Challenges of Resolving Merge Conflicts: A Mining and Survey Study

Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel

Abstract—In collaborative software development, merge conflicts arise when developers integrate concurrent code changes. Practitioners seek to minimize the number of merge conflicts because resolving them is difficult, time consuming, and often an error-prone task. Despite a substantial number of studies investigating merge conflicts, the challenges in *merge conflict resolution* are not well understood. Our goal is to investigate which factors make merge conflicts longer to resolve in practice. To this end, we performed a two-phase study. First, we analyzed 66 projects containing around 81 thousand merge scenarios, involving 2 million files and over 10 million chunks. For this analysis, we use rank correlation, principal component analysis, multiple regression model, and effect-size analysis to investigate which independent variables (e.g., number of conflicting chunks and files) mostly influence our dependent variable (i.e., time to merge). We found that the *number of chunks*, *lines of code*, *conflicting chunks*, *developers involved*, *conflicting lines of code*, *conflicting files*, and the *complexity of the conflicting code* influence the merge conflict resolution time. Second, we surveyed 140 developers from our subject projects aiming at cross-validating our results from the first phase of our study. As main results, (i) we found that committing small chunks makes merge conflict resolution faster when leaving other independent variables untouched, (ii) we found evidence that merge scenario characteristics (e.g., the number of lines of code or chunks changed in the merge scenario) are stronger correlated with our dependent variable than merge conflict characteristics (e.g., the number of lines of code or chunks in conflict), (iii) we devise a taxonomy of four types of challenges in merge conflict resolution, and (iv) we observed that the inherent dependencies among conflicting and non-conflicting code is one of the main factors influencing the merge conflict resolution time.

Index Terms—Merge Conflict Resolution, Collaborative Software Development, Three-way Merge

1 Introduction

Version control systems help developers to manage code changes over time by tracking all code contributions, especially when involving collaborations of multiple developers [66]. This allows developers to address different programming tasks (e.g., bug fixing and adding new features) simultaneously without losing changes. After fulfilling their tasks, developers can merge their changes to the main repository. A *merge scenario*, also called three-way merge, includes the whole timeline of creating a project branch, committing changes independently to the project main branch, and creating a merge commit [28], [39].

Simultaneous code changes may introduce problems of their own during integration, often manifesting as merge conflicts. A *merge conflict* occurs during the integration of changes made by one or more developers that occurred in the same chunk of code. In Figure 1, we present an exemplary merge scenario resulting in three merge conflicts. While DevC and DevD created the *source* branch adding a new feature, DevA and DevB refactored the same two files in the *target* branch. Practitioners and researchers seek to minimize the number of merge conflicts, because resolving them is difficult, time-consuming, and often an error-prone task [39], [47].

- *G. Vale is with the Department of Computer Science, Saarland University, Germany. E-mail: vale@cs.uni-saarland.de*
- *C. Hunsen is with the Department of Computer Science, University of Passau, Germany. E-mail: hunsen@fim.uni-passau.de*
- *E. Figueiredo is with the Department of Computer Science, Federal University of Minas Gerais, Brazil. E-mail: figueiredo@dcc.ufmg.br*
- *S. Apel is with the Department of Computer Science, Saarland University, Germany. E-mail: apel@cs.uni-saarland.de*

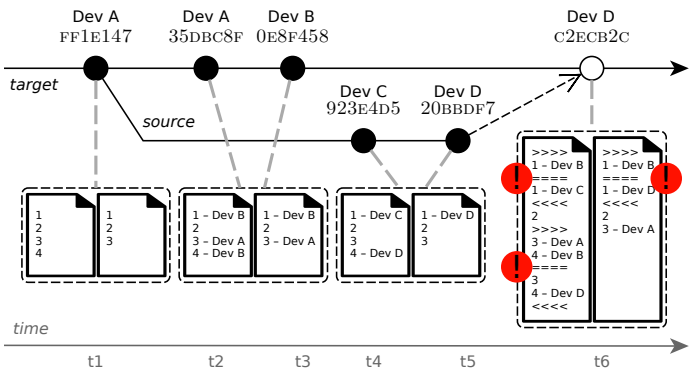

Fig. 1. Example for a merge scenario with conflicts. Four developers contributed to two files on the branches *target* and *source*, resulting in three merge conflicts.

Previous studies related to merge conflicts address: (i) *merge strategies* (e.g., structured [5] or semi-structured [6]), (ii) *prediction strategies* (e.g., continuous integration [29] and speculative merging [12]), (iii) *awareness tools* (e.g., CollabVS [18], Palantír [53], Cassandra [38], and FastDash [7]), (iv) *understanding the nature of merge conflicts* (e.g., identifying the types of code changes that lead to conflicts) [1], [24], [39], [51], and (v) *strategies for merge conflict resolution* [11], [35], [45], [49].

Despite the number of studies investigating merge conflicts, the understanding of challenges and strategies on the *resolution* of merge conflicts is limited [49]. Previous studies have asked developers about barriers and strategies when

resolving merge conflicts [11], [45], [49], empirically investigated developers' choice when resolving merge conflicts [24], or empirically investigated merge conflict resolutions in Git rebases [35]. A study showing which factors make merge conflicts longer to resolve is still missing. Such a study can guide practitioners to avoid the creation of time-consuming conflicts, to better coordinate their tasks, and to avoid delaying core-tasks on the project life-cycle (e.g., developing new features and fixing bugs). So far, there is no reliable knowledge on the factors that make merge conflict resolution longer in practice. While previous studies provide an initial understanding of merge conflict resolution, an empirical study investigating factors that increase merge conflict resolution time in practice may not only confirm and add nuances to previous findings but also pin down the most impacting and recurring factors. These factors together with the knowledge acquired from previous studies may either serve as best practices for developers saving time on merge conflict resolution or as guidelines for tool builders to better support practitioners. At the same time, our results outline opportunities for researchers to improve the state of the art of merge conflicts. Our study is guided on the overarching research question:

RQ: Which factors do make merge conflicts longer to resolve in practice?

To answer this research question, we conduct a two-phase study. First, we rebuild and extract information from all merge scenarios of 66 GitHub projects, selected based on their popularity. Inspired by previous work [24], [39], [49], we extracted 11 variables for each subject merge scenario. These variables include the *time* (in seconds) to resolve the merge conflict, *measures directly related to merge conflicts*, such as the number of conflicting chunks (*#ConfChunks*), the number of conflicting files (*#ConfFiles*), and the complexity of code in conflict (*CodeComplexity*), and *measures indirectly related to merge conflicts*, such as the number of developers involved (*#Devs*), the number of lines of code (*#LoC*), and the number of chunks (*#Chunks*) of the merge scenario. We group the independent variables into: *directly* and *indirectly* related to merge conflicts, since we suspect that merge conflict resolution depends not only on conflicting code but also on changes not in conflict. In this phase, we performed three main analyses: (1) a *correlation analysis* for each pair of the investigated variables (using correlation matrix and principal component analysis), (2) a *multiple regression model* analysis, and (3) an *effect-size analysis* using Cohen's f^2 measuring the impact of independent variables on our dependent variable.

Second, we conducted a survey with 140 developers from subject projects to triangulate our results and provide a broader understanding of the challenges on merge conflict resolution. For short, we asked developers: (1) to describe how they estimate how hard/time-consuming a merge conflict is to be resolved. Therewith, we checked if our independent variables are in line with measures used in practice, (2) a few statements to understand their processes on merging their contributions and resolving conflicts. The main goals of this analysis were to minimize threats to validity of our dependent variable and to cross-validate our results, and (3) to share their experiences when dealing with merge conflicts to help us understand their challenges and needs.

Summarizing our results, the correlation analysis indicates that measures *indirectly* related to merge conflicts (i.e., measures related to the merge scenario changes) are stronger correlated with merge conflict resolution time than measures *directly* related to merge conflicts (i.e., merge conflict characteristics). The regression model analysis reveals that *#LoC*, *#ConfChunks*, *#Devs*, *#ConfFiles*, *#ConfLoC*, and *#Files* have a positive correlation with merge conflict resolution time. Surprisingly, *#Chunks* and *CodeComplexity* show a negative correlation with merge conflict resolution time. In the effect-size analysis, we found that *#Chunks* has a medium effect on merge conflict resolution time, whereas *#Devs*, *#LoC*, *#ConfChunks*, and *CodeComplexity* only have a small effect. Cross-validating our results, survey participants mentioned 25 measures used to quantify how hard/time-consuming is the resolution of merge conflicts. Measures indirectly related to merge conflicts are among the most cited by them. In addition, they reported that they often merge their changes right after finishing addressing an issue, resolve conflicts right after they occur, and usually look at changes not in conflict to resolve merge conflicts. These results increase internal validity on the variables used in our empirical study. Related to developers' experience dealing with merge conflicts, survey participants pointed out four major challenges on merge conflict resolution: *lack of coordination*, *lack of tool support*, *flaws in the system architecture*, and *lack of testing suite or pipeline for continuous integration*. Indirectly, these challenges bring explanations of why some relatively simple merge conflicts (in terms of the subject variables in our empirical study) took a while to be resolved.

Aiming at deeper explanations for our results, we analyzed specific factors individually and triangulated our data with manual analyses. These manual analyses include a comparison of the 100 shortest and 100 longest conflicting scenarios and observations of how developers resolved the merge conflicts, for instance. As a major result of these analyses, we found: (i) a dependency among conflicting and non-conflicting code, which normally increases the time necessary for developers resolving merge conflicts, (ii) reasons for why it is better to commit many small chunks of code instead of few large chunks, and (iii) why characteristics of merge scenarios influence the merge conflict resolution time more than characteristics of the merge conflicts themselves.

Overall, we make the following contributions:

- We propose a taxonomy of challenges on merge conflict resolution acquired by quantitative empirical data and by surveying developers of subject projects.
- We provide evidence that *#Chunks*, *#LoC*, *#ConfChunks*, *#Devs*, and *CodeComplexity* have an effect on merge conflict resolution time.
- We found that variables indirectly related to merge conflicts (e.g., number of chunks changed in a merge scenario) have a higher influence on the merge conflict resolution time than variables directly related to merge conflicts (e.g., the number of conflicting chunks).
- We found a positive correlation between *#LoC*, *#Devs*, *#Files*, *#ConfChunks*, *#ConfFiles*, and *#ConfLoC* and merge conflict resolution time and a negative correlation between *#Chunks* and *CodeComplexity* and the merge conflict resolution time.

- By a manual analysis of the 100 shortest and the 100 longest merge scenarios, we observed that file extension and dependencies among conflicting and non-conflicting code make developers take longer to resolve merge conflicts.
- We found that, in more than 50 % of the cases, developers have changed the files that are in conflict before they resolve the merge conflicts.
- We found that despite 30 out of the 66 projects having at least one conflicting merge scenario, due to formatting changes, formatting changes result in merge conflicts in only 2.42 % of the merge scenarios.
- We make our infrastructure to mine fine-grained merge scenario information from software repositories and all data publicly available on our supplementary Web site [63] for replication and follow-up studies.

2 Background and Related Work

In this section, we present a classification of merge commits necessary to understand how we retrieve merge scenario information and an overview on merge conflict resolution studies.

Version control systems help developers to manage source-code changes over time by tracking all code modifications [66]. This allows developers to construct software systems by means of merge scenarios, which is a widely collaborative development pattern called three-way merge [28], [39]. A *merge scenario* includes the whole timeline of creating a project branch, committing changes independently to this branch, and creating a merge commit.

To retrieve merge scenario information, it is necessary to rebuild each merge scenario of each subject project and get information from each commit of a subject merge scenario. Commits can be classified into four groups: merge commits, base commits, parent commits, and common commits. The *merge commit* is the commit that integrates the branches in a merge scenario. The *base commit* is the common ancestor among the integrated branches. *Parent commits* are the last commits of each branch before integration. *Common commits* are all other commits that are not the merge, base, or parent commits. In the example of Figure 1, the base commit is the commit on the left with hash FF1E147, the merge commit is found on the right side with hash C2ECB2C, the two parent commits have hashes 0E8F458 and 20BBDF7, and the other two remaining commits with hashes 35DBC8F and 923E4D5 are common commits.

One of the main problems of collaborative software development are merge conflicts because resolving them is difficult, time-consuming and often an error-prone task [39]. As mentioned, there are dozens of studies investigating the whole merge conflict life-cycle. Aiming at avoiding the emergence of merge conflicts, studies have investigated *merge strategies* (e.g., [5], [6]), *prediction strategies* (e.g., [12], [29]), and *awareness tools* (e.g., [7], [18], [38], [53]). After merge conflicts occur, studies have investigated *their cause and nature* to know how they look like exactly [24], which type of code changes lead to each type of merge conflict [1], [10], [43], and how to predict them [19], [39], [51]. More recently, a few studies started to investigate *merge conflict resolution* [11], [24], [35], [45], [49]. In the following, we discuss studies more closely related to ours that investigate merge conflict resolution.

Ji et al. [35] investigated merge conflicts and resolutions in GIT rebases of JAVA repositories from GITHUB. Their results show that (i) 7.6% of pull requests have rebases, (ii) merge conflicts arise in 24.3% – 26.2% of rebases, (iii) the likelihood of conflicts from (GIT) rebases is not significantly different from three-way merge conflicts (*git merge* command), and (iv) new code is introduced in 28.3% – 29.4% of conflict rebases.

Brindescu et al. [11] conducted an in-situ observation of 7 developers resolving 10 merge conflicts. Their results show that developers search for information on seven sources (diff between merged versions, commit history, source code, output running the application, build and tests output, documentation, and colleagues), the conflicts resolution took from 40 to 2 190 seconds (36.5 minutes), developers normally follow 6 steps on the conflict resolution (1) look at external data sources, (2) open a particular file to work on, (3) read or scroll through the source code, (4) edit source code, (5) read a chunk on either side, and (6) run the build or perform test. In addition, the authors observed two patterns when developers had issues in the conflict resolution (*stuck foraging* and *hunting for evidence*) and other two patterns when developers wanted a quick solution (*skipping directly to step 5* and *skipping the hypothesis* (i.e., skip step 4).

Ghiotto et al. [24] analyzed thousands of merge scenarios characterizing merge conflicts in terms of number of chunks, size, and programming language constructs and analyzing developer strategies to resolve merge conflicts. Their results show that (i) 87 % of conflicting chunks had all the information needed to resolve merge conflicts without writing any new code, (ii) 94 % of these conflicting chunks involved less than 50 lines of code in each of their versions, (iii) 60 % of merge conflicts involve multiple conflicting chunks, and (iv) 29 % of conflicting chunks are dependent on other chunks.

