Pull Requests Integration Process Optimization: An Empirical Study

Agustín Olmedo, Gabriela Arévalo, Ignacio Cassol, Quentin Perez, Christelle Urtado, Sylvain Vauttier

Pull Requests Integration Process Optimization: An Empirical Study

Agustín Olmedo[1][0000−0003−2844−6816], Gabriela Arévalo[2], Ignacio Cassol[1][0000−0002−6309−7503], Quentin Perez[3][0000−0002−1534−4821], Christelle Urtado[3][0000−0002−6711−8455], and Sylvain Vauttier[3][0000−0002−5812−1230]

[1] LIDTUA (CIC), Facultad de Ingeniería, Universidad Austral, Buenos Aires, Argentina
[2] DCyT (UNQ), CAETI (UAI), CONICET, Buenos Aires, Argentina
[3] EuroMov Digital Health in Motion, Univ. Montpellier & IMT Mines Ales, Ales, France

Abstract. Pull-based Development (PbD) is widely used in collaborative development to integrate changes into a project codebase. In this model, contributions are notified through Pull Request (PR) submissions. Project administrators are responsible for reviewing and integrating PRs. In the integration process, conflicts occur when PRs are concurrently opened on a given target branch and propose different modifications for a same code part. In a previous work, we proposed an approach, called *IP Optimizer*, to improve the Integration Process Efficiency (IPE) by prioritizing PRs. In this work, we conduct an empirical study on 260 open-source projects hosted by GitHub that use PRs intensively in order to quantify the frequency of conflicts in software projects and analyze how much the integration process can be improved. Our results indicate that regarding the frequency of conflicts in software projects, half of the projects have a moderate and high number of pairwise conflicts and half have a low number of pairwise conflicts or none. Futhermore, on average 18.82% of the time windows have conflicts. On the other hand, regarding how much the integration process can be improved, *IP Optimizer* improves the IPE in 94.16% of the time windows and the average improvement percentage is 146.15%. In addition, it improves the number of conflict resolutions in 67.16% of the time windows and the average improvement percentage is 134.28%.

Keywords: Collaborative Software Development · Distributed Version Control System · Pull-based Development · Pull Request · Integration Process Efficiency · Software Merging · Merge Conflicts.

1 Introduction

Distributed Version Control Systems (DVCSs) have transformed collaborative software development [26]. Each developer has a personal local copy of the entire project history. Changes are first applied by developers to their local copy

and then integrated into a new shared version. If they exists, conflicts between concurrent changes must be solved during this integration process [23].

The Pull-based Development (PbD) model is widely used in collaborative software development [15, 17]. In this model, developments are performed on new branches, forked from the latest version currently available on the main development branch [7]. Contributors work separately using individual copies of project files. When the development is finished, they submit a *Pull Request* (PR) so that the core team members (aka project administrators) integrate it into the main development branch.

Project administrators must review and integrate opened PRs in the project. While trying to integrate different PRs, project administrators might have to deal with conflicting changes between them. Conflicting changes are changes that modify the same part of the code (*i.e.,* same lines of the same file) in the versions to be integrated. When two PRs are concurrently opened on a given target branch, proposing different modifications for identical code parts, a *pairwise conflict* exists. In such case, only one PR can be integrated automatically, while the other requires conflict resolution (CR).

The integration process is defined as the PR integration sequence according to the chronological order. In a previous work, we proposed an approach to improve the Integration Process Efficiency (IPE) by automatically prioritizing Pull Request integration [24], considering integration process efficiency as the fact that for a given integration cost (*i.e.,* number of pairwise conflicts to be solved) the highest possible gain is reached (*i.e.,* the largest number of PRs are integrated) and taking as an hypothesis that all pairwise conflict resolutions have the same mean cost. In this paper, we refer to this work as Integration Process Optimizer (*IP Optimizer*).

In this paper, we present an empirical study conducted on 260 open-source projects hosted by GitHub[4] that use PRs intensively with the aim of quantifying the frequency of conflicts in software projects and analyzing how much the integration process can be improved in software projects. We perform this analysis with a sliding time window approach. For each time window, we extract the historical integration sequence corresponding to the integrated PRs in the window ordered chronologically by integration date and apply *IP Optimizer* to that set of PRs.

Our results indicate that, regarding the frequency of conflicts in software projects, half of the projects have a moderate to high number of pairwise conflicts and the other half have a low number of pairwise conflicts or none. Futhermore, on average 18.82% of the time windows have conflicts. On the other hand, regarding how much the integration process can be improved, *IP Optimizer* improves the integration process compared to the historical one by 94.16% of the time windows and the average improvement percentage is 146.15%. In addition, it improves the number of conflict resolutions in 67.16% of the time windows and the average improvement percentage is 134.28%.

[4] https://github.com/

We also perform the analysis for time windows of different sizes and confirm our intuition that the larger the size of the window, the greater the percentage of time windows with conflicts but also found that the larger the size of the time window the better *IP Optimizer* improves the integration process as compared to the historical one. This verifies what the practice of continuous integration indicates [6]: integrating the PRs quickly is the best way to avoid conflicts and this make the integration process more efficient.
The remainder of this article is structured as follows. Section 2 includes background knowledge and definitions. Section 3 introduces the research questions. Section 4 explains the design of the empirical study. Section 5 reports the results. Section 6 discusses the results and answers the research questions. Section 7 contains the threats to validity. Related works are included in Section 8 before providing our conclusions and perspectives for this work.

2 Background

In collaborative software development, contributors choose or are provided a *ticket* to deal with. Tickets are stored in an issue tracking system (IST) such as Jira[5] and define a task to correct, maintain or improve the software. To take the ticket into account, a contributor creates a dedicated development branch in his/her local repository from the latest software version (fetched from the main development branch in the shared project repository), check-out this version to get a working copy and make the needed changes. These changes can then be committed on the dedicated development branch in the local repository to enact them. When the development is finished, the versions locally commited on the dedicated development branch are checked-in (pushed) to the remote shared repository. A last step is to submit a PR to ask the project administrators to merge the changes into the project's main development branch.
A PR is a request to the project administrators to pull versions from a branch to another one. Therefore, a PR contains a source branch from which versions are pulled from and a target branch to which versions are pushed. Project administrators must review the opened PRs. As a result of the review process, PRs can be closed as accepted or rejected.
The accepted PRs are integrated, that is, the changes commited on the source branch since it was forked from the target branch, *i.e.,* the changes in the last version on the source branch, are merged into the target branch. If versions have been commited on the target branch since the source branch of the PR was forked, there may be merge conflicts since the head version of the source branch do not derive anymore from the head version of the target branch. We consider changes to be text file modifications like Git does [8]. So, merge conflicts between changes are due to the fact that the same line(s) of the same file are modified in both branches.
When there are merge conflicts between the changes of the source branch and the changes of the target branch of a PR, the PR is considered as a conflicting

[5] https://www.atlassian.com/en/software/jira

PR. Conflicting PRs require a conflict resolution to integrate their changes into the target branch whereas for unconflicting PRs, the changes are automatically integrated. In the PbD model, a pairwise conflict exists between two PRs when the integration of a PR would entail a future conflict when integrating the other one to their shared target branch (and reciprocally). Figure 1 shows a simple pairwise conflict scenario where two PRs conflict with each other.

