as a hidden integration scenario. Since the frequency of the rebase command is four times bigger than the frequency of the merge command in our sample, our conclusion would remain the same: hidden integration is more frequent than the visible ones in the private repositories. So the use of approximations, in this case, might affect the exact reported numbers and percentages but not the overall conclusion and direction of the result.

This result varies per developer, having a similar behavior as the merge command. As shown in the bottom part of Figure 14, the hidden integration distribution varies, showing more occurrences in the extremes sides. It means some developers frequently use this integration command, whereas others rarely use it. Comparing both distributions in the previous figure, we can see that they complement each other, as expected.

Result 1: The frequency of code integration scenarios created by the git merge command is significantly smaller than those created by other git code integration commands, representing only 13.3% of the captured integration scenarios in our sample. However, it can substantially vary per developer, with some developers heavily relying on the merge command for code integration. This result represents an underestimated frequency of merge occurrences, which could be higher in the actual event. This limitation also happens with previous studies focused only on the remote Git repository history. Regarding the hidden integration scenarios, the frequency of these scenarios is more frequent than the visible ones in the private repositories, representing 86.7% of all integrations. However, it also substantially varies per developer. The most used command among the hidden integration scenarios is rebase.

3.2.2 RQ2: How frequently do merge scenarios result in conflicts? How frequently do hidden integration scenarios result in conflicts?

3.2.2.1 How frequently do merge scenarios result in conflicts?

According to existing studies (BRUN et al., 2013; KASI; SARMA, 2013; BRUN et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; ACCIOLY; BORBA; CAVALCANTI, 2018a) on merge conflicts and resolutions in public repositories, it has been reported that from 5% to 20% of all merges fail, with some projects achieving rates of almost 50% (ZIMMERMANN, 2007; BRUN et al., 2011). But, what about the merge conflict rate in the developer's local repositories?

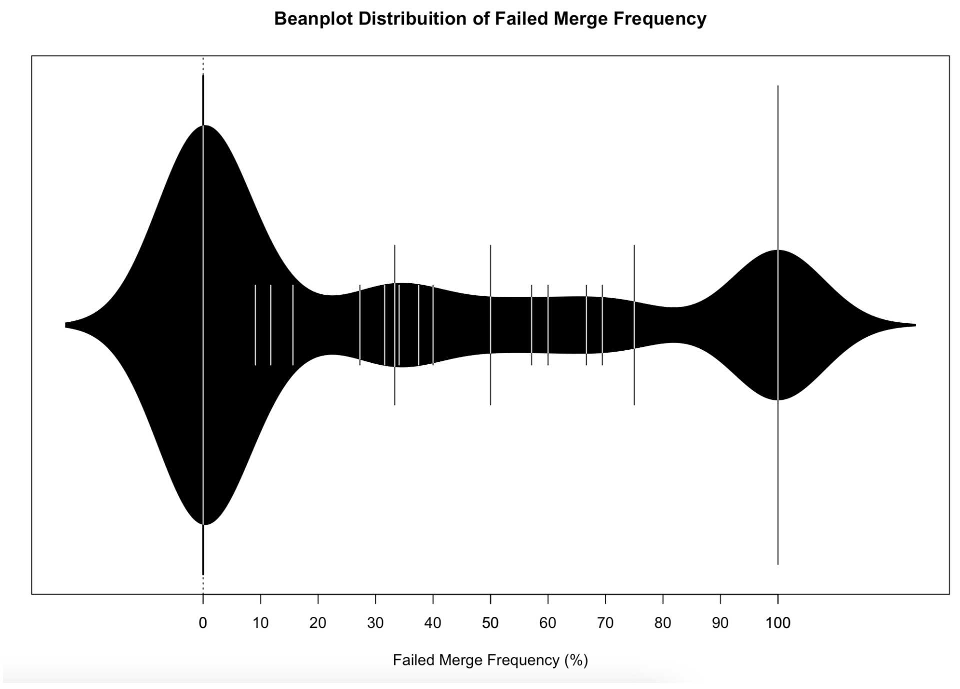
Examining our sample, we have a total of 548 merges identified. Among these merges, 204 (37.2%) failed, that is, ended up having at least one merge conflict. Let us analyze the distribution of failed merges frequency by developer, as Figure 15 shows.

Analyzing Figure 15, we notice that the median of this graph is almost on axis 0 and many occurrences of failure appear throughout the distribution, meaning that developers had failed when merging in different proportions compared to each other. Remember that we do not count the merge conflict scenarios followed by the abort or the skip action due to our script limitation for identifying these actions (see Table 1). Because of this, this result represents a lower bound number of failed merge scenarios.

Our results show a higher merge conflict rate in the developer's local repository (37.2%) than in studies that analyzed merge scenarios from the shared remote repository (5–20%). Still, this result should be interpreted as a complement to future analyses because we are just demonstrating that merge conflicts are happening locally and maybe with a higher frequency.

As explained before, the squash command can rewrite the Git history, and the commit merge can be lost if it belongs among the squashed commits. Thus, there is a possibility that merge occurrences are less when analyzing only the remote repository history than

Figure 15 – Beanplot Distribution of Failed Merge Frequency.



Source: The author (2023)

analyzing the local one. Since the merge can occur more and fail at a higher frequency in the developer's local repository, the studies that analyze merge conflicts based only on the remote repository's history present a fraction of the actual result.

According to our sample, at the same time that merge occurrences are less frequent when compared to the hidden integration ones in local repositories, it tends to fail often. Therefore, this evidence suggests that studies might miss merge conflict data by not analyzing the developer's private repository. Such merge commits might not appear in the shared remote repository history due to the use of Git commands that rewrites history, such as rebase and squash.

3.2.2.2 How frequently do hidden integration scenarios result in conflicts?

We have identified 3583 hidden integration scenarios in our sample. 405 of them ended up in conflicts, representing 11.3% of the total. Among these conflicts, rebase was the command which failed the most (80.5%), followed by cherry-pick (19%) and squash (0.5%).

Because of the script limitation, the result of the failed hidden integration scenarios represents the lower bound number of conflict scenarios. Hence, we can say that the hidden scenarios' situation is the opposite of the merge because it is more frequent in local repositories but tends to fail less in our sample.

This result represents an underestimated frequency of failed integration commands. Even representing an underestimated result, the percentage of merge conflict (11.3%)

found in our sample in the hidden integration scenario is consistent with the rates of conflicts found in previous studies regardless of the integration command. Still, since one of our research goals is to inform that studies are losing integration conflict data by not analyzing the developer repository, reporting a minimum frequency of code integration conflict would not invalidate our result. No matter how imprecise the approximation is in this case, it's still solid evidence that studies are losing integration conflict data by not analyzing the developer repository. The exact percentage of loss, however, is not accurate.

Moreover, we can compare the failed scenarios of the Git commands individually. This way, we can analyze the frequency of this scenario among the usage of each command, as Table 2 shows. Analyzing Table 2, we noticed that the merge command was the one which failed most when the developer tried to use it. Followed by the hidden integration commands (rebase, cherry-pick, and squash). Remember that these frequencies represent an underestimated result, so the actual number could have been higher. This result reaffirms that developers deal with conflicts in their private repositories, perhaps more frequently than those identified in the Git remote history.

Table 2 – The frequency of the failed integration scenarios of the Git commands individually

Git command	**Failed Integration Frequency**
Stash Apply	Not Available
Merge	37.2%
Rebase	13.9%
Cherry-Pick	7%
Squash	1%

Source: The author (2023)

Result 2: Our results show that *hidden* and *visible* code integrations happen in the developers' local repository and both could end up in merge conflicts (hidden integrations with lower bound of 11.3% and visible integrations with lower bound of 37.2%). These conflicts are resolved locally before the developers synchronize their contributions to shared remote repositories, so we can see no trace of them when analyzing only remote repositories. In resume, these results bring evidence that studies that focus only on GitHub project history might be losing integration conflict data by not considering the information in local repositories, reinforcing the need to consider both the *public* and the *private* life of merge conflicts.

3.2.3 RQ3: Which factors can influence developers' choice of Git integration commands and strategies?

Since the choice of git code integration commands and strategies can significantly impact the frequency of integration scenarios and merge conflicts that are missed by code integration studies that focus only on shared remote repositories, it's important to investigate why developers choose a particular integration command or strategy in their work routine. In particular, it's important to identify which factors influence their choice. To understand that and answer this research question, we conducted semi-structured interviews with 9 developers from the same company, and we collected additional information by conducting a participant observation study in the remaining 3 participants companies (See section 3.1).

We also performed an additional Git reflog analysis to complement the interviews by investigating secondary research questions derived from the interview transcripts' analysis. This analysis groups the developers into two categories: those with a high visible integration frequency and those with a high hidden integration frequency. We grouped only developers that use the same code integration strategy (visible or hidden) in more than 70% of the situations in which they have to integrate code. This way, we leave out of the analysis only four developers that adopt a less unbalanced mix of strategies. After the grouping, we analyzed which company each developer works for, comparing it with the information collected from interviews and the participant observation notes. This way, we can shed light on which aspects are more likely to affect code integration command or strategy choice.

Developers with high visible integration frequency

Among the 61 developers from our sample, 25 have visible integration frequency ranging from 79% to 100%, constituting one of the mentioned groups. Of those 25, 19 developers only invoked visible integration commands, whereas 6 invoked merge and hidden integration commands. These 25 developers are employees of 2 specific software companies in this group.

All 9 interviewed developers belong to this group of high visible integration frequency, and all work at the same company. The interviews showed that many of them are not sufficiently familiar with the git integration commands that lead to hidden integration. They are also unaware of any company guidance for using specific git integration commands in their work routine. Regarding the other software company representing this group, we learn from the email exchange that there is a company-wide recommendation for using the merge command instead of rebase, for example. This is a light recommendation, though, not being mandatory to use the git merge command.