Nelson et al. [49] presented an extension of the work of McKee et al. [45]. These two studies had insights into developers' process and perceptions on merge conflict resolution. To achieve their goal, they investigated nine factors pointed out by developers to measure the difficulty of resolving a merge conflict. The factors ordered by difficulty are (i) complexity of conflicting lines of code, (ii) expertise in area of conflicting code, (iii) complexity of files with conflicts, (iv) number of conflicting lines of code, (v) time to resolve a conflict, (vi) atomicity of change-sets in conflict, (vii) dependencies of conflicting code, (viii) number of files in the conflict, and (ix) non-functional changes in codebase. As further results, they found that 88 % of the surveyed developers rely on version control systems (e.g., GIT and SVN) and 21 % use continuous integration systems (e.g., JENKINS[1] and TRAVIS CI[2]) when observing merge conflicts. In addition, they found that when developers feel that their experience is not sufficient to resolve the merge conflict, they generally seek help from other developers to resolve the conflicts.

Our study extends, complements, and add nuance to these studies because by looking at factors that make merge conflict resolution longer with an empirical analysis (Section 3), a survey with developers (Section 4), and manual analyses (Section 5). Despite Ji et al. [35] investigating merge conflict resolution, their study relies on rebase scenarios which is the

1. https://jenkins.io/
2. https://travis-ci.org/

main difference from other studies and ours. Brindescu et al. [11] provided a sensemaking perspective. Their approach provides a qualitative in-depth investigation, but generalization is limited. In any event, their findings corroborate our discussions. Ghiotto et al. [24] and Nelson et al. [49] present studies closer to ours, but we investigate more factors than they investigated (see Section 5). Furthermore, while Ghiotto et al. [24] empirically characterized merge conflicts and looked at developer strategies for resolving them, we look at factors that make the merge conflict resolution longer. McKee et al. [45] and Nelson et al. [49] surveyed developers to understand how developers estimate the difficulty of merge conflict resolution, the main barriers, and strategies developers follow when resolving merge conflicts. Our study differs from theirs because while they asked developers about subjective difficulty factors, we objectively analyzed factors that make merge conflicts resolution longer in practice and triangulated our results asking developers from subject projects.

3 Empirical Study

In this section, we present our empirical study setup and results. Our overall goal in this study is to *investigate which factors make merge conflicts resolution longer in the practice of collaborative software development.*

3.1 Study Settings

In what follows, we describe the experiment variables, subject selection process, data acquisition, and statistics we use.

3.1.1 Experiment Variables

To quantify the time of resolving merge conflicts, that is, our *dependent variable*, we measure the time difference between the merge commit and the latest commit of the merged branches (parent commits). To learn which factors may influence the time for resolving merge conflicts, we defined a set of ten *independent variables*, inspired by the literature [39], [45], [49] and described in Table 1. Note that all variables have been suggested by developers for the investigation of merge conflicts, although some of them were suggested for other phases of the merge conflict life-cycle [39], [45], [49]. Our survey confirms that developers use these variables for estimating the time/difficulty of merge conflict resolution (Section 4).

We explain below the rationale of choosing each variable. For a better overview, we classify the variables into three groups: *time*, *variables directly related to merge conflicts*, and *variables indirectly related to merge conflicts*.

Time. With *#SecondsToMerge*, we aim at capturing how much time (in seconds) passed for resolving a merge conflict (Table 1). Note that this is our operationalization. In our eyes, it represents the sweet spot of accuracy that is achievable in a post-hoc analysis. Considering that this variable is central to our study and might not be very precise, we surveyed 140 developers (Section 4) and provided a broader discussion about this variable in Sections 5 and 6.

Variables directly related to merge conflicts. This group contains the majority of variables investigated in this study. As our goal is to analyze factors that influence merge conflict resolution time, it is reasonable to choose measures that directly quantify the size, complexity, and the knowledge of the integrator (i.e., who solved the merge conflict) on the conflicting code: *CodeComplexity* (via Lizard[3]), *#ConfChunks*, *#ConfFiles*, *#ConfLoC*, *%FormattingChanges*, and *%IntegratorKnowledge*. The last two variables are important because they can control for other variables. For instance, we may observe very large conflicting chunks, although these chunks occurred largely because of formatting changes (i.e., adding/removing line breakers and changing code spacing). Hence, these conflicting chunks would be easier and faster to resolve than other conflicting chunks (see Section 5).

TABLE 1
Variables of our study, along with their descriptions

Variable	Description
Dependent variable	
#SecondsToMerge	The shortest time difference between the parent commits and the merge commit
Independent variables, directly related to merge conflicts	
CodeComplexity	Sum of the cyclomatic complexity of conflicting chunks
#ConfChunks	Number of conflicting chunks
#ConfFiles	Number of conflicting files
#ConfLoC	Number of conflicting lines of code changed
%FormattingChanges	Percentage of formatting changes of conflicting chunks among all chunks
%IntegratorKnowledge	Percentage of the sum of conflicting chunks in files that the integrator had committed before the merge commit among all chunks
Independent variables, indirectly related to merge conflicts	
#Chunks	Number of chunks
#Devs	Number of developers changing code
#Files	Number of files
#LoC	Number of lines of code changed

Variables indirectly related to merge conflicts. As merge conflict resolution may depend on code changes not involved in a conflict, we considered also properties not directly related to merge conflicts (i.e., all the code changes in the merge scenario): *#Chunks*, *#Devs*, *#Files*, and *#LoC*.

Example. In the merge scenario of Figure 1, four developers (*#Devs*), namely DevA, DevB, DevC, and DevD, changed ten lines of code (*#LoC*), of which eight are in conflict (*#ConfLoC*). These lines of code changed four chunks (*#Chunks*), of which three are in conflict (*#ConfChunks*). These chunks belong to two files (*#Files*) and, as there are conflicts in both files, the number of conflicting files (*#ConfFiles*) is two. As DevD is the developer who solved the merge conflict (i.e., the integrator), and she committed to both files before the merge commit, we reason that DevD had knowledge of all changed files (*%IntegratorKnowledge*). Regarding the time to resolve the merge conflict (*#SecondsToMerge*), we compute the time difference between the source branch's parent commit (hash: 20BBDF7) and the merge commit (hash: C2ECB2C). That is, *#SecondsToMerge* is equal to T6 minus T5 in seconds. As Figure 1 is only a simple and abstract example, it is not possible (or meaningful) to calculate *%FormattingChanges* and *CodeComplexity*.

3. https://pypi.org/project/lizard/

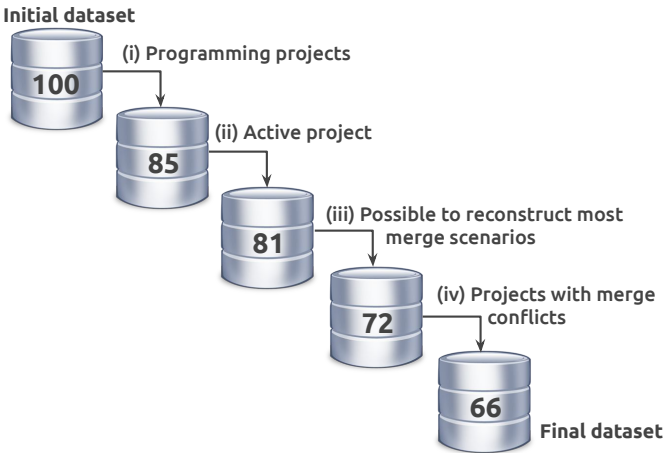

Fig. 2. Number of subject projects after each filter

3.1.2 Subject Projects

We selected the corpus of subject projects as follows. First, we retrieved the 100 most popular projects on GitHub, as determined by the number of stars [8]. Then, we applied the following four filters which we created based on the work of Kalliamvakou et al. [37]: (i) keep only programming projects (i.e., projects that have a programming language classified as the main file extension), (ii) keep only active projects (i.e., at least two commits per month in the last six months), (iii) keep only projects in which we were able to reconstruct more than 50% of the merge scenarios (see Section 3.1.3), and (iv) keep only projects with merge conflicts.

In Figure 2, we show the number of projects after each filter. These filtering steps aim at selecting active projects in terms of code contributions with an active community and at increasing internal validity. The first filter captures only software development projects, excluding projects that are, for example, repositories of books and interview tips. The third filter excludes projects such as KUBERNETES[4] and MOBY[5] because these projects do not mostly use the three-way merge [27] which could bias our analyses. Details on how we rebuild merge scenarios are in Section 3.1.3.

We restricted our selection to GitHub because it is one of the most popular platforms to host repositories and it has been investigated and used in prior work [17], [28], [56], [59], [62]. We limited our analysis to Git repositories because it simplifies the identification of merge scenarios in retrospect.

After applying all filters, we obtained 66 projects developed in 12 programming languages (e.g., JavaScript, Java, C++, and Python), containing 81 005 merge scenarios that involve more than 2 million files changed, 10.8 million chunks, and 2 608 conflicting merge scenarios.

3.1.3 Data Acquisition

We rebuilt all merge scenarios from the subject projects, since their creation. Our strategy for data acquisition consists of five steps. First, we clone a subject project's repository. Second, as the integration of multiple branches can be identified in Git when the number of parent commits is greater than one, we identify merge scenarios by filtering commits with multiple parent commits. Third, for each commit with more than one parent, we retrieve the base commit for both parent commits. Fourth, we (re)merge the parent commit of the source branch into the parent commit of the target branch by using the standard *git merge* command and retrieve measurement data for the metrics presented in Table 1 by comparing the changes that occurred since the base commit until the merge commit. Finally, we store all data and repeat Steps 3 to 5 for each merge scenario found in Step 2.

Note that we have excluded merge scenarios that do not have a base commit (e.g., rebase, fast-forward, or squash integrations [36]), and we ignore binary files, because we cannot track their changes. In the analyses that focus on the merge conflict resolution time, we retrieve data only for merge scenarios that resulted in merge conflicts. Note that the integration of two branches is not tied to pull-requests. Once we identified an integration of one or more branches into another, we rebuilt the merge scenario.

Framework and Data Availability. Our analysis framework (Java and Python) and analysis scripts (R) are open-source. All data necessary for replicating this study are stored in a MySQL database and replicated as CSV files. All tools, links to the subject projects, and data used in this study are available at our supplementary Web site [63].

3.1.4 Statistical Analysis

The statistical analysis of our study is threefold. First, we perform a ***correlation analysis*** of all covariables, using the Spearman rank-based correlation, which is invariant for linear transformations of covariates. This analysis is simple and useful to understand the relation among our covariables, to build a consistent regression model, and to support the discussions in Section 5. Spearman rank-based correlation is -1 in the case of a perfect negative correlation, +1 in the case of a perfect positive correlation, and values around 0 imply that there is no correlation between the investigated variables [34]. Note that the purpose of this first correlation analysis is data exploration, we are not going to draw conclusions on such correlation coefficients. So, a correction on the p-values to account for the multiple comparisons is not necessary. In any event, when looking at the correlation between dependent and independent variables, we performed a Bonferroni p-value adjustment for each independent variable (i.e., p-value < 0.005). Still in the first analysis, to better understand the correlations among variables and reduce the number of dimensions (i.e., variables) in the regression model, we perform a principal component analysis (PCA). PCA is important because, by removing correlated variables from the regression model, we avoid common pitfalls on modelling data [61].

With the insights of the first analysis, we build a ***multiple linear regression model*** in our second analysis for understanding the relation between our dependent and our independent variables. Multiple linear regression models are relatively simple yet powerful to achieve our goal and significantly easier to explain and interpret than other models such as neural networks and deep learning models. Coefficients in the regression model are interpreted similarly to the Spearman rank-based correlation coefficients from the first analysis. The multiple linear regression model of a y dependent variable on the $x_{1...n}$ independent variables is represented by

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \varepsilon_0. \quad (1)$$

4. https://github.com/kubernetes/kubernetes
5. https://github.com/moby/moby

The β coefficients measure the association between the independent variables and the dependent variable. β_j can be interpreted as the average effect on y of one unit increase in x_j, holding all other independent variables fixed [33]. To define the used model, we compare the variance of different models, such as the model with all independent variables and the model with a simplified number of independent variables and choose the model with greater variance, as suggested by previous work [21], [26], [33] (see details in Section 3.2.3).

Finally, we performed an ***effect-size analysis*** in the context of an analysis of variance. For short, the effect-size analysis is necessary because independent variables may change differently and, even with the results of the regression model, we are not able to classify the most influencing factors. The Cohens's f^2 for sequential multiple regression is:

$$f^2 = \frac{R^2_{AB} - R^2_A}{1 - R^2_{AB}} \quad (2)$$

where B is the variable of interest, A is the set of all other variables, R^2_{AB} is the proportion of variance accounted for A and B together, and R^2_A is the proportion of variance accounted only for A. By default, Cohen's f^2 effect size values from 0.02 to 0.15 are small, from 0.15 to 0.35 are medium and greater than 0.35 are termed large values [15]. In addition to Cohens's f^2, we also report the η^2 and ω^2 values to increase the confidence of our effect-size analysis. η^2 is the proportion of the total variability in the dependent variable that is accounted for by the variation in the independent variable [40]. It is the ratio of the sum of squares for each group level to the total sum of squares. It can be interpreted as the percentage of variance accounted for by a variable. ω^2 is widely viewed as a less biased alternative to η^2, especially when sample sizes are small [40]. η^2 and ω^2 effect size values smaller than 0.01 are very small, from 0.01 to 0.06 are small, from 0.06 to 0.14 are medium, and greater than 0.14 are termed large values [23].

3.2 Results

In this section, we present the results of our empirical study. First, we present an analysis in the distribution of our dependent variable. Then, the rest of the section is structured according to the three analyses presented in Section 3.1.4.

3.2.1 Dependent Variable Distribution

In Figure 3, we show two boxplots with descriptive statistics of our dependent variable (*#SecondsToMerge*). The statistics include the minimum value, first quartile, median, third quartile, and maximum value in seconds. As seen, the fastest merge conflict resolution took 30 seconds and the maximum took 45 857 805 seconds (around 530 days). This is definitely an outlier scenario that has been forgotten by developers for one and half years and integrated later. Median is a reasonable measure to analyze since outlier scenarios would distorce the mean. Looking at the median, we see that half of the merge conflicts took up to 11 minutes to be resolved (697 seconds). Looking at the first and third quartile, we see that 25% of the conflicting merge scenarios took 149 seconds (≈2.5 minutes) and 75% of the conflicting merge scenarios took up to 6 372 seconds (≈1.77 hours). In the right-most boxplot, we show data in the interquartile range. We show this boxplot since the box-plot with all data does not give the real idea of the

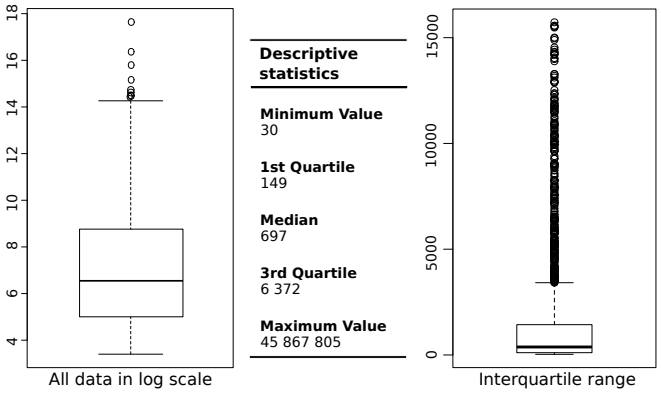

Fig. 3. Dependent variable distribution in seconds

time to resolve merge conflicts. In the interquartile range the median is 377 seconds (≈6 minutes).