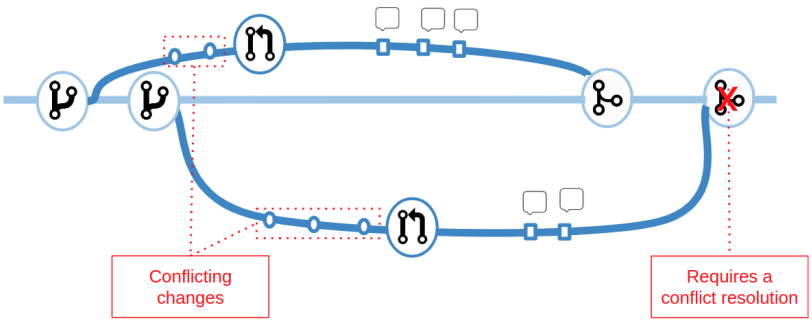

Fig. 1. Pairwsie conflict scenario where two PRs conflict with each other.

3 Research Questions

The goal of this work is to quantify the frequency of conflicts in software developments based on DVCSs and to analyze to what extent the integration process can be optimized in these projects. To achieve this goal, we aim to answer the following research questions.

Research Question 1 (RQ1): To what extent do conflicts occur in software projects managed with DVCS?
In order to answer this question, we obtain the pairwise conflicts from the history of projects that use PRs intensively and analyze how many pairwise conflicts there are in the history of each project and how they are distributed over time considering integration time windows of different sizes.

Research Question 2 (RQ2): How much can the integration process of software projects be optimized?
To answer this question, we perform a sliding window analysis. Specifically, we obtain the number of time windows in which *IP Optimizer* improves the IPE and the number of conflict resolutions compared to the historical integration sequence and in what proportion this improvement occurs. For this, we obtain for

each time window the historical integration sequence and the PR group integration sequence obtained by *IP Optimizer* for the set of PRs in the time window. Afterwards, we calculate the IPE and the number of conflict resolutions corresponding to the historical integration sequence and the PR group integration sequence obtained by *IP Optimizer*. Based on this information, we obtain the number of time windows in which *IP Optimizer* improves the IPE and the number of conflict resolutions with respect to the history of each project. We also calculate the IPE improvement percentage and the improvement percentage of the number of conflict resolutions that *IP Optimizer* achieves with respect to the historical one.

4 Empirical Study Setup

We set up an empirical study on projects hosted in GitHub that use PRs intensively. We extracted the information on projects and their PRs from the GHTorrent [6] dataset [14]. All the scripts and data used in this study are available in our online appendix [1].

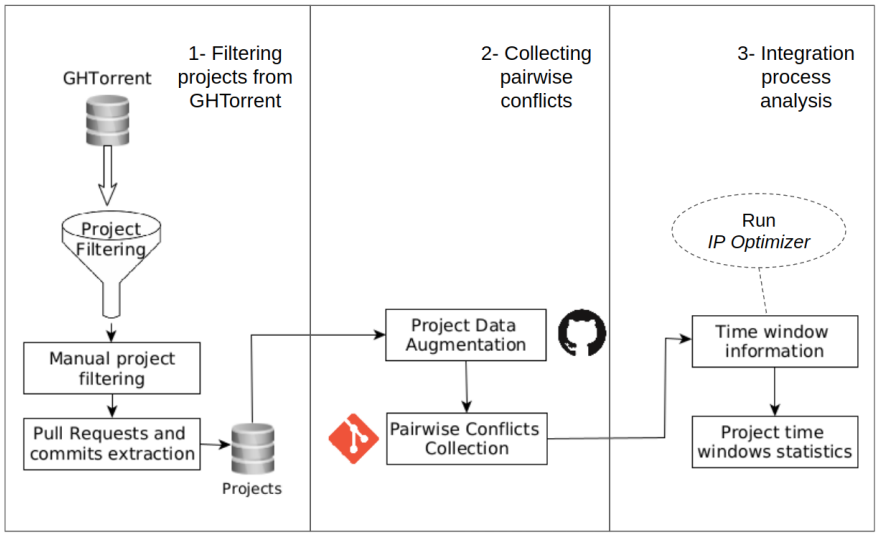

Fig. 2. Study design.

Figure 2 illustrates the study design consisting of three stages. The first stage filters and extracts the project data from the GHTorrent dataset according to four selection criteria. The second stage obtains all the pairwise conflicts of each of the projects. The third stage performs an integration process analysis that

[6] https://ghtorrent.org/

consists of extracting N-days time windows and comparing the related historical integration sequence data with the PR group integration sequence data for the time window obtained using *IP Optimizer*. Then, from the time windows data, we calculate statistical data for each project.
We conduct the reproducibility assessment for our empirical study according to the methodological framework of González-Barahona and Robles [13]. For reasons of space we include it in the online appendix [1].

4.1 Stage 1: Project Filtering & Project Data Extraction

Project selection criteria. We defined four selection criteria that are described below. Figure 3 shows the filter funnel applied to the GHTorrent dataset.

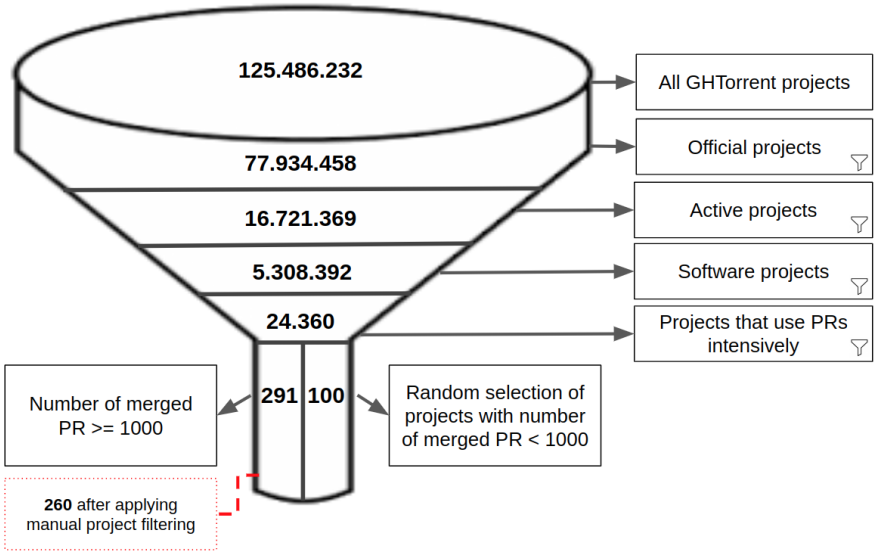

Fig. 3. GHTorrent filter funnel and filtering criteria.

Official projects. Consider only the official / original projects, that is, not the forked projects. The reason of this filter is that forked projects are not project by themselves; they are usually only used to contribute to the official projects.

Active projects. Consider only active projects. Many inactive projects are temporary or discontinued projects. Moreover, most of these are deleted after a period of time and deleted projects are not accessible. Thus, we filter deleted projects or projects that have not had a commit in the last year[7].

[7] Since the GHTorrent dataset has data up to 2019-06-01, last year corresponds to 2018-06-01 and after.

Software projects. Consider only software projects. Software projects involve a software development process and thus are of interest to evaluate the integration process. So, we filter projects by their main programming language. We select the projects that are implemented in the 25 most popular programming languages according to the PYPL ranking [2]: Python, Java, JavaScript, C#, C/C++, PHP, R, TypeScript, Objective-C, Swift, Matlab, Kotlin, Go, Rust, Ruby, VBA, Ada, Scala, Visual Basic, Dart, Abap, Lua, Groovy, Perl, Julia.

Projects that use PRs intensively. Consider only the projects that use PRs intensively, that is, projects that have at least 100 PRs. Therefore, we filter projects that have less than 100 PRs.