Developers with high hidden integration frequency

The second group consists of 32 developers having hidden integration frequencies ranging from 73% to 100%. From this total, 26 developers only invoked hidden integration commands, and six invoked also the merge command besides the hidden integration commands. We find that all employees of 2 specific software companies belong to this group.

Based on our conversation with the team leaders, and experience from working as an engineer at one of the software companies participating in this study, we know that one of the companies has strict guidelines for not allowing the use of the git merge command. In contrast, the other company recommends using rebase instead of merge, but not in a mandatory way. One of the reasons for the orientation to use rebase was because of the benefits of linear history.

Outside the two polarized groups, only 4 developers performed both kinds of integration with balance, representing 6.6% of the total developers. Those developers do not belong to any software company described before. Also, those developers did not fill in any personal information on the website to help us reach them. Due to a lack of information, we cannot inform the factors that influence their choice of balanced integration commands. We consider these developers as outliers in our study.

According to our sample and investigation, company guidance is one of the factors that influenced the code integration strategy. Depending on how strictly the company guidance is spread among the collaborators, the developers will not have the autonomy to self-decide which code integration strategy to follow. For future studies, it would be interesting to investigate how the companies decided on the Git integration strategy for all the employees.

Another influencing factor is the Git experience. In our sample, being unfamiliar with the Git integration commands that lead to hidden integration drove the developers to opt for the merge command. In contrast, because of the knowledge of making Git history linear, some team leaders guide their teams to follow this strategy. Thus, Git experience is an important factor in the choice of code integration.

Result 4: Git experience and company guidance might influence which Git integration command or strategy a developer chooses. For example, a developer less fluent in Git may prefer to use the merge command, as it is often considered simpler.

3.3 DISCUSSION

Before this research, our main questions were whether hidden integration scenarios happened frequently, and why developers preferred to use such commands instead of `git merge`. Based on our results, hidden integration scenarios might occur even more frequently than visible ones; 6 times more frequently in our sample. This is an alarming difference. We expect similar or even greater differences to be observed in contexts that recommend or demand the use of commands such as `git rebase`. We also bring evidence that hidden integration scenarios lead to conflicts, although at a lower rate than can be observed from the visible scenarios in our sample.

Such expectations are reinforced by the analysis of the performed interviews, which show that the motivation to adopt different code integration strategies might come from personal Git experiences or simply from company guidance. This way, our study brings implications for both software engineering researchers and practitioners, as we detail further.

Previous empirical studies analyze merge conflicts characteristics based only on the history of the main GitHub repository (CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018b; NGUYEN; IGNAT, 2018; ACCIOLY; BORBA; CAVALCANTI, 2018a; CAVALCANTI; ACCIOLY; BORBA, 2015; GHIOTTO et al., 2020; AHMED et al., 2017), that is, a public shared remote repository. Regarding identifying the merge command occurrences, these studies can only recognize when the integration was successful or when it failed and the developer tried to fix it and continue the action. This same behavior was detected in our study.

However, after digging deeper into identifying Git commands, we concluded that this was a limitation because we were not counting the scenarios when the integration failed, but the developers opted to skip not to solve the issue or opted to abort the action. This scenario identification limitation implies that our result could be underestimated compared to the developer's actual numbers. As this limitation impacts these studies, their results may also be underestimated.

As detailed in this research, there are Git commands that can rewrite Git commit history. These scenarios cannot be identified for these studies because these commands leave no integration trace, like a commit merge. Consequently, the conflicts generated by these scenarios that were resolved locally are not identifiable by only analyzing the shared remote repositories. Thus, the hidden integration scenarios were not analyzed in these types of studies.

Answering the questions carried out by this chapter, our results suggest that studies, which analyze merge conflicts based only on the history of the main GitHub repository, might be missing both integration scenarios and merge conflicts. By not including the developer's local repositories, these studies are analyzing and reporting just a fraction of the actual integration scenarios and conflicts faced by developers working on those projects. So at least part of the results in those studies should be interpreted as lower bounds, for example, of how often integration conflicts occur in practice. A few studies even list that as a possible threat but do not bring further evidence of the problem nor give an idea of the dimension of the threat, as we do here.

Detailed further work is then necessary to understand how precisely each previous work is affected by the findings we present here. Some might even not be affected at all, if their sample consists only of projects that demand developers to use only git merge for code integration; but, given how projects are often selected for these studies, we would expect such situation to be rare. Others might not be affected because their outcomes are not dependent on the existence of hidden integration scenarios; this is certainly not the case for outcomes that are related to conflict frequencies or causes, for example. To understand how each work is affected, a researcher has to analyze carefully the study outcomes and how they could be potentially impacted by the existence of hidden integration scenarios and conflict. To measure that with precision, in the cases of risk, one should have access to the reflog files from the developers of the analyzed repositories. Still, we affirm that related works might be losing integration conflict data because there is a previous study with the same concerns (ACCIOLY; BORBA; CAVALCANTI, 2018a).

Besides how previous work is affected by our findings, further work is necessary to understand the complexity of resolving conflicts caused by hidden integration scenarios. Since our research goal focuses on identifying other types of integration scenarios and their merge conflicts, we do not have data to inform the complexity of resolving these conflicts because our research did not involve replicating such scenarios.

In summary, our results recommend studies focusing on shared remote repositories present their results as lower bounds to the actual number of integration scenarios and conflicts. We recommend adopting a strategy that, relying on our scripts and the analysis of reflog files, includes counting the hidden merge scenarios to complement the research results for future studies.

Furthermore, in order to discover the factors that influence the strategy to integrate code, our sample of interviewers shows that developers tend to use the merge command when the company has no explicit guidance on how to use Git or because developers don't have experience in other Git commands. This reason could indicate that the merge command is easier to understand and apply. Companies and developers should then invest in better understanding and recommending alternative git code integration commands if they are more appropriate for the contexts of the projects carried on in these companies.

Even though the rebase command can be more complex, it makes the commit history clean and easier to understand. With our findings, the software industry can reflect on the different integration scenarios' impact and create the guideline with the best fit.

3.4 THREATS TO VALIDITY

In this section, we discuss some factors that might have influenced or limited our results.

Construct Validity

In order to detect hidden merge scenarios, we decided to analyze the developer's Git reflog files. Git creates the reflog file locally when the developer clones or starts a new repository using Git. However, the reflog file gets deleted when the developer deletes the project from the local machine. Even if the developer clones again the previously deleted project, the reflog file's old content cannot be recovered. Besides that, Git limits the size of the reflog file. Meaning that it erases the reflog content at some point so that future Git actions can be saved. All those situations we describe mean that we might not be analyzing the full history of the developer's action in a repository. This, nevertheless, does not compromise our results showing that studies that focus only on shared remote repositories might be missing code integration scenarios and conflicts. In fact, since we might be considering only part of the local repository history, we know that the studies could be missing more than we report here, but not less.

Since our chosen strategy to identify hidden integration scenarios depends on reflog files, we created a script responsible for detecting these scenarios. Still, we could not detect all the successful and failed scenarios. To minimize the threat, we specify whether the numbers associated with a particular command are underestimated or overestimated. Nevertheless, even with estimated results, we could inform the ranges of command occurrence and therefore analyze potential worst and best situations.

Internal Validity

A potential threat is our approach to identifying hidden integration scenarios. We implemented a script responsible for determining the occurrence of those actions. The script reads each reflog file line and detects which Git command that line corresponds to. We implemented this script based on our manual analysis while we were identifying the syntactic patterns of the Git commands in the Git log files. As a result, the script may not be counting the exact number of occurrences of the Git commands that hide integration. The reason can be a specific scenario that does not appear in the log file or a new pattern of a particular Git command that we did not implement in the script. We made manual analysis in different log files before running the study to reduce this threat.

External Validity

In this research, we collect our data via a website. This website was available for voluntary submission. This aspect could lead our study to be impacted by self-selection bias, which would only represent a particular subset of people who were comfortable participating. However, more than half of our sample came from developers teams from software companies we contacted with, and some of these developers had the option to self decide if they would participate because their manager asked them to participate. Therefore, we reduce the threat of self-selection bias. Another threat is that our semi-structured interviews were with developers from the same software company; so our answer for RQ4 is strongly associated with this company context.

Most reflog files collected in this study came from 4 different software companies described in the previous sections. Although our research does not restrict any Git projects from participating in the study, it was not possible to collect a significant number of log files. The reason provided by other companies was the sensitive information that could be in the commit message or even in the branch name provided by the developer. Even though our research would not analyze this kind of information, some companies refused to share their log files. Thus, our sample might not be large and representative enough to generalize our results. For future work, we could work on increasing the sample size.

4 EVALUATION OF CODE INTEGRATION FREQUENCY AND ITS CONFLICTS

In Chapter 3, we analyzed the integration scenarios of the Git commands from the developers' local repositories and their conflicts. In this study, it was possible to obtain integration scenarios that are not traceable in the remote repositories. We named this type of scenario a hidden integration, while the scenario caused by the merge command was called visible integration. We analyzed the frequency of both integration scenarios and considered factors that led the developers to choose particular code integration commands and strategies.

In this chapter, we are still investigating the *private* life of merge conflicts from developers' local repositories. However, we focus on understanding the code integration frequency and its association with code integration conflicts. Our goal in this chapter is to discover if there is a correlation between the frequency of developers' code integration and how frequently these scenarios result in conflicts.

To assess this correlation, we applied statistical tests to our sample. Since we have data on visible and hidden integration, we can investigate this relationship by the type of integration scenario or with both scenarios. Our expectation about running the study into these different groups is to examine if the result would differ between them. In other words, verify if the relationship would be statistically significant and if its degree would change when grouping developers with the same strategy of integrating code or with different strategies. This way, we can compare our results with previous studies, as explained in Chapter 2.