Comparing to a recent study [11] that recorded the merge conflict resolution time from seven developers resolving 10 merge conflicts, they found that these developers took from 40 to 2 190 seconds (≈36.5 minutes). Even though, the number of conflicting scenarios in this previous study is quite limited and our variable might not be precise on measuring the time developers really spent resolving merge conflicts, which makes difficult to compare these variables, it is worth to mention that *#SecondsToMerge* are not far from their records. In addition to this comparison, we provide follow-up analyses and discussions to increase construct validity of the choice of our dependent variable (see Sections 4.2, 5, and 6).

3.2.2 Correlation Analysis

In Figure 4, we present a correlation matrix among all co-variables of our analysis. As expected, merge scenario size measures (i.e., *#Chunks*, *#Files*, and *#LoC*) have a high correlation among themselves (above 0.8). Merge conflict size measures (i.e., *#ConfChunks*, *#ConfFiles*, and *#ConfLoC*) show a moderate to high correlation among themselves (above 0.5). The other merge conflict related measures do not have a strong correlation. For instance, *%IntegratorKnowledge* and *%FormattingChanges* have a correlation coefficient smaller than 0.1 with most of the merge conflict related measures. The only exception is *%IntegratorKnowledge* with a positive correlation coefficient with *CodeComplexity* (0.115).

Next, we pay more attention to the correlation between the dependent variable and each independent variable. For short, the correlation is significant with a confidence interval of 99.5 % for all independent variables, except *%IntegratorKnowledge* and *%FormattingChanges*. In Table 2, we present the correlation coefficients for the significant ones. Note that the correlation coefficients of these eight variables are rather small, but significant. Also note that the top three variables with highest correlation coefficient are variables that measure the merge scenario size (*#LoC*, *#Chunks*, and *#Files*) and not merge conflicts.

Aiming at reducing the number of dimensions and grouping similar variables, we performed a principal component analysis (PCA). It reduces the number of dimensions to the first two principal components that retain a maximum share of common variance, which simplifies the discussion

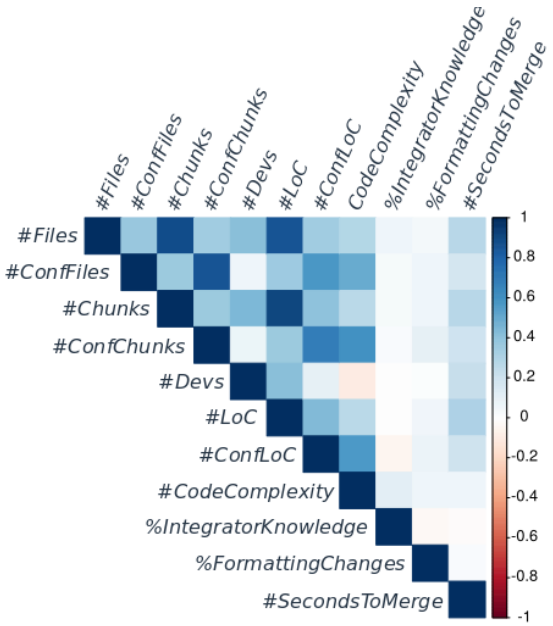

Fig. 4. Correlation matrix for all pairs of variables

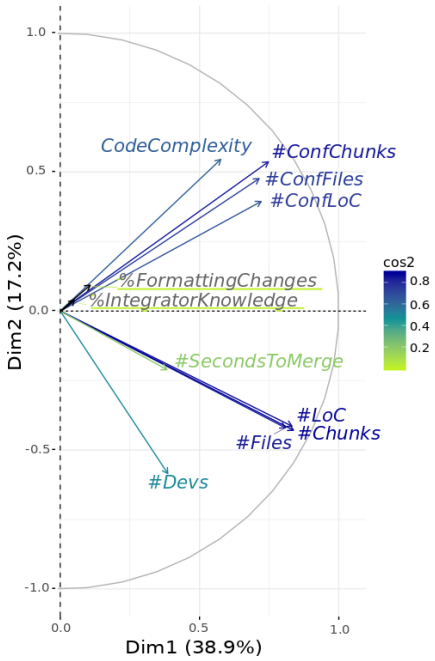

Fig. 5. Principal component analysis of our variables

TABLE 2
Correlation coefficients for independent variables with the dependent variable

Measure	Coefficient	Measure	Coefficient
#LoC	0.308	*#ConfLoC*	0.206
#Chunks	0.279	*#ConfChunks*	0.194
#Files	0.270	*#ConfFiles*	0.180
#Devs	0.228	*CodeComplexity*	0.061

of the correlation structure. In Figure 5, we show the two-dimensional output from the principal component analysis, which covers *56.1 %* (*38.9 % + 17.2 %*) of the total variance of our data. The arrows represent the weights of each variable in the respective principal component and their colors represent the square cosine (cos2). The square cosine represents the share of original variation in the variable retained in the dimensionality reduction. The longer the arrow, the larger is the share of a variable's variance. Arrows pointing to the same direction have a large share of common variance and can be assumed to belong to the same group.

The data visualized in Figure 5 suggest to classify the independent variables into four groups: *merge scenario size*, *merge conflict size*, *social activity*, and *integrator's prior knowledge/type of change*. The arrows representing *#Chunks*, *#Files*, and *#LoC* point to the same direction; they represent the size of a merge scenario. Pointing to another direction, *#ConfLoC*, *#ConfFiles*, and *#ConfChunks* represent the merge conflict size. The *#Devs* point to a third direction and, hence, we call it social activity. The factors *CodeComplexity*, *%FormattingChanges*, and *%IntegratorKnowledge* compose the fourth group, which we named integrator's prior knowledge/type of change.

To summarize, we see that (i) by clustering our independent variables into four groups, we do not need all of them in our regression model which increases internal validity avoiding overfitting and multicollinearity, as we explain in Section 3.2.3; (ii) measures from the *integrator's prior knowledge/type of change* group are almost orthogonal to the merge conflict resolution time which means a small share of variance among these variables with the merge conflict resolution time; and, (iii) measures from the *merge scenario size* and *social activity* groups have a stronger relation with the merge conflict resolution time than measures from the groups *merge conflict size* and *integrator's prior knowledge/type of change*.

3.2.3 Multiple Regression Model Analysis

All independent variables presented in Section 3.1.1 may be in our model because there is a belief that these variables influence the merge conflict resolution [39], [45], [49], which is confirmed in with our survey (Section 4). However, including all independent variables would increase overfitting (i.e., a model that contains more parameters that can be justified by the data) and multicollinearity (i.e., high correlation between two or more independent variables) in our model. To minimize overfitting and multicollinearity in our model, we perform a transparent process, as suggested by different researchers [21], [26], [33]. Of course, we could use a variable of each group of the PCA analysis. Nonetheless, we do not know which variables better fit the multiple regression model and we may ignore hidden relationships. In our case, this process consists of four steps: (i) create a preliminary model and learn with this model; (ii) create further models with observations made from the first preliminary model; (iii) compare the variance of the created models; and, (iv) choose the model that represents the investigated relationship most accurately. We present the details for these models in Table 3.

The *Full Model* column of Table 3 presents the correlation coefficients for the 10 independent variables that compose our preliminary model. Looking at the correlation coefficients of this preliminary model, we can make three observations: (i) the coefficient of six independent variables (i.e., *#LoC*, *#ConfChunks*, *#Devs*, *CodeComplexity*, *#Chunks*,

TABLE 3
Correlation coefficients for independent variables in the multiple regression model analysis

Measure	Full Model	Simplest Model	Balanced Model
#LoC	0.2538^{***}	0.2268^{***}	0.2931^{***}
#ConfChunks	0.1239^{**}	0.1752^{***}	0.1782^{***}
#Devs	0.1221^{***}	0.1171^{***}	0.1251^{***}
CodeComplexity	-0.1067^{***}	-0.0870^{***}	-0.0841^{**}
#Chunks	-0.1013^{*}	-	-0.0783^{*}
#ConfLoC	0.0799^{**}	-	-
#Files	0.0525	-	-
#ConfFiles	0.0146	-	-
%FormattingChanges	−0.0048	-	-
%IntegratorKnowledge	−0.0041	-	-

$^{***}p-value < 0.001$, $^{**}p-value < 0.01$, $^{*}p-value < 0.05$

and *#ConfLoC*) are significant with a confidence interval of 95 %; (ii) the four independent variables with greatest correlation coefficients belong to distinct groups of our PCA analysis (see Section 3.2.2); and (iii) from the two remaining variable with coefficient significant (i.e., *#Chunks* and *#ConfLoC*), only *#Chunks* provides a different view from the variable that belongs to the same group. In other words, while *#LoC* has a positive correlation coefficient in the *Full Model*, *#Chunks* has a negative correlation coefficient. On the other hand, *#ConfLoC* and *#ConfChunks* have a positive correlation coefficient. Therefore, adding both does not provide a different view to our model. Hence, choosing only *#ConfChunks* which has a greater correlation coefficient in the *Full Model* is more promising to avoid overfitting and multicollinearity.

Taking these observations into account, we build other two regression models: *simplest model* has only the four variables with greatest correlation coefficients (see observation ii) and *balanced model* has *#Chunks* and all other variables in the simplest model (see observation iii). The correlation coefficients of these models can be seen in Table 3. While the *simplest model* and the *balanced model* minimize overfitting and multicollinearity, the *balanced model* shows the hidden relationship among *#LoC* and *#Chunks* that we did not see in the correlation analysis. From that perspective, the *balanced model* seems to be the correct model to choose, although the analysis of variance supports a more data-oriented choice.

In the *analysis of variance*, a significant p-value (i.e., < 0.05) means that adding variables to the model, in fact, adds relevant information to the regression model. We compare the balanced model with the other two models because these comparisons allow us to find out which model better represents the investigated relationship. The p-value of the analysis of variance among the *balanced model* and the *full model* is 0.1. Therefore, adding *#ConfLoC*, *#Files*, *#ConfFiles*, *%FormattingChanges*, and *%Integratornowledge* to the balanced model do not add relevant information to it. On the other hand, the p-value of the analysis of variance among the *simplest model* and the *balanced model* is 0.004. Therefore, adding *#Chunks* to the simplest model add relevant information to it. In conclusion, the *balanced model* fits better on the investigated relationship than both the simplest and the full model.

Once we have chosen the model that best represents the relationship among our dependent and independent variables, we discuss its correlation coefficients as follows. Column *Balanced Model* of Table 3 presents the coefficients obtained from the chosen regression model. We can see that *#LoC*, *#ConfChunks*, and *#Devs* show a positive correlation with *#SecondsToMerge*. Hence, if these variables increase, the time to resolve merge conflicts also increases. On the other hand, *#Chunks* and *CodeComplexity* have a negative correlation coefficient with *#SecondsToMerge*. Increasing these two variables is associated with less time to resolve merge conflicts. Note that *#LoC*, which is the variable with the highest correlation in the correlation matrix(Figure 4), remains with the highest correlation in the multiple regression model analysis.

Our regression model has a significant explanation value at any significance level (p-value $< 1/10^{16}$). It has R^2 and adjusted R^2 equal to 0.122 and 0.12. Following Falk and Miller [20] classification $R^2 < 0.1$ is negligible and $R^2 \geq 0.1$ is adequate. Therefore, the R^2 of our model is adequate. Our model has a residual standard error of 706 on 2 602 degrees of freedom. It means that on average, our estimate is 706 above or below the observed value and it considers 2 602 out of the 2 608 conflicting merge scenarios investigated. Note that the interpretation of our regression model needs to be associated with the *ceteris paribus* concept. In other words, a correct interpretation of the model has to account that changing the value of one independent variable, all other variables' values have to be equal.

A simple way to interpret our regression model for *#LoC* and *#Chunks* is described as follows. Adding 1 000 LoC in the merge scenario is associated with an increase in time by approximately 293 seconds or 5 minutes to solve the merge conflicts, for a fixed amount of *#ConfChunks*, *#Chunks*, *#Devs*, and *CodeComplexity*, on average. Regarding *#Chunks*, for a fixed number of *#LoC*, *#ConfChunks*, *#Devs*, and *CodeComplexity*, adding 1 000 chunks in the merge scenario leads to a decrease in time by approximately 78 seconds or 1.2 minute. At first sight, the negative correlation coefficients for *CodeComplexity* and *#Chunks* in the regression model seem counter-intuitive, but, in Section 5, we discuss why it is not.

3.2.4 Effect-Size Analysis

Finally, to answer our research question and be able to quantify the influence of the independent variables on merge conflict resolution time, we performed an effect-size analysis. As described in Section 3.1.4, we chose Cohen's f^2 effect size since it is adequate when using multiple regression models [15]. In Table 4, we present the results of our effect-size analysis ordered by the highest to the lowest effect-size. We can see that the effect-size of *#Chunks*, *#Devs*, *#LoC*, *#ConfChunks*, and *CodeComplexity* are 0.298, 0.135, 0.129, 0.105, and 0.064, respectively. Following Cohen's classification, *#Chunks* has a medium effect-size on merge conflict resolution time, while the other four variables have a small effect-size. Interestingly, despite *#Chunks* is the variable in the chosen regression model with weak correlation, it has the highest effect-size.

Similar values are also found for η^2 and ω^2. *#Chunks* with a medium effect-size and the other variables with a small or very small effect-size. All variables in our effect-size analysis has a p-value $\leq$ than 0.001 and our analysis has a confidence interval level of 90%. Therefore, our analysis covers 90% of the subject conflicting merge scenarios.