We are interested in projects that use PRs. On GHTorrent there are 704711 projects that have at least one PR, of which there are 208949 projects (29.65%) that have only one PR. In addition, most of the projects (96.54%) have less than 100 PRs. The projects that have few PRs do not provide much relevant information for our analysis since, in many cases, they do not have pairwise conflicts or they have very few. That is why we decided to take projects that have at least 100 PRs to avoid data disturbances.

Since getting pairwise conflicts for a project requires a lot of resources and time, we consider to study projects that have more than or equal 1000 merged PRs because that is a large enough number of PRs that will allow us to find all kinds of scenarios and the number of projects is significant for evaluating the integration process. In addition, these merged PRs correspond to PRs where the source branch and the target branch are in the same repository because it is not possible to obtain the historical pairwise conflicts of merged PRs from the project repository where the source branch is in a different repository.

To avoid any kind of bias by only analyzing in detail the projects that have more than 1000 merged PRs, we made a random selection of 100 projects that have less than 1000 merged PRs. In this way we verify that the results remain the same even if we take a larger set of projects.

Manual project filtering. After applying the filters described above automatically, we obtain 291 projects with at least 1000 PRs. Some of these projects do not meet the proposed criteria because the GHTorrent dataset has outdated data or because the main programming language defined on GitHub is not correct. Specifically, we find that there are projects that have defined one of the programming languages that interest us as the main programming language, but the project is not a software project. On the other hand, there are projects that are no longer on GitHub and the project history cannot be accessed. This is why we had to manually apply the *software projects* and *active projects* filters on the 291 projects obtained with the automatic filtering. As a result of the manual filtering, there are 260 projects that meet the defined criteria.

Pull Requests and commit extraction. We extract the PRs and commits of the selected projects from the GHTorrent database to be able to obtain the pairwise conflicts in stage 2.

4.2 Stage 2: Collecting Pairwise Conflicts

Project Data Augmentation. GHTorrent is missing information needed to obtain pairwise conflicts. In particular, we need the *default branch* of the project and the *target branch* of the PRs. So we use the GitHub API to do so.

Pairwise conflicts collection. We obtain the pairwise conflicts for merged PRs of each project. The conditions for the existence of a pairwise conflict are:

1. The target branch of the involved PRs must be the same.
2. PRs must be open simultaneously for a period of time.
3. The changes applied in the versions of the source branches of the involved PRs must conflict.

Therefore, to obtain the pairwise conflicts of a given PR, we search for PRs that have the same target branch defined (first condition) and that have been opened or closed while the PR is open (second condition). Figure 4 shows the PRs that are candidates to have a pairwise conflict with a given PR according to the second condition. Then, we check if there are conflicts between the given PR and each candidate PRs (third condition) by performing an in-memory merge [8] of the head version [9] of the source branch of each PR.

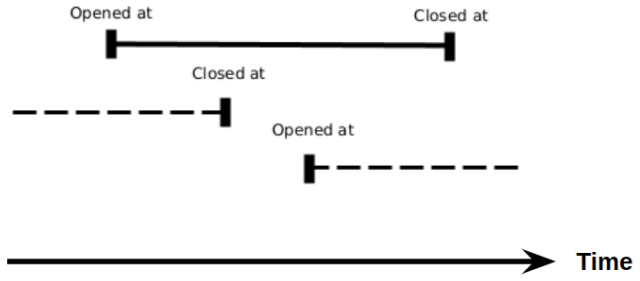

Fig. 4. Candidate PRs (dashed lines) to have a pairwise conflict with a given PR (solid line) according to the second condition.

[8] https://git-scm.com/docs/git-merge
[9] Version in which the PR was created

4.3 Stage 3: Integration Process Analysis

Based on project data extracted from GHTorrent and the pairwise conflicts, we conduct a similar analysis to the one we performed on the Antlr4 project in our previous work [24]. We extract historical integration sequences from N-days time windows. An historical integration sequence corresponds to the merged PRs in a given time window sorted by the chronological order in which the PRs were merged. We also calculate the PR group integration sequence related to the time window using *IP Optimizer*. *IP Optimizer* receives as input a set of PRs and returns a sequence of groups of PRs where the PRs of each group do not conflict with each other and the number of groups is minimal.

Given an integration sequence, we obtain the cumulative gain G_k and the cumulative cost C_k for each integration step k as shown in (1) where g_i is the number of PRs integrated in the integration step i and c_i is the number of pairwise conflicts solved in the integration step i.

$$\begin{aligned} G_k &= \textstyle\sum_{i=1}^{k} g_i \qquad k = 1, ..., \#PRs \\ C_k &= \textstyle\sum_{i=1}^{k} c_i \qquad k = 1, ..., \#PRs \end{aligned} \tag{1}$$

Next, we map the integration steps onto a cumulative gain / cumulative cost plot. Each coordinate (x,y) is an integration step and the line between points corresponds to the cost of the integration step. The trajectory obtained models the integration sequence. The area under the trajectory represents the IPE because a larger area means that a higher gain is achieved at a lower cost. Figure 5 shows an example of an integration trajectory.

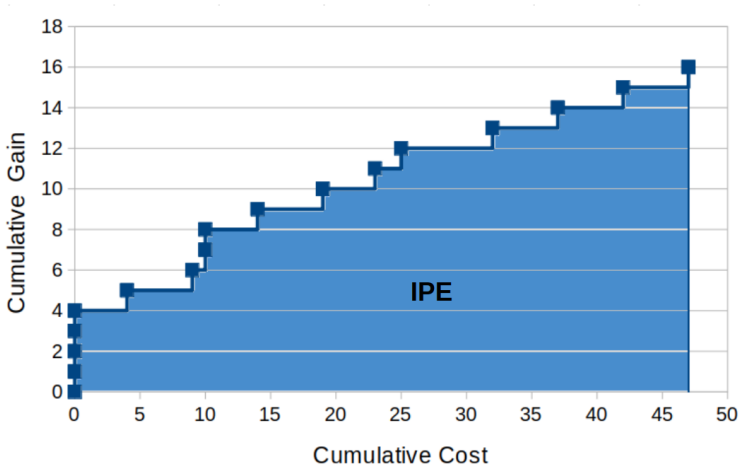

Fig. 5. Example of an integration sequence mapped onto a cost/gain trajectory. The area under the trajectory represents the IPE.

As the popularity of PRs grew over time [39, 15], projects incorporated their use at a certain point in their lives. That is why we start calculating the time windows of a project from the merge date of the first PR and we calculate them until the date we have data. Figure 6 shows how we extract the historical integration sequences of a given project.

Fig. 6. Historical integration sequences extraction

Time window information. For each project, we extract 7-days, 14-days, 28-days, 60-days and 90-days time windows. From each time window we calculate the following information.

Number of PRs (#PRs). It is the number of PRs merged between the start and end dates of the time window.

Number of pairwise conflicts (#PC). It is the number of pairwise conflicts that involve PRs merged in the time window. In (2) it is shown how the number of pairwise conflicts is calculated, where n is the number of merged PRs in the time window.

$$\begin{aligned} &\#PC = \textstyle\sum_{j=i+1}^{n} Exists\_PC(PR_i, PR_j) \quad i = 1, ..., n-1 \\ &Exists\_PC(P, Q) = \begin{cases} 1 & \text{if exists pairwise conflict between PR P and PR Q} \\ 0 & otherwise \end{cases} \end{aligned} \tag{2}$$

Number of historical CRs (Historical_CR). It is the number of conflict resolutions that actually occurred in the history of the project for the time window. In (3) it is shown how we calculate the number of conflict resolutions where n is the number of integration steps and c_k is the cost for the integration step k.

$$\begin{aligned} &CR = \textstyle\sum_{k=1}^{n} f(c_k) \\ &f(c) = \begin{cases} 1 & c > 0 \\ 0 & otherwise \end{cases} \end{aligned} \tag{3}$$

Number of optimized CRs (Optimized_CR). It is the resulting number of conflict resolutions in the PR group integration sequence, obtained by *IP Optimizer*, for the time window. Formula (3) is also used to calculate this value.