First, we will explain the methodology used in this research in Section 4.1. Then, we will present our results in Section 4.2. We will discuss the impacts of the research findings in Section 4.3. Finally, we will consider some factors that might have influenced or limited our results in Section 4.4.

4.1 METHODOLOGY

In this chapter, we analyze the frequency of developers' code integration and how frequently the integration scenarios result in conflicts. Our goal is to understand if these data are correlated. Once our sample contains visible and hidden integration scenarios, we run this study in 3 groups. The first group includes the study sample, and the other two groups are divided by the integration scenario type (visible and hidden).

To achieve this goal, we reuse the quantitative study of Chapter 3 to mine local code integration scenarios using Git and evaluate the frequency of these scenarios. Then, we will apply statistical tests to measure the relationship between these frequencies. We answer the following research question:

- **RQ**: Is there a statistically significant relationship between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts?

To answer the research question, we seek how often developers integrate code in local repositories, calculate how often integration scenarios lead to conflicts, and measure the correlation between these frequencies. We conduct a *quantitative* and *retrospective* study by collecting developers' local logs from Git projects, and analyzing them using scripts that measure the frequency of different Git commands and the successful and failure code integration scenarios. As mentioned before, part of the *quantitative* study is the same from Chapter 3. To complement this study, we apply statistical tests to analyze the interdependence between the collected data.

4.1.1 Study Setup

To answer the research question, we conduct a quantitative study. First, we define our sample. We analyze 95 git log files from 61 different developers. This sample is the same from Chapter 3. Then, we explained how our script measures the frequency of the integration scenarios. We used the same script from the last chapter (see details in Subsection 3.1.1.2). Still, we evolved it to measure other occurrences from the sample and get different frequencies than those calculated from the previous chapter.

To conclude, we run statistical tests to verify the correlation between developers' code integration frequency and integration scenarios resulting in conflicts. We create a script to run the statistic tests in R[1]. The statistical tests will be detailed in this section.

The study sample, the script of the Git log analysis, and the script of the statistic tests are available online (see Appendix A).

4.1.1.1 Sample

To answer the research question, we use a convenience sample of 95 log files (reference logs, or reflogs[2]; as detailed in Section 3.1.2.1) belonging to 61 developers. In resume, we collected these files mainly through a website[3] we developed for this study that participants used to upload their reflog files. In addition, 4 software companies agreed to participate in this research, with their developers also submitting files using our website. Of 61 developers, 17 were independent volunteers, and 44 were volunteers from the software companies we contacted. For further details about this sample, see Subsection 3.1.1.1.

[1] <https://www.r-project.org>
[2] <https://git-scm.com/docs/git-reflog>
[3] <https://tinyurl.com/privatelifemergeconflictsws>

As mentioned before, we run this study into 3 groups. The first group includes all 61 developers. The other groups were divided by the type of integration scenario. We split the sample by identifying the integration scenario that occurred in the majority of each developer. In the end, the second group represents the developers with high visible integration frequency (merge command), and the final group contains the developers with high hidden integration frequency (rebase, cherry-pick, and squash).

4.1.1.2 Script for log analysis

To automatically analyze our dataset, we implemented a script to identify and count the Git commands of interest to this study, as discussed in Chapter 2. The script takes reflog files as input and generates tables with the occurrences of the commands and related metrics.

The reflog file consists of a sequence of lines, each representing a different git action performed by the developer. The reflog file structure is detailed in Section 3.1.2. The script identifies the Git command by reading the reflog file and counts each command per developer. See Section 3.1.2 to understand how Git commands and integration scenarios are identified. For further details about this script, see Subsection 3.1.1.2.

After counting the Git commands of each reflog file, the script generates CSV files to represent the calculations made from the occurrences of each Git command. A CSV file is a comma-separated values file, which allows data to be saved in a tabular format. For example, one of the files generated is a CSV file with the Git commands as columns and the occurrences as its value.

If we want to analyze a specific Git command, we can calculate its frequency by its occurrence and generates a file only with that command frequency information. To answer the research question of this chapter, we are interested in the developers' integration frequency and how often this integration results in conflict.

To obtain the developers' integration frequency, the script counts all the integration scenarios (visible and hidden, successful and failed ones) and divides them for the period of days indicated by the reflog file. Each line of the reflog file contains the date and time information when that Git command was executed (see Subsection 3.1.2.1). So, the script calculates the period of days between the first and the last reflog line.

To obtain how often the integration scenario results in conflict, the script counts all the failed integration scenarios (visible and hidden) and divides them for the integration scenarios (visible and hidden, successful and failed ones). For example, the script identified that a reflog file has 100 integration scenarios regardless of whether it was hidden, visible, or successful. From these scenarios, it was identified that 40 have failed, regardless of whether they were derived from visible or hidden scenarios. Then, in this example, the script would divide 40 by 100 to obtain the failed integration frequency.

4.1.1.3 Statistical Tests

In this chapter, we aim to understand if there is a correlation between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts. It means that we will have to measure the association between these events.

To achieve this goal, we will have to run statistical tests such as correlation coefficient and normality tests. We run the statistical tests using a script written in R language, which is a programming language for statistical computing and graphics.

First, the correlation coefficient is a statistical test to measure the degree to which two variables tend to change together. The null hypothesis of this test is that there is no significant correlation between the 2 variables. When two variables are correlated, a change in one variable implies a change in the other (and vice versa). The coefficient describes the strength of the relationship.

When running a correlation test, we must pay attention to the result of two variables: ρ-value and correlation coefficient. A ρ-value is a measure of probability used for hypothesis testing. A low ρ-value would lead you to reject the null hypothesis. A typical threshold for rejecting the null hypothesis is a ρ-value of 0.05. That is, if you have a ρ-value less than 0.05, you will reject the null hypothesis.

The correlation coefficient is measured by values from -1 to 1. The closer to the extremes (-1 or 1), the greater the strength of the correlation, while values close to 0 imply weaker or non-existent correlations. When the correlation coefficient approaches 1, there is an increase in the value of one variable when the other also increases. When the coefficient approaches -1, it means that when one variable's value increases, the other's value decreases.

There are many ways to calculate the correlation coefficient. Depending on how the variables behave, one correlation coefficient test is more appropriate than another. If the data follows a normal distribution, we must opt for a parametric test, which relies on this type of distribution to validate its result. If the data does not follow a normal distribution, it is recommended to use a non-parametric test. Due to our results (which will be discussed in Section 4.2), we had to opt for a non-parametric test, and we decided to run Kendall's and Spearman's correlation coefficients. Based on this study (PUTH; NEUHäUSER; RUXTON, 2015), whenever it has to choose a non-parametric test, there is no strong reason to select either of these tests over the other. Besides that, it is not complex to run both tests given the online documentation provided by the R language. But, before running the correlation test, we must conduct a preliminary test to verify the data distribution.

To determine whether a sample comes from a normal distribution, we will run the preliminary tests: Shapiro-Wilk and Kolmogorov-Smirnov. We decided to run both tests because developing this test using the R language is not complex. Besides that, the evidence is reinforced once we verify that the results of both tests converge. The null hypothesis of these tests is that the population is normally distributed. Both tests produce a result with a corresponding ρ-value. If the ρ-value is less than the chosen alpha level, then the null hypothesis is rejected and there is evidence that the data tested are not normally distributed. On the other hand, if the ρ-value is greater than the chosen alpha level, then the null hypothesis can not be rejected. Thus, if the ρ-value is less than α=0.05, the null hypothesis is rejected and there is no evidence that the sample comes from a population that is normally distributed.

Given the two frequencies (the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts) we seek to analyze, they will be the two variables used in our statistical tests. In this study, we will measure the data distribution, so we can choose the appropriate test to calculate the correlation coefficient.

To run the statistical tests, we implemented an R script. We use the output from the Git log analysis script as the input data of the R script. As explained before, The Git log analysis script generates CSV files to represent the frequencies calculated. We will use these CSV files to populate the data and run the statistical tests: normality test and correlation test. The R script is available online.[4]

4.2 RESULTS

Following the study design presented in the previous section, we analyze 95 Git reflog files from 61 developers to investigate whether a correlation exists between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts. We run statistical tests to answer the research question. We now present the results of our analysis.

4.2.1 RQ: Is there a statistically significant relationship between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts?

To be able to answer the research question, we will need to calculate two frequencies before the statistical tests: the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts. To simplify what needs to be answered in this study, we split the research question into 3 steps, each representing a procedure to be executed so we can discover the association between these two frequencies. We summarize each item below:

[4] <https://tinyurl.com/privatelifemergeconflictsgh>

Research Question Steps	Description of steps to answer the research question
1:	Obtain how often developers integrate code by the Git log analysis script. Run the normal distribution tests in the sample.
2:	Obtain how frequently integration scenarios result in conflicts by the Git log analysis script. Run the normal distribution tests in the sample.
3:	Run the correlation tests between the samples from the previous steps.

The methodology to obtain the samples from step 1 and step 2 was discussed in Section 4.1.1.2, while the statistic tests (normal distribution and correlation) from all steps were discussed in Section 4.1.1.3. With the methodology set, we must define the sample used for the analysis. From the study sample (see Section 4.1.1.1), we decide to run the analysis in 3 groups.

The first group is the collected sample, which contains the Git reflog files from all developers that participated in this study. The second group is the developers with a high visible integration frequency. Finally, the third group comprises developers with a high hidden integration frequency.

The second and third group represents the developers that use the same code integration strategy (visible or hidden) in more than 70% of the situations in which they have to integrate code. This way of grouping developers was performed to answer a research question from the last chapter (see Section 3.2.3).

We divided the sample into these groups to analyze if the Git integration strategy has any effect on the relationship between the frequency of developers' code integration and the frequency of integration resulting in conflict. This way, we can shed light on the association between developers' code integration frequency and failed code integration frequency.