Surprisingly, the three variables with highest effect-size (*#Chunks*, *#Devs*, and *#LoC*) are not directly related to

TABLE 4
Effect-size analysis

Measure	f^2	f^2 GV	η^2	η^2 GV	ω^2	ω^2 GV
#Chunks	0.298		0.078		0.078	
#Devs	0.135		0.016		0.017	
#LoC	0.129		0.015		0.014	
#ConfChunks	0.105		0.010		0.011	
CodeComplexity	0.064		0.004		0.003	

GV stands for graphical visualization of the target measure. In the case of Cohen's f^2, it is divided into three groups: small, medium, and high effect-size. In the case of η^2 and ω^2, it has an additional group very small when compared with Cohen's f^2.

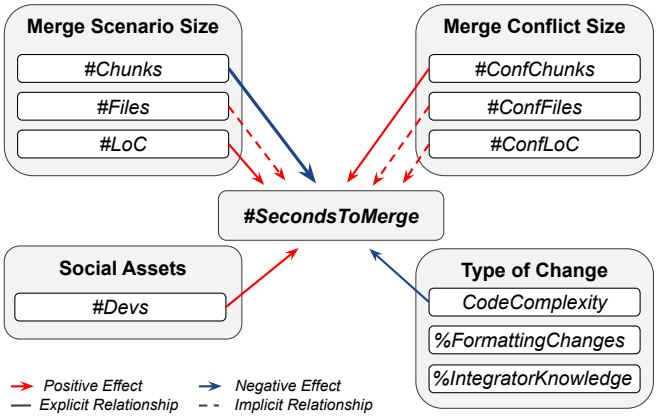

Fig. 6. Overview on our effect-size results

merge conflicts. By combining the results from the regression model and effect-size analysis, we can see that the number of chunks shows a negative correlation, whereas the other two variables indirectly related with merge conflicts (*#LoC* and *#Devs*) show a positive one. Hence, we conclude that more chunks in the merge scenario leads to shorter merge conflict resolution time with a medium effect-size. On the other hand, more lines of code and developers lead to more time to resolve the merge conflicts with a small effect-size.

Even though the correlation coefficients of the multivariable regression model are low, we obtained medium and small effect-sizes for the targeted independent variables. It highlights the importance of an effect-size analysis on measuring the impact of independent variables on the dependent variable. It may be an incentive for researchers to perform similar studies that do not stop on the correlation analysis.

Note that we present results only for the variables that compose the *balanced model* since it is the model that best represents the investigated relationship. As mentioned, further variables would only add noise to our analysis (see Sections 3.1.4 and 3.2.3) because they are highly correlated with variables that compose the regression model or do not correlate with *#SecondsToMerge*. Anyway, some of the hidden variables have a similar effect-size.

In Figure 6, we show an overview of our results considering all independent variables. Full and dashed lines represent explicit and implicit relationships investigated in the effect-size analysis, respectively. As we can see, *#ConfLoC* and *#ConfFiles* provide a similar effect-size to *#ConfChunks*. Hence, we added a dashed line from these variables to *#SecondsToMerge*. Similarly, *#Files* is correlated with *#LoC*. Hence, they have a similar effect-size with *#SecondsToMerge*. Since *%FormattingChanges* and *%IntegratorKnowledge* do not have a significant correlation with *#SecondsToMerge*, they also do not present an effect-size on *#SecondsToMerge*. For that reason, there is no line among them and *#SecondsToMerge*. We postpone a discussion of independent variables and their relationships to Section 5.

Results Summary: Our *correlation analysis* indicates that some variables are strongly correlated and, for that reason, we classified them into four groups which supported the construction of our regression model. Our *multiple regression model analysis* shows that *#LoC*, *#ConfChunks*, *#Devs*, *CodeComplexity*, and *#Chunks* are correlated with *#SecondsToMerge*. *#Chunks* and *CodeComplexity* have a negative influence while the others show a positive influence. Our *effect-size analysis* reveals that *#Chunks* has a medium effect-size on the merge conflict resolution time while *#Devs*, *#LoC*, *#ConfChunks*, and *CodeComplexity* have a small effect-size on the merge conflict resolution time.

4 Survey

In this section, we report on a survey of software developers from our subject projects (Section 3.1.2). The goal of the survey was to cross-validate our results and reduce threats to the validity of our quantitative findings. Next, we present the setting (Section 4.1) and results (Section 4.2) of our survey.

4.1 Settings

We created a seven-question survey, of which the first and last questions are open-ended. The other five questions are close-ended questions (5-point Likert-type scales). Aiming at grouping and systematically generating a theory from the answers of our open-ended questions, we used two Grounded Theory techniques [58], [60]: *open coding* and *axial coding*. We followed four steps of which two authors performed the first three steps separately. First, we extracted data from open-ended questions. Second, we segmented answers into meaningful expressions and described them in a short sequence of words (the open coding technique). Third, we relate short sequences of words to each other, combining inductive and deductive thinking (the axial coding technique). Fourth, all authors combined and discussed the outcome data repeating the second and third steps until we had a concise answer for a given question.

The survey is divided into three parts. First, we are interested in understanding factors that make the merge conflict longer/harder to resolve (Q_1). Then, with Q_2 to Q_6, we address potential threats to validity asking developers for

confirmation about results in the empirical study (Section 3). In the third part, we are interested in the experience survey participants had when dealing with merge conflicts (Q_7).

We recruited participants from subject projects that faced merge conflicts (obtained as described in Section 3.1.3) by directly contacting them via e-mail. We followed a learn-and-improve approach, in which we adapt questions based on the participants' feedback. For instance, in the first version of the survey, we asked about participants' experience and team size. However, a few developers replied to our email or reported in their survey response that, despite their interest in the topic, the survey was too long containing unnecessary questions and, for that reason, they or their colleagues did not answer the survey. Aiming at getting more informative answers, we decided to modify/shorten the survey.

The survey was available for about 6 months. We received 140 responses (response rate around 2%). Individual parts of the survey had varying response rates since open-ended questions were optional. The full set of survey versions are available at our supplementary Web site [63].

4.2 Results

We divide the discussion of the results of our survey in three sections according to the parts as previously described.

4.2.1 Participants' Perception of Factors that Make Merge Conflict Longer/Harder to Resolve

To understand factors that make the merge conflict resolution longer/harder, we asked the survey participants *Q_1: How do you estimate how hard/time-consuming a merge conflict is to be resolved?* We got 89 answers for this question. In Table 5, we present 25 measures pointed out by survey participants sorted by the number of suggestions. Measures used in the empirical study (Section 3) are highlighted with the acronym in parenthesis. As seen, the top three suggested measures are: the number of conflicting lines of code (*#ConfLOC*), the number of conflicting chunks (*#ConfChunks*), and the number of lines of code changed (*#LOC*). Notably, we used these three measures in our empirical study (Section 3). Interestingly, five participants mentioned they do not believe that it is possible to measure how hard/time-consuming is the resolution of merge conflicts.

A few participants mentioned that the difficulty/time of resolving merge conflicts is somehow related to the files' characteristics. For instance, nine participants mentioned that it is related to the number of files changed in the merge scenario (*#Files*), other four participants mentioned that it is related to the number of conflicting files (*#ConfFiles*), and another mentioned that it is related to the frequency files are changed, respectively. Further three participants mentioned that the difficulty/time of resolving conflicts is somehow related to the conflict's location. One of them mentioned "*fixing a conflict in the view layer* (i.e., referring to the Model-View-Controller design pattern) *is simpler than resolving a conflict in the controller layer*".

First Part Summary: As expected and in line with previous work [39], [49], the measures used in our empirical study reflect what survey participants think about merge conflict resolution. Surprisingly, measures not directly related to conflicts are among the most suggested ones.

TABLE 5
Measures to estimate the difficulty/time to resolve merge conflicts

Measure	#Sug.
Number of conflicting lines of code (*#ConfLOC*)	19
Number of conflicting chunks (*#ConfChunks*)	16
Number of lines of code changed (*#LOC*)	13
Number of files changed (*#Files*)	9
Time between the base commit and the merge commit	5
Developer experience responsible for conflicting changes (*∼%IntegratorKnowledge*)	4
Number of conflicting files (*#ConfFiles*)	4
Frequency target file changed	4
Semantically diff between conflicting code	4
Number of active developers (*#Devs*)	3
Number of commits with conflicts	3
Developer knowledge on the project (*∼%IntegratorKnowledge*)	3
Number of callers and callees functions in the conflicting code	3
Conflicts location	3
Number of chunks (*#Chunks*)	2
Number of commits	2
Number of conflicting lines per file in conflict	1
Number of commits affecting a file	1
Number of whitespace changes (*∼%FormattingChanges*)	1
Code complexity of conflicting code (*CodeCompexity*)	1
Number of conflicts per file	1
Average size of conflicting chunks	1
Ratio number of chunks by the number of conflicting chunks	1
Number of conflicts multiplied by the average of the number of conflicting lines of code	1
Character diff	1

#Sug. stands for the number of participants suggested a target measure.

TABLE 6
Responses on 5-point Likert-type scale indicating the agreement with questions (1 means hardly ever true, 5 means nearly always true)

#Q	Description	1 2 3 4 5	$\tilde{x}$	$\bar{x}$
Q_2	The more time it takes to resolve a conflict, the more difficult the conflict		3	3.4
Q_3	I merge my changes right after addressing an issue		4	3.9
Q_4	I resolve merge conflicts right after they occur		4	4.2
Q_5	I look at non-conflicting changes to resolve conflicts		3	3.4
Q_6	I change non-conflicting code to resolve merge conflicts and avoid introducing unexpected behavior to the project		3	2.8

#Q, $\tilde{x}$, and $\bar{x}$ stand for questions, median, and mean, respectively.

4.2.2 Cross-validating the Quantitative Results

In the second part of our survey, we address potential threats to validity and cross-validate our quantitative results from the empirical study (Section 3). In Table 6, we present statements and answers for Q_2 to Q_6. The 5-point Likert-type scale means: 1 – hardly ever true, 2 – rarely true, 3 – sometimes true, 4 – often true, and 5 – nearly always true.

As the term "difficulty" is subjective, in Q_2, we asked whether survey participants agree with the statement *"the more time it takes to resolve a conflict, the more difficult the conflict is"*. 15% of them mentioned that it is hardly ever true

or rarely true, 37.9% mentioned that it is sometimes true, and 47.1% that it is often true or nearly always true. In Q_1, a participant, who mentioned that this statement is rarely true, made an interesting comment: *"Time is not perfect because there may be lots of simple changes but it's time consuming to rectify and potentially error prone"*. Despite her mentioning that time is not perfect, she indirectly assumes that small changes may become a difficult task due to potential bug introduction. Even though most survey participants agree with time as a measure of difficulty, our study is more straightforward by searching for factors that make conflict resolution longer.

With Q_3 and Q_4, we investigate whether survey participants merge their changes right after addressing an issue and whether they resolve merge conflicts right after they occur. These questions were motivated by the fact that we are not able to detect unexpected events that happened on the merge-conflict resolution (e.g., the developer responsible for resolving the conflict had a break). Yet, 72.9% and 77.5% of the participants agree that statements of Q_3 and Q_4 are often true or nearly always true (median 4 and mean around 4). Of course, this does not automatically mean that *#SecondsToMerge* precisely measures the time a developer spent resolving merge conflicts. However, with the survey answers, we have evidence that they normally merge their changes right after addressing an issue and resolve conflicts right after they occur.

With Q_5 and Q_6, we investigate whether survey participants look at non-conflicting changes to resolve conflicts and whether they change non-conflicting code to resolve conflicts avoiding introducing a new bug. The main motivation for these two questions is a result of our quantitative study indicating that merge conflict resolution time is strongly correlated with measures not directly related to the merge conflicts (e.g., the number of chunks changed and the number of developers active in the merge scenario). Regarding Q_5, 50.7% of the survey participants mentioned they often or nearly always look at non-conflicting code, and 25.7% of the participants sometimes look at non-conflicting code. Regarding Q_6, 25.7% said that they often or nearly always change non-conflicting code and 34.3% of the participants sometimes change non-conflicting code. We expected that developers would often look at non-conflicting code, but change it only rarely. In any event, scanning all changes in the merge scenario is time-consuming and influences the merge conflict resolution. These results provide evidence that resolving merge conflicts is much more than only fixing lines in conflict (see Section 5).

Second Part Summary: We found evidence that developers from our subject projects usually think that, the more time it takes to resolve a conflict, the more difficult the conflict is. With our survey, we confirm our assumption that developers usually merge after addressing an issue and resolve merge conflicts right after they occur, increasing construct validity of the dependent variable of our empirical study. Finally, we found that developers often look at non-conflicting code and sometimes change non-conflicting code when fixing merge conflicts to avoid bug introduction, cross-validating a not very intuitive result from our empirical study.

4.2.3 Experience of Dealing with Merge Conflicts

In the third part of the survey, we asked participants to share experience of dealing with merge conflicts (Q_7). We got 43 responses to this part.

Challenges of merge conflict resolution. In Table 7, we present the 4 main challenges on merge conflict resolution brought up by our survey participants. We describe these challenges next.

Lack of coordination. We found four sub-challenges that deteriorate coordination: (i) lack of communication and awareness, (ii) large commits and rare merges, (iii) monitoring changes at coarse-grained level, and (iv) lack of an overall workflow. A participant mentioned "*good communication might avoid most hard conflicts*". Another participant suggested that most time-consuming conflicts arise from refactoring: "*Code moving from place to place is also a very hard scenario (in part because it makes diffs harder to obtain)*". Thirteen participants reported that their strategy to avoid conflicts is simply based on small commits and repeated merging. Interestingly, 8 participants mentioned that they rebase their changes often. We discuss rebase scenarios in Section 5. Related to the third sub-challenge, a participant mentioned "*I manage changes at the chunk level (not as files)*". Regarding lack of overall workflow, a participant mentioned "*good development processes avoid most merge conflicts*".

Lack of tool support. We identified three sub-challenges related to tool support: (i) inappropriate development environment, (ii) inappropriate tools for showing diffs and supporting merge conflicts resolution, and (iii) mismanaging the backlog. A participant stated "*Never resolve conflicts by hand. Use a tool*". Other six participants mentioned that merge conflict resolution is much easier with an appropriate IDE. One of them said: "*I use the included git merge conflict tool in IntelliJ. The 'magic wand' is a really powerful tool which can solve some merge conflicts, for example if there are 20 diffs in a file that magic wand button can usually figure out what to change, and only leave you with one or two lines which it can't figure out by itself*". Other participants mentioned tools they use to support diffing and merge conflict resolution. The reported tools are: P4MERGE[6], FILEMERGE[7], BEYONDCOMPARE[8], OPENDIFF[9], BBEDIT[10], TORTOISE[11], GIT DIFF[12], and GITK[13]. A few of them reported the reasons for choosing a tool. For instance, a participant mentioned that she uses TORTOISE because it shows her changes and the remote changes side by side and the file for merging them below. Other participants just mentioned avoiding duplicated work (e.g., avoiding addressing the same JIRA task) and working on the same parts of the source code at the same time.