CRs improvement percentage (CR_IP). It is the percentage of improvement in the number of conflict resolutions between the PR group integration sequence obtained by *IP Optimizer* and the historical one. It is calculated using formula (4).

$$CR\_IP = \left(\frac{Historical\_CR}{Optimized\_CR} - 1 \right) * 100 \tag{4}$$

Historical IPE. It is the Integration Process Efficiency of the historical integration sequence for the time window. In (5) it is shown how we calculate the IPE, where n is the number of integration steps, G_k is the cumulative gain for the integration step k and C_k is the cumulative cost for the integration step k. Note that this formula calculate the area under the cost/gain trajectory by adding the area of the rectangles where the base is the integration step cost and the height is the cumulative gain of the step.

$$IPE = \sum_{i=0}^{n-1} G_i * (C_{i+1} - C_i) \tag{5}$$

Optimized IPE. It is the Integration Process Efficiency of the PR group integration sequence, obtained by *IP Optimizer*, for the time window. This value is also calculated using formula (5).

IPE improvement percentage (IPE_IP). It is the improvement percentage of the IPE between the PR group integration sequence obtained by *IP Optimizer* and the historical one for the time window. It is calculated using formula (6).

$$IPE\_IP = \left(\frac{Optimized\_IPE}{Historical\_IPE} - 1 \right) * 100 \tag{6}$$

Project time windows statistics. Once the information of the time windows of the projects has been calculated, we calculate the following statistical information of each project by time window size (7-days, 14-days, 28-days, 60-days and 90-days).

Number of time windows. It is the number of time windows between the merge date of the first merged PR to the date we have data.

Number and percentage of time windows without conflicts. Corresponds to the number and the percentage of time windows in which there is no pairwise conflict.

Number and percentage of time windows with conflicts. Corresponds to the number and the percentage of time windows in which there is at least one pairwise conflict.

The following statistical information is calculated considering only the time windows that have pairwise conflicts.

Mean number of pairwise conflicts. It is the mean number of pairwise conflict for a time window.

Number and percentage of time windows that improve the historical number of CRs. It is the number and percentage of time windows in which the number of conflict resolutions obtained by *IP Optimizer* is better than the historical one.

Number and percentage of time windows that maintain the historical number of CRs. It is the number and the percentage of time windows in which the number of conflict resolutions obtained by *IP Optimizer* is equal to the historical one.
It should be noted that since *IP Optimizer* obtains groups of PRs that do not conflict with each other, there will only be conflict resolutions for each group that is integrated except the first one that is integrated without conflicts. Since the number of groups calculated by *IP Optimizer* is minimal, then the number of conflict resolutions of the integration process obtained by *IP Optimizer* is minimal. This is why we do not calculate a measure where *IP Optimizer* worsens the number of conflict resolutions compared to the historical integration process.

Number and percentage of time windows that improve the historical IPE. It is the number and percentage of time windows in which the IPE obtained by *IP Optimizer* is better than the historical IPE.

Number and percentage of time windows that worsen the historical IPE. It is the number and the percentage of time windows in which the IPE obtained by *IP Optimizer* is worse than the historical IPE.

Number and percentage of time windows that maintain the historical IPE. It is the number and the percentage of time windows in which the IPE obtained by *IP Optimizer* is equal to the historical IPE.

Mean percentage of improvement in the number of CRs. It is the mean percentage of improvement in the number of conflict resolutions between the integration sequence obtained by *IP Optimizer* and the historical one.

Mean percentage of IPE improvement. It is the mean percentage of IPE improvement between the integration sequence obtained by *IP Optimizer* and the historical one.

5 Results

In this section, we report the main results achieved in our work. We first analyze the distribution of pairwise conflicts in the projects and the proportion of time windows with and without pairwise conflicts for each project. Next, we make a comparative analysis of the historical integration process and the one obtained by *IP Optimizer* on the time windows with pairwise conflicts. The results are shown in detail for time windows of 14 days because it is both the usual sprint length in agile methodologies such as Scrum [9] and the size of the merge window used in the Linux kernel project [11]. In addition, in section 5.3 we make a comparative analysis of the results obtained using time windows of different sizes.

Category	Projects (%)	# Projects
no conflict	20.00%	52
low amount of conflicts	30.00%	78
moderate amount of conflicts	28.85%	75
high amount of conflicts	21.15%	55

Table 1. Percentage and number of projects by pairwise conflict quantity

5.1 Projects time windows: Pairwise conflicts

Figure 7 shows a histogram of pairwise conflicts. We can see 52 projects without pairwise conflicts (20%), 78 projects (30%) with less than 50 pairwise conflicts in their entire history, 75 projects (28.85%) with a moderate number of conflicts (between 51 and 530 pairwise conflicts) and finally 55 projects (21.15%) with a large number of pairwise conflicts. Therefore, we classify projects as **no conflict**, **low amount of conflicts**, **moderate amount of conflicts** and **high amount of conflicts**. Table 1 shows the percentage and number of projects for each category.

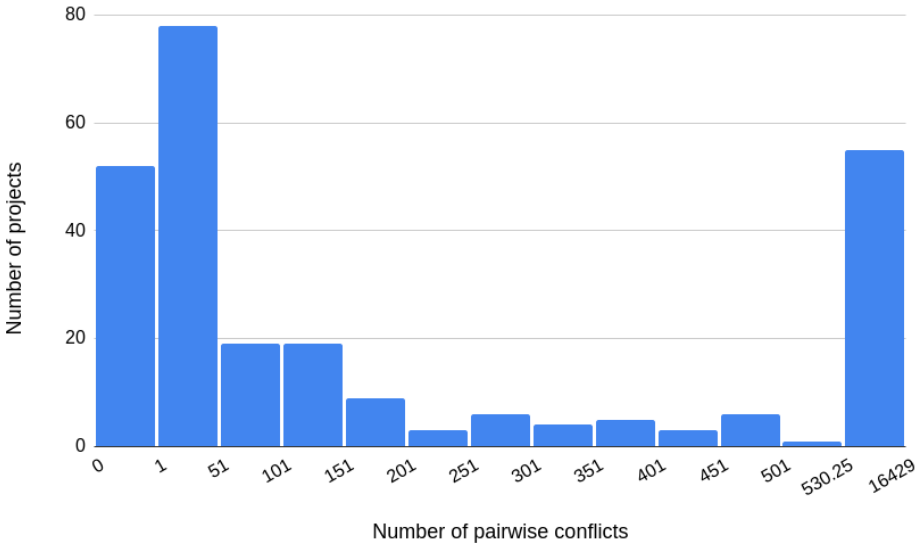

Fig. 7. Pairwise conflicts histogram

For the rest of the analysis we do not take into account the projects that do not have any pairwise conflicts since they do not provide relevant information for the analysis. Considering the remaining projects, on average 18.82% of the time windows have pairwise conflicts while the remaining 81.18% are time windows without conflicts. Figure 8 shows the percentage of time windows with and without pairwise conflicts per project ordered by the highest percentage of time windows with pairwise conflicts. So on the right we have the projects (3.85%) that have no time window with pairwise

Category	Projects (%)	# Projects
no conflicting time windows	3.85%	8
low amount of conflicting time windows	48.56%	101
moderate amount of conflicting time windows	38.46%	80
high amount of conflicting time windows	9.13%	19

Table 2. Percentage and number of projects by amount of conflicting time windows category

conflicts. Then, on the left, we can see the projects that have the highest proportion of time windows with pairwise conflicts: 9.13% of the projects have more than 50% of the time windows with conflicts; 17.31% have between 25% and 50% of time windows with conflict. It is followed by 21.15% of projects that have between 10% and 25% conflicting time windows. And finally, 48.56% of projects that have less than 10% of time windows with conflicts. Therefore, we classify projects as **no conflicting time windows**, **low amount of conflicting time windows**, **moderate amount of conflicting time windows** and **high amount of conflicting time windows**. Table 2 shows the percentage and number of projects for each category.