We present the results of the analysis of each group individually. Then, we present our interpretation of these results.

4.2.1.1 Developers with visible and hidden integration frequency

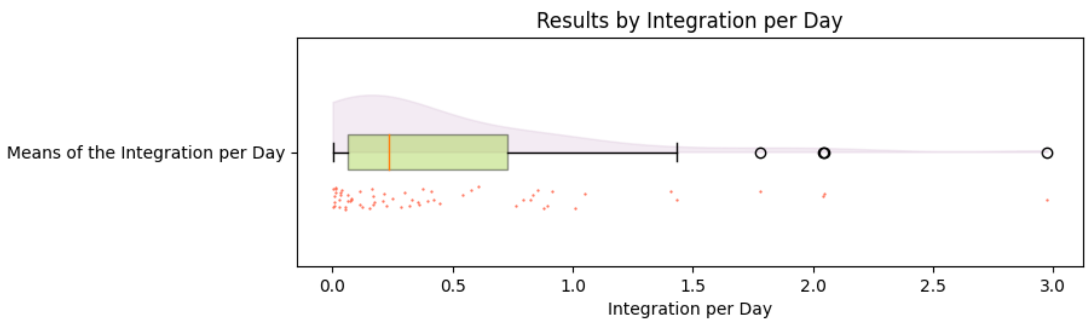
In this subsection, we will use the sample collected for this study, which comprises 61 developers. First, we collected the developers' code integration frequency by the Git log analysis script. Then, we collected the frequency of failed integration scenarios by the Git log analysis script.

Figure 16 shows the distribution of the frequency of developers' code integration. In other words, it means to display how often developers integrate code. This figure is a raincloud plot (ALLEN et al., 2019). This type of plot combines several chart types to visualize the raw data, the distribution of the data as density, and key summary statistics at the same time. There are 3 graphics combined in this type of plot: the violin plot, the jitter plot, and the box plot. The upper half of the violin plot is to visualize the data distribution, representing the cloud. The jitter plot displays the data as single dots in the plot, then it represents the rain. And lastly, the box plot helps to understand the key statistics as median, mean, and quartiles.

Since this chart is for understanding how often developers integrate code, the data represents the total integration hits divided by the period of days. Thus, we can interpret that their integration frequency was daily when the graph data is close to 1.0. Analyzing the plot, we notice that most of the sample does not integrate code daily. This can be observed by the median and the third quartile of the box plot, their values lower than the value 1.0, and from the distribution of the data (violin plot and jitter plot).

The median of the box plot indicates that 50% of the selection integrates code every five days (values equal to 0.2) or even extends the interval up to ten days (values equal to 0.1). From the jitter plot, we noticed a lot of data close to the value 0. However, this is due to the scale of the graph and could mean that the developer has an integration frequency closer to 10 days or more. Also, we have some outliers that integrate code more than once a day (values greater than 1.0), which can be seen as dots and circles on the chart. The average of this sample indicates that developers integrate code every two days (value equal to 0.5). However, this data has an asymmetric distribution since the median line is not at the center of the box.

Figure 16 – Raincloud Plot of Integration per Day.

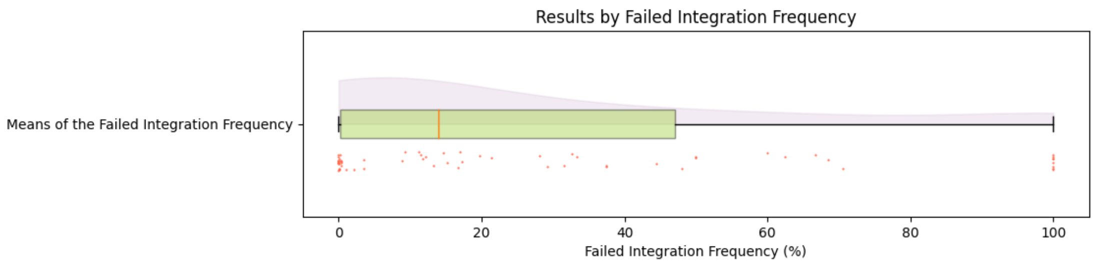


Source: The author (2023)

Figure 17 is a raincloud plot and shows the distribution of the failed integration frequency. From the data distribution, we can see that the percentage of failures varies in our sample. The median of the box plot indicates that 50% of the selection has a rate of failure in their integration scenarios close to 15%. However, if we analyze 75% of the data, the percentage of failures increases to values higher than 40%. From this difference and exploring the plot shape, we see the data has an asymmetric distribution.

This plot has a few outliers, representing extreme behavior regarding integration conflicts. Or the developer has succeeded in every tentative of integrating code (values close to 0) or has failed in every each of them (values close to 100). By manual analysis of the reflog file, we notice little code integration tentative in these extreme cases.

Figure 17 – Raincloud Plot of Failed Integration Frequency.



Source: The author (2023)

Before running the statistical tests, we provide a graphical representation to illustrate the relationship between these two samples. As shown in Figure 18, we use a scatter plot to represent the relationship between the distribution of the frequency of developers' code integration and the distribution of the failed integration frequency. In a scatter plot, the dots represent values for two numeric variables measured for the same individuals. The position of each dot on the horizontal and vertical axis indicates values for an individual data point.

Analyzing Figure 18, we can notice that the individuals that have a higher failed integration frequency are the ones that do not integrate daily, extending the code integration interval up to ten days (x-axis values close to 0.1). Moreover, we can notice the failed integration frequency decreasing when the developers integrate at least every two days (x-axis values equal to or higher than 0.5).

Comparing it with the raincloud plots, we can also identify the outliers. Since the outliers represent extreme behavior, we can see their values marked in the extremes on the vertical axis (y-axis values equals 0 and 100). As explained, we notice little code integration tentative in these extreme cases.

In order to run the correlation test, we must verify the data distribution of each one by running the normal distribution test. As described in Section 4.1, we run the *Shapiro-Wilk* and *Kolmogorov-Smirnov* tests on both samples (code integration frequency and failed integration frequency). The ρ-value from each normality test is presented in Table 3.

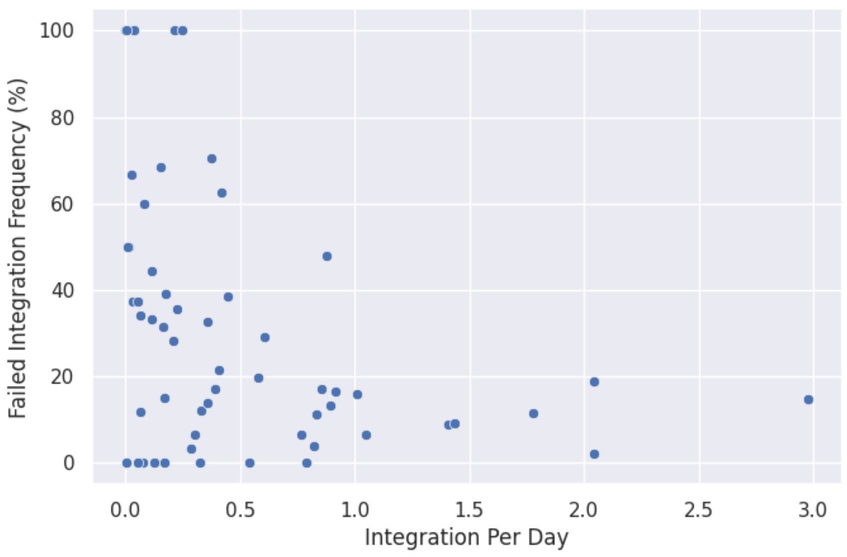
Table 3 – Normality Tests Results.

Normality Test	**Code Integration Frequency**	**Failed Integration Frequency**
Shapiro-Wilk	5.952e-09	7.136e-07
Kolmogorov-Smirnov	7.772e-15	6.883e-14

Source: The author (2023)

The normality tests results point in the same direction. Since the ρ-value is less than 0.05, we reject the null hypothesis (the population is normally distributed). Then, we do not have evidence that both samples come from a normal distribution.

Figure 18 – Scatter Plot Between Integration per Day and Failed Integration Frequency



Source: The author (2023)

Since we cannot affirm both samples are normally distributed, we must opt for non-parametric tests to run the correlation test between the previous samples. We run the *Kendall* and *Spearman* correlation tests. Each correlation test results in the ρ-value and the correlation coefficient value (as explained in Section 4.1.1.3), and can be observed in Table 4.

Table 4 – Correlation Tests Results.

Correlation Test	**ρ-value**	**Correlation Coefficient**
Kendall	0.002041	-0.2730529
Spearman	0.002802	-0.3733331

Source: The author (2023)

The correlation tests present similar results. The ρ-value is less than 0.05, we reject the null hypothesis (correlation statistically equal to zero). This means that there is a relationship between these variables. Regarding the strength of this relationship, the correlation coefficient results indicate a weak correlation since their values approach zero (Kendall test = -0.2730529, Spearman test = -0.3733331). Besides that, the coefficient presents a negative result, meaning that this is a negative correlation (when one variable's value increases, the other's value decreases).

4.2.1.2 Developers with high visible integration frequency

In this subsection, we will group only the developers who integrated code using a visible integration strategy (the merge command) in more than 70% of the situations. Among the 61 developers from our sample, 25 have visible integration frequency ranging from 79% to 100%.

We ran the same study on this group, following the steps described before. Our results showed that we did not find evidence that our samples come from a normal distribution because of the ρ-value less than 0.05. Moreover, our correlation tests indicate no significant correlation between these frequencies. Since the ρ-value is higher than 0.05, we could not reject the null hypothesis (there is no significant correlation between the 2 variables). Hence, the relationship between these frequencies is not statistically significant.