Flaws in the system architecture. We found two sub-challenges related to system architecture flaws: (i) highly coupled code and (ii) technical debt introduction. A participant mentioned that conflicting code that is highly coupled is

6. https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge
7. https://developer.apple.com/xcode/
8. https://www.scootersoftware.com/
9. https://developer.apple.com/xcode/
10. https://developer.apple.com/xcode/
11. https://tortoisegit.org/
12. https://git-scm.com/docs/git-difftool
13. https://git-scm.com/docs/gitk

TABLE 7
Challenges on merge conflict resolution

Challenge	Sub-Challenge	Solution
Lack of coordination	Lack of communication and awareness	Create communication channels for all stakeholders and channels (e.g. slack or Microsoft teams) focused on developers or specific components (e.g. backend and frontend developers)
		Fix conflicts as soon as you are aware
		Keep others aware of refactoring changes
		Use adequate tool support to avoid developers working on the same region of code (see solution for the sub-challenge *mismanaging the backlog*)
	Large commits and rare merges	Create minimal commits (i.e., small chunks)
		Pull/push changes often (i.e., merge often)
		Create tasks/pull-requests small and focused
	Monitor changes at coarse-grained level	Manage code changes at fine-grained level (e.g., at method- or at chunk-level)
	Lack of an overall workflow	Create well-defined and documented development process
		Create and document contribution rules (e.g. formatting styling)
Lack of tool support	Inappropriate development environment	Use appropriate IDEs and, if possible, developers should use the same IDE
	Inappropriate tools for showing diffs and support merge conflicts resolution	Some IDEs provide support for that. If it is not your case, use (ad-hoc) tools to support this task
	Mismanaging the backlog	Use issue trackers (e.g., GitHub or Bitbucket) and/or tools for managing work (e.g., Jira or Asana)
Flaws in the system architecture	Highly coupled code	Refactoring code to minimize coupling and increasing cohesion
		Create an architecture that follows well-known design patterns (e.g., Singleton, Decorator, and Observer)
	Technical debt introduction	Always review code changes. Especially, more experienced developers should carefully review code changes from less experienced developers
Lack of testing suite or pipeline for continuous integration	Lack of tests and their maintenance	Always create test cases for new features and integrate them with existing test cases ensuring that no unexpected behavior was introduced
		Update test cases always such that something changes in the project related to existing test cases
	Lack of continuous integration pipeline and its maintenance	Create and maintain a pipeline for continuous integration

normally harder to resolve, since this requires looking into files that have not changed in the merge scenario or have changed but have no conflict. Another participant complemented this by mentioning that non-trivial merge conflicts tend to be a symptom of architectural flaws that make it difficult to apply a given change without touching a lot of different files/systems. Regarding the introduction of technical debt, a participant suggested that the introduction of new features/code should be reviewed by more experienced developers aiming at reducing the introduction of technical debt. This case is either related to the deterioration of the system code/design or to future refactorings. As mentioned by participants, both are prone to introduce conflicts. In fact, previous studies have reported the relation of merge conflicts with code smells [2], [4] and their effects on software quality [9]. In most cases, merge conflicts deteriorate the software quality. Furthermore, researchers [16], [50] have found that sometimes developers may not have the expertise or knowledge to make the right decisions, which might degrade the quality of the merged code. This highlights the importance of proper code review by experienced developers.

Lack of testing suite or pipeline for continuous integration. We classified factors related to this challenge into: (i) lack of tests and their maintenance and (ii) lack of continuous integration pipeline and its maintenance. A participant stated: "*my harder conflicts are often when integrating two different large feature branches, the tests may at most ensure that specific isolated scenarios keep working, not that the involved features interact well, until newer tests are written for that purpose*". Another participant mentioned that, when describing her process on resolving merge conflicts, she tries to merge everything as much automated as possible. If it does not parse, or does not build, or does not pass on tests, then she uses reverse engineering.

Participants' desires, needs, and alerts. Participants articulated four desires, needs, or alerts.

Improve diffs. A participant mentioned "*a semantic diff would be amazing*", another articulated the desire of a diff of each version against the common ancestor, and several participants highlighted the importance of good visualization/interfaces in diff tools.

Keep awareness when others are refactoring. As mentioned, keeping awareness in the project is very important. There are some awareness tools proposed in the literature, for

instance, CollabVS [18], Palantír [53], Cassandra [38], and FASTDash [7]. However, what called our attention is a participant expressing the interest in a tool informing when others are refactoring the source code. As mentioned, for some participants, hard merge conflicts normally occur because of refactoring changes. Related to that, Mahmood et al. [43] found that refactoring was the most frequent change, which often collided with other refactoring or feature introductions and enhancements on the other branch. Furthermore, Mahmoudi et al. [44] have found evidence that refactoring operations are involved in 22% of merge conflicts and that conflicts that involve refactoring are more complex than conflicts with no refactoring. Putting awareness and refactoring together, Shen et al. [54] have proposed IntelliMerge, a graph-based refactoring-aware merging algorithm for Java programs and Cavalcanti et al. [13] have proposed jFSTMerge the state-of-the-art semi-structured merging algorithm for Java programs.

Show a merge-conflict difficulty estimation. A participant suggested a tool to show the merge conflict difficulty. We see it as an opportunity for tool builders building tools that work either reactively (i.e., when developers pull code from other branches) or proactively (i.e., the tool checks changes from other pre-selected branches periodically). Indeed we found some studies predicting indicators for merge conflicts [19], [39], [51], however, we did not find any tool that estimate the merge-conflict difficulty.

Improve Git *conflict message and merge strategy.* We got three opinions specific to Git. A participant complained about Git's conflict report: "*often the most confusing parts are the guides informing about incoming and current changes. Literally 2 in 3 times I have to refresh my memory about those*". Another participant reported: "*sometimes the 'differ' erroneously show the conflict to be across two functions while in reality just one function was changed significantly (also happens often when merging xml files) such cases should best be approached with a differ/merger that is aware of the underlying semantic but detecting such cases and assigning them high merge difficulty would be nice*". A third participant mentioned: "*The worst problems are when* Git *doesn't detect a merge conflict because the change appears to merge cleanly, but then bugs are introduced*".

Third Part Summary: Based on the participants' reported experience, we defined four challenges on the merge conflict resolution: lack of coordination, lack of tool support, flaws in the system architecture, and lack of testing suite or pipeline for continuous integration. Furthermore, we collected desires, needs, and alerts reported by survey participants in which we classified into: improve diffs, keep awareness when others are refactoring, show a merge conflict difficulty estimator, and improve Git's conflict message and merge strategy.

5 Discussion

Aiming at achieving a deep understanding of what happens when developers are resolving merge conflicts, we triangulate our analyses with a manual analysis of the 100 shortest and 100 longest merge scenarios (Section 5.1). Next, we discuss the outcome of variables individually (Section 5.2) as well as some relationships among them (Section 5.3). After that, we provide a comparison of our results with previous work results (Section 5.4), followed by a reflection of the importance of any improvement in the merge conflict life-cycle (Section 5.5).

5.1 Manual Analysis

In this section, we inspect our data deeply and manually to understand the resolution of merge conflicts and support further discussion points.

Is it possible to identify any difference between the most quickly resolved conflicts and the ones that took the longest to be resolved? To answer this question, we manually investigated the 100 shortest conflicting merge scenarios (Group 1) and the 100 longest conflicting merge scenarios (Group 2) performing three analyses. First, we check if the independent variables are statistically significant across these two groups to confirm the results presented in Section 3.2 and to increase the internal validity of our study. Second, we compare the file extension of the conflicting files for both groups to see if the content of files of specific extensions might influence the merge conflict resolution. Third, we look at each conflicting code and observe how developers resolved them to investigate merge conflicts resolution from a different perspective that our independent variables might have not been able to catch.

Regarding time, we see a huge difference: while the shortest conflicting merge scenarios took less than 40 seconds to be resolved, the median for longest ones is 6.62 days. In Figure 7, we present a comparison of the two groups for the ten independent variables investigated in the study. Except *CodeComplexity*, *%IntegratorKnowledge*, and *%FormattingChanges*, the other variables show a statistically significant difference for these two groups (Wilcoxon signed-rank test with $p\text{-}value < 0.001$). This result is similar to the results we presented in Section 3.2, except that there is no significant difference for *CodeComplexity*. Note that, although significant, the difference for variables measuring the merge conflict size is small. For instance, the average number of conflicting files is 1.06 and 1.75 for the shortest and longest conflicting merge scenarios, respectively. In other words, the 100 shortest and the 100 longest conflicting merge scenarios have on average around 1 and 2 files in conflict, respectively. It explains why the correlation between the merge conflict size measures and *#SecondsToMerge* is not strong, as we found in Section 3.2. In fact, previous studies [24], [43] found that merge conflicts are normally small. For instance, Mahmood et. al [43] found that 28 out of 40 investigated conflicts had only one line of code conflicting in the merge branches.

Regarding the second analysis, Table 8 presents a comparison of the extension of conflicting files for both groups divided into five categories: 1) *Minified files* pass for a minification process for markup Web pages and script files, for example. Although the minification process reduces readability, it dramatically improves site speed and accessibility [30], [52]. This minification process is normally automated by tools, such as Minify[14] and JSCompress[15]. 2) *Markdown files* describe the next category of files. Markdown is a lightweight markup language with plain text formatting syntax. 3) *Package manager files* are files automatically generated to manage the project.

14. https://www.minifier.org/
15. https://jscompress.com/

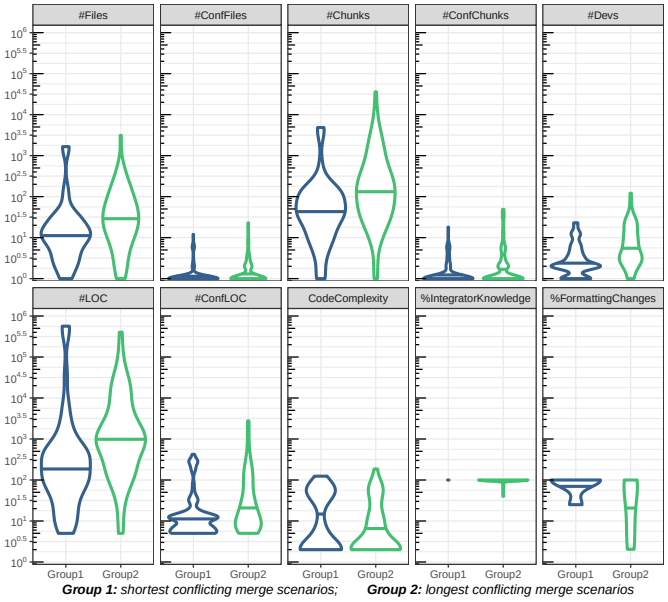

Fig. 7. Violin plots distinguishing shortest and longest conflicting merge scenarios.

TABLE 8
Comparing the number (and percentage) of conflicting files per category in the shortest (Group 1) and the longest (Group 2) scenarios

File Extension	Group 1	Group 2
Minified files	45 (42.45%)	16 (9.14%)
Markdown files	39 (36.79%)	21 (12.00%)
Package manager files	7 (6.61%)	14 (8.00%)
Programming language files extension	14 (13.21%)	119 (68.00%)
Other files	1 (0.94%)	5 (2.86%)

It includes, for instance, pom.xml and package.json files that are used in Java and by Node.js package manager (NPM), respectively, to identify the project as well as to handle project dependencies. 4) *Programming language files* are related to software development including programming project files, source code files, code libraries, header files, class files, and compiled objects. They are files not included in previous categories and have extensions, such as *.js*, *.rb*, *.css*, *.c*, *.h*, *.py*, and *.java*. 5) *Other files* represent files without extension (e.g., change-log files) or .gitignore files.

We argue that developers need more time to resolve merge conflicts in files with native programming language extensions because of the inherent structure including dependencies among methods, functions, procedures, class, modules, or components that may exist in the conflicting code of these files. Such dependencies may not happen in plain text files and can be automatically generated in package manager and minified files. Survey participants mentioned that number of callers and callees for added/removed functions and the conflict's location might influence the merge conflict resolution time (see Table 5). Furthermore, previous studies [10] [19] investigated the diffusion changes and the conflict's location. Brindescu et al. [10], found that diffusion changes (e.g., the number of files changed and the dependency among files) are important when predicting the difficulty of merge conflicts. Dias et al., [19] found that the likelihood of merge conflict occurrence significantly increases when contributions to be merged are not modular in the sense that they involve files from the same MVC slice (related model, view, and controller files). As we can see in Table 8, while only 13.21% of the conflicting files in Group 1 are files with native programming language extensions, 68.00% of the conflicting files in Group 2 are files with native programming language extensions. It might be an indication that source code files influence the time of resolving merge conflicts. Aiming for a more adequate answer, we look at the conflicting code as well as at how developers resolve merge conflicts in these files next.

Conflicts in Group 1. Looking at the conflicting code for Group 1, we found that (i) for all 45 minified files found with conflicts, the original file was also changed in the merge scenario. Hence, to resolve the merge conflicts developers only regenerated the minified files after the merge; (ii) for all package manager files found with conflicts, the number of the version of the document or of some dependencies were different. Hence, developers only chose the newest version to solve the conflicts; (iii) for the 14 files in native programming language extension, 5 of them had a timestamp problem in the header of the document and, hence, developers only chose the newest timestamp. For 4 files, we found code additions between existing methods in both branches. Hence, to remove the merge conflicts, developers only removed the conflict markers (e.g., ">>>>>>>"). For the remaining 5 files, we found a combination of formatting, with small refactorings (e.g., renaming) and fixing small issues and typos (e.g., adding an extra condition in an existing if-statement, or changing the background color of an object, or removing a "\9" that appeared in a JavaScript file). In these cases, developers chose the changes of one branch or a combination of both; (iv) for 39 markdown files and the .gitignore file with conflicts, conflicts were basically emerged from refactoring (e.g., rewriting phrases improving grammar) and code addition (e.g., adding new phrases to give more details about some topic) without any dependency with other files. The conflicts of these files are similar to the ones we found with the files in native programming language extension. To resolve the merge conflicts, developers chose one version, or removed the conflict markers. Even without knowing the code before, with a diff checker tool, we quickly understood why the conflict arose and we would resolve conflicts similar to how developers did.