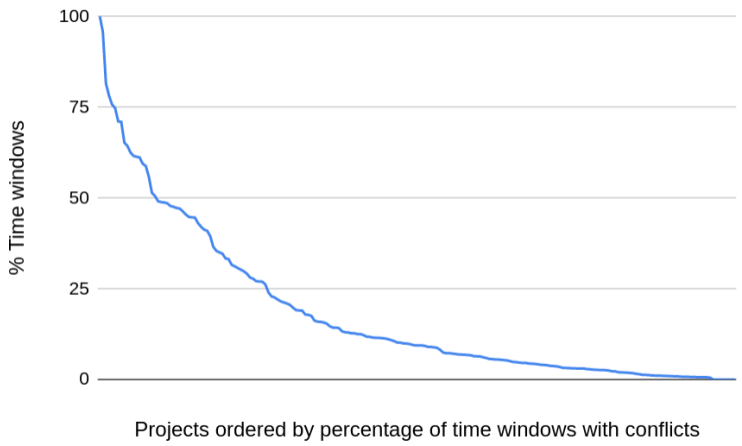

Fig. 8. Percentage of time windows with and without pairwise conflicts by project

5.2 Time windows with pairwise conflicts

In this section, we analyze in detail the time windows with pairwise conflicts since for the time windows without pairwise conflicts the historical integration sequence and the PR group integration sequence obtained by *IP Optimizer* both give the same IPE and the same number of CRs and do not provide any relevant information.

IPE. Considering the time windows of all the remaining projects, *IP Optimizer* improves the historical IPE for 94.16% of the time windows; for the 3.83% of the time windows *IP Optimizer* achives the same IPE compared to the historical one and only for 2.01% *IP Optimizer* worsens the historical IPE.
Figure 9 shows the percentage of time windows where *IP Optimizer* improves, worsens or mantains the same IPE compared to the historical one per project ordered by the percentage of time windows that improve the historical IPE. Table 3 shows the percentage and number of projects in which the IPE is improved. We can see that *IP Optimizer* improves all the time windows for 52% of the projects. 25% of the projects improves more than 90% of the time windows and 18% improves between 75% and 90% of the time windows. 4.50% improves between 50% and 75% of the time windows. Only one project (0.50%) mantains the same IPE for all the time windows, but this project only has one time window with pairwise conflicts.

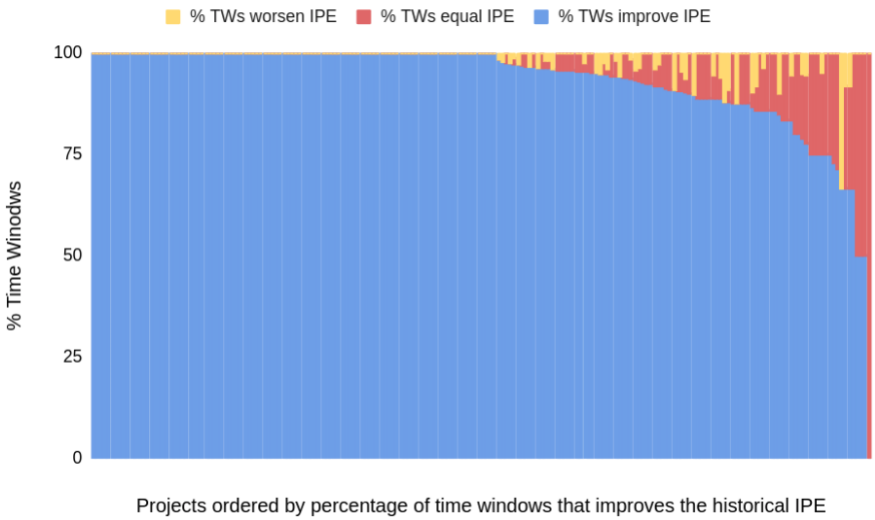

Fig. 9. Percentage of time windows that improves, maintains and worse the historical IPE by project

Regarding the magnitude of the improvement of the historical IPE achieved by *IP Optimizer*, the average of the mean improvement of the historical IPE of all the projects is 146.15%. That is, on average, the IPE obtained by *IP Optimizer* is 146.15% higher than the historical IPE.
Figure 10 shows the mean IPE improvement achieved by *IP Optimizer* compared to the historical one per project ordered by the highest mean number of pairwise conflicts. Table 4 shows the percentage and number of projects by range of mean improvement of the IPE. For 37.50% of the projects, the mean IPE improvement is less than 1. This means that the mean improvement percentage is between 0% and 100%. In 42% of the projects, *IP Optimizer* improves on average the historical IPE between 1 and 2 times, that is, the mean IPE improvement percentage is between 100% and 200%. Then 14%

Percentage range of time windows in which the IPE is improved	Percentage of projects	Number of projects
= 100%	52.00%	104
>= 90%, < 100%	25.00%	50
>= 75%, < 90%	18.00%	36
>= 50%, < 75%	4.50%	9
= 0%	0.50%	1

Table 3. Percentage and number of projects by percentage range of time windows in which the IPE is improved

of the projects have a mean IPE improvement between 2 and 3 times and 6.50% of the projects have a mean IPE improvement higher than 3 times. There is a project where *IP Optimizer* improves on average the historical IPE more than 7 times and another project that *IP Optimizer* improves on average the historical IPE more than 10 times. It should be noted that when the number of pairwise conflicts is larger, the IPE tends to improve less than when the number of pairwise conflicts is smaller. The green line shows this trend.

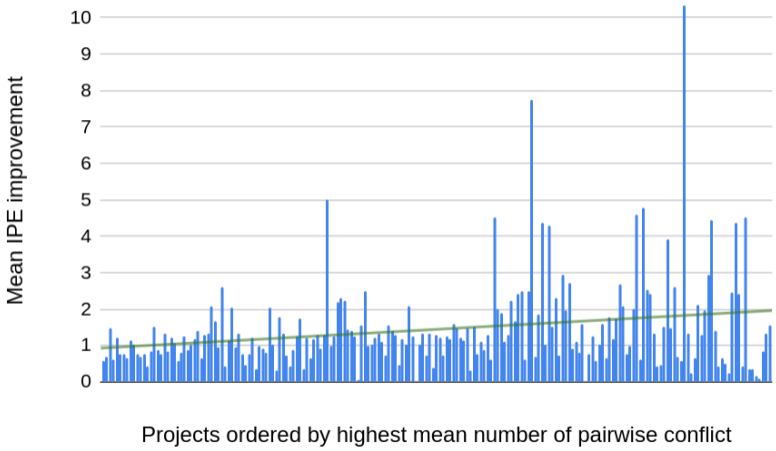

Fig. 10. Mean IPE improvement by project

Number of CRs. Considering the time windows of all the projects, *IP Optimizer* improves the historical number of CRs for 67.16% of the time windows and achieved the same number of CR for 32.84% of the time windows.
Figure 11 shows the percentage of time windows where *IP Optimizer* improves or mantains the number of CRs with respect to the historical one per project, ordered by the percentage of time windows that improves the CRs number. Table 5 shows the percentage and number of projects in which the number of CRs is improved. The

Mean IPE improvement range	Projects (%)	# Projects
< 1	37.50%	75
$>= 1, < 2$	42.00%	84
$>= 2, < 3$	14.00%	28
>3	6.50%	13

Table 4. Percentage and number of projects by range of mean improvement of the IPE

projects in which *IP Optimizer* improves the number of CRs for all time windows (5.50%) and those in which the same number of CRs is maintained compared to the historical one for all time windows (16%) have few time windows with pairwise conflicts. There is 18.50% of projects in which the number of CRs is improved for more than 75% of the time windows and 31% that improve between 50% and 75% of the time windows. Finally, for 29% of the projects, the number of CRs is improved in less than 50% of the time windows.