4.2.1.3 Developers with high hidden integration frequency

In this subsection, we will group only the developers who integrated code using a hidden integration strategy (the Git commands: rebase, cherry-pick, squash) in more than 70% of the situations. Among the 61 developers from our sample, 32 have hidden integration frequency ranging from 73% to 100%.

We ran the same study on this group, following the steps described before. Our results showed that we did not find evidence that our samples come from a normal distribution because of the ρ-value less than 0.05. Moreover, our correlation tests indicate no significant correlation between these frequencies. Since the ρ-value is higher than 0.05, we could not reject the null hypothesis (there is no significant correlation between the 2 variables). Hence, the relationship between these frequencies is not statistically significant.

4.2.1.4 Results Interpretation

According to our analysis, there is a relationship between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts if we consider the complete study sample. This relationship is represented by a negative correlation coefficient, which indicates that when one variable's value increases, the other's value decreases. Converting to our study case, it could mean that when a developer integrates code with a higher frequency, the failed integration scenarios will be at a low rate (or vice versa). Our scatter plot (Figure 18) reinforces this result because we notice a decrease in the failed integration frequency (y-axis values less or equal to 20%) when the developer integrates code more frequently (x-axis values equal to or higher than 0.5).

Besides the negative correlation, the correlation coefficient indicates a weak relationship since its value approaches zero. In our study, it means that the events are not strongly connected, but when one event tends to increase, the other will tend to decrease.

Apart from the correlation analysis in the study sample, we decide to run the same analysis in two subgroups: developers with a high visible integration frequency and developers with a high hidden integration frequency. By running this analysis in these groups, we expect to compare the correlation and its strength between the study sample and a sample with a code integration strategy representing the majority (visible or hidden integration scenario). This way, we could analyze if a specific code integration strategy would strengthen this relationship and how one event would imply a change in the other. However, our results were negative in groups with the same code integration strategy. Then, we were not able to go deep into this comparison.

Interpreting the results from groups of developers that performed the same strategy of integrating code (visible or hidden), the outcomes were similar for both. The results indicate that there is no statistically significant relationship between the developer's integration frequency and the failed integration scenarios frequency. In other words, there is no significant correlation when a developer integrates code that the failed integration scenarios will reduce or increase.

Analyzing the correlation result from these two groups, we could conclude that no code integration strategy (when representing the majority in the sample) affects the relationship between the two events: the developer integrating code and the frequency of integration scenarios resulting in conflicts. However, our sample size from both groups was small. Correlations obtained with small samples are quite unreliable (SCHöNBRODT; PERUGINI, 2013).

The sample size is a threat to these groups in this study. We would have to increase the study sample to minimize this threat in future work. Doing so would give us more confidence when analyzing the correlation result. Until this future work, we cannot support the existence or non-existence of a statistically significant relationship between the developer's code integration frequency and the failed integration scenarios frequency in a sample that contains a specific code integration strategy. Thus, in our study, we cannot indicate if there is a code integration strategy that would have less impact on the failed integration scenarios frequency.

Although we cannot indicate a specific code integration strategy, we still can suggest that if a developer integrates code often, the failed code integration frequency will tend to decrease. From our study sample, we have evidence that the event of integrating code frequently at a high rate will indicate a decrease in the frequency of integration scenarios resulting in conflicts (because of the negative correlation result).

Result: According to our sample, there is a statistically significant relationship between the frequency of developers' code integration and the frequency of code integration scenarios resulting in conflicts. This relationship is represented by a negative correlation (the higher values of one event are associated with the lower values of the other). As the correlation result approaches zero, this relationship can be considered weak (events are not strongly connected). In addition, we run the same process for the developers with a high visible integration frequency and those with a high hidden integration frequency. The results for both groups were similar: there is no statistically significant relationship between the events analyzed in this study. Since both groups were represented by a small sample and correlations obtained with small samples are quite unreliable, we cannot support the existence or non-existence of this statistically significant relationship and we neither can indicate a code integration strategy that would have less impact on the failed integration scenarios frequency based on our research. Still, from our study sample result, we can suggest that if a developer integrates code often, the failed code integration frequency will tend to decrease.

4.3 DISCUSSION

As discussed in Chapter 2, understanding how the frequency of integrating code could impact merge conflicts has been a goal of previous studies. Previous studies have shown the frequency of integrating code as a factor that could cause merge conflicts and also as a way to reduce such conflicts (BIRD; ZIMMERMANN, 2012; ADAMS; MCINTOSH, 2016; MCKEE et al., 2017; DIAS; BORBA; BARRETO, 2020).

Each of these mentioned studies had their own goal and methodologies. Bird and Zimmerman (BIRD; ZIMMERMANN, 2012) conducted a survey to characterize how developers use branches in large industrial projects and the common problems they face. Adams and McIntosh (ADAMS; MCINTOSH, 2016) provided a checklist of release engineering decisions to avoid incorrect conclusions when mining software repository data. While that, McKee et al. (MCKEE et al., 2017) performed interviews and deployed a survey to understand how practitioners approach merge conflicts and their difficulties. And finally, Dias, Borba, and Barreto (DIAS; BORBA; BARRETO, 2020) aimed to understand how merge conflict occurrence is affected by technical and organizational factors.

These researchers either had the merge conflicts as the study's main goal, like these studies (MCKEE et al., 2017; DIAS; BORBA; BARRETO, 2020), or found the merge conflict as a factor impacting their research goal, as these studies (BIRD; ZIMMERMANN, 2012; ADAMS; MCINTOSH, 2016) did. Either way, these studies discovered an association between code integration frequency and its effect on merge conflicts. In resume, it indicates that a way to reduce merge conflicts is to integrate often. Each study with its goal but with a similar conclusion about code integration frequency and the merge conflicts rate.

Our research follows this same path. As we aim to analyze Git's private life and its merge conflicts, our results indicate a statistically significant relationship between the frequency of developers' code integration and the frequency of integration scenarios resulting in conflicts. Since this relationship has a negative correlation, one possible scenario is that when the frequency of developers' code integration is high, the frequency of failed integration scenarios will tend to have lower values. This result leads us to have a finding about code integration frequency and its conflicts following the same direction as the ones from these previous studies, even though our methodology differs.

In addition to our research reinforcing the findings of these previous studies, we complement studies based only on remote Git repository history. In our research, we collected integration scenarios that are not visible in the remote Git repository history, such as the rebase command. Therefore, our research complements and reinforces previous studies because we came to a similar conclusion even though analyzing more than one type of code integration scenario.

During our investigation, we decide to run the study process in three different groups. The first group contains the study sample, and the other two groups were divided by the Git integration command that the developer used the most. One group represents the developers with high visible integration frequency (merge command), and the final group contains the developers with high hidden integration frequency (rebase, cherry-pick, and squash).

Our expectation about running the study into these different groups is to investigate if the result would differ between them. In other words, verify if the relationship would be statistically significant and if its degree would change when grouping developers with the same strategy of integrating code (visible or hidden). Then, given the frequency of integrating code, we would analyze if a specific strategy for integrating code would contribute or not to the failed integration scenarios.

Based on our findings, only one group showed a statistically significant relationship between the frequency of developers integrating code and how frequently these scenarios result in conflicts: the study sample. According to our analysis, this correlation coefficient is negative (Kendall test = -0.27, Spearman test = -0.37), meaning that when one variable's value increases, the other's value tends to decrease. In our study case, it can mean that when a developer integrates code with a higher frequency, the failed integration scenarios will reduce (or vice versa).

Besides that, this correlation coefficient indicates a weak relationship because its value approaches zero. Still, this finding supports the results from previous works (BIRD; ZIMMERMANN, 2012; ADAMS; MCINTOSH, 2016; DIAS; BORBA; BARRETO, 2020). These events (code integration and its conflicts) are associated, and this discovery has been validated by different methodologies as described in this study and prior studies, for example. These studies and this research suggest that when a developer integrates code frequently, it tends to decrease the merge conflicts occurrences.

This suggestion is because low code integration frequency more likely increases contributions made in the meantime from other developers. The consequence is a greater chance of overlapping changes and merge conflicts occurrences. Thus, the frequency a developer integrates code is likely inversely proportional to the number of other developers' contributions he will get when he decides to incorporate his work.

Apart from this, when we analyze the results from the other groups (developers with high visible integration frequency and developers with high hidden integration frequency), none presented a statistically significant relationship between the two events: the developer integrating code and the frequency of integration scenarios resulting in conflicts. Then, we cannot support that there is a specific strategy for integrating code that would have less impact on the failed integration scenarios. Nevertheless, these results could have been affected by their small sample size (it will be discussed in Section 4.4).

Since correlations are often inaccurate in small samples (SCHöNBRODT; PERUGINI, 2013), further work would be necessary to increase this study sample. Then, we would have to rerun this analysis to verify if the same result remains. Considering this threat, there is a chance that a correlation exists in a group with these specific characteristics (developers with high visible integration frequency and developers with high hidden integration frequency). Until this future work, we are unsure if this relationship would exist and would be statistically significant. Because of this uncertainty, we cannot recommend a strategy for integrating code that would impact less the failed integration scenarios.

Based on our study sample, our results reinforce the findings in the literature. There is a correlation between the frequency of integrating code and the frequency of integration scenarios resulting in conflicts. Depending on its frequency, code integration can be considered a factor in mitigating or contributing to merge conflicts. Maintaining a high frequency of integrating code is a way to mitigate the risk of merge conflicts. Integrating code regularly minimizes this risk because it will reduce the time other developers could have made conflicting code changes.

Decreasing the merge conflicts rate is a common desire shared between the software industry, software practitioners, and researchers. Previous studies have shown that merge conflicts frequently occur and impair developers' productivity because it is a demanding and error-prone task (BIRD; ZIMMERMANN, 2012; BRUN et al., 2013; KASI; SARMA, 2013; SARMA; REDMILES; HOEK, 2012; ZIMMERMANN, 2007). As discussed, this research suggests the increase of code integration frequency as a way to reduce integration conflicts.