Conflicts in Group 2. Looking at the conflicting code for Group 2, the longest conflicting merge scenarios, we found that: (i) regarding the 21 markdown files, 10 of them are in merge scenarios of which there are also merge conflicts in files from the native programming language extension category. Files from the native programming language extension category might have taken longer to resolve the merge conflicts, which might have influenced the resolution of conflicts in the 10 markdown files. The remaining 11 markdown files present URLs for other files changed in the merge scenario or with external links. We do not believe that this is the only reason that developers took longer to resolve merge conflicts of these files. However, it was a pattern that we noted; (ii) regarding the 16 minified files, in 13 of them the original files also changed and in 1 of these 13 files the original file also had merge conflicts. For the remaining 3 files, the original files did

not change. Looking only at code changes, we could not find a plausible reason for developers taking so long to resolve merge conflicts in these files; (iii) regarding the 14 package manager files, 8 of them had changes in the structure that were beyond formatting and version of dependencies. For the other 6 files, we found only formatting and versioning problems. However, 5 of them are in a merge scenario with other conflicting files. Hence, the other conflicting files might be the reason why these conflicts took so long to be resolved; (iv) the changes in all files in the *other files* category were simply code addition, although, all of them are in merge scenarios that have additional conflicting files. These additional files might be the reason for the long time needed to resolve their merge conflicts; (v) for the 119 files with native programming language, 78 have at least another file in conflict and 117 of them have at least another file that was changed in the merge scenario. Looking at the code changes, in 79 of the 119 files, we could not understand the code changes only by looking at the file in conflict because it contained a call to a method, procedure, or import file or module also changed in the merge scenario. Therefore, in 66.39% of the cases of source code files, we found a dependency that made the merge conflict resolution longer to resolve. For the remaining 40 files without any explicit dependency, in 29 there is at least another conflicting file with further dependency in the merge scenario. Hence, at the end, 90.76% of native programming language files are related to a non-trivial solution (i.e., involving complex dependencies with libraries and other packages). Only 11 of them do not have any other file with dependency and we could not find a plausible reason for developers needing so much time to resolve these merge conflicts.

As shown in Section 3.2.1, merge conflicts normally take up to 11 minutes to be resolved and this result is in line with previous work [11]. In that study, researchers measured the time spent resolving merge conflicts directly by observing developers. Even looking at the 100 longest merge scenarios, we could not find plausible reasons for why the merge conflict resolution took so long for only 11 of them. With the survey (Section 4), participants confirmed that they normally merge their changes and resolve conflicts right after they arise. Therefore, based on our manual analysis and on the answers of survey participants, we found that extraordinary events occur in practice, but not too often to the point of biasing our results. These results altogether, strengthens that *#SecondsToMerge* is a reasonable choice as a dependent variable for a post-hoc analysis as we presented in our empirical study.

Analysis Summary. Indeed, the longest conflicting merge scenarios are larger and more complex than the shortest conflicting merge scenarios for most of the independent variables, which is in line with the results presented in Section 3.2 and with survey participants. It shows that our choice of *#SecondsToMerge* as a dependent variable is plausible. Our subsequent analysis shows an indication that developers need more time to resolve merge conflicts in programming language files. With the follow-up analysis, we see that the content of files with some extensions remains an indicator for the merge conflict resolution time. However, the dependency among the code in conflict with files changed in the merge scenario may be a better indicator of the merge conflict resolution time. While in the files of Group 1 we did not find a dependency among the conflicting code, we found such dependency for 90.76% of the programming language files in Group 2 (see a concrete example in Section 5.3).

5.2 Investigating Non-correlated Variables

In this section, we investigate reasons of why the *%IntegratorKnowledge* and *%FormattingChanges* are not correlated with *#SecondsToMerge*.

Do integrators have knowledge of conflicting files? Are there differences between conflicting merge scenarios that are resolved by integrators with previous knowledge on the involved files and those integrators without previous knowledge? To answer these two questions, we distinguished between merge scenarios for which the merge-scenario integrator had previous knowledge of the involved files from those merge scenarios that the integrator did not have previous knowledge of. As an immediate result, we found that about 56 % of the conflicting merge scenarios have been integrated by a developer with some knowledge on the files in conflict. From these scenarios, integrators previously changed all files in conflict in about 94 % of such cases. Therefore, more than half of the merge scenarios are integrated by developers that already touched all files in conflict.

For further investigation, we found that the number of chunks tends to be higher, on average, when integrators have some knowledge about the conflicting files (see the violin plots in Figure 8) with a statistically significant difference (Wilcoxon signed-rank test, $p \approx 0.001$, $W = 891\,559$). We choose the number of chunks for this further analysis because it is the variable with the greatest impact on the merge conflict resolution time (see Section 3.2.4). Furthermore, as can be seen in Figure 8, integrators with previous knowledge seem to handle more complex merge scenarios with respect to the other measures that represent size and complexity of the merge scenario (e.g., *#Files* ($p \approx 0.001$, $W = 13\,030$), *CodeComplexity* ($p \approx 0.001$, $W = 13\,207$), and *#Devs* ($p \approx 0.002$, $W = 12\,382$)). Given that, we can assume that these developers are tasked with handling merge scenarios that spread farther across the code base and potentially inherit more semantic changes, resulting in locally restricted merge scenarios given to less knowledgeable developers.

Inspired by the response of survey participants (Section 4.2) and previous work [11], [49], we see two main reasons for this finding in the previous knowledge of the integrators

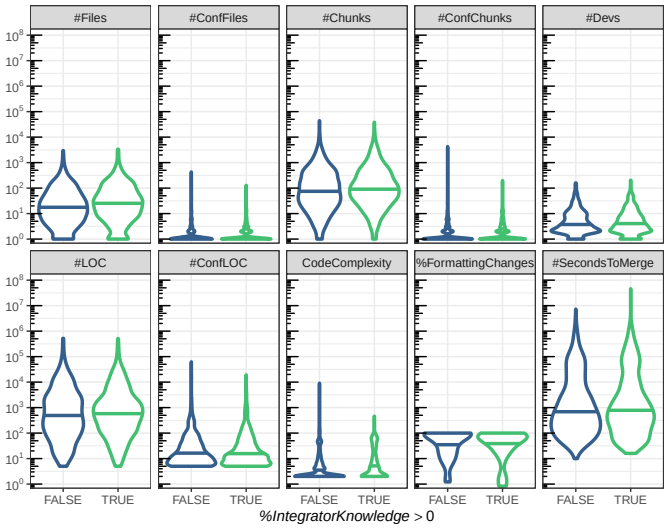

Fig. 8. Violin plots distinguishing merge scenarios by the predicate *%IntegratorKnowledge* > 0.

itself: (i) integrators that have profound knowledge of the involved files of a merge scenario tend to ask more questions regarding the proposed code and are able to identify more potential semantic problems avoiding future problems [11] and (ii) when developers feel that their experience is not sufficient to resolve the merge conflict, they generally seek for help from other developers to resolve the merge conflict [49]. Hence, an integrator without previous knowledge finds the merge conflict, but, at the end, a knowledgeable integrator that will normally solve the problem. This transition among integrators may take some time. Still, we cannot draw a conclusion that prior knowledge on conflicting files supports integrators solving merge conflicts faster than integrators without prior knowledge mainly because of the inherent structure including dependencies among methods, functions, class, modules, or components that may exist in the conflicting code as discussed in Section 5.1.

Implications for practitioners: In more than half of the conflicting merge scenarios, integrators with previous knowledge on the conflicting files are the ones that resolve merge conflicts and they normally solve more complex and larger merge scenarios. We argue that integrators with previous knowledge on the merge scenario are recommended to resolve more complex and larger conflicting merge scenarios because they may provide more solid solutions. However, we cannot affirm that previous knowledge makes the merge conflict resolution faster.

What is the percentage of conflicting merge scenarios due to formatting changes? 2.42 % of the conflicting merge scenarios occurred because of formatting changes. This small percentage is likely the reason that it does not correlate with other variables. For short, from the 66 subject projects, 30 have, at least, one merge scenario with merge conflicts arising due to formatting changes; 15 have, at least, one merge scenario of which all merge conflicts arose from formatting changes. What intrigued us, was the fact that despite the effort of researchers on proposing merge strategies [5], [6] and that simple definitions of contribution rules (e.g., defining expected code style) could extinguish this type of conflict, they are still present in some subject projects. As suggested by survey participants, formatting style from different IDEs might be an indicator of why this kind of conflict occurs even when contribution rules are well-defined in the project. An investigation aiming at finding out the reasons of why the merge conflicts due to formatting changes emerged and the actions developers took over the evolution of the projects would be welcome to both researchers and practitioners. As it is far from our goal, we leave it for future work.

Implications for researchers: Our results imply that formatting changes are not really relevant for the merge conflict resolution time since, despite 30 (out of 66) projects have conflicting merge scenarios due to formatting changes, it represents only 2.42 % of the merge scenarios analyzed. A deeper investigation on domains and a temporal analysis of the contribution rules may be fruitful to better understand the reasons why developers proposed them and how it impacts on the occurrence of merge conflicts as well as on their resolution time.

5.3 Investigating Relationships Among Subject Variables

In this section, we investigate relationships between subject variables that at the first glance seem counter-intuitive.

Why merge conflict resolution time is stronger correlated with merge scenario size measures than with merge conflict size measures? Some merge conflicts are not trivial to resolve and, for that reason, there are many studies investigating it [1], [5]–[7], [12], [18], [24], [29], [38], [39], [45], [49], [53], [64]. In some cases, unexpected results are found. For instance, Leßenich et al. [39] aimed at predicting merge conflicts, but even though they have used factors that practitioners indicated to be related to the emergence of merge conflicts (e.g., scattering degree among classes, commit density, and number of files), none of these factors showed a strong correlation with the occurrence of merge conflicts. One may say that the merge conflict resolution depends on the type of conflict [1], others say that it depends on the language constructs that are involved in the conflict [24]. It is reasonable to expect that resolving merge conflicts involves much more than only taking a look at the conflicting code, especially when there is a dependency between conflicting and non-conflicting code introduced in the merge scenario (see Section 5.1). Resolving merge conflicts without looking at dependent code changes may introduce unexpected behavior to the project, even when it passes the test suite. Hence, the most prominent action is to understand, at least, the non-conflicting code related to the conflicting code.

To get a concrete example, we choose a merge scenario from project NODE[16]. Despite of being a large merge scenario (*#LoC* = 36 794, *#Chunks* = 7 305, and *#Files* = 669), we see only 86 lines in conflict (*#ConfLoC*) in three conflicting chunks (*#ConfChunks*) of three conflicting files (*#ConfFiles*). The developers needed around 40 hours to solve the merge conflict. Looking at one of the three files in conflict, *"src/node_crypto.cc"*, we could see that this file refers to five other non-conflicting files changed in the merge scenario. In addition, despite this file having only one conflicting chunk,

16. https://github.com/nodejs/node – COMMIT HASH: 61CCAF

other 87 chunks were changed in this file. Therefore, despite the merge conflict being small, the merge scenario changes are quite large and we found some dependencies among conflicting and non-conflicting code. These dependencies may explain the large amount of time needed to resolve this small conflict. In this direction, a recent study [11] accompanied by 7 developers resolving 10 merge conflicts noted that developers usually first look at files changed and then resolve the merge conflicts. Our results give nuance to this finding in a broader perspective and using different research methods.

Implications for researchers and tool builders: Merge scenario characteristics impact more on the merge conflict resolution time than merge conflict characteristics especially when dependencies among conflicting and non-conflicting code are found. Therefore, researchers should pay more attention to measures related to the merge scenario when exploring merge conflicts and tool builders should consider them when creating solutions to support practitioners on resolving merge conflicts.

Why does the number of chunks and the code complexity of the conflicting code show a negative correlation with the time needed to resolve merge conflicts? Before a data-driven answer, let us make an analogy. Once you find a very long method, it might be an indicator that this method does more than it should do. In several cases, one or more methods can be extracted from this very long method to make it more concise. The same is valid for commits. Previous studies have shown that committing small chunks of code make it easier to understand the code changes [3], [31]. In the context of merge conflicts, thirteen survey's participants reported that their strategy to avoid conflicts is simply based on small commits and merge often. As a data-driven and simpler discussion, our arguments here focus on three measures used in our regression model: *#Chunks*, *#LoC*, and *CodeComplexity*. It is worth remembering that the results of our regression model are valid for one variable only when the values of the others remain the same. Therefore, by increasing the number of chunks and keeping the same number of lines of code and the complexity of the conflicting code, we have small chunks which are much easier to understand and resolve [3], [31], [49]. In other words, a fast understanding of each chunk makes it easier to figure out which ones have a dependence on conflicting code. Hence, integrators can focus only on the dependent ones to resolve the merge conflict, also avoiding unexpected behaviors. In the end, we have merge scenarios easier to understand which will also reflect on a faster merge conflict resolution.

As a concrete example related to the number of chunks, we selected two merge scenarios from VSCode[17] with similar *#LoC* and *CodeComplexity*, but different values of *#Chunks*. In each of these merge scenarios, developers have changed around 10 thousand lines of code and have only one conflicting chunk with code complexity equal 2. However, while in the first scenario the code is divided into 1 458 chunks, in the second it was divided into 268 chunks. Taking all arguments into account that we discussed before, we assume that the first merge scenario would be easier to understand. In this particular case, it was true. The (same) integrator needed around 9 minutes to resolve the first merge scenario while she needed around 24 minutes to resolve the second. Looking at the code changes, we found nested scenarios in both cases. In other words, a developer was working in the source branch to add a feature while other developers were working in parallel to address other issues in other branches. At the end, their changes were integrated into the target branch before the subject source branch. At the end, most of the changes occurred in the target branch. As already discussed in Section 5.1, the location, content, and dependencies among non-conflicting code and conflicting code might have influenced the resolution time. It is worth mentioning that in both exemplified scenarios the changes in the source branch were quite simple, however, the merge scenario took around one week. Extracting some further information from GitHub, we found a possible reason that might have made these two merge scenarios longer. In both scenarios, the contributor was participating for the first time in the project. As Microsoft requests a contribution license agreement (CLA) for first-time contributors, it might have taken some time. Note that it does not impact on the merge conflict resolution, however, it might have influenced the time the merge scenario lasted, opening space for other integrations and introduction of merge conflicts.