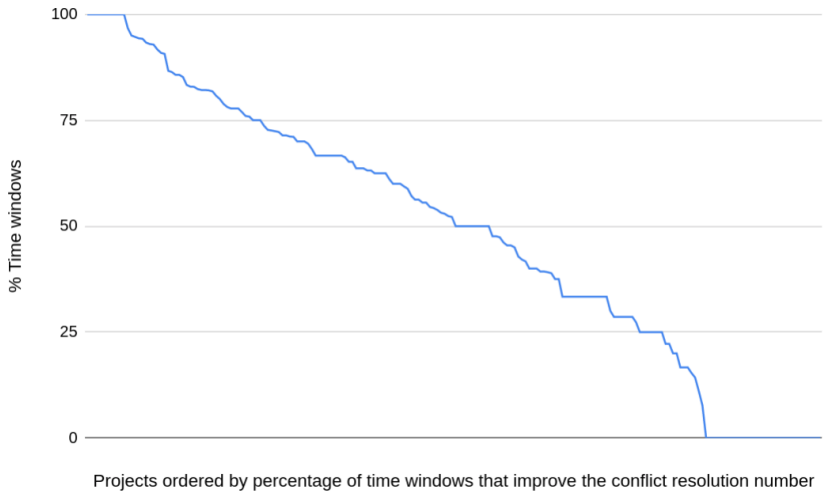

Fig. 11. Percentage of time windows that improves and maintains the number of CRs by project

Regarding the magnitude of the improvement of the historical number of CRs achieved by *IP Optimizer*, the average of the mean improvement of the historical number of CRs of all the projects is 134.28%. That is, on average, the number of CRs obtained by *IP Optimizer* is 134.28% lower than the historical one.
Figure 12 shows the mean number of CRs improvement achieved by *IP Optimizer* compared to the historical one per project ordered by the highest mean number of pairwise conflicts. Table 6 shows the percentage and number of projects by range of mean improvement of the number of CRs. For 48.50% of the projects, the mean number of CRs improvement is less than 1. This means that the mean improvement percentage

Percentage range of time windows in which the number of CRs is improved	Percentage of projects	Number of projects
= 100%	5.50%	11
>= 75%, < 100%	18.50%	37
>= 50%, < 75%	31.00%	62
> 0 %, < 50%	29.00%	58
= 0%	16.00%	32

Table 5. Percentage and number of projects by percentage range of time windows in which the number of CRs is improved

is between 0% and 100%. In 29.50% of the projects, *IP Optimizer* improves on average the historical number of CRs between 1 and 2 times, that is, the mean number of CRs improvement percentage is between 100% and 200%. Then, 12.50% of the projects have a mean number of CRs improvement between 2 and 3 times and 9.50% of the projects have a mean number of CR improvement higher than 3 times. There is a project where *IP Optimizer* improves on average the historical number of CRs more than 10 times. It should be noted that when the number of pairwise conflicts is larger, the number of CRs tends to improve more than when the number of pairwise conflicts is smaller. The green line shows this trend.

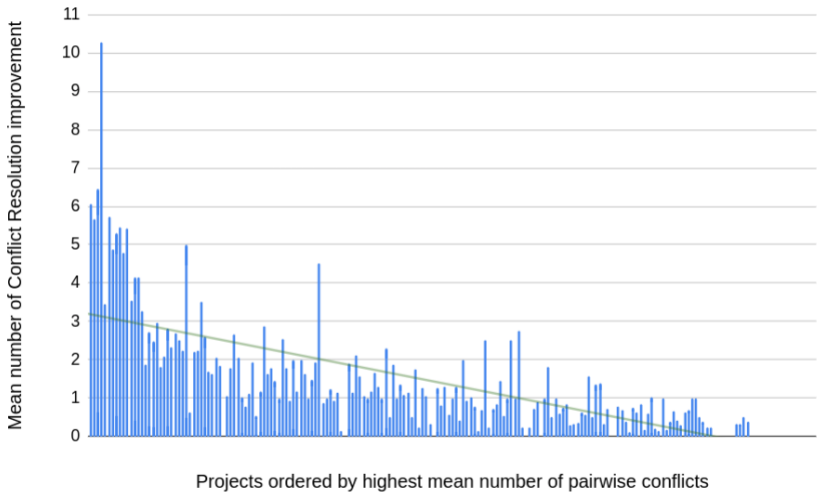

Fig. 12. Mean number of CR improvement by project

5.3 Comparison of time windows of different sizes

In this section, we make a comparison of the main results obtained using different time window sizes. Table 7 shows the results for 7-days, 14-days, 28-days, 60-days and

Mean number of CR improvement range	Projects (%)	# Projects
< 1	48.50%	97
$>= 1, < 2$	29.50%	59
$>= 2, < 3$	12.50%	25
>3	9.50%	19

Table 6. Percentage and number of projects by range of mean improvement of the number of CRs

Time window size	7-days	14-days	28-days	60-days	90-days
# Time windows (avg)	226.61	113.88	57.22	27.06	18.04
% Time windows with conflicts	12.57%	18.82%	26.79%	35.35%	41.63%
% Time windows without conflicts	87.43%	81.18%	73.21%	64.65%	58.37%
% Time windows that improve historical IPE	89.83%	94.16%	96.88%	98.28%	98.58%
% Time windows that do not change historical IPE	7.39%	3.83%	2.12%	0.95%	0.90%
% Time windows that worsen historical IPE	2.78%	2.01 %	1.00%	0.77%	0.52%
% Time windows that improve historical number of CR	59.55%	67.16%	72.76%	76.65%	77.30%
% Time windows that do not change historical number of CR	40.45%	32.84%	27.24%	23.35%	22.70%
Mean IPE improvement percentage	128.69%	146.15%	178.25%	186.40%	211.69%
Mean number of CR improvement percentage	95.21%	134.28%	198.10%	306.84%	385.08%

Table 7. Comparison of the main results obtained for different time window sizes

90-days time windows. These results correspond to the average of all the projects that have at least one time window with conflicts.
We can see that, not surprisingly, as the window size is greater, the percentage of time windows with pairwise conflicts is also greater. The same happens with the percentage of time windows that improve the historical IPE and the number of historical CRs. Moreover, the magnitude of that improvement is also greater when the time window size is greater.

6 Discussion

In this section, we disccuss the results and we answer the research questions.