One way to support a high code integration frequency is to adopt the practice of continuous integration (CI). Continuous integration is a software development practice of automating the integration of code changes from multiple contributors into a single software project. With continuous integration, code changes are automatically built and tested.

Because of this automation, developers are dismissed from doing manual validations, which could improve team productivity. Besides that, CI allows the developer to discover and address bugs earlier due to frequent testing. Eventually, continuous integration will help the team deliver updates to their costumer faster and more frequently.

Since collaborative development is essential for the success of most software projects, developers working in parallel is inevitable. Therefore, we recommend that the software development teams work on shorter commit cycles. Applying this strategy, code integration would occur more frequently, and it could tend to decrease the merge conflicts and, consequently, their negative side effects. This recommendation extends to any software practitioner.

4.4 THREATS TO VALIDITY

This section discusses the potential threats that might have affected or influenced our evaluation.

Construct Validity

The greater goal of this research is to investigate Git's private life and its merge conflicts. In this chapter, we analyze the frequency of developers' code integration and how frequently these scenarios result in conflicts. This analysis aims to verify the correlation between these frequencies.

In order to explore Git's private life, as these frequencies and this correlation, we must investigate the developer's local history. As discussed before, one way to investigate the developer's local history is by the reference logs file (or reflog file) maintained by Git. This file contains information about the Git commands performed that changed branch references.

The reflog file is created when the developer clones or starts a new repository using Git. The reflog file is automatically saved and updated by Git as the user performs Git commands. The reflog is strictly local, then cannot be pushed or pulled from a remote repository.

However, the reflog file gets deleted when the developer deletes the project from the local machine. Even if the developer clones again the previously deleted project, the reflog file's old content cannot be recovered. Besides that, Git limits the size of the reflog file. Meaning that it erases the reflog content at some point so that future Git actions can be saved.

All of these situations we've described mean that we might not be analyzing the complete Git history in the developers' private repositories. This, nevertheless, does not compromise the results of our correlation test from our sample because it showed statistical significance in the relationship between developers integrating code and the frequency of integration scenarios resulting in conflicts (although our sample may be considering only part of the Git history).

Regarding the statistical test, there is a risk related to rejecting the correlation test's null hypothesis. To minimize this threat, we set a lower threshold for statistical significance. This threshold is the maximum risk of making a false positive conclusion. In our study, we set the significant level to 0.05 (or 5%). That means our results must have a 5% or lower chance of occurring under the null hypothesis to be considered statistically significant.

Internal Validity

Our analysis strongly depends on the reflog files and how we identified the Git commands from them. We created a script that reads these files, identifies integration scenarios, and presents metrics from the occurrences. From these metrics, we were able to achieve this study.

A potential threat is our approach to identifying hidden integration scenarios. The script was created based on our manual analysis while placing the syntactic patterns of the Git commands in the Git log files. As a result, the script may not count the exact number of occurrences of the Git commands that hide integration. The reason can be a specific scenario that does not appear in the log file or a new pattern of a particular Git command that we did not implement in the script. We made manual analysis in different log files before running the study to reduce this threat.

From our correlation test results, our findings did not show statistical significance in the groups with high specific integration frequency (visible or hidden). We failed to reject our null hypothesis (there is no significant correlation between the 2 variables). However, this result is not quite the same as accepting the null hypothesis because hypothesis testing can only tell you whether to reject the null hypothesis. In this case, we failed to conclude the significance statically of this relationship.

Because of that, our study may not have had enough statistical power to detect an effect of a certain size. Many factors can determine statistical power. One is the sample size: a larger sample reduces sampling error and increases power.

Since these two groups had at most 32 developers and both failed to reject the null hypothesis in the correlation test, we could consider their size a potential threat in our findings. A small sample size may make it difficult to determine if a particular outcome is a true finding. Thus, small sample sizes decrease statistical power. For this reason, we could increase the sample size for future work.

External Validity

Regarding this type of threat, we must discuss our sample and how selection bias could have impacted it. As detailed before, we use the same sample from the last chapter. Then, this analysis shares the same possibility of this threat as discussed in the previous chapter.

We collect our data via a website. This website was available for voluntary submission. This aspect could lead our study to be impacted by self-selection bias, which would only represent a particular subset of people who were comfortable participating. However, more than half of our sample came from developers teams from software companies we contacted, and some of these developers had the option to self decide if they would participate because their manager asked them to participate. Therefore, we reduce the threat of self-selection bias.

Regarding representation, we considered our sample diverse because we collected most of our sample from 4 different software companies. Moreover, we have evidence from the last chapter of their different characteristics related to the Git integration strategy. Two companies guide their collaborators to use commands that could hide integration, and the other two present the opposite, the collaborators using the merge command.

Even though we have a diverse sample, our sample might not be large and representative enough to generalize our results. For future work, we could work on increasing the sample size.

5 RELATED WORK

In a software development environment, concurrent work is inevitable. Developers work on tasks independently of each other. Once the work is done, changes must be integrated and shared with other developers. One way to do it is by applying the merge command provided by Git.

As explained in this research, integration scenarios via merge command are considered visible integrations since it creates a new (merge) commit in the history of the remote shared repository. However, there are other ways to integrate code using Git. These integration scenarios are often omitted from history because there are Git commands that can rewrite Git commit history, leaving no trace in the remote shared repository history. We named this type of integration a hidden integration scenario.

Previous studies provide evidence about collaborative software development. As mentioned earlier, those studies focus on analyzing merge conflicts and their resolution. In addition, these studies based their research on collecting integration scenarios via merge command on remote shared repositories such as the ones available in GitHub.

For example, Ghiotto et al. (GHIOTTO et al., 2020) focus on analyzing merge conflicts of 2731 open-source Java projects resulting in recommendations for future merge techniques that could help resolve certain types of conflicts. Likewise, other studies (BRUN et al., 2011; KASI; SARMA, 2013; NGUYEN; IGNAT, 2017; AHMED et al., 2017; ACCIOLY; BORBA; CAVALCANTI, 2018a; CAVALCANTI; ACCIOLY; BORBA, 2015; SARMA; REDMILES; HOEK, 2012; ZIMMERMANN, 2007) also analyzed large open-source projects with a similar purpose.

In contrast, our dissertation focuses on examining developer local repository logs instead of remote shared repositories. This way of mining Git allows us to go deeper into various integration scenarios. Compared to these works, our research explains how to identify ways of integrating code that does not leave traces or can even erase traces of code integration in the history of remote project repositories. These hidden integration scenarios can be possible by performing the following Git commands: rebase, cherry-pick, squash, and stash-apply (as detailed before, we identify rebase, cherry-pick, and squash commands in this research).

According to our sample, *hidden* code integrations are much more frequent (approximately 6 times) than potentially *visible* code integrations. This difference is alarming for studies that analyzed projects where developers rely on the rebase command to integrate code, for example. Hence, this result shows that previous studies might be missing integration data from their analysis.

With our findings, we show evidence that studies that focus only on remote project history might be missing integration conflict data by not considering the information in local repositories. We have observed conflicts at a lower bound rate of 37% resulting from `git merge`, revealing developers that often have to deal locally with conflicts when invoking `git merge` in their private repositories.

Regarding the visible integration scenarios, our result of the merge conflict rate is bigger if we compare it with previous studies. Accioly, Borba and Cavalcanti (ACCIOLY; BORBA; CAVALCANTI, 2018a) found a merge conflict rate of 9.38%, meanwhile, Brun et al. (BRUN et al., 2011) and Kasi and Sarma (KASI; SARMA, 2013) found that textual conflicts occurred in an average of 15% of all merge scenarios. Moreover, our result is similar to Zimmermann's (ZIMMERMANN, 2007), with detected rates of 23% to 47%. Compared to these studies, our results could be up to 3 times bigger, indicating a considerable difference. This difference reinforces the importance of not just focusing on the public repositories.

Regarding the hidden integration scenarios, we have evidence that this type of scenario also faces merge conflicts. Such conflicts are also missed by analyses focusing only on remote repositories, as rebase and cherry-pick scenarios are not visible with only remote history information. According to our sample, our results show that the hidden integration scenario ended up in conflict with a lower bound rate of 10%. This evidence indicates that hidden integration scenarios lead to conflicts, although at a lower rate than can be observed from the visible scenarios in our sample. Then, it reinforces the need to consider the *public* and the *private* life of merge conflicts.

In contrast, Ji et al. (JI et al., 2020) investigate the hidden integration scenario performed by the rebase command. This study examines how developers rebase their working branches in the pull requests. They collected 82 Java repositories from GitHub and identified 51,183 rebase scenarios from the pull requests of these repositories. It was possible to identify the rebase scenario in the pull request by using APIs of GitHub, especially events that identify a force-push to the head branch of the pull request (BaseRefForce-PushedEvents and HeadRefForcePushedEvents). They utilize the GitHub APIs to collect all the closed and merged pull requests and the rebase scenarios events.

Even though Ji et al. and previous studies are based on remote shared repositories, the way chosen to identify integration scenarios differs. Ji et al. research can identify hidden integration scenarios by examining pull requests. Otherwise, if it examines only the remote repository history, it would be limited to analyzing only the merge command, as in previous studies.

Moreover, Ji et al.'s study was the only one we found so far that analyzed hidden integration scenarios. In comparison with our research, our work goes further by identifying other hidden integration scenarios beyond the rebase: cherry-pick and squash. Since we collect our data through the developer's local history, we identified integration scenarios and conflicts that occur locally and are resolved before the developers synchronize their contributions to shared remote repositories, so it can see no trace of them when analyzing only remote repositories. Therefore, our research identifies more integration scenarios by including local repositories than Ji et al.'s study investigating rebase scenarios in pull requests.