To illustrate the code complexity of conflicting code, we selected two merge scenarios from next.js[18] with similar *#LoC* (387 vs. 661), *#Chunks* (107 vs. 147), *#ConfChunks* (9 vs. 9), and *#ConfFiles* (2 vs. 2), but different values of *CodeComplexity* (2 vs. 20). It is more intuitive to think that the conflicts resolution of the first scenario was faster than the second one since it is slightly smaller and less complex in terms of *#LoC* and *CodeComplexity*. However, while the same integrator took 47 minutes to resolve the first scenario, she took 18 minutes to resolve the second. Deeply looking at the code changes, we found that in the first scenario, there were changes in the yarn.lock and package.json files which only some of these changes were in conflict with. Ignoring the file content/extention/location (discussed in Section 5.1) and the slight difference in the number of chunks discussed above, we assume that the integrator fixed the conflicts fast and missed some dependencies. Once rebuilding and running the project, other dependency errors were found and she needed more time to complete the merge. In the second scenario, 8 of the 9 conflicts were in the taskfile.js with some nested loops. However, most of the changes are related to formatting, renaming, addition of new functions in the same region of code, and new parameters in asynchronous functions. All in all, for the first scenario, we assume that non-conflicting code influenced the long resolution time, for the second scenario, we assume that the type of the changes supported a faster resolution even though it looked more complicated in the beginning.

17. https://github.com/Microsoft/vscode – commit hashes: bd8108 and 44c395

18. https://github.com/vercel/next.js – commit hashes: f34262 and c92bde

Implications for practitioners: Committing small chunks of code makes the code understanding easier and, consequently, merge conflict resolution faster. Indeed, we are not the first ones to recommend developers committing small chunks of code. However, to the best of our knowledge, we did not find any study showing that committing small chunks of code makes the merge conflict resolution faster. Related to the code complexity of the conflicting code, we see that changes that look more complex on first sight might have simpler solutions depending on their type and location as discussed in Section 5.1.

5.4 Comparison of Previous Work Results

What factors related to the merge conflict resolution have been explored in the literature and how do our results relate to them? In Section 2, we presented related work showing the main differences of our study. Next, we compare our results with two other studies [24], [49]. As it is hard to quantitatively compare the results, we put all factors into Table 9 differing them as: (i) factors that make merge conflict resolution longer/harder (↗), (ii) factors that do not impact on the time/difficulty of resolving merge conflicts (→), and (iii) factors that make merge conflict resolution faster/easier (↘).

Nelson et al. [49] investigated nine factors while Ghiotto et al. [24] investigated four factors of which one is in both studies (see Table 9). Our study explores six factors from Nelson et al. and two factors from Ghiotto et al. In addition, we explore five factors not explored by previous studies. Our study is in-line with previous studies for most of the factors. The only exceptions are with factors: *complexity of conflicting lines of code* and *expertise in area of conflicting code*. We acknowledge that our result in this case is counter-intuitive at first sight. The justification for it is given by the setup and results of our study. We can use the same argumentation when explaining the negative correlation between the number of chunks of the merge scenario and the merge conflict resolution time (see Section 5.3). For short, in our study the increase of the complexity of conflicting lines of code has a negative influence on the merge conflict resolution time only when the other variables in the regression model (*#LoC*, *#Chunks*, *#Devs*, and *#ConfChunks*) remain fixed. This way, we would keep with the same merge-conflict and merge-scenario size which are factors that strongly impact the merge conflict resolution time. In other words, since the chunks remain small it is not a problem they are slightly more complex. Regarding the expertise in areas of conflicting code, prior knowledge in the conflicting files does not always help people to resolve their tasks [25]. Hence, we believe that developers with previous knowledge do not always resolve merge conflicts faster than developers without prior knowledge, as discussed in Section 5.2.

Related to the factor *time to resolve a conflict*, we consider the results from our survey comparable with the results of Nelson et al. [49]. We both asked for the developer's agreement with a similar statement "the more time it takes to resolve a conflict, the more difficult the conflict" in a 5-point Likert-type scale. Despite our developers' set being different from theirs, we got a median of 3 and a mean of 3.4, while they got a median of 3 and a mean of 2.82. Therefore, we both found that the time of resolving merge conflicts is perceived by the developers as a factor of difficulty of merge conflicts.

TABLE 9
Comparison of our results with previous studies

Factors	Nel.	Ghi.	Our Study
Factors directly related to merge conflicts			
Complexity of conflicting lines of code	↗	-	↘
Expertise in area of conflicting code	↗	-	→
Complexity of files with conflicts	↗	-	-
Number of conflicting lines of code	↗	↗	↗
Time to resolve a conflict	↗	-	↗
Atomicity of changesets in conflict	↗	-	-
Dependencies of conflicting code	↗	-	↗*
Number of files in the conflict	↗	-	↗
Non-functional changes in codebase	↗	-	-
Number of chunks in conflict	-	↗	↗
Language constructs involved in conflicting chunks	-	↗	-
Language constructs involving dependencies among conflicting chunks	-	↗	-
Formatting changes in the code in conflict	-	-	→
Factors indirectly related to merge conflicts			
Number of lines of code of the merge scenario	-	-	↗
Number of chunks of the merge	-	-	↘
Number of files of the merge scenario	-	-	↗
Number of developers involved in the merge scenario	-	-	↗

*Nel. and Ghi. stand for Nelson et al. [49] and Ghiotto et al. [24]. "↗" means that the factor makes the merge conflict resolution longer/harder, "→" means that the factor does not impact on the time/difficulty of resolving merge conflicts, and "↘" means that the factor makes the merge conflict resolution faster/easier. * highlights that the conclusion for this factor came by further analysis.*

The factor *non-functional changes in the code base* is similar to the factor *formatting changes in the code in conflict*. We prefer to classify them separately because the formatting changes in the code in conflict are only a subset of the non-functional changes in the code base. We consider that non-functional changes in the code base also include refactoring (e.g., renaming and reordering methods). Taking into account previous work [43] and survey participants, one of the main changes related to merge conflicts is refactoring. We agree with the developers surveyed by Nelson et al. [49] that non-functional changes in the code base (when refactorings are included) make the merge conflict resolution longer/harder. However, when considering only formatting changes in the code in conflict (i.e., excluding refactorings), it does not influence the merge conflict resolution time. In any event, it is worth remembering that the setup of our empirical study considers all variables together while previous work [49] asks developers individually.

Looking at Table 9 and considering the results of our effect-size analysis, we see that most factors that only we explore are related to the merge scenario size, i.e., not directly related to the merge conflict. In addition, *#Chunks*, *#Devs*, and *#LoC* are the three factors in our study with the highest effect on the merge conflict resolution time. Taking into account our cross-validation surveying developers, we are confident that, when resolving merge conflicts, developers usually are aware of the changes in the merge scenario. Being aware of the changes might take some time which influences the merge conflict resolution time. Together with our previous

discussions it shows that, in practice, merge scenario characteristics should be considered when exploring merge conflicts resolution.

Implications for practitioners and researchers: Merge conflict resolution theory and practice are in-line for most of the factors involved in both types of investigation. As mentioned, researchers and practitioners should also consider factors not directly related to merge conflicts (e.g., *#Chunks*, *#Devs*, and *#LoC*) in their analyses since these factors influence more the merge conflict resolution time than factors directly related to merge conflicts (e.g., *#ConfChunks* and *#ConfLoC*).

5.5 Reflections on the Merge Conflict Life-Cycle

Is rebasing a good solution for avoiding/dealing with merge conflicts? In our survey, 8 participants reported that they use rebases to integrate branches, whence, it is worth discussing this topic. Despite rebases drawn in a more linear evolutionary history view, rebase commits change the order code changes in fact occurred damaging the project history. Therefore, rebase should be used with care. The main difference between *git merge* and *git rebase* scenarios on the merge conflict resolution is that, in *git merge* scenarios, code changes are shown once all together and in *git rebase* scenarios, Git individually reapplies the commits off the to-do list. Hence, developers need to resolve conflicts first, and then Git continues to reapply the remaining commits. There is a chance that these resolutions would conflict with these remaining commits in the to-do list [36]. Ji et. al [35] investigated merge conflicts in Git rebases. Their results show that conflicts arise in 24.3% – 26.2% of rebase scenarios and no significant difference was found between the likelihood of conflicts arose given *git merge* from previous work [24], [65], [67] and the *git rebase* scenarios from their study. Considering that real-evolutionary history can be used to support developers on different types of tasks, we leave an open question to practitioners: *is the rebase really worth the damage in the project's history?*

Policy for resolving merge conflicts. In the third question of our survey, we asked the participants to share their experience of dealing with merge conflicts. Unfortunately, we did not get any answer explicitly reporting policies to deal with conflicts at project level. In fact, we searched in the GitHub page of each subject project, but we also did not find any report of it. Creating a policy might provide an organized and planned way to deal with conflicts. The creation of such a policy might be an opportunity for practitioners to improve their work-day tasks, as well as, for researchers to collect different experiences from practitioners and create a catalogue of best practices for dealing with merge conflicts.

Why is even a small improvement regarding the time to resolve single merge conflicts relevant for practitioners? The results we presented in Section 3.2 do not indicate that the time to resolve merge conflicts can be improved by a very large extent when solely referring to the observed variables. Even though this improvement would be enough to resolve 15.84% of the conflicting merge scenarios of our dataset, this is still a small fraction of them. Considering the difficulty of predicting merge conflicts as well as of creating strategies to resolve them faster, we argue that even a small improvement may help developers in practice. To this end, we see five relevant points of discussion to support our claim: (i) Developers may get frustrated when they are unaware of merge conflicts [49], it may diminish their satisfaction to work on a project since it is a tedious and error-prone task [39], [47]. (ii) Developers potentially need to handle multiple merge conflicts during the evolution of the project. For instance, the developers of the project d3[19] have resolved 286 conflicting merge scenarios; if it was possible to save five minutes per conflicting merge scenario, they would have saved around 3 full-time working days. (iii) Developers may lose focus on the tasks they were doing to resolve merge conflicts. As merge conflicts normally interfere with other developers' work, they have high priority and developers should stop what they are doing to resolve the conflicts. A break in the software development may lead them to lose track of previous tasks; (iv) By reducing the time/difficulty of merge conflicts, which is a key challenge for developers [39], [45], [49], may also decrease the chance of introducing an error during the merge. (v) An anticipated reduction of merge-conflict difficulty will also benefit other parts of the software project. A reduction in time to resolve a merge conflict is only the result of improving the developers' daily work. For example, when developers introduce one-off contributions, they may also introduce more modularized code that may result also in an improvement to the software architecture.

6 Threats To Validity

In this section, we discuss limitations as well as internal and external threats to the validity of our study.

Internal Validity. There are basically six main threats to internal validity. First, we did not measure the software development experience of developers integrating conflicting code. This is a limitation of our study, since more experienced developers may need less time to solve the same merge conflict than less experienced developers. In any case, we argue that large samples average this effect out. Second, we selected subject projects from different programming languages, hence, one language could have dominated our dataset (see the programming language of each subject project in our supplementary Web site [63]). We checked whether a programming language dominated half of our subject projects. Fortunately, it did not happen. Third, we rebuild merge scenarios by using the standard *git merge* command; if developers used other merge strategies, merge conflicts emerged that may differ from the ones we found. However, as developers normally use standard *git merge*, and avoid using external tools when merging [49], it does not affect our results considerably. Fourth, we could have classified merge scenarios based on their type (e.g., merge scenarios integrated using pull-requests and not using pull-requests) or based on the type of change (e.g., refactoring changes). We minimize this limitation by looking at characteristics of merge scenarios and merge conflicts and by differing formatting changes from other types of changes. Survey participants suggested that refactoring operations are conflict-prone. In that direction there is a previous work already [44]. Unfortunately, this is a limitation of our study and we suggest investigations in that direction in future work (see Section 7).

19. https://github.com/d3/d3

Fifth, we are not able to measure unexpected events that happened on the merge-conflict resolution (e.g., the developer responsible for solving the conflict had a break or asked other developers for support). As discussed in Sections 4 and 5, we believe that it does not change our results considerably. Sixth, there may be a better measure than *#SecondsToMerge*; changing the measure may also change our results. We are confident that we chose the best option for a post-hoc analysis based on the following arguments: 1) we could not find plausible reasons for only 11 out of the 100 longest conflicting scenarios of our dataset having taken so long. It suggests that unexpected events occur, however, they do not occur very often in practice; 2) comparing the values of *#SecondsToMerge* with the values of a previous study [11] that precisely measured the time spent to resolve merge conflicts, our results are not that far from their results. For instance, while developers in their study needed at least 389 seconds (6.48 minutes) to resolve the half of the longest merge conflicts (i.e., 5/10), developers from our subject projects took at least 697 seconds (11.62 minutes) to resolve the half of the longest merge conflicts (1 304/2 608); and, 3) survey participants agreed that they usually merge their code changes right after addressing an issue/task, and resolve conflicts right after they occur (see Section 4.2). Again, it does not mean that *#SecondsToMerge* indeed measures the time developers spent resolving merge conflicts, however, it is an evidence that subject developers normally proceed as we expect (i.e., what we aim to measure with *#SecondsToMerge*).

External Validity. External validity is threatened mainly by three factors. First, our restriction to Git and GitHub as a platform, the three-way merge pattern as well as to the set of measures. Generalizability to other platforms, projects, development patterns, and set of measures is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity [55]. While more research is needed to generalize to other version control systems, development patterns, and measures, we are confident that we selected and analyzed a practically relevant setting, measures estimating different software properties, and a substantial number of software projects from various domains, programming languages, longevity, size, and coordination practices. In addition, our filters applied during subject project selection guarantee, for instance, that we sampled active projects (see Section 3.1.2). Second, we are not able to retrieve information from binary files, hence, we may miss a piece of information from some merge scenarios. Unfortunately, we cannot do anything about that, however, the number of binary files is usually small in software projects. Third, the response rate of our survey study is very small (%2), it might be because of external factors that we are not able to control. For instance, (i) emails are not valid anymore given developers that used student emails and finished their studies or developers that used corporative emails and moved to another company or (ii) the great number of surveys in pandemic times as reported by a few developers that replied to our invitation. We tried to increase our response rate following guidelines of previous studies [46] [48] [57], however, it remained low. In any event, we got 140 answers for our survey, which is similar to previous studies investigating merge conflicts (e.g., [45], [49], and [39] had 162, 102, and 41 participants, respectively).

7 Conclusion and Future Work

In this study, we investigated the main challenges on merge conflict resolution with a two-phase study. First, we empirically looked at thousands of merge scenarios from 66 subject projects aiming at identifying factors that make the merge conflict resolution longer. Second, we minimized threats to validity of our empirical study and cross-validated non-intuitive results by surveying developers from subject projects. Furthermore, we manually checked hundreds of merge scenarios to dive deep in merge conflict resolutions. In Table 7, we presented 4 major challenges detailed into 11 sub-challenges and with a body of 19 solutions to minimize the emergence of merge conflicts as well as make conflict resolutions faster.

Despite several studies investigating merge conflict resolution, we are the first to investigate factors that influence the merge conflict resolution in practice with a triangulated approach (mining, survey and manual analyses). In fact, some of our results were already mentioned in the literature. However, our quantitative and qualitative results complement and add nuance to the recommendations from previous work.