6.1 RQ1. To what extent do conflicts occur in software projects managed with DVCS?

To answer this question, we focus on Figure 7 and 8. From Figure 7, we can extract the number of pairwise conflicts that the projects have throughout their history and

in Figure 8 we can see how they are distributed over time by analyzing how many integration time windows are affected by conflicts.
We see that half of the projects have a moderate and high number of pairwise conflicts and the other half have a low number of pairwise conflicts or no conflict. We also see that, on average, 18.82% of project time windows have conflicts of which 47.59% of projects have a moderate to high number of conflicting integration time windows and the rest of the projects (52.41%) have a low number of integration time windows with pairwise conflicts or no conflict.
Therefore, in 50% of the projects, conflicts are very frequent while in the other half they occur infrequently. This can depend on many factors such as the way the projects are managed, the size of the projects or the number of contributors that deserve a study exclusively dedicated to this topic.
On the other hand, Table 7 shows how the percentage of time windows with pairwise conflicts increases when the size of the window is larger. This means that if the integration process is prolonged, that is, if it takes a long time to integrate the PRs, there is possibly more conflicts. This information verifies what continuous integration practice proposes, which suggests integrating changes as soon as possible, even several times a day. In this way, it is avoided that the code where the programmer is working is very outdated and therefore conflicts are avoided.

6.2 RQ2. How much can the integration process of software projects be optimized?

We answer this question by considering two factors of the integration process: the efficiency and the number of CRs. So we compare the integration process efficiency and the number of CRs obtained by *IP Optimizer* with the historical ones.
In previous section, we mentioned that *IP Optimizer* improves the integration process efficiency (IPE) by an average of 146.15% and reduces the number of CRs by an average of 134.28%.
In addition, *IP Optimizer* improves the efficiency of the historical integration process for most (94.16%) of the time windows. This indicates the level of usefulness of the proposal since it not only improves efficiency but also achieves it most of the time. Something similar, but to a lesser extent, occurs with the number of CRs. For 67.16% of the time windows, *IP Optimizer* reduces the number of CRs while it remains the same for the remaining time windows.
For 64.50% of the projects, *IP Optimizer* improves the integration process efficiency by more than double, and for 51.50% of the projects, it reduces the number of CRs by half or more.
Something interesting to note about Figure 10 and Figure 12 is that, while the IPE improvement is greater when there are fewer pairwise conflicts, the reduction in the number of CRs is greater when there are more pairwsie conflicts. The IPE trend according to the number of pairwise conflicts has little slope, that is to say that the difference between there being many pairwise conflicts or few does not modify the IPE improvement to a great extent. On the other hand, in the case of the number of CRs, the trend is very steep, so there is a great difference in the reduction of the number of CRs according to the number of pairwise conflicts.
Table 7 shows how *IP Optimizer* further improves the historical IPE and the number of historical CRs when the size of the windows increases. This makes a lot of sense since there are more conflicts and the project administrator does not have the information

on the existence of conflicts between the PRs to be integrated, so the decisions made about the integration order of the PRs do not take into account the efficiency of the integration process.

7 Threats To Validity

We carefully analyzed the threats to validity based on the work of Feldt and Magazinius [10] and did our best to mitigate them.

7.1 Threats to construct validity

To obtain the pairwise conflicts from the project history, we use the version from which the branch to be integrated was forked and the version in which the PR was created. In many cases, it happened that when looking for one of these versions they were not in the repository. This may be because the versions to be integrated have been squashed when integrating[10]. In these cases, if there was a conflict, we could not find it. This issue was mitigated by the number of projects we used to validate the proposal since the number of conflicts we found is large enough to have relevant data.

7.2 Threats to internal validity

It would be argued that the analysis was not performed on all software projects that have more than 100 PRs. We prioritize projects with the most number of merged PRs (those with more than 1000) because they include all relevant cases. To mitigate this risk, we performed the analysis on 100 randomly selected projects that have less than 1000 merged PRs and verified that the results do not provide relevant information.

7.3 Threats to conclusion validity

We are validating *IP Optimizer* in static scenarios. We take the merged PRs in a time window of the project's history and apply *IP Optimizer* on that set of PRs, comparing the efficiency of the historical integration sequences and that obtained by *IP Optimizer*. However, further studies are needed in a dynamic environment. That is, to implement a tool integrated with the code repositories and evaluate it in real time considering all the open PRs and offering the information to the project administrator so that she/he can carry out the integration process efficiently.

7.4 Threats to external validity

Due to the fact of having to take the merged PRs where the source branch and the target branch are in the same repository, we were not able to evaluate projects that use the Forking workflow[11], which are very popular in open source software development. Likewise, we consider that we were able to evaluate the proposal in a sufficiently large number of projects which allows us to generalize the results to apply them to projects that use this workflow as well.

[10] https://git-scm.com/docs/git-merge#Documentation/git-merge.txt—squash
[11] https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

8 Related Work

In a previous work [24], we proposed an approach to optimize the efficiency of the integration process through the prioritization of PRs and we validated it by analyzing 7 representative historical integration sequences of the Antlr4 project[12]. In the present work, we conduct an empirical study that seeks to understand how frequent conflicts are in software projects managed with DVCS and analyze how much the integration process carried out in the history of software projects can be improved establishing as baseline the results obtained by *IP Optimizer*.

In recent years, several empirical studies have been conducted to understand how contributors and project administrators work with the PbD model from a general perspective, or from the perspective of contributors, project aministrators or contributions [29, 30, 15, 17, 16]. The results obtained in our work have a direct implication for project administrators, although they also provide information about the development process in general and thus increase the body of knowledge on software engineering practices.

Some works study latency factors [15, 38, 33, 35, 20] proposing different quantitative / qualitative analyzes to identify latency factors in the PR review process. In our work, we find that the latency in the integration of the PRs can cause more conflicts. There are works that seek to reduce latency by recommending the most suitable project administrator for reviewing a given PR [28, 32, 34, 19, 18]. Other works seek to lower the latency of the most important PRs, prioritizing them by response and acceptance likelihood [31, 4, 5]. Zhao *et al.* [40] also propose a learning-to-rank approach to prioritize PRs that can be quickly reviewed by project administrators in order to review more PRs in a period of time or be able to review any PR when they have a few minutes. Recently, Saini and Britto [27] use a Bayesian Network to prioritize PRs based on acceptance probability, change type (*i.e.,* bug fixing, new feature, refactoring) and presence or absence of merge conflicts. Their main goal is to decrease the overall lead time of code review process along with helping to reduce the workload of project administrators.

Other works study the PR acceptance factors [29, 25, 17, 36, 21] identifying and analyzing the social and technical factors behind PR acceptance or rejection. Gousios *et al.* [15] indicate that 27% of rejected PRs are conflicting PRs. Our work indicates that there are integration processes that reduce the number of conflict resolutions, so the acceptance rate can also be improved according the integration process.

Finally, there are works that study conflicts in software projects. Some works [22, 37] analyze the frequency and the difficulty of resolution. Accioly *et al.* [3] conduct an empirical study that analyze the effectiveness of two types of code changes as conflict predictors in open-source Java projects. Ghiotto *et al.* [12] study the merge conflicts found in the histories of 2731 open-source Java projects. They characterize merge conflicts in terms of number of chunks, size, and programming language constructs involved, classify the manual resolution strategies that developers use to address these merge conflicts, and analyze the relationships between various characteristics of merge conflicts and chosen resolution strategies. Our work also performs an analysis of merge conflicts but between PRs. Furthermore, our analysis is independent of the code structures, so we are not limited to evaluating a single programming language.

[12] https://www.antlr.org/

9 Conclusions

In this paper, we conduct an empirical study on 260 open-source projects hosted in GitHub that use PRs intensively to understand how frequent pairwise conflicts are in the Pull-based Development model and evaluate our integration process optimization proposal.

We find that half of the projects have a moderate and high number of pairwise conflicts and the other half have a low or none number of pairwise conflicts. Futhermore, on average, there is a 18.82% of the time windows that have conflicts.