According to this study, rebasing is widely used in pull requests because it can relieve the burden of reviewing changes and keep the commit history clean. Their result shows that conflicts arise in 24.3%-26.2% of rebases. However, they claim no significant difference between the possibilities of merge conflicts arising in rebases and merges.

This result diverges from ours because merge conflict in the visible integration scenario (lower bound of 37% / $\uparrow$37%) is more frequent than the conflicts in the hidden integration scenarios (lower bound of 10% / $\uparrow$10%) in our sample. Thus, our results complement theirs since conflicts that developers resolve locally may not appear in the pull request history.

Related to the influence of choosing between different Git integration commands, we concluded that company guidance can be an essential factor of influence because it can lead or prohibit the usage of a specific command. The other factor was personal Git experience, making less experienced developers choose easier commands, such as merge, instead of rebase. As we aim to investigate why developers choose a particular integration command or strategy in their work routine, Ji et al. investigate when and why developers decide to rebase branches in pull requests with a somewhat different focus.

Overall, our research focuses on local repositories, and we analyze the visible and hidden integration commands and their characteristics: their frequency and how often they failed, factors that can influence developers' choice of Git integration command. Besides that, we also analyzed the relationship between the frequency of developers' code integration and how frequently these scenarios result in conflicts. In contrast, Ji et al. focus on examining rebase scenarios in pull requests. Even though we have different approaches and goals, we could be one of the first studies to research other ways to integrate code besides merge command and enhance their importance to provide comprehensive insights on software merging.

Regarding the analysis of code integration frequency and its association with code integration conflicts, this dissertation proposes examining whether a relationship exists between developers' code integration frequency and how frequently these scenarios result in conflicts in private repositories. Comparing to the literature, our result is similar to findings from previous studies. There is evidence regarding the frequency of integrating code as a factor that could cause merge conflicts and also as a way to reduce such conflicts (NGUYEN; IGNAT, 2017; BIRD; ZIMMERMANN, 2012; ADAMS; MCINTOSH, 2016; MCKEE et al., 2017; DIAS; BORBA; BARRETO, 2020).

Nguyen and Ignat (NGUYEN; IGNAT, 2017) analyzed four open source projects to understand the relationship between the integration rate (i.e., number of concurrently modified files over all modified files) and conflict rate (i.e., number of files with unresolved conflicts over the ones concurrently modified). The evidence found was the lower the integration rate, the higher the conflict rate. Meanwhile, McKee et al. (MCKEE et al., 2017) performed 10 interviews and deployed a survey with 162 other developers to understand how practitioners approach merge conflicts and their difficulties. Among their findings, they informally claim that practitioners should attempt to commit often to prevent and alleviate the severity of merge conflicts. In addition, Dias, Borba, and Barreto (DIAS; BORBA; BARRETO, 2020) aimed to understand how merge conflict occurrence is affected by technical and organizational factors. They showed the higher the contribution duration (geometric mean of the number of days between the common ancestor and the last commit in each contribution), the higher the chances of merge conflict occurrence.

In this research, we evaluate the frequencies of integration scenarios from the local history of each developer and apply statistical tests to measure the correlation between these two frequencies (code integration frequency and failed integration frequency). Our results show a statistically significant relationship between developers' code integration frequency and code integration scenarios resulting in conflicts. This relationship is represented by a negative correlation coefficient (-0.373), meaning, in our study case, that when a developer integrates code with a higher frequency, the failed integration scenarios will reduce (or vice versa). Then, from our study sample result, we can suggest that if a developer integrates code often, the failed code integration frequency will tend to decrease. In resume, it indicates that a way to reduce merge conflicts is to integrate often.

All these studies dedicated their goal to analyzing merge conflicts. Although we share a similar goal with these studies, our methodology differs. Their methods involved analyzing GitHub repositories or conducting interviews and surveys, while ours examined the private developer repository. Besides the method, our sample differs from previous studies because we can identify more than one type of integration scenario. Furthermore, we present a similar conclusion about code integration frequency and its conflict rate, even with the differences. Therefore, this dissertation complements previous studies because we present similar results, even having a more diversified sample of integration scenarios. Thus, we suggest that other types of integration be considered in the sample when analyzing merge conflicts.

In contrast, Leßenich et al. (LEßENICH et al., 2018) proposed seven indicators to identify merge conflicts, for example, based on the number of commits in a branch, the number of commits in a time interval, and the number of files changed in parallel. These indicators were drawn from survey responses targeted at developers to understand their experience of what factors cause merge conflicts. However, none of the seven indicators is related to the frequency of integrating code. After an empirical study on open-source projects, the result is that none of the seven indicators have a strong correlation with the number of failed merges.

Compared with our research, our work complements previous studies that did not analyze the frequency of integrating code as a factor that impacts the merge conflict rate. Since our research and previous studies have found that integrating code often could reduce merge conflicts, we suggest that studies that explore merge conflict factors include code integration frequency in their research context.

Furthermore, no previous work has shed as much light on the private life of Git and its merge conflicts as ours. The reason is that no previous work has analyzed developers' private repositories and used them to identify integration scenarios that cannot be identified by analyzing only the remote repository history. Moreover, our main contribution is to show the importance of including information about the developer's local repository to improve future research on code integration and its conflicts.

6 CONCLUSIONS

In collaborative development environments, developers work simultaneously on several tasks independently. As a result, they may encounter conflicts when integrating code with the main repository. Previous studies indicate that these conflicts occur frequently and can negatively impact developer productivity. To minimize this negative impact, previous studies recommend integrating often to reduce merge conflicts.

In this dissertation, we present research focusing on actions performed in the private repositories of the developers. In our research, we perform two main studies. The first study aims to analyze Git commands history data extracted from the local repositories, mainly the integration scenarios. The second study focuses on discovering if there is a correlation between the frequency of developers' code integration and how frequently these scenarios result in conflicts.

In the first study, we report a quantitative and a qualitative study to measure the frequency of the integration scenarios in developers' local repositories. The main difference from previous merge studies was the detection of hidden integration scenarios, that is, scenarios that do not appear in the shared remote repository history. In order to do so, we collected a total of 95 logs from 61 different developers and analyzed the actions recorded in their local repository log files. We implemented a script that counts and computes the metrics used to answer our research questions from Chapter 3. Additionally, we conducted semi-structured interviews with nine developers to learn more about their habits while using Git to merge code.

Our results indicate that *hidden* code integrations are much more frequent (approximately 6 times) than potentially *visible* code integrations (performed by git merge and that might reach remote repositories). Additionally, we observed conflict rates of up to 37% resulting from `git merge`, revealing developers that often have to locally deal with conflicts when invoking `git merge` in their private repositories. Since many merge conflicts are resolved locally before the developers synchronize their contributions to shared remote repositories, we can see no trace of them when analyzing only remote repositories. The hidden integration scenarios also lead to conflicts, with a lower bound rate of 10% in one private repository in our sample. Such conflicts are also missed by analyses that focus only on remote repositories. In resume, these results bring evidence that studies that focus only on GitHub project history might be missing integration conflict data by not considering the information in local repositories, reinforcing the need to consider both the *public* and the *private* life of merge conflicts.

Regarding the factors that could influence the choice among the ways of integrating code, we find that Git experience and company guidance can affect which Git integration command a developer chooses. Our sample of interviewers shows that developers tend to use the merge command when the company has no explicit guidance on how to use Git or because developers don't have experience in other Git commands. With our findings, the software industry can reflect on the different integration scenarios' impact and create a guideline with the best fit.

In the second study of this dissertation, we focus on evaluating the code integration frequency and how it is associated with the failed integration frequency in private repositories. Our study differs from previous ones by analyzing if this relationship exists when involving visible and hidden integration scenarios. To assess this, we apply statistical tests to measure the correlation between these two frequencies (code integration frequency and failed integration frequency).

Since we have data on visible and hidden integration, we analyze the correlation by considering the type of integration scenario. This way, we can examine if the result would differ between the strategies of integrating code. So, we consider 3 groups: all the developers' participants of our sample (which contains visible and hidden integration scenarios), the developers with high visible integration frequency, and the ones with high hidden integration frequency. For each group, we run statistical tests to verify the correlation.

According to our results, when running the study in the first group (all developers), the correlation coefficient resulted in -0.37 when running the Spearman test. Thus, there is a statistically significant relationship between developers' code integration frequency and code integration scenarios resulting in conflicts. This relationship is represented by a negative correlation (the higher values of one event are associated with the lower values of the other), and it can be considered weak since the correlation approaches zero. In our study case, it can mean that when a developer integrates code with a higher frequency, the failed integration scenarios will reduce (or vice versa).

Based on our study sample, this result reinforces the findings in the literature because we came to a similar conclusion about the association between code integration frequency and its effect on merge conflicts: a way to reduce merge conflicts is to integrate often. In addition, our results complement previous studies because we show evidence of this relationship even though analyzing more than one type of code integration scenario and mining a different repository (the private ones).

However, this result did not reflect when running the study process for the developers with a high visible integration frequency and those with a high hidden integration frequency. The results for both groups were similar: there is no statistically significant relationship between the events analyzed in this study. Yet, both groups were represented by a small sample, and correlations obtained with small samples are quite unreliable (SCHöNBRODT; PERUGINI, 2013). Still, from our research sample result, we can suggest that if a developer integrates code often, the failed code integration frequency will tend to decrease.

6.1 FUTURE WORK

The main goal of this dissertation is to shed some light on the private life of merge conflicts and how they relate to the code integration commands that generate them. We investigate and analyze the many forms of integrating code using Git in the developer's local repository. Besides that, we seek to discover if there is a correlation between the frequency of developers' code integration and how frequently these scenarios result in conflicts.