As future work, we suggest a replication of this study involving other variables that we are not able to properly control, such as, the dependencies among files, chunks, conflicting files, and conflicting chunks. Such an analysis could add nuance to our findings from the manual analysis. With the knowledge acquired in this study we suggest six directions for future work: 1) Interview developers asking them about specific outlier merge scenarios and how merge conflicts impact on their mood and their contribution to the project. With this study we would find technical or social reasons that influence the resolution of merge conflicts. For instance, why did a merge conflict of a given characteristic take so long to be addressed? Was that because of lack of experience, lack of knowledge in the piece of code, dependence with other conflicts or changed code? 2) Interview developers aiming at building a taxonomy of which code they consider relevant to resolve merge conflicts. What does really matter when solving a merge conflict? How does the answer vary among developers? 3) An investigation on the impact of the merge conflict resolution time comparing merge scenarios with refactorings and without refactorings. Our study does not distinguish scenarios which involve refactoring or not, an empirical study emphasizing on refactoring operations could add nuance to our finding of the impact of non-conflicting changes on the merge conflict resolution time, as well as, bring justifications abroad our study. 4) A similar investigation on the merge conflict resolution time classifying merge scenarios into different types (e.g., merge scenarios integrated using pull-requests and the ones integrated without using pull-requests). Such a study could measure how efficient pull requests are to keep awareness among developers and its influence on the merge conflict resolution time. 5) a similar investigation classifying the different types of conflicts that might happen (e.g., lexical, structural, and semantic). As mentioned, our decision to cover several programming languages makes this investigation harder in practice since it is necessary an analysis in the abstract syntax tree (AST) and in several cases the conflicting code would not be enough. A study focusing on specific programming languages instead might be feasible in practice and provides an understanding on the influence of the conflict

type on its resolution. 6) a similar investigation using other models such as neural networks and deep learning techniques. Once our study presents a solid understanding of factors and challenges on merge conflict resolution, a study with other models could bring a more accurate outcome for the empirical analysis.

Acknowledgment

This work was partially supported by CNPq (grant 290136/2015-6). Apel's work has been funded by the German Research Foundation (AP 206/14-1). We would like to thank you Angelika Schmid and Zohaib Brohi for support on initial versions as well as in the statistical analysis.

References

[1] P. Accioly, P. Borba, and G. Cavalcanti. "Understanding Semi-structured Merge Conflict Characteristics in Open-source Java Projects". In Empirical Software Engineering, vol. 23(4), Springer, pp. 1–35, 2017.

[2] I. Ahmed, C. Brindescu, U. Mannan, C. Jensen, A. Sarma. "An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts". In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM/IEEE, pp. 58–67, 2017.

[3] A. Alali, H. Kagdi, J. I. Maletic. "What's a Typical Commit? A Characterization of Open Source Software Repositories". In Proceedings of the International Conference on Program Comprehension (ICPC), IEEE, pp. 182–191, 2008.

[4] L. Amaral, M. Oliveira, W. Luz, J. Fortes, R. Bonifacio, D. Alencar, E. Monteiro, G. Pinto, D. Lo. "How (Not) to Find Bugs: The Interplay Between Merge Conflicts, Co-Changes, and Bugs". In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp. 441-452, 2020.

[5] S. Apel, O. Leßenich, and C. Lengauer. "Structured Merge with Autotuning: Balancing Precision and Performance". In Proceedings of the International Conference on Automated Software Engineering (ASE). ACM, pp. 120–129, 2012.

[6] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. "Semistructured Merge: Rethinking Merge in Revision Control Systems". In Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 190–200, 2011.

[7] J. Biehl, M. Czerwinski, G. Smith and G. Robertson "FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams". In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), ACM, pp. 1313–1322, 2007.

[8] H. Borges and M.T. Valente. "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform". In Journal of Systems and Software (JSS), vol. 146 (1), pp. 112–129, 2018.

[9] C. Brindescu, I. Ahmed, C. Jensen, A. Sarma. "An Empirical Investigation into Merge Conflicts and Their Effect on Software Quality". In Empirical Software Engineering, Springer, vol. 25, pp. 562–590, 2020.

[10] C. Brindescu, I. Ahmed, R. Leano, A. Sarma. "Planning for Untangling: Predicting the Difficulty of Merge Conflicts". In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 801–811, 2020.

[11] C. Brindescu, Y. Ramirez, A. Sarma, C. Jensen. "Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective". In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp. 534–545, 2020.

[12] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. "Proactive Detection of Collaboration Conflicts". In Proceedings of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE). ACM, pp. 168–178, 2011.

[13] G. Cavalcanti, P. Borba, and P. Accioly. "Evaluating and Improving Semistructured Merge". In Proceedings of the ACM on Programming Languages (OOPSLA), ACM, 59, pp.1–27, 2017.

[14] C. Chatfield. "The Analysis of Time Series: An Introduction". 5th Edition, Chapman and Hall, 1996.

[15] J. Cohen. "Statistical Power Analysis for the Behavioral Sciences". Elsevier Inc., p. 490, 1977.

[16] C. Costa, J. Figueiredo, G. Ghiotto, L. Murta. "Characterizing the Problem of Developers' Assingment for Merging Branches". In International Journal of Software Engineering and Knowledge Engineering (SEKE), World Scientific, vol. 24, pp. 1489–1508, 2014.

[17] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository". In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 1277–1286, 2012.

[18] P. Dewan and R. Hegde. "Semi-synchronous Conflict Detection and Resolution in Asynchronous Software Development". In Proceedings of the Conference on European Computer Supported Cooperative Work (ECSCW). ACM, pp. 159–178, 2007.

[19] K. Dias, P. Borba, M. Barreto. "Understanding predictive factors for merge conflict". In Information and Software Technology (IST), vol. 121, Science Direct, pp. 1–12, 2020.

[20] R. Falk and N. Miller. "A Primer for Soft Modeling". In University of Akron Press, 1992.

[21] L. Fahrmeir, T. Kneib, S. Lang, and B. Marx. "Regression: Models, Methods and Applications". Springer, pp. 698, 2013.

[22] Y. Fang, H. Sun, G. Li, R. Zhang, and J. Huai. "Context-Aware Result Inference in Crowdsourcing". In Information Sciences, vol. 460 (1), pp. 346–363, 2018.

[23] A. Field. "Discovering Statistics Using IBM SPSS Statistics". Sage, Fifth Edition, p.1104, 2017.

[24] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek. "On the Nature of Merge Conflicts a Study of 2,731 Open Source Java Projects Hosted by Github". In Transactions on Software Engineering (TSE), vol. 99 (1), IEEE, pp. 1–25, 2018.

[25] N. Goode, J.F. Beckman. "You Need to Know: There is a Causal Relationship Between Structural Knowledge and Control Performance in Complex Problem Solving Tasks". In Intelligence, Elsevier, vol 38(3), pp. 345–352, 2010.

[26] R. A. Gordon. "Regression Analysis for the Social Sciences". Routledge, p.566, 2015.

[27] G. Gousios, M. Pinzger, and A. Deursen. "An Exploratory Study of the Pull-based Software Development Model". In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 345–355, 2014.

[28] G. Gousios, M.A. Storey, and A. Bacchelli. "Work Practices and Challenges in Pull-based Development: The Contributor's Perspective". In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 285–296, 2016.

[29] M. L. Guimarães and A. R. Silva. "Improving Early Detection of Software Merge Conflicts". In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 342–352, 2012.

[30] M. Hague, A. Lin, C. Hong. "CSS Minification via Constraint Solving". In Transactions on Programming Languages and Systems (TOPLAS). ACM, vol. 41(2), pp. 12–76, 2017.

[31] A. Hindle, D. M. German, R. Holt. "What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits". In Proceedings of the International Working Conference on Mining Software Repositories (MSR). ACM, pp. 99–108, 2008.

[32] P. W. Holland. "Statistis and Causal Inference". Journal of the American Statistical Association, vol. 8 (81), Taylor & Francis, Ltd, pp. 945–960, 1986.

[33] G. James, D. Witten, T. Hastie, and R. Tibshirani. "An Introduction to Statistical Learning". Springer, p.426, 2013.

[34] H. Z. Jerrold. "Significance Testing of the Spearman Rank Correlation Coefficient". Journal of the American Statistical Association, vol. 67 (339), Taylor & Francis, Ltd, pp. 578–580, 1972.

[35] T. Ji, L. Chen, X. Yi, and X. Mao. "Understanding Merge conflicts and Resolutions in Git Rebases". In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp. 70-80, 2020.

[36] S. Just, K. Herzig, J. Czerwonka, and B. Murphy. "Switching to Git: The Good, the Bad, and the Ugly". In Proceeding of the International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 400–411, 2016.

[37] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, D. Damian. "The Promises and Perfils of Mining GitHub". In Proceedings of the Working Conference on Mining Software Repositories (MSR), ACM, 92–101, 2014.

[38] B.K. Kasi and Anita Sarma. “Cassandra: Proative Conflict Minimization through Optimized Task Scheduling”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 732–741, 2013.

[39] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. “Indicators for Merge Conflicts in the Wild: Survey and Empirical Study”. Automated Software Engineering, vol. 25 (2), Springer, pp. 1–35, 2017.

[40] T. Levine and C. Hullet. “Eta Squared, Partial Eta Squared, and Misreporting of Effect Size in Communication Research”. In Human Communication Research, vol. 28(4), pp. 612–625, 2002.

[41] Y. Li and N.J. Belkin. “A Faceted Approach to Conceptualizing Tasks in Information Seeking”. Information Processing and Management, vol. 44, pp. 1822–1837, 2008.

[42] J. Liu, J. Gwizdka, C. Liu, and N.J. Belkin. “Predicting Task Difficulty for Different Task Types”. In Proceedings of the American Society for Information Science and Technology (ASIST). Wiley, pp.1–10, 2010.

[43] W. Mahmood, M. Chagama, T. Berger, R. Hebig. “Causes of Merge Conflicts: A Case Study of ElasticSearch”. In Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, pp.1–9, 2020.

[44] M. Mahmoudi, S. Nadi and N. Tsantalis, “Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts”. In Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp. 151–162, 2019.

[45] S. McKee, N. Nelson, A. Sarma, and D. Dig. “Software Practitioner Perspectives on Merge Conflicts and Resolutions”. In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 467–478, 2017.

[46] R. Mello and G. Travassos.“Surveys in Software Engineering: Identifying Representative Samples”. In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM, article 55, pp. 1–6, 2016.

[47] T. Mens. “A State-of-the-Art Survey on Software Merging”. In IEEE Transactions on Software Engineering. IEEE, 28(5):449–462, 2002.

[48] J. Molléri, K. Petersen, and E. Mendes. “Survey Guidelines in Software Engineering: An Annotated Review”. In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM, article 58, p.1–6, 2016.

[49] N. Nelson, C. Brindescu, S. McKee, A. Sarma, D. Dig. “The Lifecycle of Merge Conflicts: Processes, Barriers, and Strategies”. Empirical Software Engineering, Springer, vol. 24, pp. 2863–2906, 2019.

[50] A. Nieminen. “Real-time Collaborative Resolving of Merge Conflicts.” In Proceedigns of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), IEEE, pp. 540–543, 2012.

[51] M. Owhadi-Kareshk, S. Nadi, and J. Rubin. “Predicting Merge Conflicts in Collaborative Software Development”. In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 1–11, 2019.

[52] Y. Sakamoto, S. Matsumoto, S. Tokunaga, C. Saiki, M. Nakamura. “Empirical Study on Effects of Script Minification and HTTP Compression for Traffic Reduction”. In Proceedings of the International Conference on Digital Information, Networking, and Wireless Communications (DINWC). IEEE, pp. 127–132, 2015.

[53] A. Sarma, D.F. Redmiles, A. van der Hoek. “Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes”. IEEE Transactions on Software Engineering. IEEE, vol. 38(4), pp. 889–908, 2012.

[54] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang. “IntelliMerge: A Refactoring-Aware Software Merging Technique”. In Proceedings of the ACM Programmaing Languages (OOPSLA), ACM, 170, pp. 1–28, 2019.

[55] J. Siegmund and J. Schumann. “Confounding Parameters on Program Comprehension: A Literature Survey”. Empirical Software Engineering vol. 20 (4), pp. 1159–1192, 2015.

[56] L. Singer, F. Figueira Filho, B. Cleary, C. Treude, M.A. Storey, and K. Schneider. “Mutual Assessment in the Social Programmer Ecosystem: An Empirical Investigation of Developer Profile Aggregators”. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW). ACM, pp. 103–116, 2013.

[57] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird and T. Zimmermann, “Improving Developer Participation Rates in Surveys”. In Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), IEEE, pp. 89–92, 2013.

[58] K. Stol, P. Ralph and B. Fitzgerald, “Grounded Theory in Software Engineering Research: A Critical Review and Guidelines,” In Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp. 120–131, 2016.‘

[59] M.A. Storey, A Zagalsky, F. Figueira Filho, L. Singer, D. M. German. “How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development”. In IEEE Transactions on Software Engineering. vol. 43 (2), pp. 185–204, 2016.

[60] A. Strauss and J. Corbin. “Grounded Theory in Practice”, Sage, p. 280, 1997.

[61] C. Tantithamthavorn, and A. Hassan. “An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges”. In Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SIEP), ACM, pp. 286–295, 2018.

[62] J. Tsay, L. Dabbish, and J. Herbsleb. “Influence of Social and Technical Factors for Evaluating Contribution in GitHub”. In Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 356–366, 2014.

[63] G. Vale, C. Hunsen, E. Figueiredo, S. Apel. “Challenges of Resolving Merge Conflicts: A Mining and Survey Study – Supplementary Web site” Available: https://gustavovale.github.io/merge-conflict-resolution-analysis/, [Accessed: 25/07/2021].

[64] G. Vale, A. Schmid, A., Santos, E. Almeida, and S. Apel. “On the Relation Between Github Communication Activity and Merge Conflicts. In Empirical Software Engineering, vol 25, pp. 402–433, 2020.

[65] R. Yuzuki, H. Hate, and K. Matsumoto. “How We Resolve conflict: An Empirical Examination of Method-level Conflict Resolution”. In Proceedings of the International Workshop on Software Analytics (SWAN), IEEE, pp. 21–24, 2015.

[66] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. “Mining Version Histories to Guide Software Changes”. In Proceedings of the International Conference on Software Engineering (ICSE). IEEE, pp. 563–572, 2004.

[67] T. Zimmermann. “Mining Workspace Updates in CVS”. In Proceedings of the International Workshop on Mining Software Repositories (MSR), IEEE, pp. 11–11, 2007.