Regarding how much the integration process can be improved, *IP Optimizer* improves the historical integration process for 94.16% of the time windows and does so on average by 146.15%. In addition, it reduces the number of CRs for 67.16% of the time windows and does so on average by 134.28%. Therefore, we can conclude that the integration process can be greatly improved.

It should be noted that the analysis performed is in a static environment and the real environment is dynamic. In addition, the application of the proposal in a real environment could have side effects on work habits of the involved people in the project. Therefore, we plan to develop a tool integrated into platforms that support PbD (*e.g.,* GitHub, Bitbucket), in order to evaluate *IP Optimizer* in a real environment and study the side effects on work habits when using our approach.

We also plan to further study the occurrence of conflicts in projects that use PbD, in order to discover the factors for some projects to be more conflictive than others. Finally, it is also a perspective for our work to study accurate models for the evaluation of the cost and gain of PR merges.

References

1. Online appendix homepage, https://anonymous.4open.science/r/pull-request-conflicts-7884/docs/index.md
2. Pypl popularity of programming language homepage, https://pypl.github.io/PYPL.html
3. Accioly, P., Borba, P., Silva, L., Cavalcanti, G.: Analyzing conflict predictors in open-source java projects. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 576–586 (2018)
4. Azeem, M.I., Panichella, S., Di Sorbo, A., Serebrenik, A., Wang, Q.: Action-based recommendation in pull-request development. In: Proceedings of the International Conference on Software and System Processes. pp. 115–124 (2020)
5. Azeem, M.I., Peng, Q., Wang, Q.: Pull request prioritization algorithm based on acceptance and response probability. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). pp. 231–242. IEEE (2020)
6. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley (2000)
7. Bird, C., Zimmermann, T.: Assessing the value of branches with what-if analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11 (2012)
8. Chacon, S., Straub, B.: Pro git. Springer Nature (2014)
9. Diebold, P., Ostberg, J.P., Wagner, S., Zendler, U.: What do practitioners vary in using scrum? In: International Conference on Agile Software Development. pp. 40–51. Springer (2015)

10. Feldt, R., Magazinius, A.: Validity threats in empirical software engineering research-an initial survey. In: Seke. pp. 374–379 (2010)
11. German, D.M., Adams, B., Hassan, A.E.: Continuously mining distributed version control systems: an empirical study of how linux uses git. Empirical Software Engineering **21**(1), 260–299 (2016)
12. Ghiotto, G., Murta, L., Barros, M., Van Der Hoek, A.: On the nature of merge conflicts: A study of 2,731 open source java projects hosted by github. IEEE Transactions on Software Engineering **46**(8), 892–915 (2018)
13. González-Barahona, J.M., Robles, G.: On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. Empirical Software Engineering **17**(1-2), 75–89 (2012)
14. Gousios, G.: The ghtorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 233–236. MSR '13, IEEE Press, Piscataway, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2487085.2487132
15. Gousios, G., Pinzger, M., Deursen, A.v.: An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering. pp. 345–355 (2014)
16. Gousios, G., Storey, M.A., Bacchelli, A.: Work practices and challenges in pull-based development: the contributor's perspective. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 285–296. IEEE (2016)
17. Gousios, G., Zaidman, A., Storey, M.A., Van Deursen, A.: Work practices and challenges in pull-based development: The integrator's perspective. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 358–368. IEEE (2015)
18. Jiang, J., Lo, D., Zheng, J., Xia, X., Yang, Y., Zhang, L.: Who should make decision on this pull request? analyzing time-decaying relationships and file similarities for integrator prediction. Journal of Systems and Software **154**, 196–210 (2019)
19. Jiang, J., Yang, Y., He, J., Blanc, X., Zhang, L.: Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. Information and Software Technology **84**, 48–62 (2017)
20. Kononenko, O., Rose, T., Baysal, O., Godfrey, M., Theisen, D., De Water, B.: Studying pull request merges: a case study of shopify's active merchant. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. pp. 124–133 (2018)
21. Legay, D., Decan, A., Mens, T.: On the impact of pull request decisions on future contributions. arXiv preprint arXiv:1812.06269 (2018)
22. Ma, P., Xu, D., Zhang, X., Xuan, J.: Changes are similar: Measuring similarity of pull requests that change the same code in github. In: Software Engineering and Methodology for Emerging Domains, pp. 115–128. Springer (2017)
23. Mens, T.: A state-of-the-art survey on software merging. IEEE transactions on software engineering **28**(5), 449–462 (2002)
24. Olmedo, A., Arévalo, G., Cassol, I., Urtado, C., Vauttier, S.: Improving integration process efficiency through pull request prioritization. In: ENASE 2022-17th International Conference on Evaluation of Novel Approaches to Software Engineering. pp. 62–72. SCITEPRESS-Science and Technology Publications (2022)
25. Rahman, M.M., Roy, C.K.: An insight into the pull requests of github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 364–367 (2014)

26. Rodríguez-Bustos, C., Aponte, J.: How distributed version control systems impact open source software projects. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). pp. 36–39. IEEE (2012)
27. Saini, N., Britto, R.: Using machine intelligence to prioritise code review requests. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 11–20. IEEE (2021)
28. Thongtanunam, P., Kula, R.G., Cruz, A.E.C., Yoshida, N., Iida, H.: Improving code review effectiveness through reviewer recommendations. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering. pp. 119–122 (2014)
29. Tsay, J., Dabbish, L., Herbsleb, J.: Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th international conference on Software engineering. pp. 356–366 (2014)
30. Tsay, J., Dabbish, L., Herbsleb, J.: Let's talk about it: evaluating contributions through discussion in github. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. pp. 144–154 (2014)
31. Van Der Veen, E., Gousios, G., Zaidman, A.: Automatically prioritizing pull requests. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. pp. 357–361. IEEE (2015)
32. Ying, H., Chen, L., Liang, T., Wu, J.: Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In: 2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE). pp. 29–35. IEEE (2016)
33. Yu, Y., Wang, H., Filkov, V., Devanbu, P., Vasilescu, B.: Wait for it: Determinants of pull request evaluation latency on github. In: 2015 IEEE/ACM 12th working conference on mining software repositories. pp. 367–371. IEEE (2015)
34. Yu, Y., Wang, H., Yin, G., Wang, T.: Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? Information and Software Technology **74**, 204–218 (2016)
35. Yu, Y., Yin, G., Wang, T., Yang, C., Wang, H.: Determinants of pull-based development in the context of continuous integration. Science China Information Sciences **59**(8), 1–14 (2016)
36. Zampetti, F., Bavota, G., Canfora, G., Di Penta, M.: A study on the interplay between pull request review and continuous integration builds. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 38–48. IEEE (2019)
37. Zhang, X., Chen, Y., Gu, Y., Zou, W., Xie, X., Jia, X., Xuan, J.: How do multiple pull requests change the same code: A study of competing pull requests in github. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 228–239. IEEE (2018)
38. Zhang, Y., Yin, G., Yu, Y., Wang, H.: A exploratory study of@-mention in github's pull-requests. In: 2014 21st Asia-Pacific Software Engineering Conference. vol. 1, pp. 343–350. IEEE (2014)
39. Zhang, Y., Yin, G., Yu, Y., Wang, H.: Investigating social media in github's pull-requests: a case study on ruby on rails. In: Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies. pp. 37–41 (2014)
40. Zhao, G., da Costa, D.A., Zou, Y.: Improving the pull requests review process using learning-to-rank algorithms. Empirical Software Engineering **24**(4), 2140–2170 (2019)