In general, we could not generalize our results since our sample might not be large and representative enough. One example is when we interpret the correlation result from groups of developers that performed the same strategy of integrating code (visible or hidden), the outcomes were similar. The results for these groups indicate no significant correlation when a developer integrates code that the failed integration scenarios will reduce or increase.

The sample size could have impacted this result when analyzing these groups. In resume, our study may not have had enough statistical power to detect an effect of a certain size. Many factors can determine statistical power. One is the sample size: a larger sample reduces sampling error and increases power. A small sample size may make it difficult to determine if a particular outcome is a true finding. Therefore, we could increase the sample size as an improvement for future work. To do this, we can focus on open-source companies. This would minimize the risk of companies refusing to share their log files because of any confidentiality clause in the employment contract.

During our research, we could not identify the command stash-apply by focusing only on the reference logs from the developers. As a perspective for future work, we can discover a way to identify this command in local repositories. Then, we could complement our findings with this new hidden integration scenario.

Another perspective for future work is to perform a study about merge conflicts involving the shared remote repository history with the reflog files from the developers' participants in the analyzed repositories. This future work would enable us to explore how the hidden integration scenario may contribute to integration conflicts in the main repository.

Regarding the factors that may have influenced a developer to adopt a particular integration strategy, company guidance is one of them. Depending on how strictly the company guidance is spread among the collaborators, the developers will not have the autonomy to self-decide which code integration strategy to follow. For future work, it would be interesting to investigate how the companies decided on the Git integration strategy for all the employees.

REFERENCES

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source java projects (journal-first abstract). In: _____. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. p. 955. ISBN 9781450359375. Disponível em: <https://doi.org/10.1145/3238147.3241983>.

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source java projects (journal-first abstract). In: _____. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. p. 955. ISBN 9781450359375. Disponível em: <https://doi.org/10.1145/3238147.3241983>.

ACCIOLY, P.; BORBA, P.; SILVA, L.; CAVALCANTI, G. Analyzing conflict predictors in open-source java projects. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2018. (MSR '18), p. 576–586. ISBN 9781450357166. Disponível em: <https://doi.org/10.1145/3196398.3196437>.

ADAMS, B.; MCINTOSH, S. Modern release engineering in a nutshell – why researchers should care. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 5, p. 78–90.

aES, M. L. G.; SILVA, A. R. Improving early detection of software merge conflicts. In: *Proceedings of the 34th International Conference on Software Engineering*. [S.l.]: IEEE Press, 2012. (ICSE '12), p. 342–352. ISBN 9781467310673.

AHMED, I.; BRINDESCU, C.; MANNAN, U. A.; JENSEN, C.; SARMA, A. An empirical examination of the relationship between code smells and merge conflicts. In: *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017. (ESEM '17), p. 58–67. ISBN 9781509040391. Disponível em: <https://doi.org/10.1109/ESEM.2017.12>.

ALLEN, M.; POGGIALI, D.; WHITAKER, K.; MARSHALL, T. R.; KIEVIT, R. A. Raincloud plots: a multi-platform tool for robust data visualization. *Wellcome open research*, The Wellcome Trust, v. 4, 2019.

APEL, S.; LEßENICH, O.; LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012. p. 120–129.

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KäSTNER, C. Semistructured merge. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. [S.l.]: ACM Press, 2011.

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2012. (FSE '12). ISBN 9781450316149. Disponível em: <https://doi.org/10.1145/2393596.2393648>.

BRUN, Y.; HOLMES, R.; ERNST, M.; NOTKIN, D. Early detection of collaboration conflicts and risks. *Software Engineering, IEEE Transactions on*, v. 39, p. 1358–1375, 10 2013.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 168–178. ISBN 9781450304436. Disponível em: <https://doi.org/10.1145/2025113.2025139>.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2015. p. 1–10.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. OOPSLA, oct 2017. Disponível em: <https://doi.org/10.1145/3133883>.

CAVALCANTI, G.; BORBA, P.; SEIBT, G.; APEL, S. The impact of structure on software merging: Semistructured versus structured merge. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2019. p. 1002–1013.

CHACON, S.; STRAUB, B. *Pro Git*. 2nd edition. ed. [S.l.]: Apress, 2014.

CLEMENTINO, J.; BORBA, P.; CAVALCANTI, G. Textual merge based on language-specific syntactic separators. In: ______. *Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. p. 243–252. ISBN 9781450390613. Disponível em: <https://doi.org/10.1145/3474624.3474646>.

DIAS, K.; BORBA, P.; BARRETO, M. Understanding predictive factors for merge conflicts. *Information and Software Technology*, v. 121, p. 106256, 2020. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S095058492030001X>.

GHIOTTO, G.; MURTA, L.; BARROS, M.; HOEK, A. van der. On the nature of merge conflicts: A study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, v. 46, n. 8, p. 892–915, 2020.

HOEK, A. van der; SARMA, A. Palantir: enhancing configuration management systems with workspace awareness to detect and resolve emerging conflicts. In: . [S.l.: s.n.], 2008.

JI, T.; CHEN, L.; YI, X.; MAO, X. Understanding merge conflicts and resolutions in git rebases. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.: s.n.], 2020. p. 70–80.

KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 2013 International Conference on Software Engineering*. [S.l.]: IEEE Press, 2013. (ICSE '13), p. 732–741. ISBN 9781467330763.

LEßENICH, O.; SIEGMUND, J.; APEL, S.; KäSTNER, C.; HUNSEN, C. Indicators for merge conflicts in the wild: Survey and empirical study. *Automated Software Engg.*, Kluwer Academic Publishers, USA, v. 25, n. 2, p. 279–313, jun 2018. ISSN 0928-8910. Disponível em: <https://doi.org/10.1007/s10515-017-0227-0>.

MAHMOOD, W.; CHAGAMA, M.; BERGER, T.; HEBIG, R. Causes of merge conflicts: a case study of elasticsearch. In: . [S.l.: s.n.], 2020. p. 1–9.

MCKEE, S.; NELSON, N.; SARMA, A.; DIG, D. Software practitioner perspectives on merge conflicts and resolutions. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 467–478.

MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, v. 28, n. 5, p. 449–462, 2002.

NGUYEN, H. L.; IGNAT, C.-L. Parallelism and conflicting changes in Git version control systems. In: *IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems*. Portland, Oregon, United States: [s.n.], 2017. Disponível em: <https://hal.inria.fr/hal-01588482>.

NGUYEN, H. L.; IGNAT, C.-L. An analysis of merge conflicts and resolutions in git-based open source projects. *Computer Supported Cooperative Work (CSCW)*, v. 27, 12 2018.

NISHIMURA, Y.; MARUYAMA, K. Supporting merge conflict resolution by using fine-grained code change history. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 1, p. 661–664.

PERRY, D.; SIY, H.; VOTTA, L. Parallel changes in large scale software development: an observational case study. In: *Proceedings of the 20th International Conference on Software Engineering*. [S.l.: s.n.], 1998. p. 251–260.

PUTH, M.-T.; NEUHäUSER, M.; RUXTON, G. D. Effective use of spearman's and kendall's correlation coefficients for association between two measured traits. *Animal Behaviour*, v. 102, p. 77–84, 2015. ISSN 0003-3472. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0003347215000196>.

SARMA, A.; REDMILES, D. F.; HOEK, A. van der. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, v. 38, n. 4, p. 889–908, 2012.

SCHöNBRODT, F. D.; PERUGINI, M. At what sample size do correlations stabilize? *Journal of Research in Personality*, v. 47, n. 5, p. 609–612, 2013. ISSN 0092-6566. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0092656613000858>.

SEDANO, T.; RALPH, P.; PéRAIRE, C. Software development waste. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2017. p. 130–140.

SILVA, L. D.; BORBA, P.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J. Detecting semantic conflicts via automated behavior change detection. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2020. p. 174–184.

SILVA, L. D.; BORBA, P.; PIRES, A. Build conflicts in the wild. *J. Softw. Evol. Process*, John Wiley amp; Sons, Inc., USA, v. 34, n. 4, apr 2022. ISSN 2047-7473. Disponível em: <https://doi.org/10.1002/smr.2441>.

TAVARES, A.; BORBA, P.; CAVALCANTI, G.; SOARES, S. Semistructured merge in javascript systems. In: . [S.l.: s.n.], 2019. p. 1014–1025.

ZIMMERMANN, T. Mining workspace updates in cvs. In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. [S.l.: s.n.], 2007. p. 11–11.

APPENDIX A – ONLINE RESOURCES

The author provides this appendix to give readers information about their work. The following list contains the study's resources and their URLs so readers can access this information. The links are listed in the order they appear in the document.

- Study Setup in GitHub: <https://tinyurl.com/privatelifemergeconflictsgh>
- Website used to collect the study sample: <https://tinyurl.com/privatelifemergeconflictsws>
- SBES 2022 - The Private Life of Merge Conflicts: <https://dl.acm.org/doi/10.1145/3555228.3555240>

APPENDIX B – INTERVIEW SCRIPT

In this study, we conducted semi-structured interviews. We created an interview script containing 18 questions to explore the developer experience using Git:

1. How long as a developer on the project?
2. Do you have experience or know the rebase command?
3. When using it, did you deal with any conflicts?
4. Do you have experience or know the squash command?
5. When using it, did you deal with any conflicts?
6. Do you have experience or know the cherry-pick command?
7. When using it, did you deal with any conflicts?
8. Do you use stash-apply?
9. In what situations did you use this command?
10. Do many integration conflicts happen during development?
11. Are these conflicts resolved locally in your repository before integrating the code with the main repository?
12. In your opinion, which command usually results in code integration conflicts?
13. In your project, is there a standard adopted by all members, or is it a personal choice to use git commands?
14. Ever had to deal with conflicts resulting from the merge command?
15. How did you solve them?
16. How do you usually resolve merge conflicts?
17. Do you often consult with the team to resolve merge conflicts?
18. Is there any standard for the workflow?